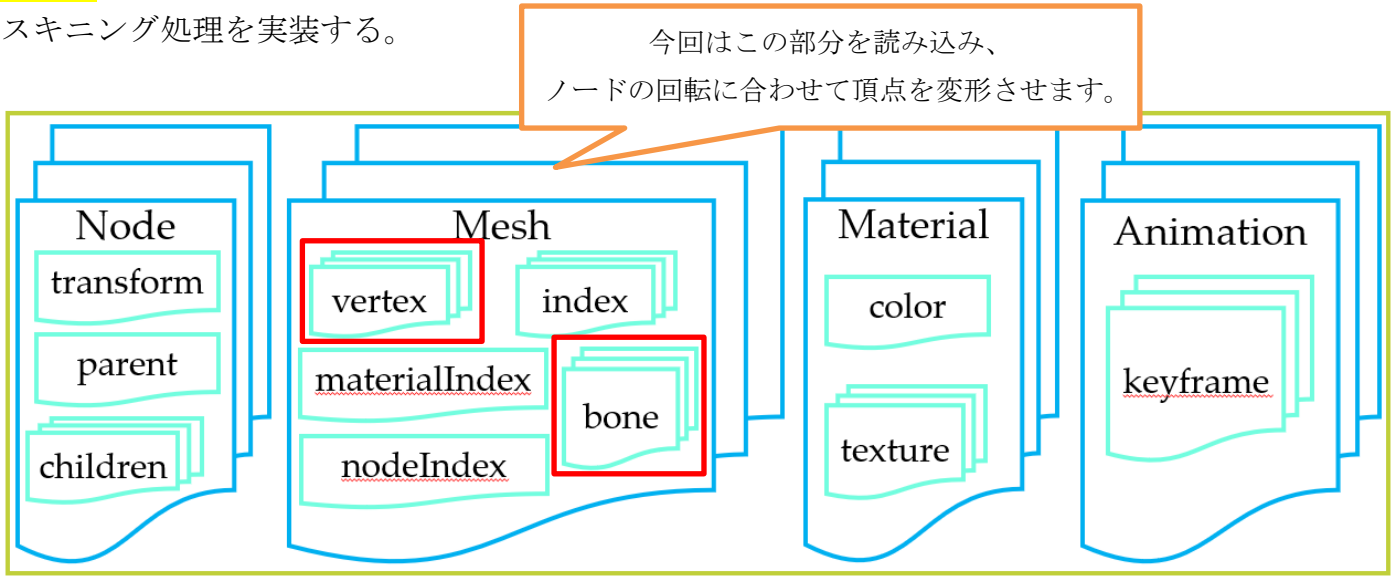
描画エンジン開発 EX

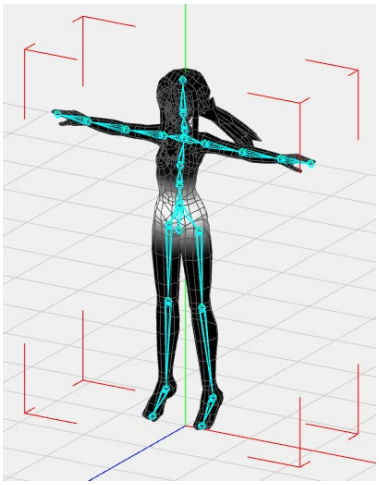
○概要

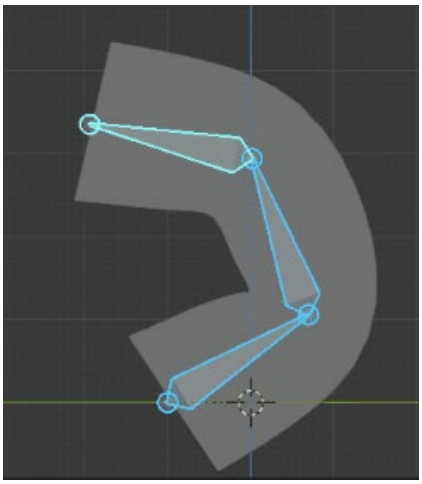
スキニング処理を実装する。

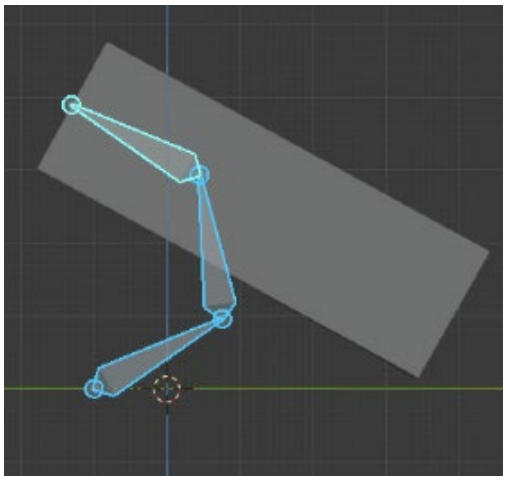


○スキニング

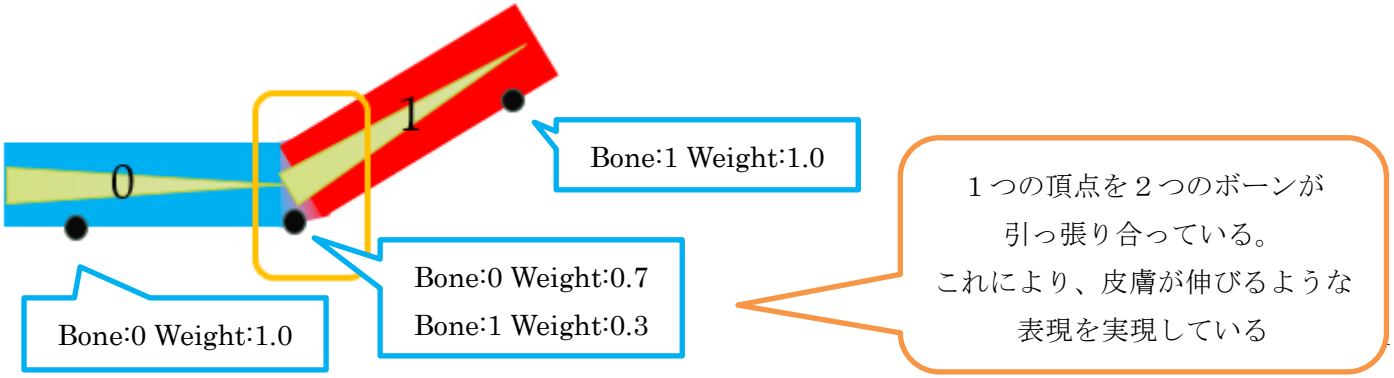
スキニングとは人間で言うところの皮膚や関節を表現するための技術です。
3D モデルで人間を表現する場合、「ボーン」と呼ばれる骨格データが内蔵されています。
ボーンを動かすことで関連しているメッシュの形状を変形させる技術をスキニングと呼びます。





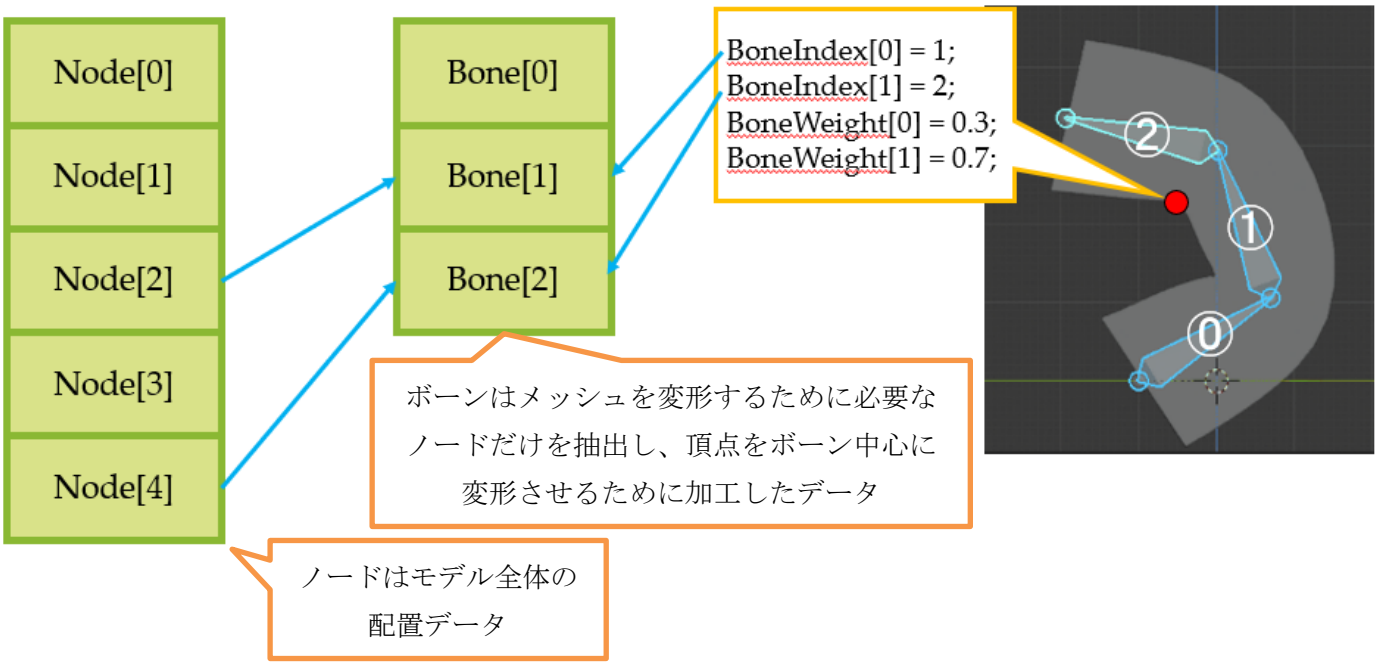


スキニングをするためには頂点毎に「関連するボーン」と「影響する重み」の２つのデータが必要になってきます。



○ボーン

ボーンとはメッシュを変形させるための骨格データのことです。
これまでに学習した「ノード」と同意味で使用することもあるのですが、ここではメッシュを変形させること限定で使用する行列データとして区別します。



○スキニングに必要な頂点データの収集

まず頂点データにスキニングに必要な「関連するボーン」と「影響する重み」データの２つを追加します。

Model.h

```
---省略---

class Model
{
public:
    ---省略---

    struct Vertex
    {
        DirectX::XMFLOAT3     position = { 0, 0, 0 };
        DirectX::XMFLOAT4     boneWeight = { 1, 0, 0, 0 };
        DirectX::XMUINT4      boneIndex = { 0, 0, 0, 0 };
        DirectX::XMFLOAT2     texcoord = { 0, 0 };
    };

    ---省略---
};
```

boneWeight は
ｘｙｚｗの合計が 1.0 になる

AssimpImporter で頂点がボーンに影響するデータを収集します。

AssimpImporter.cpp

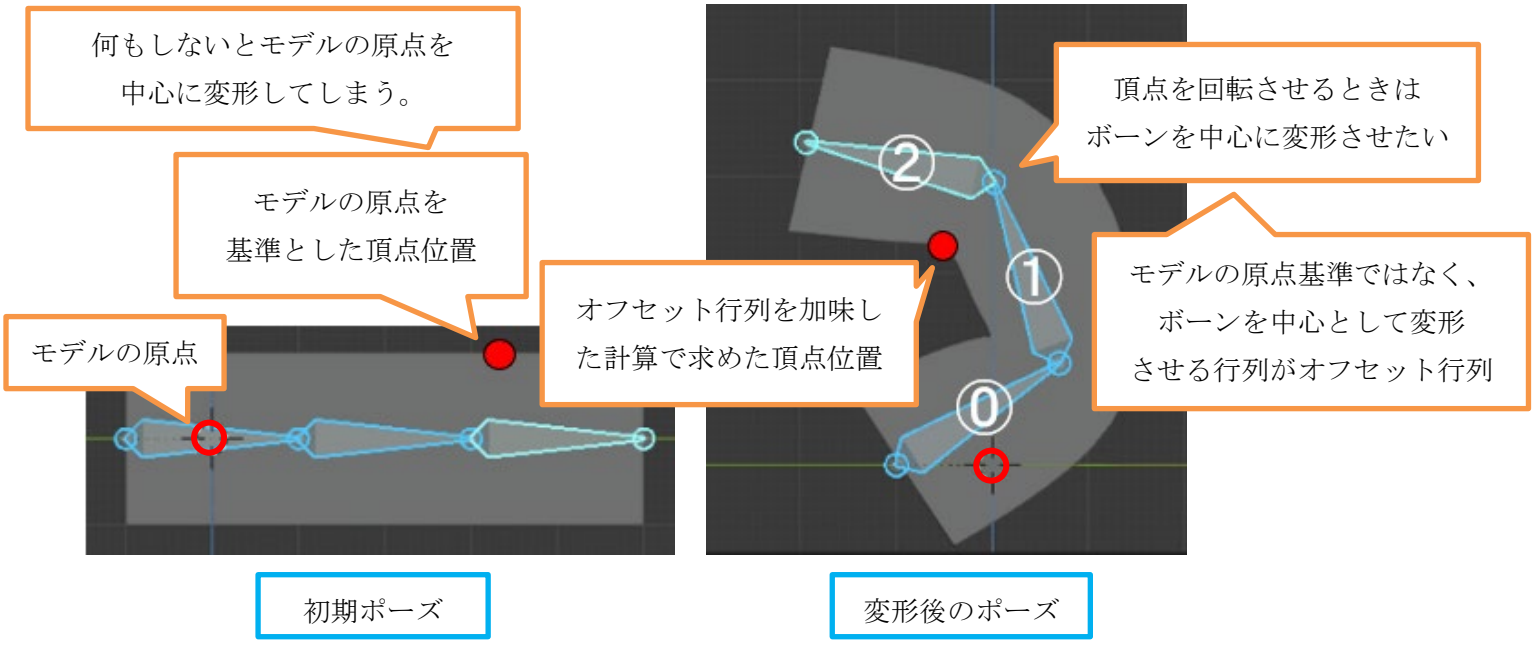
```
---省略---

// コンストラクタ
AssimpImporter::AssimpImporter(const char* filename)
{
    ---省略---
    // インポート時のオプションフラグ
    uint32_t aFlags   = aiProcess_Triangulate             // 多角形を三角形化する
                      | aiProcess_JoinIdenticalVertices   // 重複頂点をマージする
                      | aiProcess_LimitBoneWeights;       // １頂点の最大ボーン影響数を制限する
    ---省略---
}

// メッシュデータを読み込み
void AssimpImporter::LoadMeshes(MeshList& meshes, const NodeList& nodes, const aiNode* aNode,
                                                                          std::string nodePath)
{
    ---省略---

    // メッシュデータ読み取り
    for (uint32_t aMeshIndex = 0; aMeshIndex < aNode->mNumMeshes; ++aMeshIndex)
    {
        ---省略---

        // スキニングデータ
        if (aMesh->mNumBones > 0)
        {
            // ボーン影響力データ
            struct BoneInfluence
            {
                uint32_t indices[4] = { 0, 0, 0, 0 };
                float    weights[4] = { 1, 0, 0, 0 };
                int      useCount = 0;
            };
            std::vector<BoneInfluence> boneInfluences;
            boneInfluences.resize(aMesh->mNumVertices);

            // メッシュに影響するボーンデータを収集する
            for (uint32_t aBoneIndex = 0; aBoneIndex < aMesh->mNumBones; ++aBoneIndex)
            {
                const aiBone* aBone = aMesh->mBones[aBoneIndex];

                // 頂点影響力データを抽出
                for (uint32_t aWightIndex = 0; aWightIndex < aBone->mNumWeights; ++aWightIndex)
                {
                    const aiVertexWeight& aWeight = aBone->mWeights[aWightIndex];
                    BoneInfluence& boneInfluence = boneInfluences.at(aWeight.mVertexId);
                    boneInfluence.indices[boneInfluence.useCount] = aBoneIndex;
                    boneInfluence.weights[boneInfluence.useCount] = aWeight.mWeight;
                    boneInfluence.useCount++;
```

１つの頂点で影響するボーンの最大数を４つに制限する
※シェーダーで扱えるデータ型が float4 のため。

```
                }
            }
            // 頂点影響力データを格納
            for (size_t vertexIndex = 0; vertexIndex < mesh.vertices.size(); ++vertexIndex)
            {
                Model::Vertex& vertex = mesh.vertices.at(vertexIndex);
                BoneInfluence& boneInfluence = boneInfluences.at(vertexIndex);
                vertex.boneWeight.x = boneInfluence.weights[0];
                vertex.boneWeight.y = boneInfluence.weights[1];
                vertex.boneWeight.z = boneInfluence.weights[2];
                vertex.boneWeight.w = boneInfluence.weights[3];
                vertex.boneIndex.x = boneInfluence.indices[0];
                vertex.boneIndex.y = boneInfluence.indices[1];
                vertex.boneIndex.z = boneInfluence.indices[2];
                vertex.boneIndex.w = boneInfluence.indices[3];
            }
        }
    }
    ---省略---
}
```

○オフセット行列

オフセット行列とはメッシュの頂点をボーンを中心に変形させるための行列です。



○メッシュに関連するボーンデータの収集

Model クラスに Bone 構造体を定義し、ボーンデータを収集します。

Model.h

```
---省略---
```

```
class Model
{
public:
    ---省略---

    struct Bone
    {
        int                         nodeIndex;
        DirectX::XMFLOAT4X4         offsetTransform;
    };

    struct Mesh
    {
        ---省略---
        std::vector<Bone>      bones;
    };

    ---省略---
};
```

位置情報などはノードに入っている
ので参照するためのインデックス

AssimpImporter.h

```
---省略---

class AssimpImporter
{
    ---省略---

private:
    ---省略---

    // 名前からノードインデックス取得
    static int GetNodeIndexByName(const NodeList& nodes, const char* name);

    // aiMatrix4x4 → XMFLOAT4X4
    static DirectX::XMFLOAT4X4 aiMatrix4x4ToXMFLOAT4X4(const aiMatrix4x4& aValue);

    ---省略---
};
```

AssimpImporter.cpp

```
---省略---

// メッシュデータを読み込み
void AssimpImporter::LoadMeshes(MeshList& meshes, const NodeList& nodes, const aiNode* aNode,
                                                                          std::string nodePath)
{
    ---省略---

    // メッシュデータ読み取り
    for (uint32_t aMeshIndex = 0; aMeshIndex < aNode->mNumMeshes; ++aMeshIndex)
    {
        ---省略---
```

```
        // スキニングデータ
        if (aMesh->mNumBones > 0)
        {
            ---省略---

            // メッシュに影響するボーンデータを収集する
            for (uint32_t aBoneIndex = 0; aBoneIndex < aMesh->mNumBones; ++aBoneIndex)
            {
                ---省略---

                // ボーンデータ取得
                Model::Bone& bone = mesh.bones.emplace_back();
                bone.nodeIndex = GetNodeIndexByName(nodes, aBone->mName.C_Str());
                bone.offsetTransform = aiMatrix4x4ToXMFLOAT4X4(aBone->mOffsetMatrix);
            }
            ---省略---
        }
    }
    ---省略---
}

// 名前からノードインデックス取得
int AssimpImporter::GetNodeIndexByName(const NodeList& nodes, const char* name)
{
    int index = 0;
    for (const Model::Node& node : nodes)
    {
        if (node.name == name)
        {
            return index;
        }
        index++;
    }
    return -1;
}

// aiMatrix4x4 → XMFLOAT4X4
DirectX::XMFLOAT4X4 AssimpImporter::aiMatrix4x4ToXMFLOAT4X4(const aiMatrix4x4& aValue)
{
    return DirectX::XMFLOAT4X4(
        static_cast<float>(aValue.a1),
        static_cast<float>(aValue.b1),
        static_cast<float>(aValue.c1),
        static_cast<float>(aValue.d1),
        static_cast<float>(aValue.a2),
        static_cast<float>(aValue.b2),
        static_cast<float>(aValue.c2),
        static_cast<float>(aValue.d2),
        static_cast<float>(aValue.a3),
        static_cast<float>(aValue.b3),
        static_cast<float>(aValue.c3),
        static_cast<float>(aValue.d3),
        static_cast<float>(aValue.a4),
        static_cast<float>(aValue.b4),
        static_cast<float>(aValue.c4),
        static_cast<float>(aValue.d4)
```

```
    );
}
```

ボーンデータからノードを参照しやすいようにポインタ設定します。

Model.h

```
---省略---

class Model
{
public:
    ---省略---

    struct Bone
    {
        ---省略---
        Node*                       node = nullptr;
    };

    ---省略---
};
```

Model.cpp

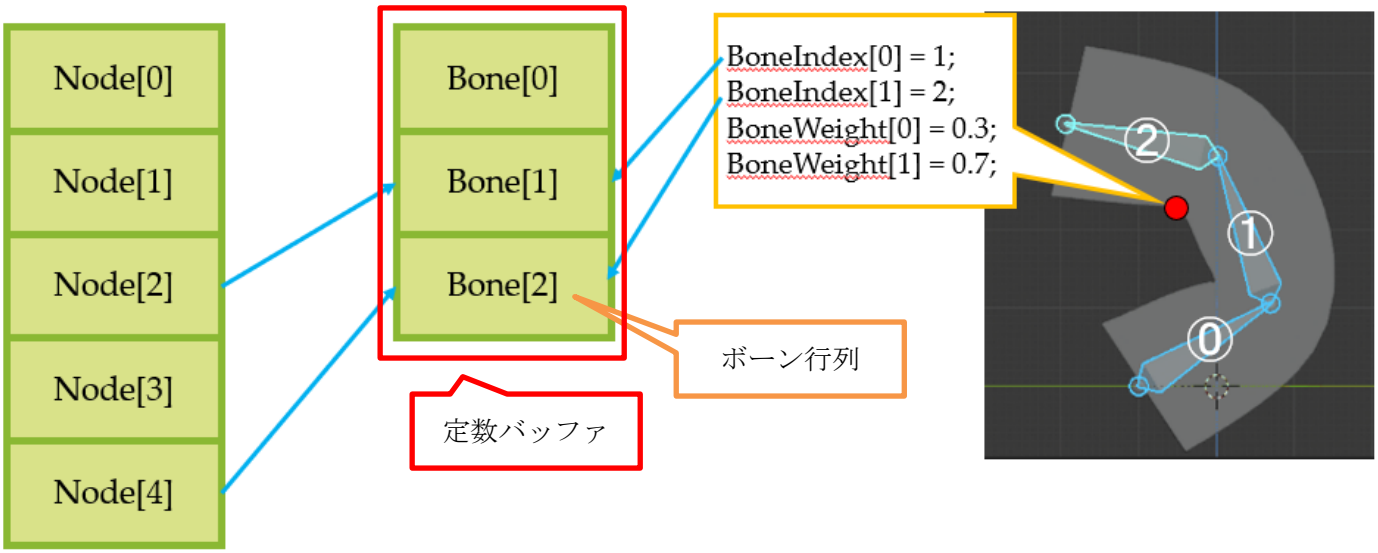
```
---省略---

// コンストラクタ
Model::Model(ID3D11Device* device, const char* filename)
{
    ---省略---

    // メッシュ構築
    for (Mesh& mesh : meshes)
    {
        ---省略---

        // ボーン構築
        for (Bone& bone : mesh.bones)
        {
            // 参照ノード設定
            bone.node = &nodes.at(bone.nodeIndex);
        }
    }
}
```

○スキニング計算

オフセット行列がボーンを中心に頂点を変形させるための行列と説明しました。
このオフセット行列にワールド行列を乗算することで変形する頂点位置がワールドのどこに存在するか計算することができます。

この行列をボーン行列と呼び、ボーン行列データを定数バッファに入れてシェーダーに渡します。

ワールド行列 ＝ ローカル行列 × 親のワールド行列
ボーン行列 ＝ オフセット行列 × ワールド行列
頂点ワールド座標 ＝ 頂点座標 × ボーン行列



スキニングをするための定数バッファとスキニング計算関数を定義します。
Skinning.hlsli を作成しましょう。

Skinning.hlsli

```
cbuffer CbSkeleton : register(b2)
{
    row_major float4x4    boneTransforms[256];
};

float4 SkinningPosition(float4 position, float4 boneWeights, uint4 boneIndices)
{
    float4 p = float4(0, 0, 0, 0);

    [unroll]
    for (int i = 0; i < 4; i++)
    {
        p += (boneWeights[i] * mul(position, boneTransforms[boneIndices[i]]));
    }
    return p;
}
```

頂点シェーダーで Skinning.hlsli をインクルードし、頂点計算を行います。

Phong.hlsli

```
---省略---

cbuffer CbMesh : register(b1)
{
    float4                  materialColor;
    row_major float4x4      worldTransform;
};
```

ボーン行列にワールド行列の計算が加味されているので必要なくなった

PhongVS.hlsl

```
#include "Skinning.hlsli"
#include "Phong.hlsli"

VS_OUT main(
    float4 position       : POSITION,
    float4 boneWeights    : BONE_WEIGHTS,
    uint4  boneIndices    : BONE_INDICES,
    float2 texcoord       : TEXCOORD)
{
    VS_OUT vout = (VS_OUT)0;

    position = SkinningPosition(position, boneWeights, boneIndices);
    vout.vertex = mul(position, mul(worldTransform, viewProjection));
    vout.vertex = mul(position, viewProjection);
    vout.texcoord = texcoord;

    return vout;
}
```

シェーダーにボーン行列を渡すための定数バッファを作成し、ボーン行列を計算します。

PhongShader.h

```
---省略---

class PhongShader : public Shader
{
    ---省略---
private:
    ---省略---
    struct CbMesh
    {
        ---省略---
        DirectX::XMFLOAT4X4    worldTransform;
    };

    struct CbSkeleton
    {
        DirectX::XMFLOAT4X4    boneTransforms[256];
    };

    Microsoft::WRL::ComPtr<ID3D11Buffer>            skeletonConstantBuffer;
};
```

PhongShader.cpp

```
#include "Misc.h"
---省略---

PhongShader::PhongShader(ID3D11Device* device)
{
    // 入力レイアウト
    D3D11_INPUT_ELEMENT_DESC inputElementDesc[] =
    {
        { "POSITION",      0, DXGI_FORMAT_R32G32B32_FLOAT,     0, D3D11_APPEND_ALIGNED_ELEMENT,
                                                                  D3D11_INPUT_PER_VERTEX_DATA, 0 },
        { "BONE_WEIGHTS", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT,
                                                                  D3D11_INPUT_PER_VERTEX_DATA, 0 },
        { "BONE_INDICES", 0, DXGI_FORMAT_R32G32B32A32_UINT,  0, D3D11_APPEND_ALIGNED_ELEMENT,
                                                                  D3D11_INPUT_PER_VERTEX_DATA, 0 },
        { "TEXCOORD",      0, DXGI_FORMAT_R32G32_FLOAT,        0, D3D11_APPEND_ALIGNED_ELEMENT,
                                                                  D3D11_INPUT_PER_VERTEX_DATA, 0 },
    };

    ---省略---

    // スケルトン用定数バッファ
    GpuResourceUtils::CreateConstantBuffer(
        device,
        sizeof(CbSkeleton),
        skeletonConstantBuffer.GetAddressOf());
}

// 描画開始
void PhongShader::Begin(const RenderContext& rc)
{
    ---省略---

    // 定数バッファ設定
    ID3D11Buffer* constantBuffers[] =
    {
        ---省略---
        skeletonConstantBuffer.Get(),
    };
    ---省略---
}

// 描画
void PhongShader::Draw(const RenderContext& rc, const Model* model)
{
    ---省略---

    for (const Model::Mesh& mesh : model->GetMeshes())
    {
        ---省略---

        // メッシュ用定数バッファ更新
        CbMesh cbMesh{};
```

ボーン影響データを追加
※並び順を Vertex 構造体の要素と同じにする

```
struct Vertex
{
    DirectX::XMFLOAT3 position;
    DirectX::XMFLOAT4 boneWeight;
    DirectX::XMUINT4  boneIndex;
    DirectX::XMFLOAT2 texcoord;
};
```

```
        cbMesh.materialColor = mesh.material->color;
        cbMesh.worldTransform = mesh.node->worldTransform;
        dc->UpdateSubresource(meshConstantBuffer.Get(), 0, 0, &cbMesh, 0, 0);

        // スケルトン用定数バッファ更新
        CbSkeleton cbSkeleton{};
        if (mesh.bones.size() > 0)
        {
            for (size_t i = 0; i < mesh.bones.size(); ++i)
            {
                const Model::Bone& bone = mesh.bones.at(i);
                DirectX::XMMATRIX WorldTransform = DirectX::XMLoadFloat4x4(&bone.node->worldTransform);
                DirectX::XMMATRIX OffsetTransform = DirectX::XMLoadFloat4x4(&bone.offsetTransform);
                DirectX::XMMATRIX BoneTransform = OffsetTransform * WorldTransform;
                DirectX::XMStoreFloat4x4(&cbSkeleton.boneTransforms[i], BoneTransform);
            }
        }
        else
        {
            cbSkeleton.boneTransforms[0] = mesh.node->worldTransform;
        }
        rc.deviceContext->UpdateSubresource(skeletonConstantBuffer.Get(), 0, 0, &cbSkeleton, 0, 0);

        ---省略---
    }
}
```

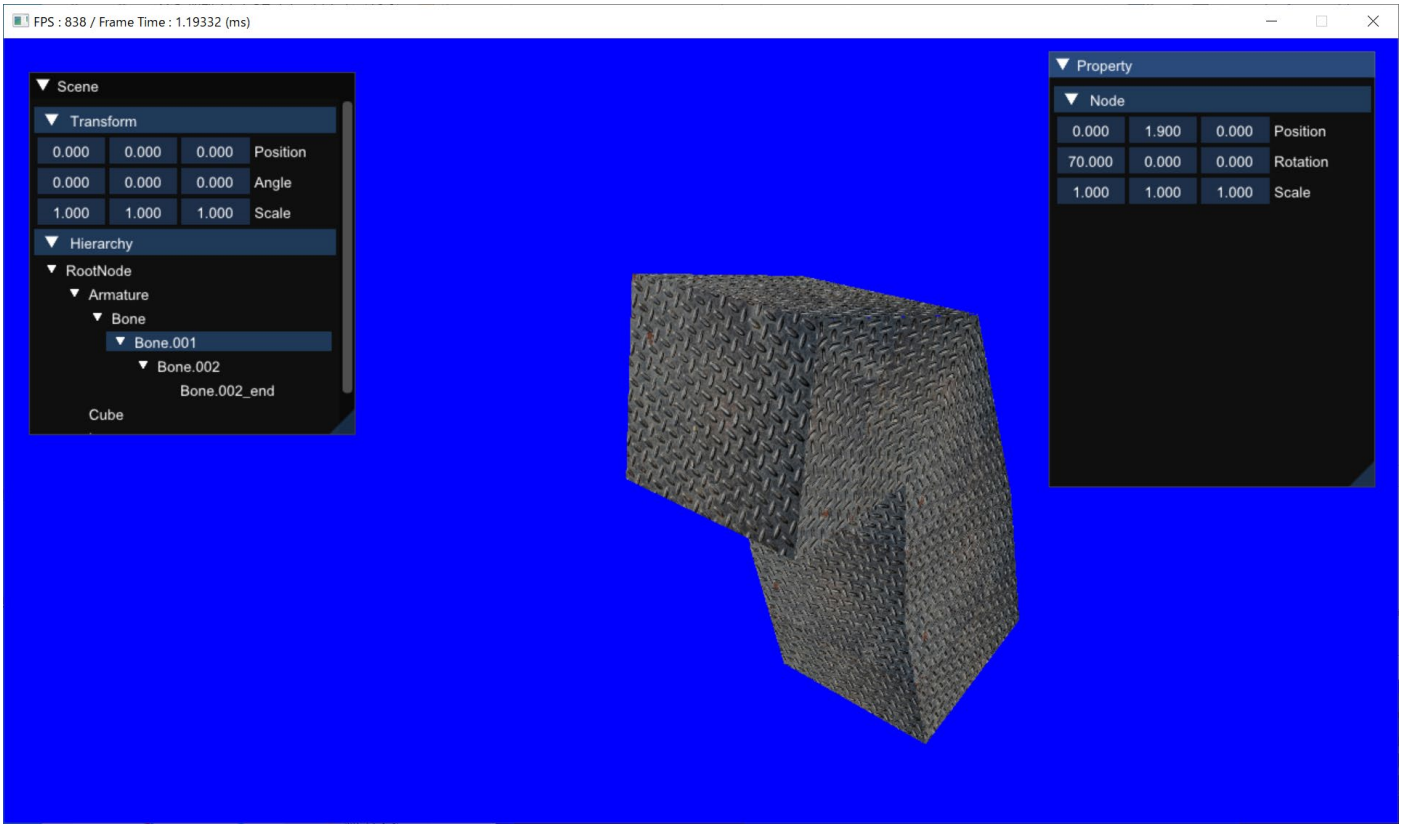
ボーン行列を計算し、
定数バッファに入れる

ボーンがない場合は
ワールド行列を入れる

Scene.cpp

```
---省略---

// コンストラクタ
ModelTestScene::ModelTestScene()
{
    ---省略---

    // モデル作成
    model = std::make_unique<Model>(device, "Data/Model/Cube/cube.003.1.fbx");
    model = std::make_unique<Model>(device, "Data/Model/Cube/cube.004.fbx");
}
```

実行確認してみましょう。
ボーンを選択し、位置や回転を編集してボーンを基準に変形できていれば OK です。


FPS : 838 / Frame Time : 1.19332 (ms)
Scene
Transform
0.000 0.000 0.000 Position
0.000 0.000 0.000 Angle
1.000 1.000 1.000 Scale
Hierarchy
RootNode
Armature
Bone
Bone.001
Bone.002
Bone.002_end
Cube
Property
Node
0.000 1.900 0.000 Position
70.000 0.000 0.000 Rotation
1.000 1.000 1.000 Scale