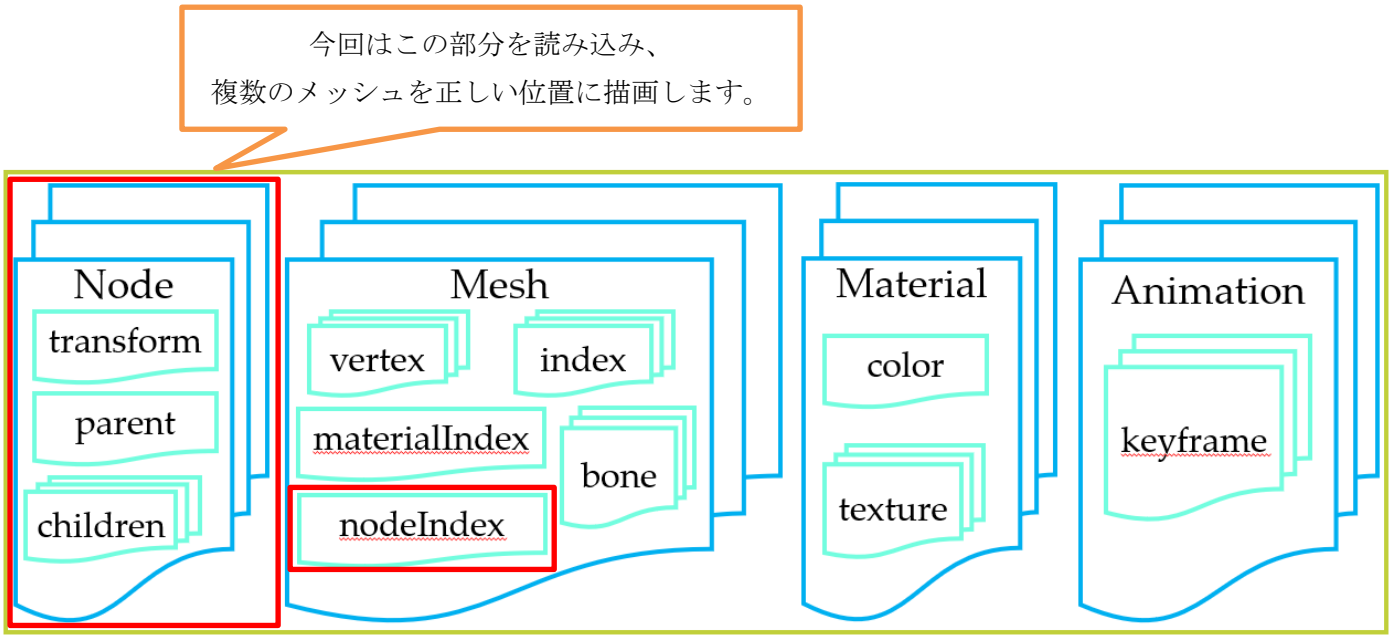
描画エンジン開発 EX

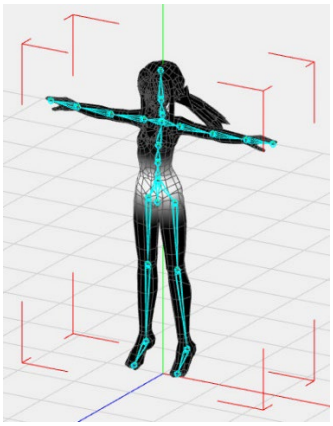
○概要

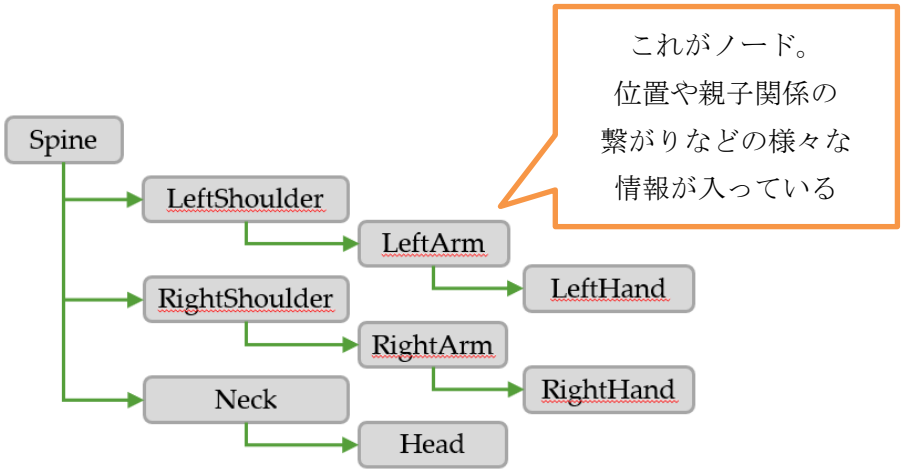
3D モデルファイルには複数のメッシュデータが存在し、それぞれをどこに配置するかの情報を持つノードデータを参照してメッシュを描画します。



○ノード

ノードとは情報の塊のことを指し、情報と情報をつなぐ役割を持っています。
3D モデルデータでの「名前」「位置」「回転」「スケール」「親子関係」などの情報の塊をノードと呼びます。





まずは Model クラスに Node 構造体を定義します。

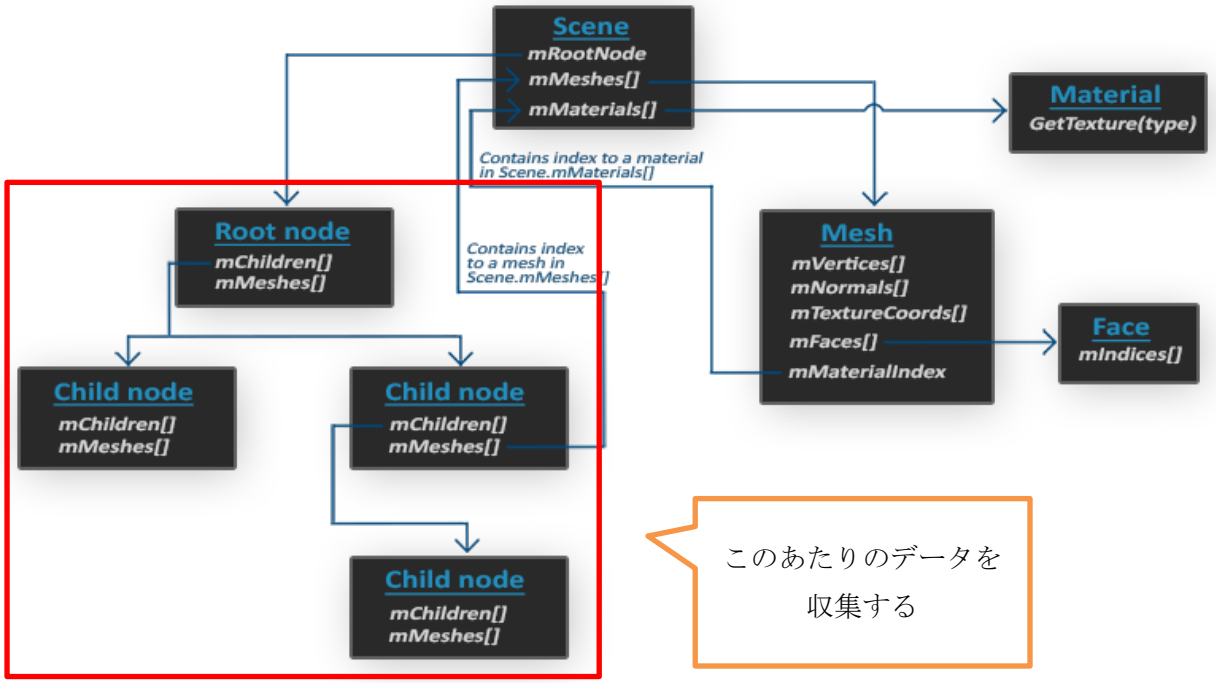
Model.h

```
---省略---

class Model
{
```

```
public:
    ---省略---

    struct Node
    {
        std::string             name;
        std::string             path;
        int                     parentIndex;
        DirectX::XMFLOAT3       position;
        DirectX::XMFLOAT4       rotation;
        DirectX::XMFLOAT3       scale;
    };

    ---省略---
};
```

これで親子関係を表現する

AssimpImporter クラスにノードデータ読み込み関数を作成しましょう。
ノードデータはツリー構造になっているので再帰処理をしてデータを取り出します。



AssimpImporter.h

```
---省略---

class AssimpImporter
{
private:
    ---省略---
    using NodeList = std::vector<Model::Node>;

public:
    ---省略---
```

```
    // ノードデータを読み込み
    void LoadNodes(NodeList& nodes);

private:
    // ノードデータを再帰読み込み
    void LoadNodes(NodeList& nodes, const aiNode* aNode, std::string nodePath, int parentIndex);

    // aiQuaternion → XMFLOAT4
    static DirectX::XMFLOAT4 aiQuaternionToXMFLOAT4(const aiQuaternion& aValue);

    ---省略---
};
```

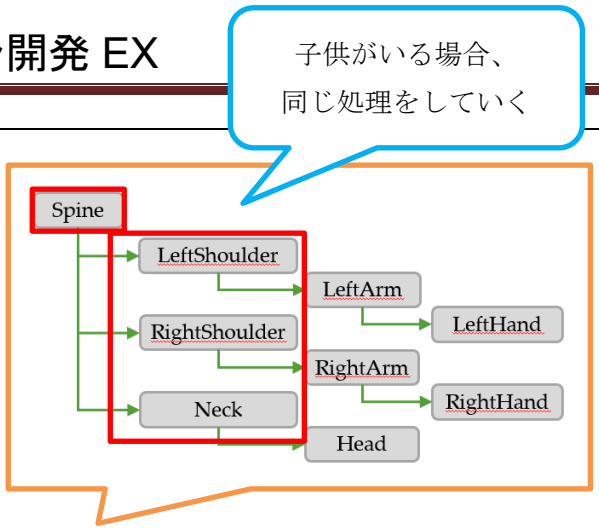
AssimpImporter.cpp

```
---省略---

// コンストラクタ
AssimpImporter::AssimpImporter(const char* filename)
    : filepath(filename)
{
    // 拡張子取得
    std::string extension = filepath.extension().string();
    std::transform(extension.begin(), extension.end(), extension.begin(), tolower); // 小文字化

    // FBXファイルの場合は特殊なインポートオプション設定をする
    if (extension == ".fbx")
    {
        // $AssimpFBX$が付加された余計なノードを作成してしまうのを抑制する
        aImporter.SetPropertyInteger(AI_CONFIG_IMPORT_FBX_PRESERVE_PIVOTS, false);
    }

    ---省略---
}

---省略---

// ノードデータを読み込み
void AssimpImporter::LoadNodes(NodeList& nodes)
{
    LoadNodes(nodes, aScene->mRootNode, "", -1);
}

// ノードデータを再帰読み込み
void AssimpImporter::LoadNodes(NodeList& nodes, const aiNode* aNode, std::string nodePath, int parentIndex)
{
    // ノードパスの作成
    nodePath += "/";
    nodePath += aNode->mName.C_Str();

    // トランスフォームデータ取り出し
    aiVector3D aScale, aPosition;
    aiQuaternion aRotation;
    aNode->mTransformation.Decompose(aScale, aRotation, aPosition);
```

Assimp が FBX を読み込む際に余計なことをしてしまう場合があるので抑制する

先頭のノードから順に処理していく

```
    // ノードデータ格納
    Model::Node& node = nodes.emplace_back();
    node.name = aNode->mName.C_Str();
    node.path = nodePath;
    node.parentIndex = parentIndex;
    node.scale = aiVector3DToXMFLOAT3(aScale);
    node.rotation = aiQuaternionToXMFLOAT4(aRotation);
    node.position = aiVector3DToXMFLOAT3(aPosition);

    parentIndex = static_cast<int>(nodes.size() - 1);

    // 再帰的に子ノードを処理する
    for (uint32_t aNodeIndex = 0; aNodeIndex < aNode->mNumChildren; ++aNodeIndex)
    {
        LoadNodes(nodes, aNode->mChildren[aNodeIndex], nodePath, parentIndex);
    }
}

// aiQuaternion → XMFLOAT4
DirectX::XMFLOAT4 AssimpImporter::aiQuaternionToXMFLOAT4(const aiQuaternion& aValue)
{
    return DirectX::XMFLOAT4(
        static_cast<float>(aValue.x),
        static_cast<float>(aValue.y),
        static_cast<float>(aValue.z),
        static_cast<float>(aValue.w)
    );
}
```



ノードデータを取得するプログラムを実装したので、次はメッシュがどのノードを参照するのかを取り出すプログラムを実装します。

Model クラスの Mesh 構造体に参照するノードのインデックスデータを追加します。

Model.h

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

    struct Mesh
    {
        ---省略---
        int             nodeIndex = 0;
    };

    ---省略---
};
```

続いてメッシュデータの読み取りを改造します。

現時点では１つのメッシュデータしか読み取っていなかったのですが、ノードを辿って複数のメッシュデータの読み取りと参照ノードインデックスの設定をしていきます。

AssimpImporter.h

```
---省略---

class AssimpImporter
{
    ---省略---
public:
    ---省略---

    // メッシュデータを読み込み
    void LoadMeshes(MeshList& meshes);
    void LoadMeshes(MeshList& meshes, const NodeList& nodes);

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

    // パスからノードインデックス取得
    static int GetNodeIndexByPath(const NodeList& nodes, const char* path);

    ---省略---
};
```

AssimpImporter.cpp

```
---省略---

// メッシュデータを読み込み
void AssimpImporter::LoadMeshes(MeshList& meshes);
void AssimpImporter::LoadMeshes(MeshList& meshes, const NodeList& nodes)
{
    LoadMeshes(meshes, nodes, aScene->mRootNode, "");
}

// 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)
 {
    // ノードパスの作成
    nodePath += "/";
    nodePath += aNode->mName.C_Str();

    // メッシュデータ読み取り
    for (uint32_t aMeshIndex = 0; aMeshIndex < aNode->mNumMeshes; ++aMeshIndex)
    {
        const aiMesh* aMesh = aScene->mMeshes[aNode->mMeshes[aMeshIndex]];

        // メッシュデータ格納
        Model::Mesh& mesh = meshes.emplace_back();
        mesh.nodeIndex = GetNodeIndexByPath(nodes, nodePath.c_str());
```

```
        ---省略---
    }

    // 再帰的に子ノードを処理する
    for (uint32_t aNodeIndex = 0; aNodeIndex < aNode->mNumChildren; ++aNodeIndex)
    {
        LoadMeshes(meshes, nodes, aNode->mChildren[aNodeIndex], nodePath);
    }
}

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

必要なデータの読み取りプログラムを実装したので Model クラスで呼び出しましょう。
また、ノード構造体には描画処理時に必要となる行列データと親子情報を参照するポインタデータを追加します。

ノードが持っている「位置」「回転」「スケール」のデータからローカル行列を計算できます。
ワールド行列は親のワールド行列とローカル行列を掛け算することで計算できます。

ローカル行列 ＝ スケール × 回転 × 位置
ワールド行列 ＝ ローカル行列 × 親のワールド行列

Model.h

```
---省略---

class Model
{
public:
    ---省略---

    struct Node
    {
        ---省略---

        DirectX::XMFLOAT4X4    localTransform;
        DirectX::XMFLOAT4X4    worldTransform;
```

計算で求める

```
        Node*                   parent = nullptr;
        std::vector<Node*>      children;
    };

    ---省略---

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

    ---省略---

    // トランスフォーム更新処理
    void UpdateTransform(const DirectX::XMFLOAT4X4& worldTransform);

private:
    ---省略---
    std::vector<Node>       nodes;
};
```

親インデックスデータから
参照しやすいようにポインタを設定する

ノードインデックスデータから
参照しやすいようにポインタを設定する

全てのノードの行列データを計算する

Model.cpp

```
---省略---

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

    // ノードデータ読み取り
    importer.LoadNodes(nodes);

    // メッシュデータ読み取り
    ~~importer.LoadMeshes(meshes);~~
    importer.LoadMeshes(meshes, nodes);

    // ノード構築
    for (size_t nodeIndex = 0; nodeIndex < nodes.size(); ++nodeIndex)
    {
        Node& node = nodes.at(nodeIndex);

        // 親子関係を構築
        node.parent = node.parentIndex >= 0 ? &nodes.at(node.parentIndex) : nullptr;
        if (node.parent != nullptr)
        {
            node.parent->children.emplace_back(&node);
        }
    }
    ---省略---

    // メッシュ構築
    for (Mesh& mesh : meshes)
```

親インデックスデータから
参照しやすいようにポインタを設定する

```
    {
        // 参照ノード設定
        mesh.node = &nodes.at(mesh.nodeIndex);

        ---省略---
    }
}

// トランスフォーム更新処理
void Model::UpdateTransform(const DirectX::XMFLOAT4X4& worldTransform)
{
    for (Node& node : nodes)
    {
        // ローカル行列算出
        DirectX::XMMATRIX S = DirectX::XMMatrixScaling(node.scale.x, node.scale.y, node.scale.z);
        DirectX::XMMATRIX R = DirectX::XMMatrixRotationQuaternion(DirectX::XMLoadFloat4(&node.rotation));
        DirectX::XMMATRIX T = DirectX::XMMatrixTranslation(node.position.x, node.position.y, node.position.z);
        DirectX::XMMATRIX LocalTransform = S * R * T;

        // ワールド行列算出
        DirectX::XMMATRIX ParentWorldTransform;
        if (node.parent != nullptr)
        {
            ParentWorldTransform = DirectX::XMLoadFloat4x4(&node.parent->worldTransform);
        }
        else
        {
            ParentWorldTransform = DirectX::XMLoadFloat4x4(&worldTransform);
        }
        DirectX::XMMATRIX WorldTransform = LocalTransform * ParentWorldTransform;

        // 計算結果を格納
        DirectX::XMStoreFloat4x4(&node.localTransform, LocalTransform);
        DirectX::XMStoreFloat4x4(&node.worldTransform, WorldTransform);
    }
}
```

ノードインデックスデータから
参照しやすいようにポインタを設定する

Phong.hlsli

```
---省略---

cbuffer CbMesh : register(b1)
{
    ---省略---
    row_major float4x4    worldTransform;
};
```

PhongVS.hlsl

```
#include "Phong.hlsli"

VS_OUT main(float4 position : POSITION, float2 texcoord : TEXCOORD)
{
    ---省略---
```

```
    vout.vertex = mul(position, viewProjection);
    vout.vertex = mul(position, mul(worldTransform, viewProjection));
    ---省略---
}
```

指定の位置に配置できるように
ワールド行列を計算に含める

PhongShader.h

```
#pragma once

#include "Shader.h"

class PhongShader : public Shader
{
    ---省略---

private:
    ---省略---

    struct CbMesh
    {
        ---省略---
        DirectX::XMFLOAT4X4    worldTransform;
    };

    ---省略---
};
```

PhongShader.cpp

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

        ---省略---
    }
}
```

ワールド行列データを
シェーダーに渡す

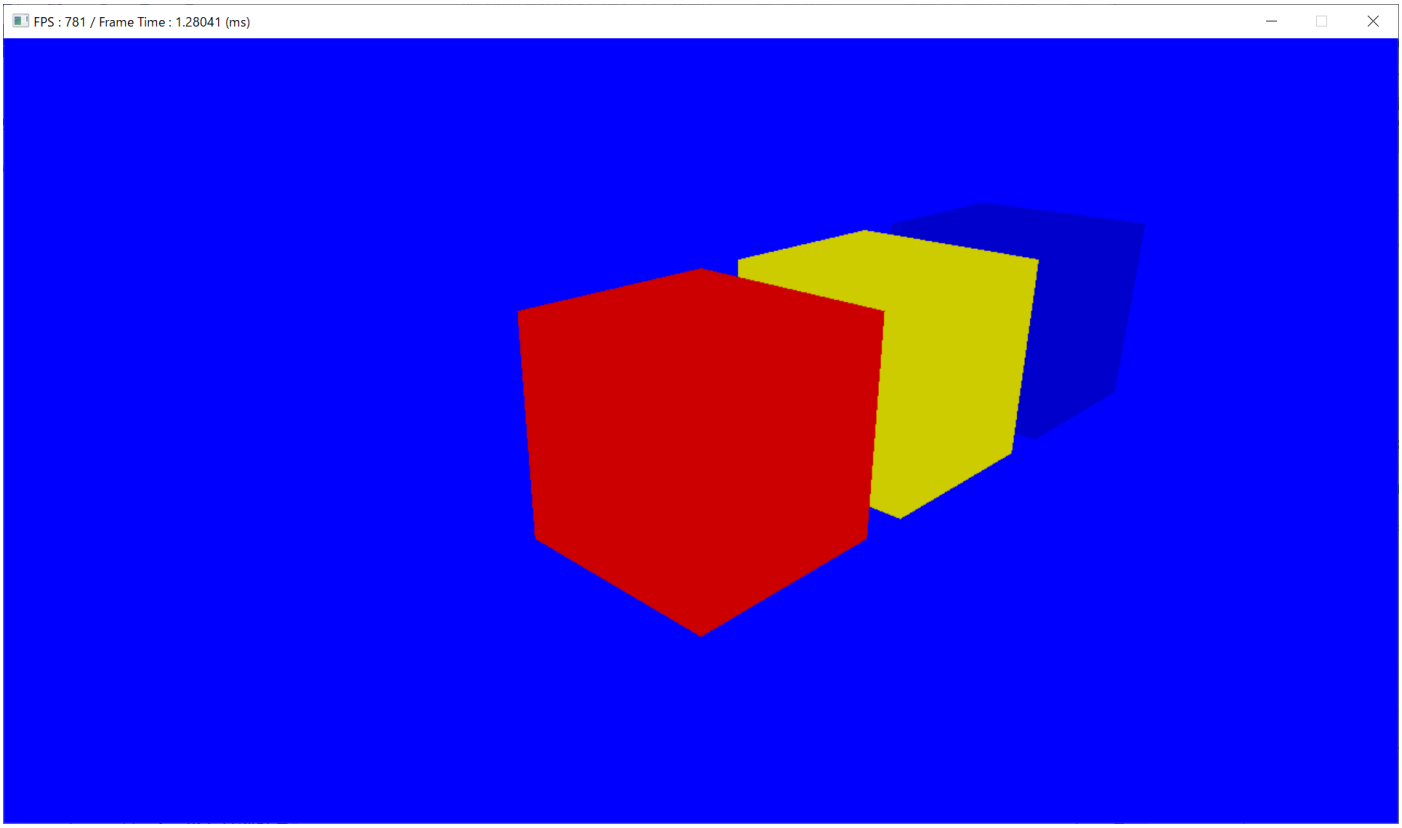
Scene.h

```
---省略---
```

```
// モデルテストシーン
class ModelTestScene : public Scene
{
    ---省略---

private:
    ---省略---
    DirectX::XMFLOAT3          position = { 0, 0, 0 };
    DirectX::XMFLOAT3          angle = { 0, 0, 0 };
    DirectX::XMFLOAT3          scale = { 1, 1, 1 };
};
```

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.001.1.fbx");
    model = std::make_unique<Model>(device, "Data/Model/Cube/cube.003.1.fbx");
}

// 描画処理
void ModelTestScene::Render(float elapsedTime)
{
    // ワールド行列計算
    DirectX::XMMATRIX S = DirectX::XMMatrixScaling(scale.x, scale.y, scale.z);
    DirectX::XMMATRIX R = DirectX::XMMatrixRotationRollPitchYaw(angle.x, angle.y, angle.z);
    DirectX::XMMATRIX T = DirectX::XMMatrixTranslation(position.x, position.y, position.z);
    DirectX::XMFLOAT4X4 worldTransform;
    DirectX::XMStoreFloat4x4(&worldTransform, S * R * T);

    // トランスフォーム更新
    model->UpdateTransform(worldTransform);

    ---省略---
}
```


FPS : 781 / Frame Time : 1.28041 (ms)