描画エンジン開発 EX

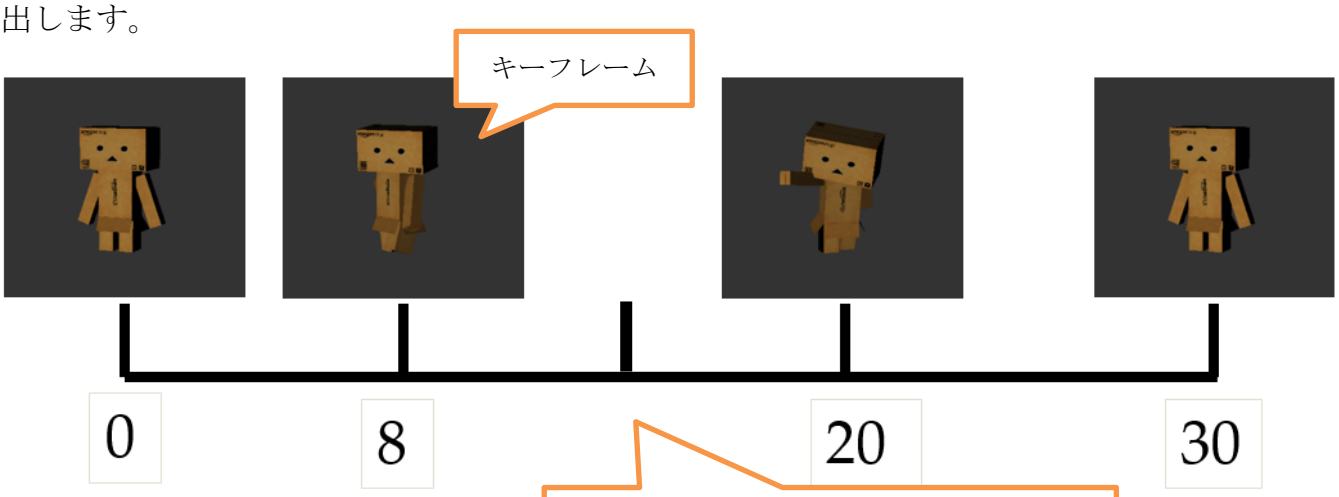
○概要

アニメーション処理を実装する。

○アニメーションとは

3D モデルのノードを動かすことによって「走る」「ジャンプ」などの動きを表現する技術です。
アニメーションは DCC ツール（Maya、Blender など）で作成し、FBX フォーマットなどで出力されたデータを利用して実装します。

アニメーションデータの中にはキーフレームというポーズデータがいくつか存在します。
下図の例では 30 フレームのアニメーションの中に 4 つのポーズデータが存在します。
この 4 つのキーフレームのデータを使ってキーフレームが存在しないフレームのポーズを計算で算出します。



○アニメーションデータの収集

アニメーションデータはノード毎にキーフレームが存在します。
ノード毎のキーフレームデータを収集しましょう。

Model.h

```
---省略---
```

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

    struct VectorKeyframe
    {
        float                   seconds;
        DirectX::XMFLOAT3       value;
    };

    struct QuaternionKeyframe
    {
        float                   seconds;
        DirectX::XMFLOAT4       value;
    };

    struct NodeAnim
    {
        std::vector<VectorKeyframe>         positionKeyframes;
        std::vector<QuaternionKeyframe>     rotationKeyframes;
        std::vector<VectorKeyframe>         scaleKeyframes;
    };

    struct Animation
    {
        std::string                 name;
        float                       secondsLength;
        std::vector<NodeAnim>       nodeAnims;
    };

    ---省略---
};
```

AssimpImporter.h

```
---省略---

class AssimpImporter
{
private:
    ---省略---
    using AnimationList = std::vector<Model::Animation>;

public:
    ---省略---

    // アニメーションデータを読み込み
    void LoadAnimations(AnimationList& animations, const NodeList& nodes);

    ---省略---
};
```

AssimpImporter.cpp

```
---省略---

// アニメーションデータを読み込み
void AssimpImporter::LoadAnimations(AnimationList& animations, const NodeList& nodes)
{
    for (uint32_t aAnimationIndex = 0; aAnimationIndex < aScene->mNumAnimations; ++aAnimationIndex)
    {
        const aiAnimation* aAnimation = aScene->mAnimations[aAnimationIndex];
        Model::Animation& animation = animations.emplace_back();

        // アニメーション情報
        animation.name = aAnimation->mName.C_Str();
        animation.secondsLength = static_cast<float>(aAnimation->mDuration / aAnimation->mTicksPerSecond);

        // ノード毎のアニメーション
        animation.nodeAnims.resize(nodes.size());
        for (uint32_t aChannelIndex = 0; aChannelIndex < aAnimation->mNumChannels; ++aChannelIndex)
        {
            const aiNodeAnim* aNodeAnim = aAnimation->mChannels[aChannelIndex];
            int nodeIndex = GetNodeIndexByName(nodes, aNodeAnim->mNodeName.C_Str());
            if (nodeIndex < 0) continue;

            const Model::Node& node = nodes.at(nodeIndex);
            Model::NodeAnim& nodeAnim = animation.nodeAnims.at(nodeIndex);

            // 位置
            for (uint32_t aPositionIndex = 0; aPositionIndex < aNodeAnim->mNumPositionKeys; ++aPositionIndex)
            {
                const aiVectorKey& aKey = aNodeAnim->mPositionKeys[aPositionIndex];
                Model::VectorKeyframe& keyframe = nodeAnim.positionKeyframes.emplace_back();
                keyframe.seconds = static_cast<float>(aKey.mTime / aAnimation->mTicksPerSecond);
                keyframe.value = aiVector3DToXMFLOAT3(aKey.mValue);
            }
            // 回転
            for (uint32_t aRotationIndex = 0; aRotationIndex < aNodeAnim->mNumRotationKeys; ++aRotationIndex)
            {
                const aiQuatKey& aKey = aNodeAnim->mRotationKeys[aRotationIndex];
                Model::QuaternionKeyframe& keyframe = nodeAnim.rotationKeyframes.emplace_back();
                keyframe.seconds = static_cast<float>(aKey.mTime / aAnimation->mTicksPerSecond);
                keyframe.value = aiQuaternionToXMFLOAT4(aKey.mValue);
            }
            // スケール
            for (uint32_t aScalingIndex = 0; aScalingIndex < aNodeAnim->mNumScalingKeys; ++aScalingIndex)
            {
                const aiVectorKey& aKey = aNodeAnim->mScalingKeys[aScalingIndex];
                Model::VectorKeyframe& keyframe = nodeAnim.scaleKeyframes.emplace_back();
                keyframe.seconds = static_cast<float>(aKey.mTime / aAnimation->mTicksPerSecond);
                keyframe.value = aiVector3DToXMFLOAT3(aKey.mValue);
            }
        }
        // アニメーションがなかったノードに対して初期姿勢のキーフレームを追加する
        for (size_t nodeIndex = 0; nodeIndex < animation.nodeAnims.size(); ++nodeIndex)
        {
            const Model::Node& node = nodes.at(nodeIndex);
```

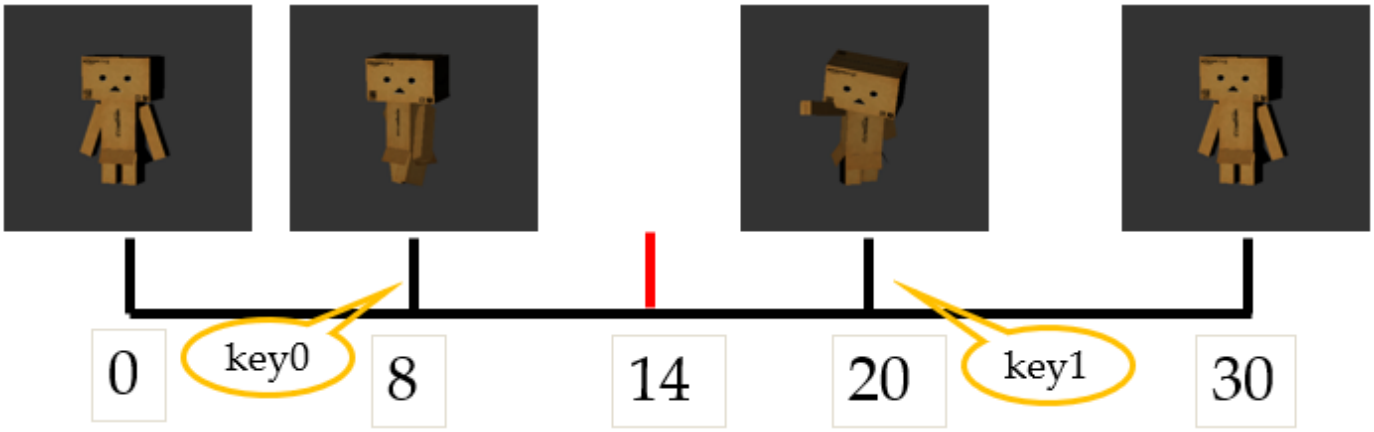
```
                Model::NodeAnim& nodeAnim = animation.nodeAnims.at(nodeIndex);
                // 移動
                if (nodeAnim.positionKeyframes.size() == 0)
                {
                    Model::VectorKeyframe& keyframe = nodeAnim.positionKeyframes.emplace_back();
                    keyframe.seconds = 0.0f;
                    keyframe.value = node.position;
                }
                if (nodeAnim.positionKeyframes.size() == 1)
                {
                    Model::VectorKeyframe& keyframe = nodeAnim.positionKeyframes.emplace_back();
                    keyframe.seconds = animation.secondsLength;
                    keyframe.value = nodeAnim.positionKeyframes.at(0).value;
                }
                // 回転
                if (nodeAnim.rotationKeyframes.size() == 0)
                {
                    Model::QuaternionKeyframe& keyframe = nodeAnim.rotationKeyframes.emplace_back();
                    keyframe.seconds = 0.0f;
                    keyframe.value = node.rotation;
                }
                if (nodeAnim.rotationKeyframes.size() == 1)
                {
                    Model::QuaternionKeyframe& keyframe = nodeAnim.rotationKeyframes.emplace_back();
                    keyframe.seconds = animation.secondsLength;
                    keyframe.value = nodeAnim.rotationKeyframes.at(0).value;
                }
                // スケール
                if (nodeAnim.scaleKeyframes.size() == 0)
                {
                    Model::VectorKeyframe& keyframe = nodeAnim.scaleKeyframes.emplace_back();
                    keyframe.seconds = 0.0f;
                    keyframe.value = node.scale;
                }
                if (nodeAnim.scaleKeyframes.size() == 1)
                {
                    Model::VectorKeyframe& keyframe = nodeAnim.scaleKeyframes.emplace_back();
                    keyframe.seconds = animation.secondsLength;
                    keyframe.value = nodeAnim.scaleKeyframes.at(0).value;
                }
            }
        }
    }
```

○アニメーション再生処理

アニメーションデータを利用してすべてのノードに対してアニメーションの計算処理を実装します。

アニメーションの再生時間から前と後のキーフレームを取得し、キーフレームのポーズを合成します。

合成する割合は再生時間とキーフレームの時間から算出します。




```
float t = (current_frame - key0.frame / key1.frame - key0.frame);
DirectX::VECTOR Position = DirectX::XMVectorLerp(Key0_Position, Key1_Position, t);
DirectX::XMVECTOR Rotation = DirectX::XMQuaternionSlerp(Key0_Rotation, Key1_Rotation, t);
```

Model クラスに再生関数とアニメーション計算関数を実装します。

Model.h

```
---省略---

class Model
{
public:
    ---省略---

    // アニメーション再生
    void PlayAnimation(int index, bool loop);

    // アニメーション再生中か
    bool IsPlayAnimation() const;

    // アニメーション更新処理
    void UpdateAnimation(float elapsedTime);

private:
    // アニメーション計算処理
    void ComputeAnimation(float elapsedTime);

private:
    ---省略---
    std::vector<Animation>    animations;

    int      currentAnimationIndex = -1;
    float    currentAnimationSeconds = 0;
    bool     animationPlaying = false;
    bool     animationLoop = false;
};
```

Model.cpp

```
---省略---

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

    // アニメーションデータ読み取り
    importer.LoadAnimations(animations, nodes);
}

// アニメーション再生
void Model::PlayAnimation(int index, bool loop)
{
    currentAnimationIndex = index;
    currentAnimationSeconds = 0;
    animationLoop = loop;
    animationPlaying = true;
}

// アニメーション再生中か
bool Model::IsPlayAnimation() const
{
    if (currentAnimationIndex < 0) return false;
    if (currentAnimationIndex >= animations.size()) return false;
    return animationPlaying;
}

// アニメーション更新処理
void Model::UpdateAnimation(float elapsedTime)
{
    ComputeAnimation(elapsedTime);
}

// アニメーション計算処理
void Model::ComputeAnimation(float elapsedTime)
{
    if (!IsPlayAnimation()) return;

    // 指定のアニメーションデータを取得
    const Animation& animation = animations.at(currentAnimationIndex);

    // ノード毎のアニメーション処理
    for (size_t nodeIndex = 0; nodeIndex < animation.nodeAnims.size(); ++nodeIndex)
    {
        Node& node = nodes.at(nodeIndex);
        const NodeAnim& nodeAnim = animation.nodeAnims.at(nodeIndex);

        // 位置
        for (size_t index = 0; index < nodeAnim.positionKeyframes.size() - 1; ++index)
        {
            // 現在の時間がどのキーフレームの間にいるか判定する
            const VectorKeyframe& keyframe0 = nodeAnim.positionKeyframes.at(index);
            const VectorKeyframe& keyframe1 = nodeAnim.positionKeyframes.at(index + 1);
```

```
            if (currentAnimationSeconds >= keyframe0.seconds && currentAnimationSeconds < keyframe1.seconds)
            {
                // 再生時間とキーフレームの時間から補完率を算出する
                float rate = (currentAnimationSeconds - keyframe0.seconds) / (keyframe1.seconds -
                                                                              keyframe0.seconds);

                // 前のキーフレームと次のキーフレームの姿勢を補完
                DirectX::XMVECTOR V0 = DirectX::XMLoadFloat3(&keyframe0.value);
                DirectX::XMVECTOR V1 = DirectX::XMLoadFloat3(&keyframe1.value);
                DirectX::XMVECTOR V = DirectX::XMVectorLerp(V0, V1, rate);
                // 計算結果をノードに格納
                DirectX::XMStoreFloat3(&node.position, V);
            }
        }
        // 回転
        for (size_t index = 0; index < nodeAnim.rotationKeyframes.size() - 1; ++index)
        {
            // 現在の時間がどのキーフレームの間にいるか判定する
            const QuaternionKeyframe& keyframe0 = nodeAnim.rotationKeyframes.at(index);
            const QuaternionKeyframe& keyframe1 = nodeAnim.rotationKeyframes.at(index + 1);
            if (currentAnimationSeconds >= keyframe0.seconds && currentAnimationSeconds < keyframe1.seconds)
            {
                // 再生時間とキーフレームの時間から補完率を算出する
                float rate = (currentAnimationSeconds - keyframe0.seconds) / (keyframe1.seconds -
                                                                              keyframe0.seconds);

                // 前のキーフレームと次のキーフレームの姿勢を補完
                DirectX::XMVECTOR Q0 = DirectX::XMLoadFloat4(&keyframe0.value);
                DirectX::XMVECTOR Q1 = DirectX::XMLoadFloat4(&keyframe1.value);
                DirectX::XMVECTOR Q = DirectX::XMQuaternionSlerp(Q0, Q1, rate);
                // 計算結果をノードに格納
                DirectX::XMStoreFloat4(&node.rotation, Q);
            }
        }
        // スケール
        for (size_t index = 0; index < nodeAnim.scaleKeyframes.size() - 1; ++index)
        {
            // 現在の時間がどのキーフレームの間にいるか判定する
            const VectorKeyframe& keyframe0 = nodeAnim.scaleKeyframes.at(index);
            const VectorKeyframe& keyframe1 = nodeAnim.scaleKeyframes.at(index + 1);
            if (currentAnimationSeconds >= keyframe0.seconds && currentAnimationSeconds < keyframe1.seconds)
            {
                // 再生時間とキーフレームの時間から補完率を算出する
                float rate = (currentAnimationSeconds - keyframe0.seconds) / (keyframe1.seconds -
                                                                              keyframe0.seconds);

                // 前のキーフレームと次のキーフレームの姿勢を補完
                DirectX::XMVECTOR V0 = DirectX::XMLoadFloat3(&keyframe0.value);
                DirectX::XMVECTOR V1 = DirectX::XMLoadFloat3(&keyframe1.value);
                DirectX::XMVECTOR V = DirectX::XMVectorLerp(V0, V1, rate);
                // 計算結果をノードに格納
                DirectX::XMStoreFloat3(&node.scale, V);
            }
        }
    }

    // 時間経過
```

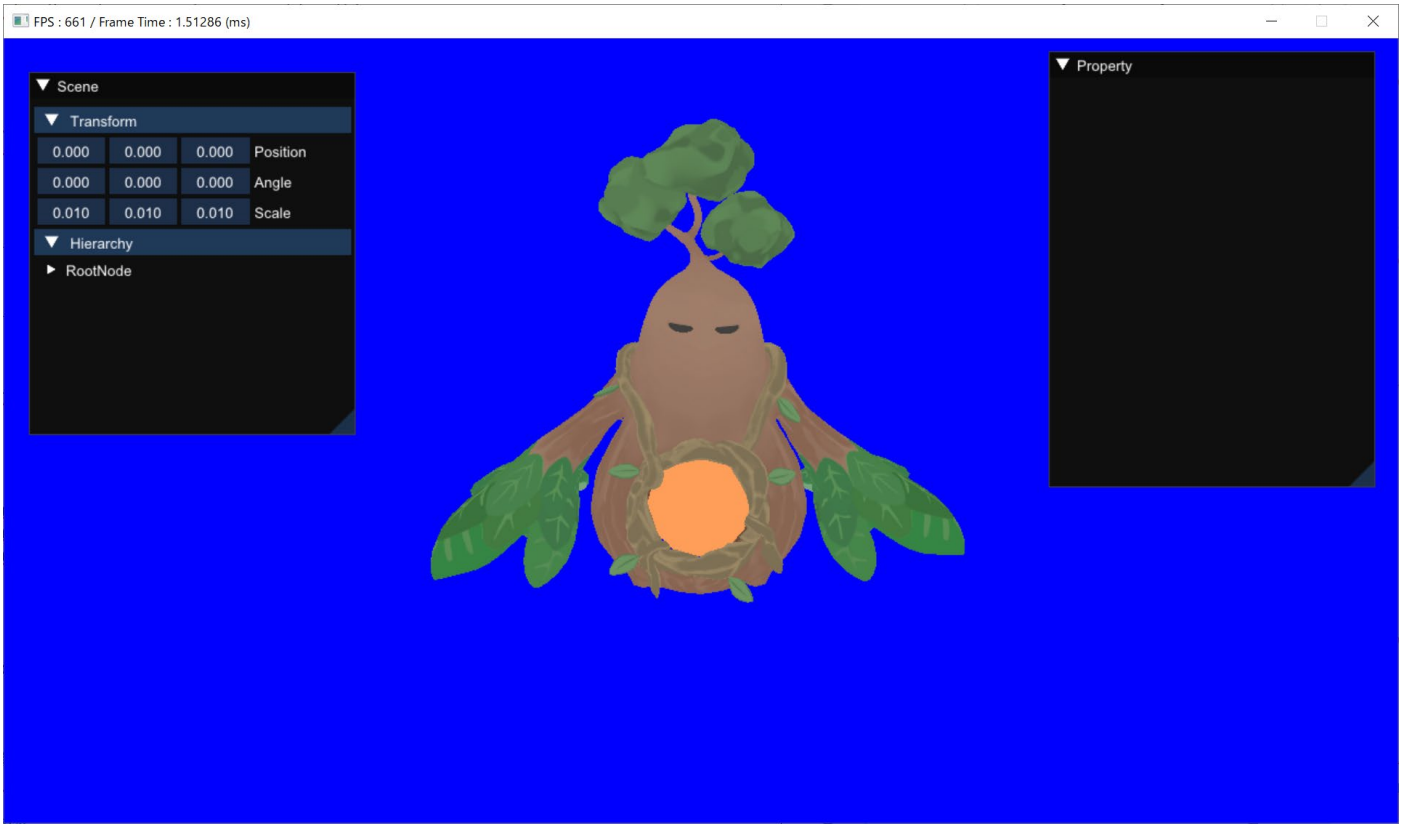
```
    currentAnimationSeconds += elapsedTime;

    // 再生時間が終端時間を超えたら
    if (currentAnimationSeconds >= animation.secondsLength)
    {
        if (animationLoop)
        {
            // 再生時間を巻き戻す
            currentAnimationSeconds -= animation.secondsLength;
        }
        else
        {
            // 再生終了時間にする
            currentAnimationSeconds = animation.secondsLength;
            animationPlaying = false;
        }
    }
}
```

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.004.fbx");
    model = std::make_unique<Model>(device, "Data/Model/Plantune/plantune.fbx");
    model->PlayAnimation(0, true);
    scale.x = scale.y = scale.z = 0.01f;
}

// 描画処理
void ModelTestScene::Render(float elapsedTime)
{
    ---省略---

    // アニメーション更新
    model->UpdateAnimation(elapsedTime);

    // トランスフォーム更新
    ---省略---
}
```

Plantune は大きいので
スケール調整しておく

実行確認してみましょう。
待機アニメーションが再生されていれば OK です。



○アニメーションを選んで再生

IMGUI でアニメーションリストを表示し、ダブルクリックで再生できるようにします。

Model.h

```
---省略---

class Model
{
public:
    ---省略---

    // アニメーションデータ取得
    const std::vector<Animation>& GetAnimations() const { return animations; }

    ---省略---
};
```

Scene.h

```
---省略---

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

```
private:
    ---省略---

    // アニメーションGUI描画
    void DrawAnimationGUI();

    ---省略---
private:
    ---省略---
    bool                        animationLoop = false;
};
```

Scene.cpp

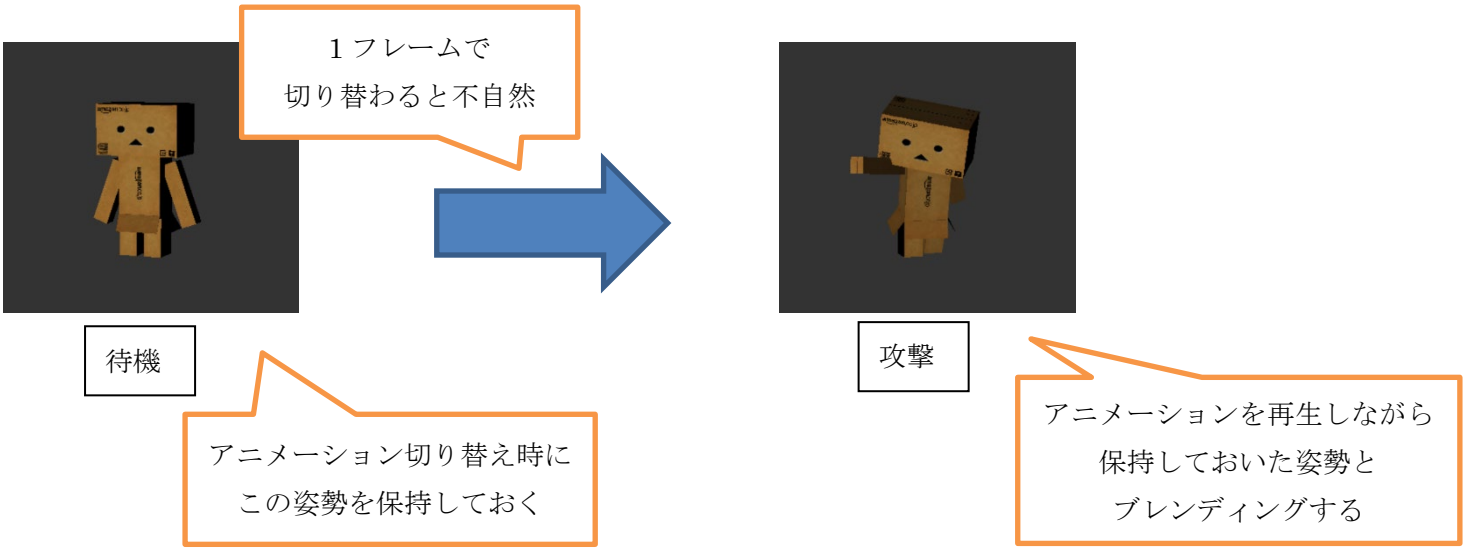
```
---省略---

// 描画処理
void ModelTestScene::Render(float elapsedTime)
{
    ---省略---

    // デバッグメニュー描画
    ---省略---
    DrawAnimationGUI();
}

// アニメーションGUI描画
void ModelTestScene::DrawAnimationGUI()
{
    ImGui::SetNextWindowPos(ImVec2(10, 350), ImGuiCond_FirstUseEver);
    ImGui::SetNextWindowSize(ImVec2(300, 300), ImGuiCond_FirstUseEver);

    ImGui::Begin("Animation", nullptr, ImGuiWindowFlags_None);

    ImGui::Checkbox("Loop", &animationLoop);

    int index = 0;
    for (const Model::Animation& animation : model->GetAnimations())
    {
        ImGuiTreeNodeFlags nodeFlags = ImGuiTreeNodeFlags_Leaf;

        ImGui::TreeNodeEx(&animation, nodeFlags, animation.name.c_str());

        // ダブルクリックでアニメーション再生
        if (ImGui::IsItemClicked())
        {
            if (ImGui::IsMouseDoubleClicked(ImGuiMouseButton_Left))
            {
                model->PlayAnimation(index, animationLoop);
            }
        }

        ImGui::TreePop();
```

```
            index++;
        }

        ImGui::End();
}
```

実行確認してみましょう。
アニメーションウインドウが表示され、ダブルクリックで指定したアニメーションが再生されればOKです。

○アニメーションブレンディング

アニメーションを自然に切り替えるためにアニメーションブレンディングを実装します。
例えば「待機」から「攻撃」へアニメーションを切り替えた場合に１フレームで攻撃ポーズに切り替わってしまうと不自然です。
アニメーションが切り替わる前の姿勢を保持しておき、切り替わった後のアニメーションと姿勢を合成することで滑らかな姿勢の切り替わりを実現します。



Model.h

```
---省略---

class Model
{
public:
    ---省略---

    // アニメーション再生
    void PlayAnimation(int index, bool loop);
    void PlayAnimation(int index, bool loop, float blendSeconds = 0);

    ---省略---

private:
    ---省略---

    // ブレンディング計算処理
    void ComputeBlending(float elapsedTime);

private:
    ---省略---

    struct NodeCache
    {
        DirectX::XMFLOAT3 position = { 0, 0, 0 };
        DirectX::XMFLOAT4 rotation = { 0, 0, 0, 1 };
        DirectX::XMFLOAT3 scale = { 1, 1, 1 };
    };
    std::vector<NodeCache>    nodeCaches;

    float    currentAnimationBlendSeconds = 0.0f;
    float    animationBlendSecondsLength = -1.0f;
    bool     animationBlending = false;
};
```

Model.cpp

```
---省略---

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

    // ノードキャッシュ
    nodeCaches.resize(nodes.size());
}

// アニメーション再生
void Model::PlayAnimation(int index, bool loop, float blendSeconds)
{
    ---省略---

    // ブレンドパラメータ
    animationBlending = blendSeconds > 0.0f;
    currentAnimationBlendSeconds = 0.0f;
    animationBlendSecondsLength = blendSeconds;

    // 現在の姿勢をキャッシュする
    for (size_t i = 0; i < nodes.size(); ++i)
    {
        const Node& src = nodes.at(i);
        NodeCache& dst = nodeCaches.at(i);

        dst.position = src.position;
        dst.rotation = src.rotation;
        dst.scale = src.scale;
    }
}

// アニメーション更新処理
void Model::UpdateAnimation(float elapsedTime)
{
    ---省略---

    ComputeBlending(elapsedTime);
}

// ブレンディング計算処理
void Model::ComputeBlending(float elapsedTime)
{
    if (!animationBlending)
    {
        return;
    }

    // ブレンド率の計算
    float rate = currentAnimationBlendSeconds / animationBlendSecondsLength;

    // ブレンド計算
    int count = static_cast<int>(nodes.size());
    for (int i = 0; i < count; ++i)
    {
        const NodeCache& cache = nodeCaches.at(i);
```

```
        Node& node = nodes.at(i);

        DirectX::XMVECTOR S0 = DirectX::XMLoadFloat3(&cache.scale);
        DirectX::XMVECTOR S1 = DirectX::XMLoadFloat3(&node.scale);
        DirectX::XMVECTOR R0 = DirectX::XMLoadFloat4(&cache.rotation);
        DirectX::XMVECTOR R1 = DirectX::XMLoadFloat4(&node.rotation);
        DirectX::XMVECTOR T0 = DirectX::XMLoadFloat3(&cache.position);
        DirectX::XMVECTOR T1 = DirectX::XMLoadFloat3(&node.position);

        DirectX::XMVECTOR S = DirectX::XMVectorLerp(S0, S1, rate);
        DirectX::XMVECTOR R = DirectX::XMQuaternionSlerp(R0, R1, rate);
        DirectX::XMVECTOR T = DirectX::XMVectorLerp(T0, T1, rate);

        DirectX::XMStoreFloat3(&node.scale, S);
        DirectX::XMStoreFloat4(&node.rotation, R);
        DirectX::XMStoreFloat3(&node.position, T);
    }

    // 時間経過
    currentAnimationBlendSeconds += elapsedTime;
    if (currentAnimationBlendSeconds >= animationBlendSecondsLength)
    {
        currentAnimationBlendSeconds = animationBlendSecondsLength;
        animationBlending = false;
    }
}
```

Scene.h

```
---省略---

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

private:
    ---省略---
    float        animationBlendSeconds = 0;
};
```

Scene.cpp

```
---省略---

// アニメーションGUI描画
void ModelTestScene::DrawAnimationGUI()
{
    ---省略---
    ImGui::DragFloat("BlendSec", &animationBlendSeconds, 0.01f);

    for (const Model::Animation& animation : model->GetAnimations())
    {
        ---省略---
```

```
        // ダブルクリックでアニメーション再生
        if (ImGui::IsItemClicked())
        {
            if (ImGui::IsMouseDoubleClicked(ImGuiMouseButton_Left))
            {
                model->PlayAnimation(index, animationLoop, animationBlendSeconds);
            }
        }

        ---省略---
    }
}
```

実行確認してみましょう。
ブレンド時間が 0.0 で再生した場合はアニメーション切り替わりが即座に切り替わり、ブレンド時間を 0.2 くらいにして再生した場合は補完しながらアニメーションが切り替わっていれば OK です。