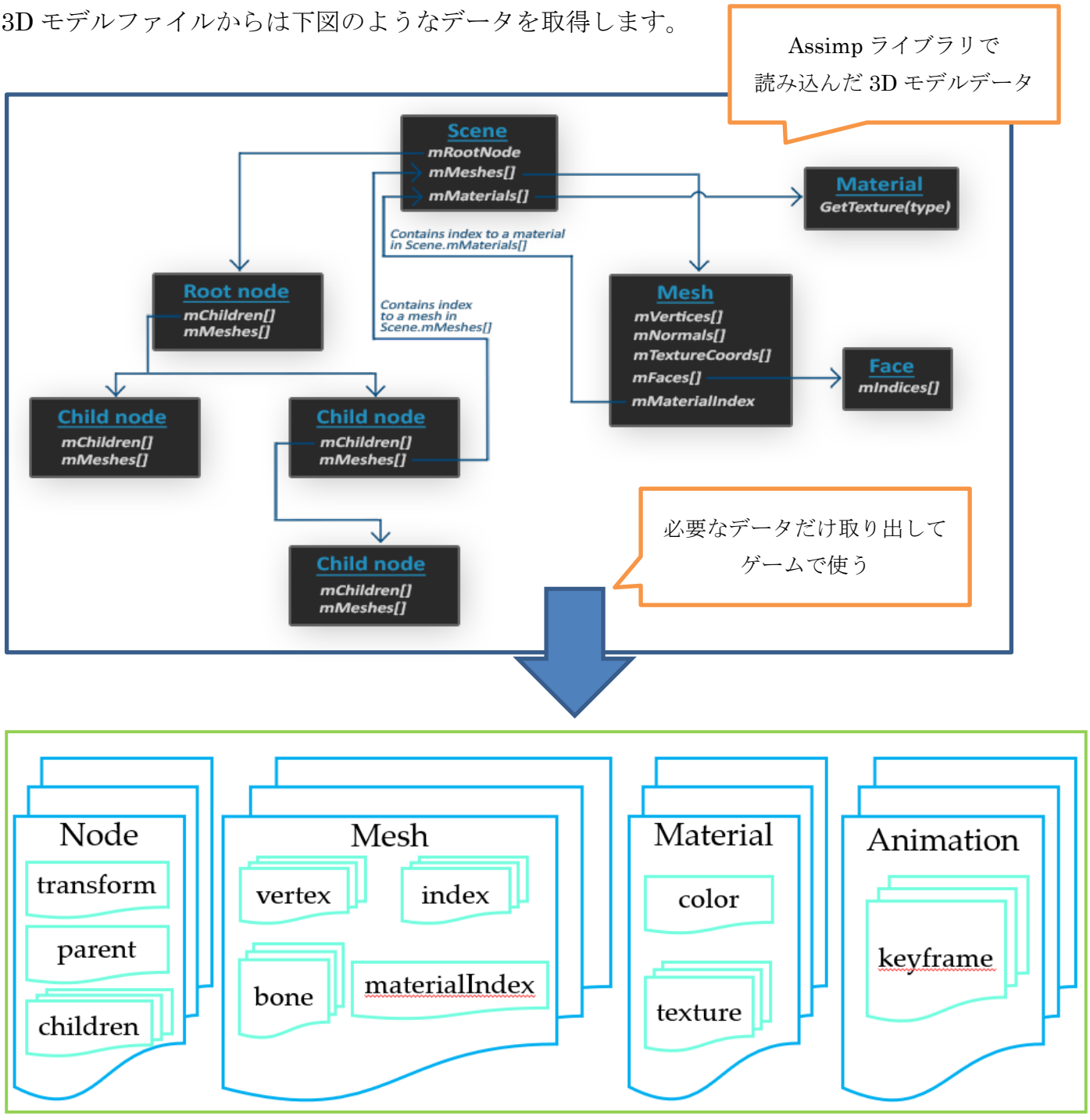
描画エンジン開発 EX

○概要

3D モデルファイルのデータを読み込み、3D 空間に 3D モデルを描画する。
3D モデルファイルの読み込みには Assimp ライブラリを利用します。

3D モデルファイルからは下図のようなデータを取得します。



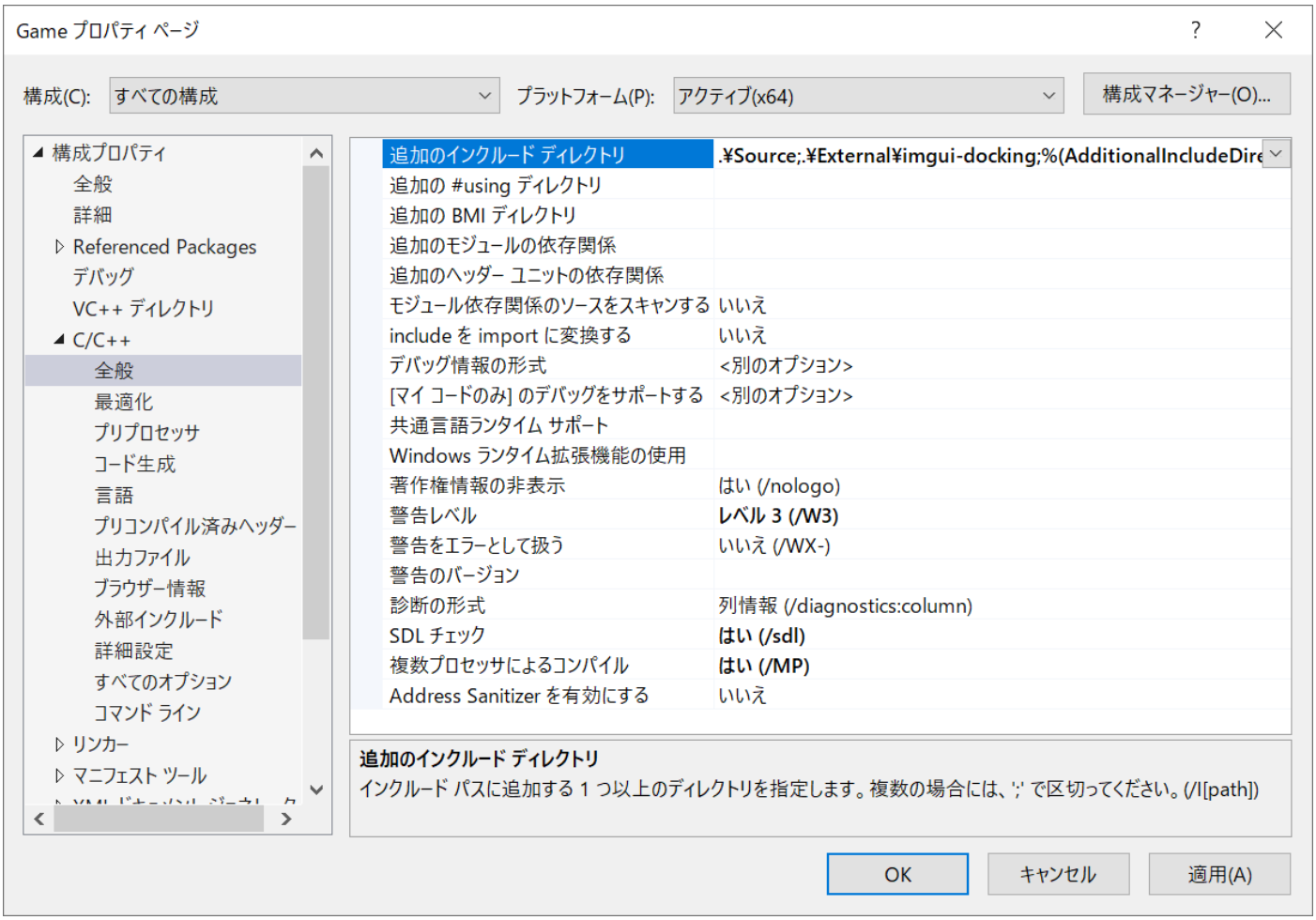
○Assimp

Assimp とは 3D モデルファイルを読み込み、頂点データやアニメーションデータなど 3D モデルを構成するデータの読み取りを支援してくれるライブラリです。

3D モデルファイルには様々な種類のデータが存在し、FBX、GLTF、OBJ など数多くのデータフォーマットの読み取りをサポートしています。

本来、Assimp ライブラリを導入するには GitHub からソースコードをダウンロードしてきて自身でビルドしたライブラリファイルをリンクするという工程が必要なります。
今回はビルド済みの Assimp ライブラリを用意していますので VisualStudio のプロジェクト設定でインクルードパスとライブラリのリンクを設定しましょう。

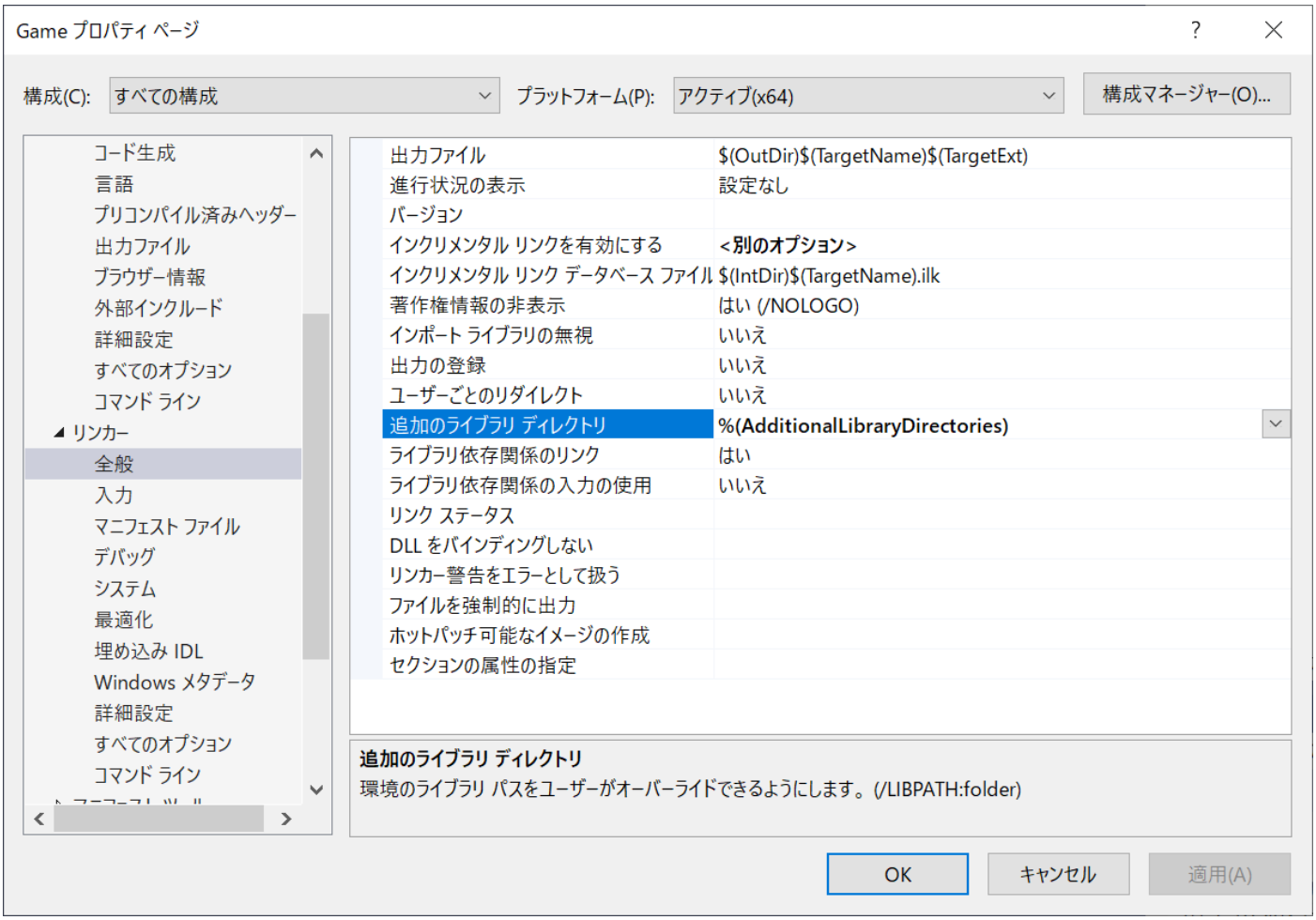
VisualStudio のプロジェクト設定で左上の「構成」を「すべての構成」に設定し、「C/C++」を選択し、「追加のインクルードディレクトリを編集しましょう。



以下のインクルードディレクトリを追加します。

.¥External¥assimp¥include

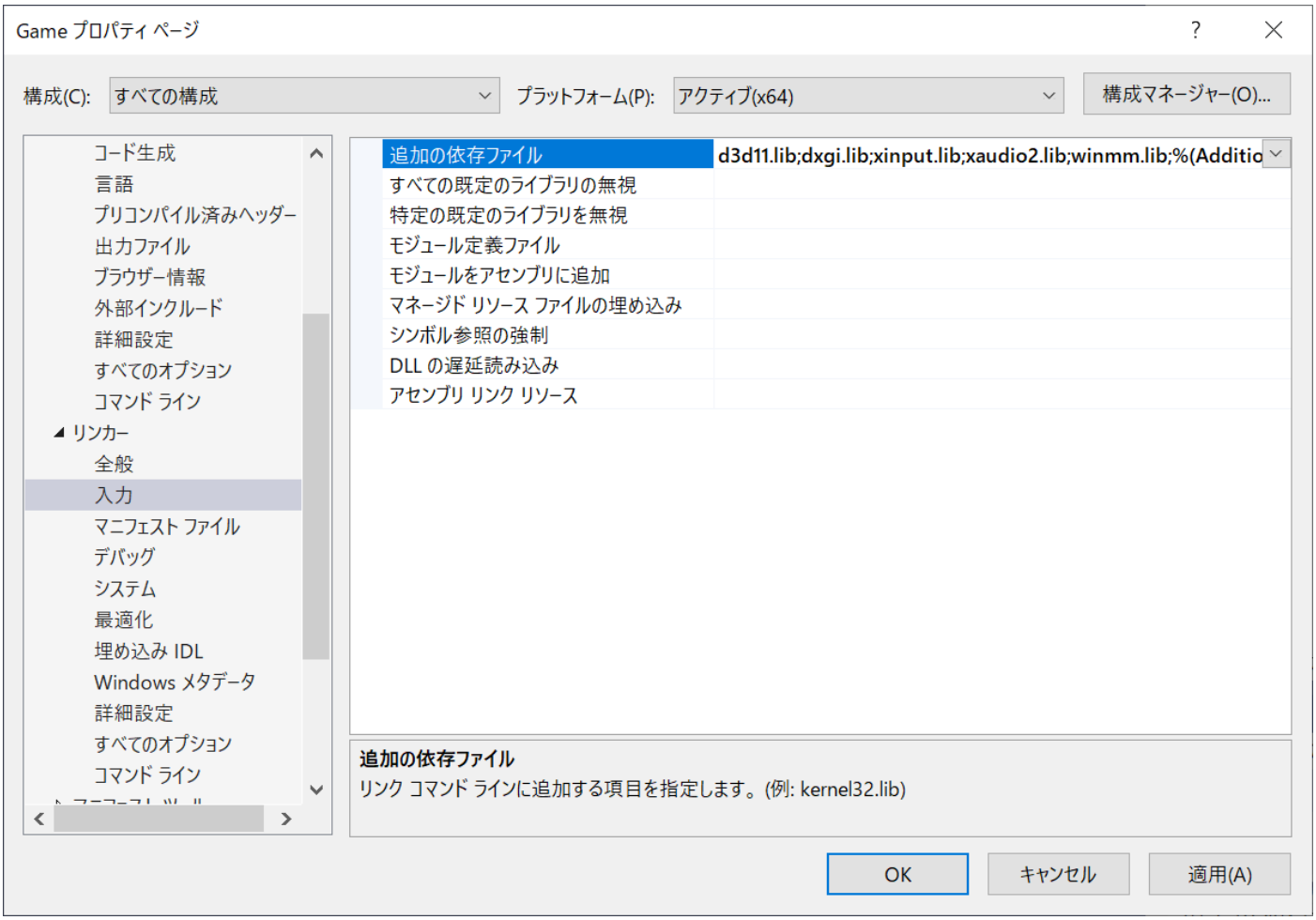
続いてリンク設定を追加します。
「リンカー」を選択し、「追加のライブラリディレクトリ」を編集しましょう。



以下のライブラリディレクトリを追加します。

.¥External¥assimp¥lib¥$(Platform)¥$(Configuration)

続いて「入力」を選択し、「追加の依存ファイル」を編集しましょう。



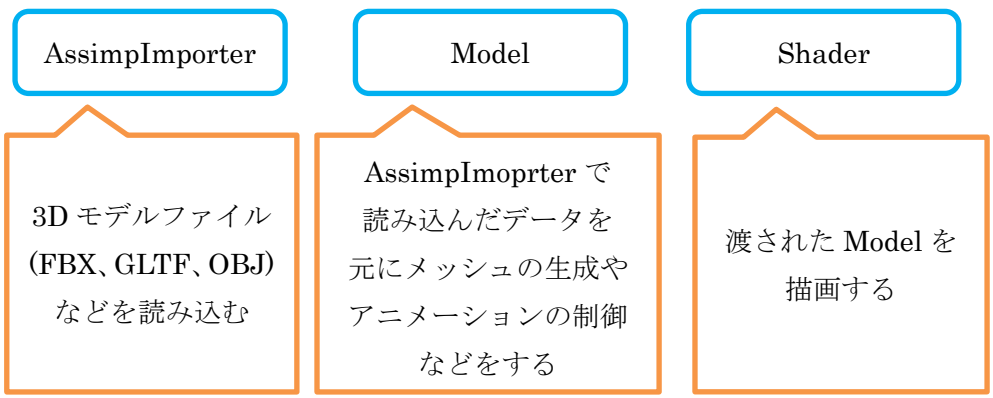
以下のライブラリファイルを追加します。

assimp.lib zlibstatic.lib

これでライブラリ設定は完了です。

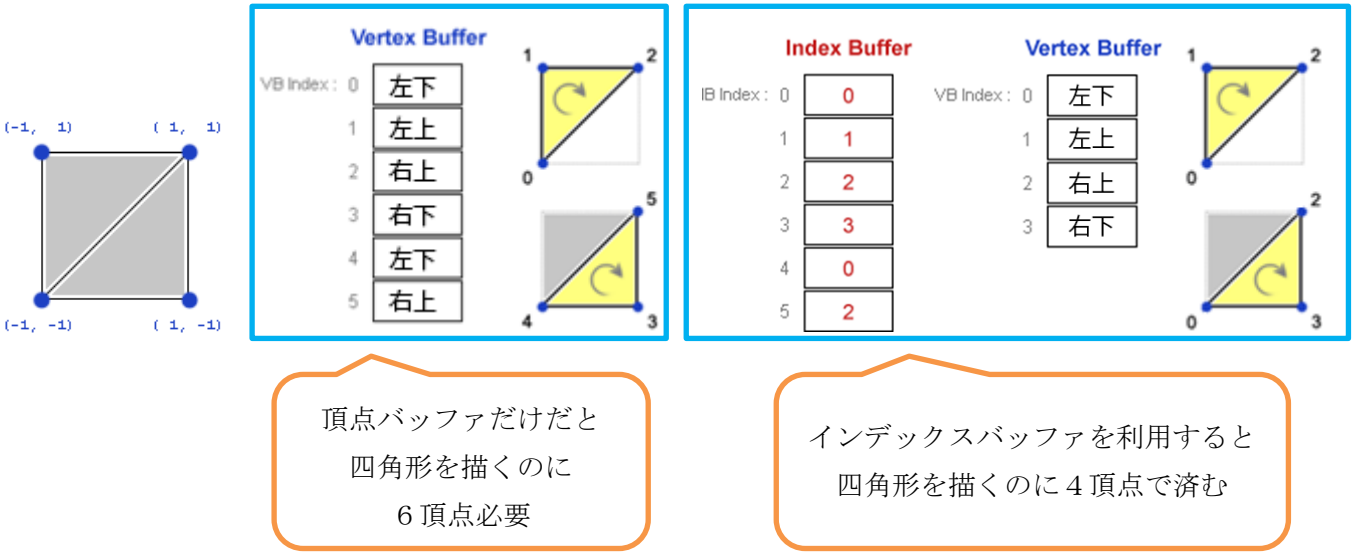
○3D モデル描画の設計

3D は 2D と比べるとデータが複雑でプログラムコードが煩雑になりやすいので、3D モデルファイルの読み込みをする前に 3D モデルを描画するまでのクラス設計をしておきます。



○メッシュ

複数のポリゴンで構成された形状のことをメッシュと呼びます。
メッシュは「頂点バッファ」と「インデックスバッファ」を元にポリゴンを描きます。
スプライトを描くときは頂点バッファのみで描きましたが、メッシュは形状が複雑でポリゴンを描く上で重複する頂点が出てくるのでインデックスバッファというデータを利用します。

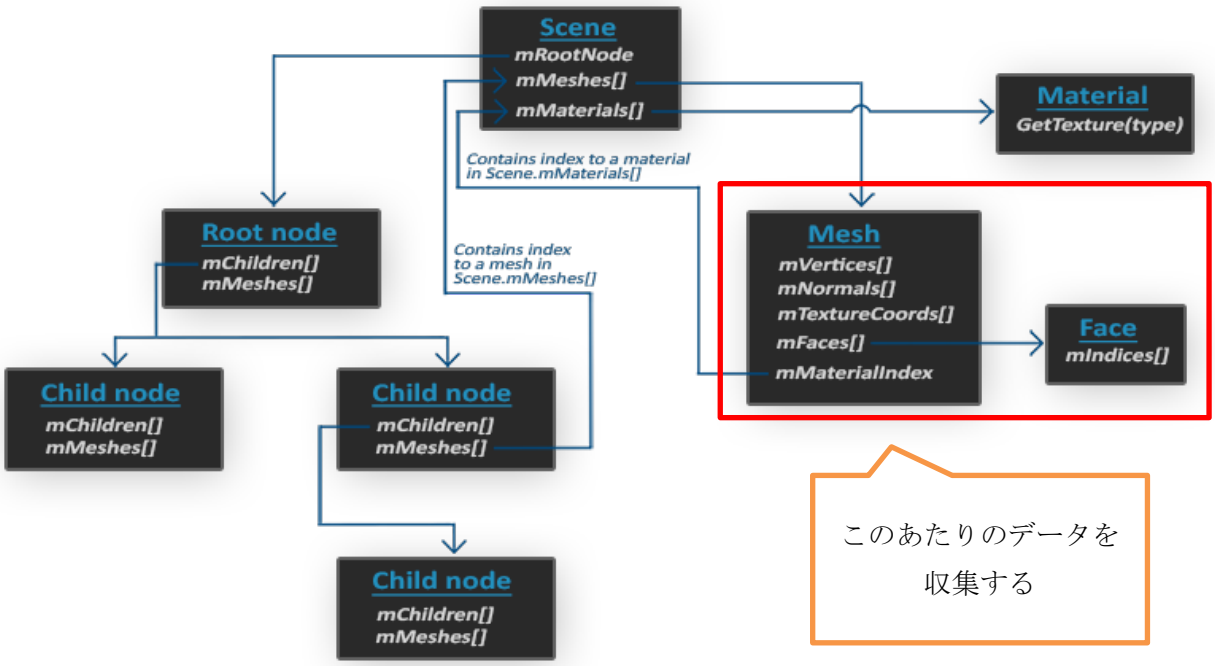


まずはメッシュに必要なデータを定義します。
Model クラスを作成し Mesh 構造体を定義します。
Model.h を作成しましょう。

Model.h

```
#pragma once

#include <vector>
#include <DirectXMath.h>

class Model
{
public:
    struct Vertex
    {
        DirectX::XMFLOAT3     position = { 0, 0, 0 };
    };

    struct Mesh
    {
        std::vector<Vertex>       vertices;
        std::vector<uint32_t>     indices;
    };
};
```

次に Assimp ライブラリを利用して 3D モデルファイルを読み込み、3D モデルデータを読み取る AssimpImporter クラスを作成し、メッシュデータを収集します。



AssimpImporter.cpp と AssimpImporter.h を作成しましょう。

AssimpImporter.h

```
#pragma once

#include <assimp/Importer.hpp>
#include <assimp/postprocess.h>
#include <assimp/scene.h>
#include "Model.h"

class AssimpImporter
{
private:
    using MeshList = std::vector<Model::Mesh>;

public:
    AssimpImporter(const char* filename);

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

private:
    // aiVector3D → XMFLOAT3
    static DirectX::XMFLOAT3 AssimpImporter::aiVector3DToXMFLOAT3(const aiVector3D& aValue);
```

```
private:
    Assimp::Importer      aImporter;
    const aiScene*        aScene = nullptr;
};
```

AssimpImporter.cpp

```
#include "Misc.h"
#include "AssimpImporter.h"

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

    // ファイル読み込み
    aScene = aImporter.ReadFile(filename, aFlags);
    _ASSERT_EXPR_A(aScene, "3D Model File not found");
}

// メッシュデータを読み込み
void AssimpImporter::LoadMeshes(MeshList& meshes)
{
    const aiMesh* aMesh = aScene->mMeshes[0];

    // メッシュデータ格納
    Model::Mesh& mesh = meshes.emplace_back();
    mesh.vertices.resize(aMesh->mNumVertices);
    mesh.indices.resize(aMesh->mNumFaces * 3);

    // 頂点データ
    for (uint32_t aVertexIndex = 0; aVertexIndex < aMesh->mNumVertices; ++aVertexIndex)
    {
        Model::Vertex& vertex = mesh.vertices.at(aVertexIndex);
        // 位置
        if (aMesh->HasPositions())
        {
            vertex.position = aiVector3DToXMFLOAT3(aMesh->mVertices[aVertexIndex]);
        }
    }

    // インデックスデータ
    for (uint32_t aFaceIndex = 0; aFaceIndex < aMesh->mNumFaces; ++aFaceIndex)
    {
        const aiFace& aFace = aMesh->mFaces[aFaceIndex];
        uint32_t index = aFaceIndex * 3;
        mesh.indices[index + 0] = aFace.mIndices[0];
        mesh.indices[index + 1] = aFace.mIndices[1];
        mesh.indices[index + 2] = aFace.mIndices[2];
    }
}

// aiVector3D → XMFLOAT3
```

Face とはポリゴンのこと。
DirectX では三角形で
ポリゴンを描画するので
3×ポリゴン数分の
インデックスデータを用意する

```
DirectX::XMFLOAT3 AssimpImporter::aiVector3DToXMFLOAT3(const aiVector3D& aValue)
{
    return DirectX::XMFLOAT3(
        static_cast<float>(aValue.x),
        static_cast<float>(aValue.y),
        static_cast<float>(aValue.z)
    );
}
```

AssimpImporter を利用して 3D モデルデータを読み込み、頂点バッファとインデックスバッファを構築します。

Model.h

```
---省略---
#include <wrl.h>
#include <d3d11.h>

class Model
{
public:
    Model(ID3D11Device* device, const char* filename);

    ---省略---

    struct Mesh
    {
        ---省略---

        Microsoft::WRL::ComPtr<ID3D11Buffer>    vertexBuffer;
        Microsoft::WRL::ComPtr<ID3D11Buffer>    indexBuffer;
    };

    // メッシュデータ取得
    const std::vector<Mesh>& GetMeshes() const { return meshes; }

private:
    std::vector<Mesh>     meshes;
};
```

Model.cpp を作成しましょう。

Model.cpp

```
#include "Misc.h"
#include "AssimpImporter.h"
#include "Model.h"

// コンストラクタ
Model::Model(ID3D11Device* device, const char* filename)
{
    // ファイル読み込み
```

```
    AssimpImporter importer(filename);

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

    // メッシュ構築
    for (Mesh& mesh : meshes)
    {
        // 頂点バッファ
        {
            D3D11_BUFFER_DESC bufferDesc = {};
            D3D11_SUBRESOURCE_DATA subresourceData = {};

            bufferDesc.ByteWidth = static_cast<UINT>(sizeof(Vertex) * mesh.vertices.size());
            bufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
            bufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
            bufferDesc.CPUAccessFlags = 0;
            bufferDesc.MiscFlags = 0;
            bufferDesc.StructureByteStride = 0;
            subresourceData.pSysMem = mesh.vertices.data();
            subresourceData.SysMemPitch = 0;
            subresourceData.SysMemSlicePitch = 0;

            HRESULT hr = device->CreateBuffer(&bufferDesc, &subresourceData, mesh.vertexBuffer.GetAddressOf());
            _ASSERT_EXPR(SUCCEEDED(hr), HRTrace(hr));
        }

        // インデックスバッファ
        {
            D3D11_BUFFER_DESC bufferDesc = {};
            D3D11_SUBRESOURCE_DATA subresourceData = {};

            bufferDesc.ByteWidth = static_cast<UINT>(sizeof(uint32_t) * mesh.indices.size());
            bufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
            bufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;
            bufferDesc.CPUAccessFlags = 0;
            bufferDesc.MiscFlags = 0;
            bufferDesc.StructureByteStride = 0;
            subresourceData.pSysMem = mesh.indices.data();
            subresourceData.SysMemPitch = 0;
            subresourceData.SysMemSlicePitch = 0;
            HRESULT hr = device->CreateBuffer(&bufferDesc, &subresourceData, mesh.indexBuffer.GetAddressOf());
            _ASSERT_EXPR(SUCCEEDED(hr), HRTrace(hr));
        }
    }
}
```

Model を描画する標準的なシェーダーを作成します。
Phong.hlsli と PhongVS.hlsl と PhongPS.hlsl を作成しましょう。

Phong.hlsli

```
struct VS_OUT
{
```

```
    float4 vertex : SV_POSITION;
};

cbuffer CbScene : register(b0)
{
    row_major float4x4    viewProjection;
};
```

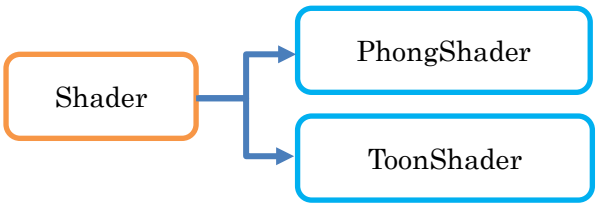
PhongVS.hlsl

```
#include "Phong.hlsli"

VS_OUT main(float4 position : POSITION)
{
    VS_OUT vout = (VS_OUT)0;

    vout.vertex = mul(position, viewProjection);

    return vout;
}
```

PhongPS.hlsl

```
#include "Phong.hlsli"

float4 main(VS_OUT pin) : SV_TARGET
{
    return float4(1, 1, 1, 1);
}
```

続いて Model を描画する Shader クラスを作成します。
今後、様々な 3D モデルの描画表現をしたいので下図のような設計にします。



Shader.h を作成しましょう。

Shader.h

```
#pragma once

#include "RenderContext.h"
#include "Model.h"

class Shader
```

```
{
public:
    Shader() {}
    virtual ~Shader() {}

    // 描画開始
    virtual void Begin(const RenderContext& rc) = 0;

    // モデル描画
    virtual void Draw(const RenderContext& rc, const Model* model) = 0;

    // 描画終了
    virtual void End(const RenderContext& rc) = 0;
};
```

Shader を継承した PhongShader を作成します。
PhongShader.cpp と PhongShader.h を作成しましょう。

PhongShader.h

```
#pragma once

#include "Shader.h"

class PhongShader : public Shader
{
public:
    PhongShader(ID3D11Device* device);
    ~PhongShader() override = default;

    // 描画開始
    void Begin(const RenderContext& rc) override;

    // モデル描画
    void Draw(const RenderContext& rc, const Model* model) override;

    // 描画終了
    void End(const RenderContext& rc) override;

private:
    struct CbScene
    {
        DirectX::XMFLOAT4X4    viewProjection;
    };

    Microsoft::WRL::ComPtr<ID3D11Buffer>            sceneConstantBuffer;

    Microsoft::WRL::ComPtr<ID3D11VertexShader>      vertexShader;
    Microsoft::WRL::ComPtr<ID3D11PixelShader>       pixelShader;
    Microsoft::WRL::ComPtr<ID3D11InputLayout>       inputLayout;
};
```

PhongShader.cpp

```
#include "Misc.h"
#include "GpuResourceUtils.h"
#include "PhongShader.h"

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 },
    };

    // 頂点シェーダー
    GpuResourceUtils::LoadVertexShader(
        device,
        "Data/Shader/PhongVS.cso",
        inputElementDesc,
        _countof(inputElementDesc),
        inputLayout.GetAddressOf(),
        vertexShader.GetAddressOf());

    // ピクセルシェーダー
    GpuResourceUtils::LoadPixelShader(
        device,
        "Data/Shader/PhongPS.cso",
        pixelShader.GetAddressOf());

    // シーン用定数バッファ
    GpuResourceUtils::CreateConstantBuffer(
        device,
        sizeof(CbScene),
        sceneConstantBuffer.GetAddressOf());
}

// 描画開始
void PhongShader::Begin(const RenderContext& rc)
{
    ID3D11DeviceContext* dc = rc.deviceContext;

    // シェーダー設定
    dc->IASetInputLayout(inputLayout.Get());
    rc.deviceContext->VSSetShader(vertexShader.Get(), nullptr, 0);
    rc.deviceContext->PSSetShader(pixelShader.Get(), nullptr, 0);

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

    // レンダーステート設定
    const float blend_factor[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
    dc->OMSetBlendState(rc.renderState->GetBlendState(BlendState::Transparency),
```

```
                                                                                blend_factor, 0xFFFFFFFF);
    dc->OMSetDepthStencilState(rc.renderState->GetDepthStencilState(DepthState::TestAndWrite),
                                                                                0);
    dc->RSSetState(rc.renderState->GetRasterizerState(RasterizerState::SolidCullBack));

    // シーン用定数バッファ更新
    CbScene cbScene{};
    DirectX::XMMATRIX V = DirectX::XMLoadFloat4x4(&rc.camera->GetView());
    DirectX::XMMATRIX P = DirectX::XMLoadFloat4x4(&rc.camera->GetProjection());
    DirectX::XMStoreFloat4x4(&cbScene.viewProjection, V * P);
    dc->UpdateSubresource(sceneConstantBuffer.Get(), 0, 0, &cbScene, 0, 0);
}

// 描画
void PhongShader::Draw(const RenderContext& rc, const Model* model)
{
    ID3D11DeviceContext* dc = rc.deviceContext;

    for (const Model::Mesh& mesh : model->GetMeshes())
    {
        // 頂点バッファ設定
        UINT stride = sizeof(Model::Vertex);
        UINT offset = 0;
        dc->IASetVertexBuffers(0, 1, mesh.vertexBuffer.GetAddressOf(), &stride, &offset);
        dc->IASetIndexBuffer(mesh.indexBuffer.Get(), DXGI_FORMAT_R32_UINT, 0);
        dc->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

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

// 描画終了
void PhongShader::End(const RenderContext& rc)
{
    ID3D11DeviceContext* dc = rc.deviceContext;
    dc->VSSetShader(nullptr, nullptr, 0);
    dc->PSSetShader(nullptr, nullptr, 0);
    dc->IASetInputLayout(nullptr);
}
```

Graphics クラスから任意のシェーダーを取得できるようにします。

Graphics.h

```
---省略---
#include "Shader.h"

enum class ShaderId
{
    Phong,

    EnumCount
};
```

```
// グラフィックス
class Graphics
{
public:
    ---省略---

    // シェーダー取得
    Shader* GetShader(ShaderId shaderId) { return shaders[static_cast<int>(shaderId)].get(); }

private:
    ---省略---
    std::unique_ptr<Shader>             shaders[static_cast<int>(ShaderId::EnumCount)];
};
```

Graphics.cpp

```
---省略---
#include "PhongShader.h"

// 初期化
void Graphics::Initialize(HWND hWnd)
{
    ---省略---

    // シェーダー生成
    shaders[static_cast<int>(ShaderId::Phong)] = std::make_unique<PhongShader>(device.Get());
}
```

Scene.h

```
---省略---
#include "Model.h"

---省略---

// モデルテストシーン
class ModelTestScene : public Scene
{
public:
    ModelTestScene();
    ~ModelTestScene() override = default;

    // 描画処理
    void Render(float elapsedTime) override;

private:
    Camera                      camera;
    std::unique_ptr<Model>      model;
};
```

Scene.cpp

```
---省略---

// コンストラクタ
ModelTestScene::ModelTestScene()
{
    ID3D11Device* device = Graphics::Instance().GetDevice();
    float screenWidth = Graphics::Instance().GetScreenWidth();
    float screenHeight = Graphics::Instance().GetScreenHeight();

    // カメラ設定
    camera.SetPerspectiveFov(
        DirectX::XMConvertToRadians(45),    // 画角
        screenWidth / screenHeight,         // 画面アスペクト比
        0.1f,                               // ニアクリップ
        1000.0f                             // ファークリップ
    );
    camera.SetLookAt(
        { 5, 3, -5 },     // 視点
        { 0, 0, 0 },      // 注視点
        { 0, 1, 0 }       // 上ベクトル
    );

    // モデル作成
    model = std::make_unique<Model>(device, "Data/Model/Cube/cube.000.fbx");
}

// 描画処理
void ModelTestScene::Render(float elapsedTime)
{
    // 描画コンテキスト設定
    RenderContext rc;
    rc.camera = &camera;
    rc.deviceContext = Graphics::Instance().GetDeviceContext();
    rc.renderState = Graphics::Instance().GetRenderState();

    // 描画
    Shader* shader = Graphics::Instance().GetShader(ShaderId::Phong);
    shader->Begin(rc);
    shader->Draw(rc, model.get());
    shader->End(rc);
}
```

Framework.cpp

```
---省略---

// コンストラクタ
Framework::Framework(HWND hWnd)
{
    ---省略---

    // シーン初期化
    scene = std::make_unique<GizmosTestScene>();
    scene = std::make_unique<ModelTestScene>();
}
```

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