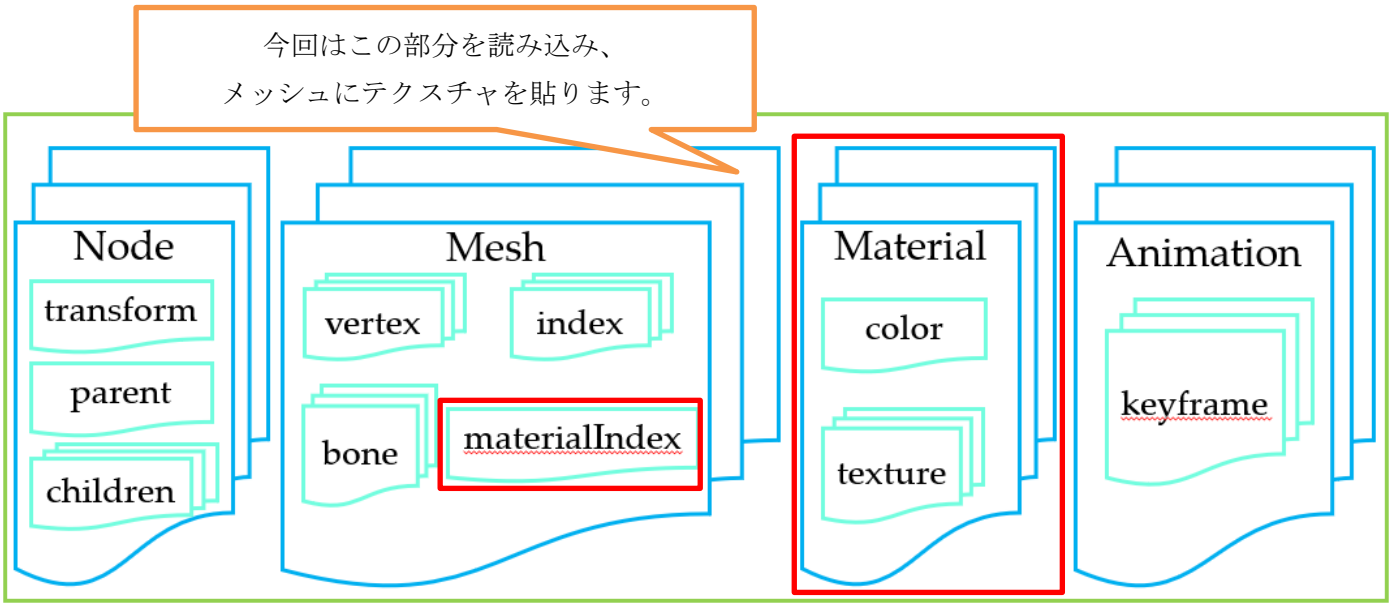
描画エンジン開発 EX

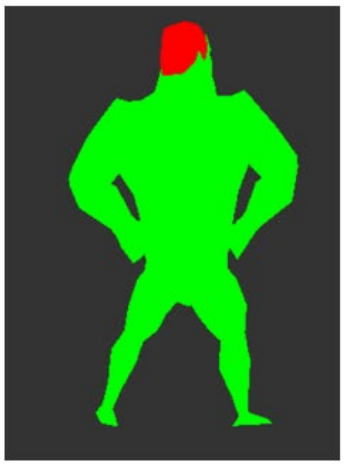
○概要

3D モデルファイルからマテリアル情報を読み取り、メッシュに着色やテクスチャを貼ります



○マテリアル

マテリアルとは 3D モデルの質感を表現するデータです。
色データやテクスチャ、光の反射率など様々なデータがあります。

今回は基本となるカラーデータとテクスチャデータを読み取ります。
Model クラスに Material 構造体を定義します。

Model.h

```
---省略---
#include <string>

class Model
{
public:
```

```
    struct Material
    {
        std::string           name;
        std::string           diffuseTextureFileName;
        DirectX::XMFLOAT4     color = { 1, 1, 1, 1 };
    };

    ---省略---
};
```

AssimpImporter にマテリアルデータの読み込み関数を作成します。

AssimpImporter.h

```
---省略---

class AssimpImporter
{
private:
    ---省略---
    using MaterialList = std::vector<Model::Material>;

public:
    ---省略---

    // マテリアルデータを読み込み
    void LoadMaterials(MaterialList& materials);

private:
    ---省略---

    // aiColor3D → XMFLOAT4
    static DirectX::XMFLOAT4 AssimpImporter::aiColor3DToXMFLOAT4(const aiColor3D& aValue);

    ---省略---
};
```

AssimpImporter.cpp

```
---省略---

// マテリアルデータを読み込み
void AssimpImporter::LoadMaterials(MaterialList& materials)
{
    materials.resize(aScene->mNumMaterials);
    for (uint32_t aMaterialIndex = 0; aMaterialIndex < aScene->mNumMaterials; ++aMaterialIndex)
    {
        const aiMaterial* aMaterial = aScene->mMaterials[aMaterialIndex];
        Model::Material& material = materials.at(aMaterialIndex);

        // マテリアル名
        aiString aMaterialName;
        aMaterial->Get(AI_MATKEY_NAME, aMaterialName);
```

```
        material.name = aMaterialName.C_Str();

        // ディフューズ色
        aiColor3D aDiffuseColor;
        if (AI_SUCCESS == aMaterial->Get(AI_MATKEY_COLOR_DIFFUSE, aDiffuseColor))
        {
            material.color = aiColor3DToXMFLOAT4(aDiffuseColor);
        }

        // テクスチャ読み込み関数
        auto loadTexture = [&](aiTextureType aTextureType, std::string& textureFilename)
        {
            // テクスチャファイルパス取得
            aiString aTextureFilePath;
            if (AI_SUCCESS == aMaterial->GetTexture(aTextureType, 0, &aTextureFilePath))
            {
                // テクスチャファイルパスをそのまま格納
                textureFilename = aTextureFilePath.C_Str();
            }
        };

        // ディフューズマップ
        loadTexture(aiTextureType_DIFFUSE, material.diffuseTextureFileName);
    }
}

---省略---

// aiColor3D → XMFLOAT4
DirectX::XMFLOAT4 AssimpImporter::AssimpImporter::aiColor3DToXMFLOAT4(const aiColor3D& aValue)
{
    return DirectX::XMFLOAT4(
        static_cast<float>(aValue.r),
        static_cast<float>(aValue.g),
        static_cast<float>(aValue.b),
        1.0f
    );
}
```

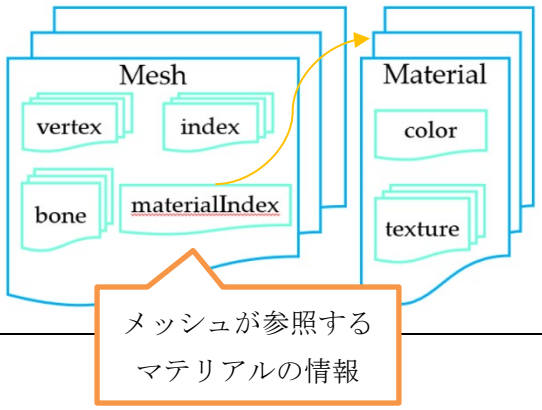
今後、他のテクスチャの読み込みも
するので関数化して読み込み

頂点データにテクスチャ座標を追加し、メッシュデータに参照するマテリアルインデックスを追加します。

Model.h

```
---省略---

class Model
{
public:
    ---省略---

    struct Vertex
    {
        ---省略---
```

```
        DirectX::XMFLOAT2 texcoord = { 0, 0 };
    };

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

    ---省略---
};
```

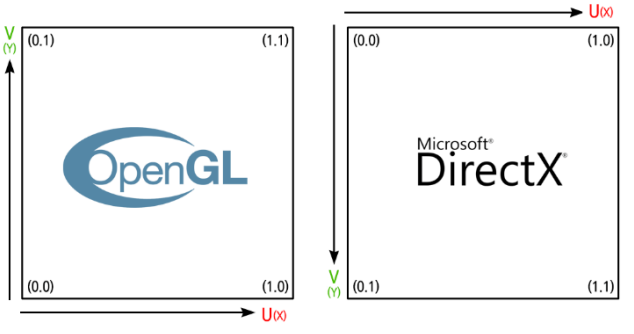


AssimpImporter.h

```
---省略---

class AssimpImporter
{
    ---省略---
private:
    // aiVector3D → XMFLOAT2
    static DirectX::XMFLOAT2 AssimpImporter::aiVector3DToXMFLOAT2(const aiVector3D& aValue);

    ---省略---
};
```

AssimpImporter.cpp

```
---省略---

// メッシュデータを読み込み
void AssimpImporter::LoadMeshes(MeshList& meshes)
{
    ---省略---

    // メッシュデータ格納
    ---省略---
    mesh.materialIndex = static_cast<int>(aMesh->mMaterialIndex);

    // 頂点データ
    for (uint32_t aVertexIndex = 0; aVertexIndex < aMesh->mNumVertices; ++aVertexIndex)
    {
        ---省略---

        // テクスチャ座標
        if (aMesh->HasTextureCoords(0))
        {
            vertex.texcoord = aiVector3DToXMFLOAT2(aMesh->mTextureCoords[0][aVertexIndex]);
            vertex.texcoord.y = 1.0f - vertex.texcoord.y;
        }
    }

    ---省略---
}
```



OpenGL と DirectX で縦方向の
テクスチャ座標が違う。
Assimp は OpenGL 基準の
データなので変換する

```
---省略---

// aiVector3D → XMFLOAT2
DirectX::XMFLOAT2 AssimpImporter::AssimpImporter::aiVector3DToXMFLOAT2(const aiVector3D& aValue)
{
    return DirectX::XMFLOAT2(
        static_cast<float>(aValue.x),
        static_cast<float>(aValue.y)
    );
}

---省略---
```

3D モデルファイルから必要なデータを取得するプログラムを実装できたので、Model クラスでマテリアルデータを読み込み、テクスチャなど GPU リソースの構築をします。

Model.h

```
---省略---

class Model
{
public:
    ---省略---

    struct Material
    {
        ---省略---
        Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> diffuseMap;
    };

    struct Mesh
    {
        ---省略---
        Material*    material = nullptr;
    };

    ---省略---

    // マテリアルデータ取得
    const std::vector<Material>& GetMaterials() const { return materials; }

private:
    ---省略---
    std::vector<Material> materials;
};
```

Model.cpp

```
#include <filesystem>
#include "GpuResourceUtils.h"
---省略---
```

```
// コンストラクタ
Model::Model(ID3D11Device* device, const char* filename)
{
    std::filesystem::path filepath(filename);
    std::filesystem::path dirpath(filepath.parent_path());

    ---省略---

    // マテリアルデータ読み取り
    importer.LoadMaterials(materials);

    // マテリアル構築
    for (Material& material : materials)
    {
        // ディフューズテクスチャ読み込み
        std::filesystem::path diffuseTexturePath(dirpath / material.diffuseTextureFileName);
        HRESULT hr = GpuResourceUtils::LoadTexture(device, diffuseTexturePath.string().c_str(),
                                                   material.diffuseMap.GetAddressOf());

        _ASSERT_EXPR(SUCCEEDED(hr), HRTrace(hr));
    }

    // メッシュ構築
    for (Mesh& mesh : meshes)
    {
        // 参照マテリアル設定
        mesh.material = &materials.at(mesh.materialIndex);

        ---省略---
    }
}
```

ディレクトリパスを取得

相対パスを解決し、
テクスチャファイルパスを作成

メッシュデータからアクセスしやすいように
マテリアルのポインタを保持する

シェーダープログラムもマテリアル対応します。

Phong.hlsli

```
struct VS_OUT
{
    float4 vertex    : SV_POSITION;
    float2 texcoord  : TEXCOORD;
};

---省略---

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

PhongVS.hlsl

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

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

    return vout;
}
```

PhongPS.hlsl

```
#include "Phong.hlsli"

Texture2D diffuseMap       : register(t0);
SamplerState linearSampler : register(s0);

float4 main(VS_OUT pin) : SV_TARGET
{
    return diffuseMap.Sample(linearSampler, pin.texcoord) * materialColor;
}
```

PhongShader クラスにメッシュ用の定数バッファを作成し、マテリアルカラーデータを GPU に渡します。

PhongShader.h

```
#pragma once

#include "Shader.h"

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

    struct CbMesh
    {
        DirectX::XMFLOAT4      materialColor;
    };

    ---省略---
    Microsoft::WRL::ComPtr<ID3D11Buffer>            meshConstantBuffer;
};
```

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

PhongShader.cpp

```
---省略---

PhongShader::PhongShader(ID3D11Device* device)
{
    // 入力レイアウト
    D3D11_INPUT_ELEMENT_DESC inputElementDesc[] =
```

```
    {
        ---省略---
        { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT,  0, D3D11_APPEND_ALIGNED_ELEMENT,
                                                                D3D11_INPUT_PER_VERTEX_DATA, 0 },
    };

    ---省略---
    // メッシュ用定数バッファ
    GpuResourceUtils::CreateConstantBuffer(
        device,
        sizeof(CbMesh),
        meshConstantBuffer.GetAddressOf());
}

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

    // 定数バッファ設定
    ID3D11Buffer* constantBuffers[] =
    {
        sceneConstantBuffer.Get(),
        meshConstantBuffer.Get(),
    };
    rc.deviceContext->VSSetConstantBuffers(0, _countof(constantBuffers), constantBuffers);
    rc.deviceContext->PSSetConstantBuffers(0, _countof(constantBuffers), constantBuffers);

    // サンプラステート設定
    ID3D11SamplerState* samplerStates[] =
    {
        rc.renderState->GetSamplerState(SamplerState::LinearWrap)
    };
    dc->PSSetSamplers(0, _countof(samplerStates), samplerStates);

    ---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;
        dc->UpdateSubresource(meshConstantBuffer.Get(), 0, 0, &cbMesh, 0, 0);

        // シェーダーリソースビュー設定
        dc->PSSetShaderResources(0, 1, mesh.material->diffuseMap.GetAddressOf());

        // 描画
        dc->DrawIndexed(static_cast<UINT>(mesh.indices.size()), 0, 0);
```

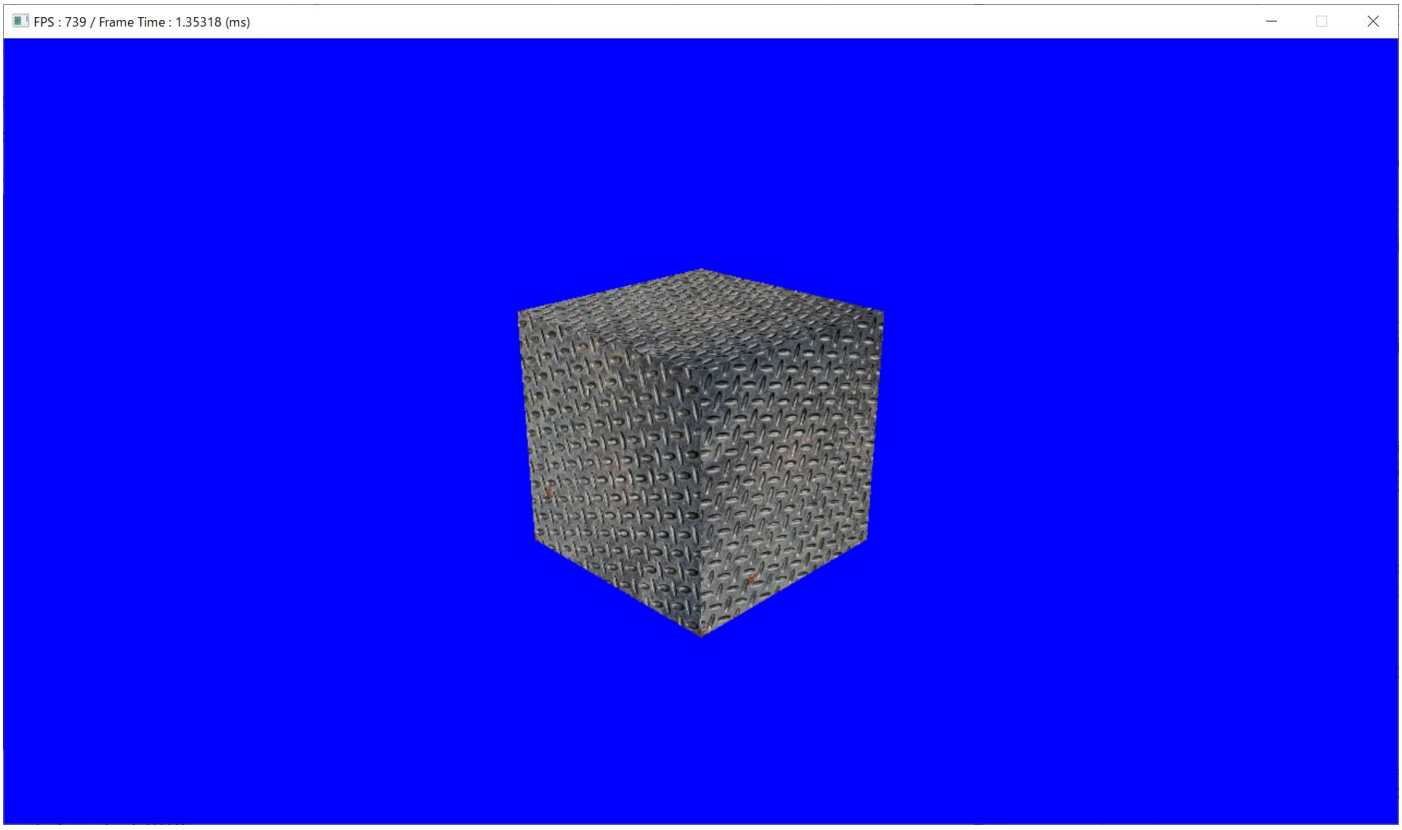
ピクセルシェーダーにも
定数バッファを設定する

```
    }
}
```

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.000.fbx");
    model = std::make_unique<Model>(device, "Data/Model/Cube/cube.001.0.fbx");
}
```

実行確認してみましょう。
下図のようにテクスチャが貼られた立方体が表示されていれば OK です。



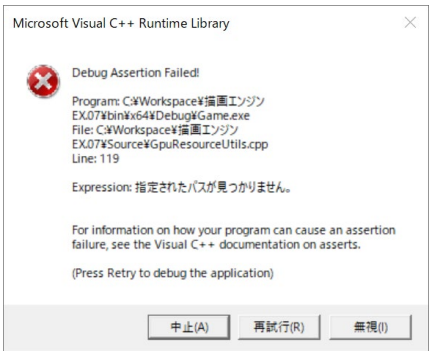
○ダミーテクスチャ対応

次はテクスチャが存在しない 3D モデルを読み込んで表示します。

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.001.0.fbx");
    model = std::make_unique<Model>(device, "Data/Model/Cube/cube.001.2.fbx");
}
```

実行確認すると下図のようなエラーが表示されます。
この 3D モデルはテクスチャがないのにテクスチャを読み込もうとしていることが原因です。



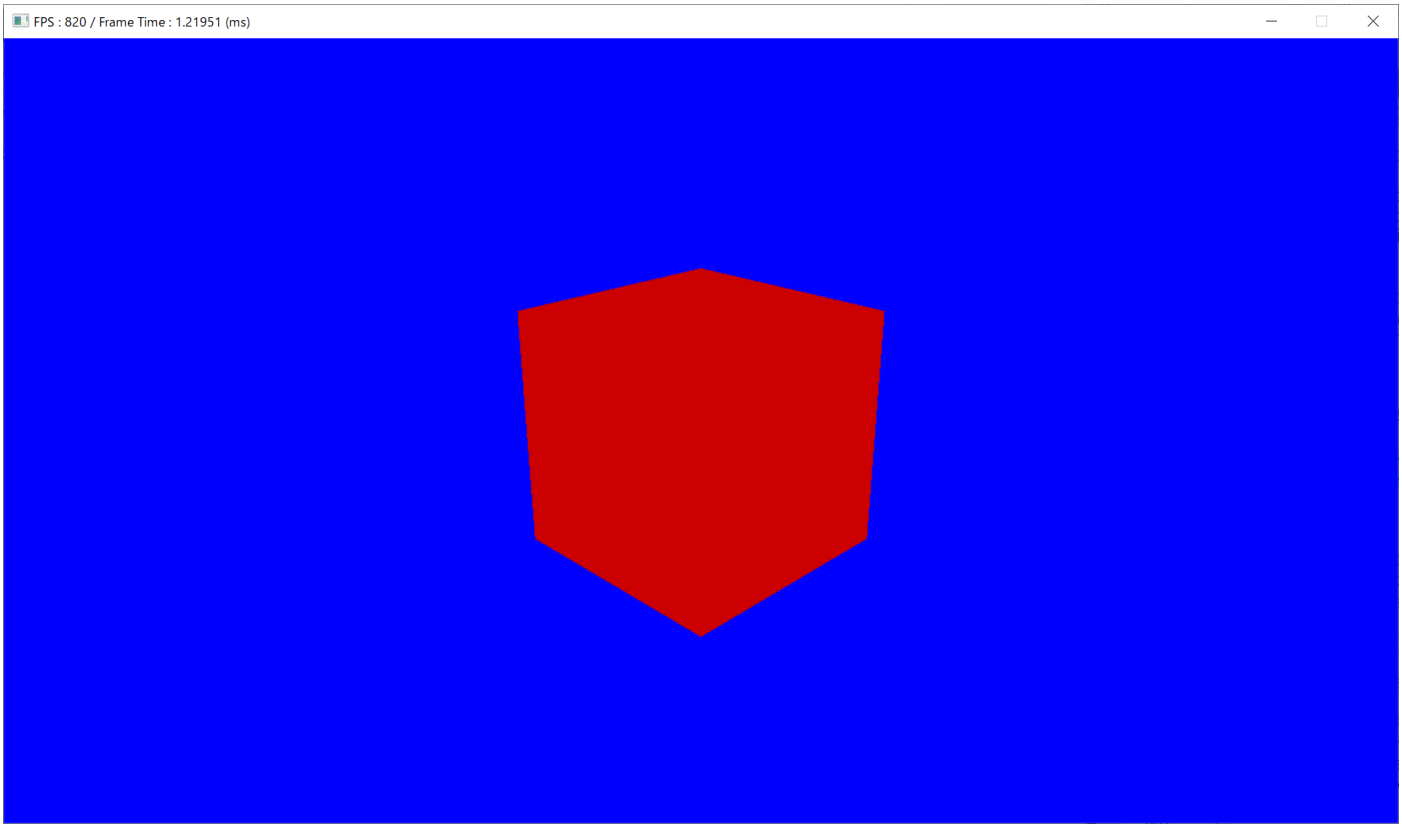
テクスチャがない 3D モデルの場合でも正常に読み込めるようにしましょう。

Model.cpp

```
---省略---

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

    // マテリアル構築
    for (Material& material : materials)
    {
        if (material.diffuseTextureFileName.empty())
        {
            // ダミーテクスチャ作成
            HRESULT hr = GpuResourceUtils::CreateDummyTexture(device, 0xFFFFFFFF,
                                                              material.diffuseMap.GetAddressOf());
            _ASSERT_EXPR(SUCCEEDED(hr), HRTrace(hr));
        }
        else
        {
            // ディフューズテクスチャ読み込み
            ---省略---
        }
```

```
    }

    ---省略---
}
```

実行確認してみましょう。
下図のように赤い立方体が表示されていれば OK です。



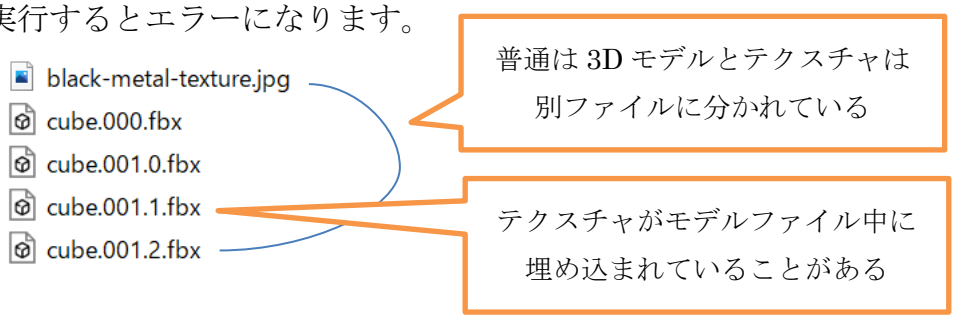
○埋め込みテクスチャ対応

次は 3D モデルファイルの中に画像データが内蔵されているファイルを読み込みます。

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

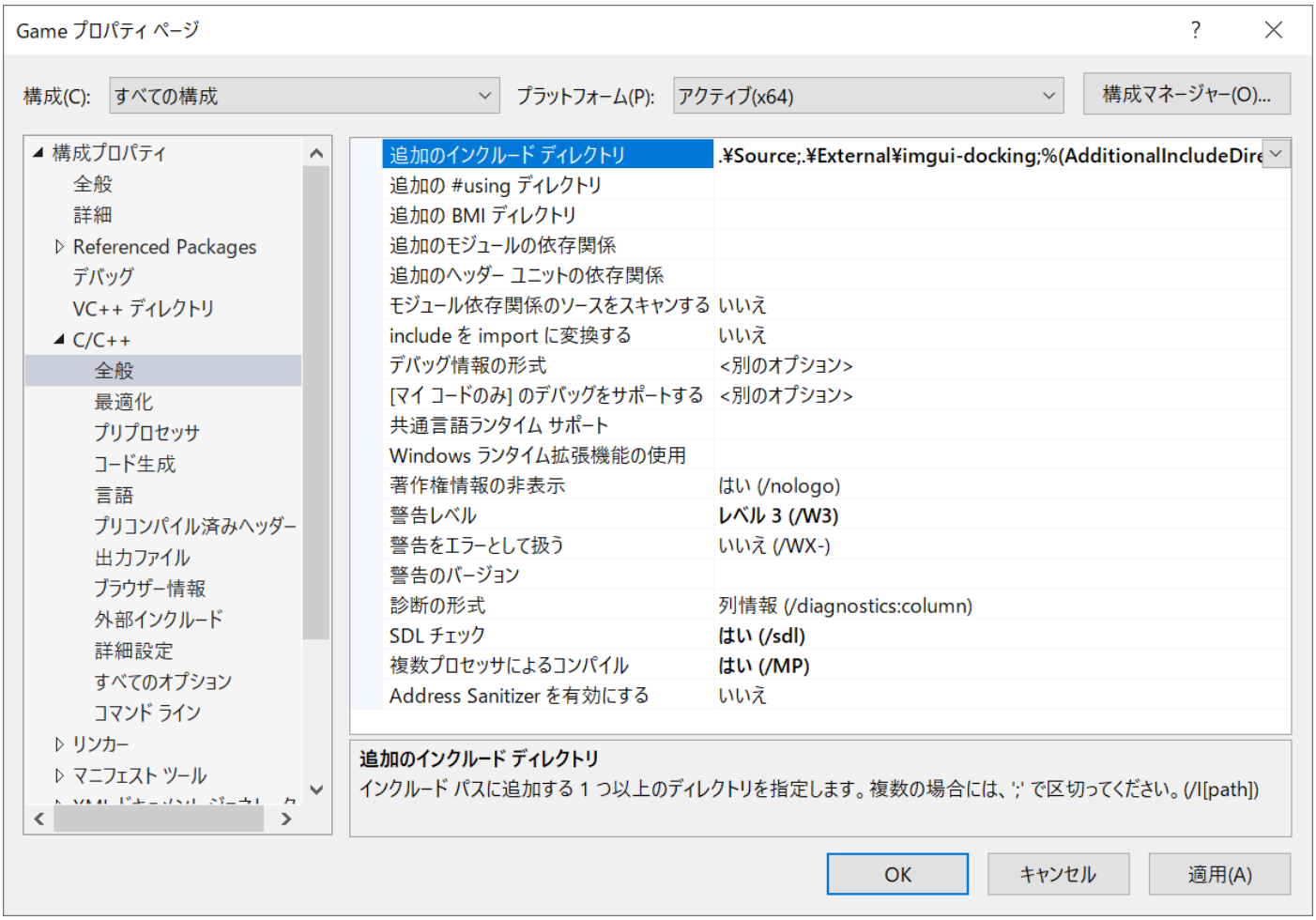
実行するとエラーになります。



Assimp ライブラリでは埋め込み画像のデータを取り出すことができます。
この画像データを指定ディレクトリに出力するプログラムを実装します。

画像ファイルを出力する stb ライブラリを使用するため、VisualStudio のプロジェクト設定でライブラリのインクルードパスを設定します。

VisualStudio のプロジェクト設定で左上の「構成」を「すべての構成」に設定し、「C/C++」を選択し、「追加のインクルードディレクトリを編集しましょう。



以下のライブラリパスを追加します。

.¥External¥stb

AssimpImporter のテクスチャ読み取り部分を改造し、埋め込みテクスチャが存在した場合は画像ファイルを指定ディレクトリに書き出すプログラムを実装します。

AssimpImporter.h

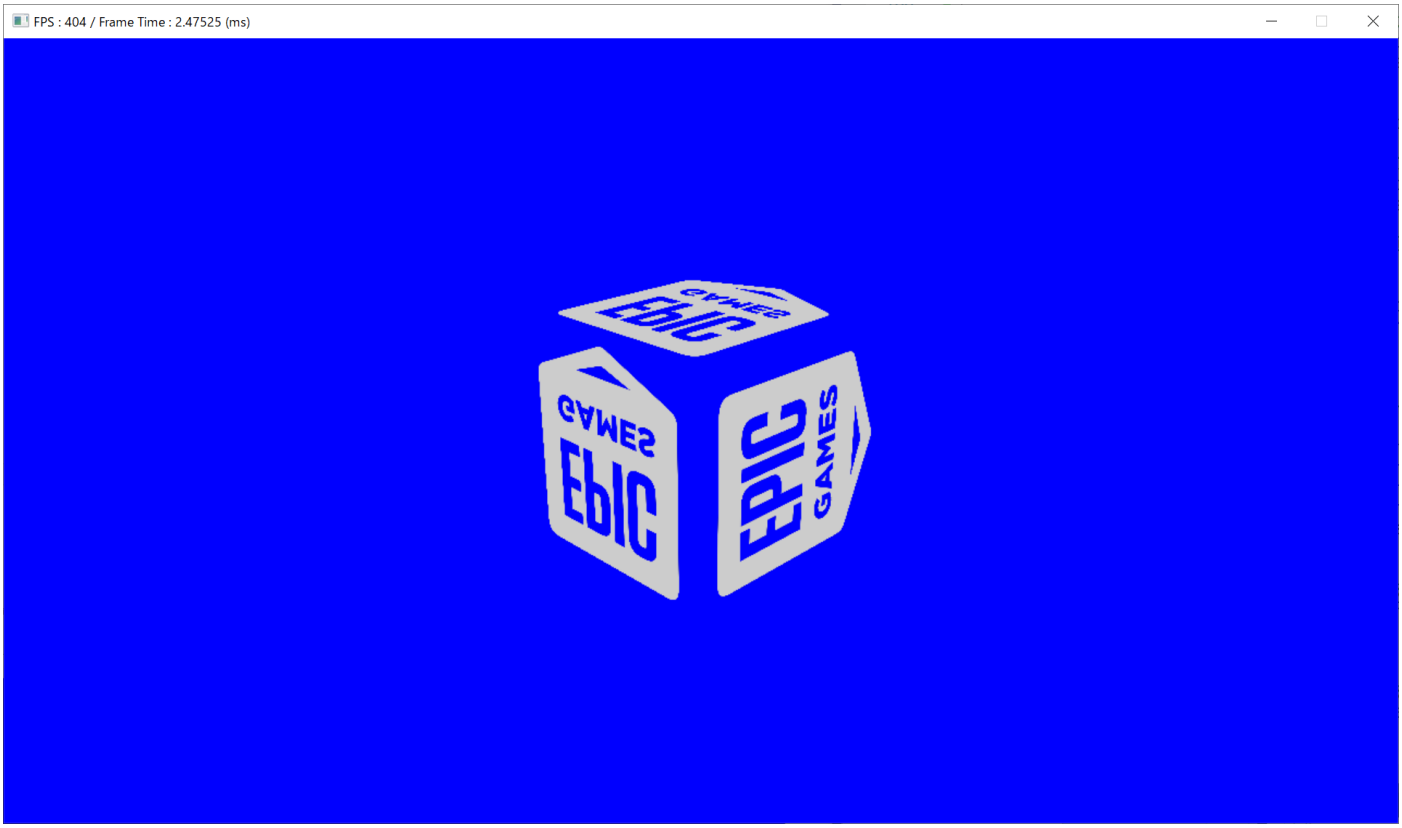
```
---省略---
#include <filesystem>

class AssimpImporter
{
    ---省略---

private:
    ---省略---
    std::filesystem::path filepath;
};
```

AssimpImporter.cpp

```
#include <fstream>
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include <stb_image_write.h>
---省略---

// コンストラクタ
AssimpImporter::AssimpImporter(const char* filename)
    : filepath(filename)
{
    ---省略---
}

---省略---

// マテリアルデータを読み込み
void AssimpImporter::LoadMaterials(MaterialList& materials)
{
    // ディレクトリパス取得
    std::filesystem::path dirpath(filepath.parent_path());

    ---省略---
    for (uint32_t aMaterialIndex = 0; aMaterialIndex < aScene->mNumMaterials; ++aMaterialIndex)
    {
        ---省略---

        // テクスチャ読み込み関数
        auto loadTexture = [&](aiTextureType aTextureType, std::string& textureFilename)
        {
            // テクスチャファイルパス取得
            aiString aTextureFilePath;
            if (AI_SUCCESS == aMaterial->GetTexture(aTextureType, 0, &aTextureFilePath))
            {
```

```
                    // 埋め込みテクスチャか確認
                    const aiTexture* aTexture = aScene->GetEmbeddedTexture(aTextureFilePath.C_Str());
                    if (aTexture != nullptr)
                    {
                        // テクスチャファイルパス作成
                        std::filesystem::path textureFilePath(aTextureFilePath.C_Str());
                        textureFilePath = "Textures" / textureFilePath.filename();

                        // 埋め込みテクスチャを出力するディレクトリを確認
                        std::filesystem::path outputDirPath(dirpath / textureFilePath.parent_path());
                        if (!std::filesystem::exists(outputDirPath))
                        {
                            // なかったらディレクトリ作成
                            std::filesystem::create_directories(outputDirPath);
                        }
                        // 出力ディレクトリに画像ファイルを保存
                        std::filesystem::path outputFilePath(dirpath / textureFilePath);
                        if (!std::filesystem::exists(outputFilePath))
                        {
                            // mHeightが0の場合は画像の生データなのでそのままバイナリ出力
                            if (aTexture->mHeight == 0)
                            {
                                std::ofstream os(outputFilePath.string().c_str(), std::ios::binary);
                                os.write(reinterpret_cast<char*>(aTexture->pcData), aTexture->mWidth);
                            }
                            else
                            {
                                // リニアな画像データは.pngで出力
                                stbi_write_png(
                                    outputFilePath.string().c_str(),
                                    static_cast<int>(aTexture->mWidth),
                                    static_cast<int>(aTexture->mHeight),
                                    static_cast<int>(sizeof(uint32_t)),
                                    aTexture->pcData, 0);
                            }
                        }
                        // テクスチャファイルパスを格納
                        textureFilename = textureFilePath.string();
                    }
                    else
                    {
                        // テクスチャファイルパスをそのまま格納
                        textureFilename = aTextureFilePath.C_Str();
                    }
                }
            };
            ---省略---
        }
    }

---省略---
```

実行確認してみましょう。
下図のようなテクスチャが貼られた立方体が表示されていれば OK です。



また、出力した画像ファイルが存在するか確認しておきましょう。