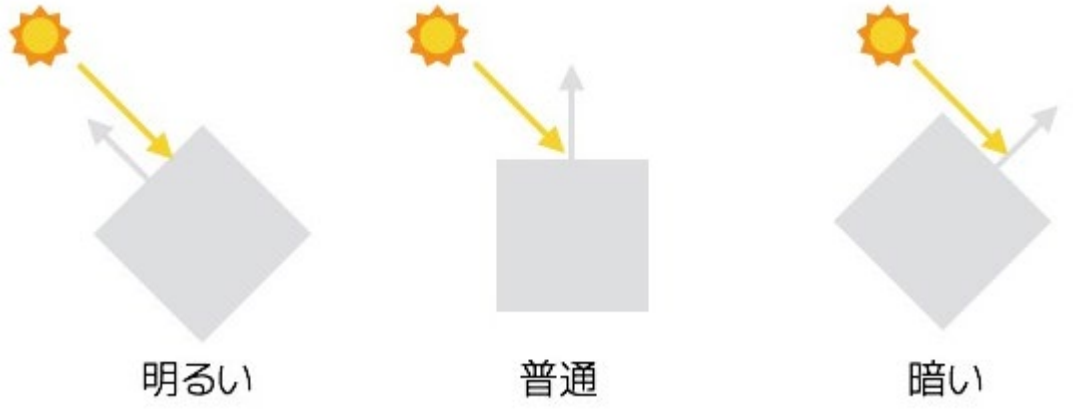
描画エンジン開発 EX

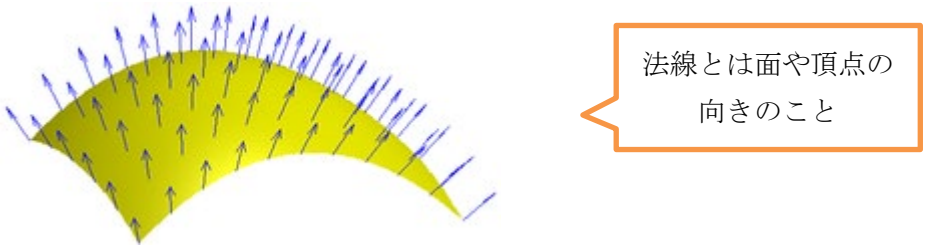
○概要

ライティング処理を実装する。

○ランバートシェーディング

メッシュの法線ベクトルと光線ベクトルとの角度差のみで明るさを決定するライティングをランバートシェーディングと呼びます。
光線と法線の角度差が大きくなるほど暗くなります。





まずはライト情報を管理する LightManager クラスと並行光の情報である DirectionalLight 構造体を定義します。
Light.h を作成しましょう。

Light.h

```
#pragma once

#include <DirectXMath.h>

struct DirectionalLight
{
    DirectX::XMFLOAT3 direction;
    DirectX::XMFLOAT3 color;
};

class LightManager
{
public:
```

```
    // ディレクショナルライト設定
    void SetDirectionalLight(DirectionalLight& light) { directionalLight = light; }

    // ディレクショナルライト取得
    const DirectionalLight& GetDirectionalLight() const { return directionalLight; }

private:
    DirectionalLight  directionalLight;
};
```

描画時にライト情報を参照できるように RenderContext に LightManager のポインタを追加します。

RenderContext.h

```
---省略---
#include "Light.h"

struct RenderContext
{
    ---省略---
    const LightManager*        lightManager;
};
```

ライティングの計算には法線データが必要なので Model クラスの頂点データに法線を追加し、AssimpImporter で法線データを収集します。

Model.h

```
---省略---

class Model
{
public:
    ---省略---

    struct Vertex
    {
        ---省略---
        DirectX::XMFLOAT3      normal = { 0, 0, 0 };
    };

    ---省略---
};
```

AssimpImporter.cpp

```
---省略---

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

```
    : filepath(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
                    | aiProcess_CalcTangentSpace;       // 接線を計算する

    ---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---

        // 頂点データ
        for (uint32_t aVertexIndex = 0; aVertexIndex < aMesh->mNumVertices; ++aVertexIndex)
        {
            ---省略---
            // 法線
            if (aMesh->HasNormals())
            {
                vertex.normal = aiVector3DToXMFLOAT3(aMesh->mNormals[aVertexIndex]);
            }
        }
        ---省略---
    }
    ---省略---
}
```

シェーダープログラムにライティングするためのデータを定義します。

Phong.hlsli

```
struct VS_OUT
{
    ---省略---
    float3 normal     : NORMAL;
};

cbuffer CbScene : register(b0)
{
    ---省略---
    float4                lightDirection;
    float4                lightColor;
```

```
};
```

ベクトルをスキニングする関数を作成します。

Skinning.hlsli

```
---省略---

float3 SkinningVector(float3 vec, float4 boneWeights, uint4 boneIndices)
{
    float3 v = float3(0, 0, 0);

    [unroll]
    for (int i = 0; i < 4; i++)
    {
        v += boneWeights[i] * mul(float4(vec, 0), boneTransforms[boneIndices[i]]).xyz;
    }
    return v;
}
```

位置要素の計算は省きたいので
w の値は 0 にしている

頂点シェーダーに送られてきた法線データをスキニングしてピクセルシェーダーに渡します。

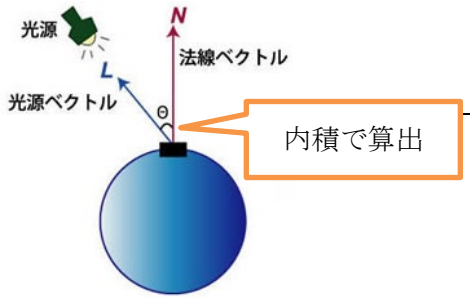
PhongVS.hlsl

```
---省略---

VS_OUT main(
    ---省略---
    float3 normal         : NORMAL)
{
    ---省略---

    vout.normal = SkinningVector(normal, boneWeights, boneIndices);

    return vout;
}
```

ピクセルシェーダーでライティングの計算をします。

PhongPS.hlsl

```
---省略---

float4 main(VS_OUT pin) : SV_TARGET
{
    float4 color = diffuseMap.Sample(linearSampler, pin.texcoord) * materialColor;

    float3 N = normalize(pin.normal);
    float3 L = normalize(-lightDirection.xyz);
    float power = max(0, dot(L, N));
```



```
    color.rgb *= lightColor.rgb * power;

    return color;
}
```

ライトデータをシェーダーに渡すため、定数バッファのデータを追加します。

PhongShader.h

```
---省略---

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

private:
    struct CbScene
    {
        ---省略---
        DirectX::XMFLOAT4 lightDirection;
        DirectX::XMFLOAT4 lightColor;
    };

    ---省略---
};
```

PhongShader.cpp

```
---省略---

PhongShader::PhongShader(ID3D11Device* device)
{
    // 入力レイアウト
    D3D11_INPUT_ELEMENT_DESC inputElementDesc[] =
    {
        ---省略---
        { "NORMAL",        0, DXGI_FORMAT_R32G32B32_FLOAT,     0, D3D11_APPEND_ALIGNED_ELEMENT,
                                                                      D3D11_INPUT_PER_VERTEX_DATA, 0 },
    };

    ---省略---
}

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

    // シーン用定数バッファ更新
    CbScene cbScene{};
    ---省略---
    const DirectionalLight& directionalLight = rc.lightManager->GetDirectionalLight();
```

```
    cbScene.lightDirection.x = directionalLight.direction.x;
    cbScene.lightDirection.y = directionalLight.direction.y;
    cbScene.lightDirection.z = directionalLight.direction.z;
    cbScene.lightColor.x = directionalLight.color.x;
    cbScene.lightColor.y = directionalLight.color.y;
    cbScene.lightColor.z = directionalLight.color.z;
    dc->UpdateSubresource(sceneConstantBuffer.Get(), 0, 0, &cbScene, 0, 0);
}
```

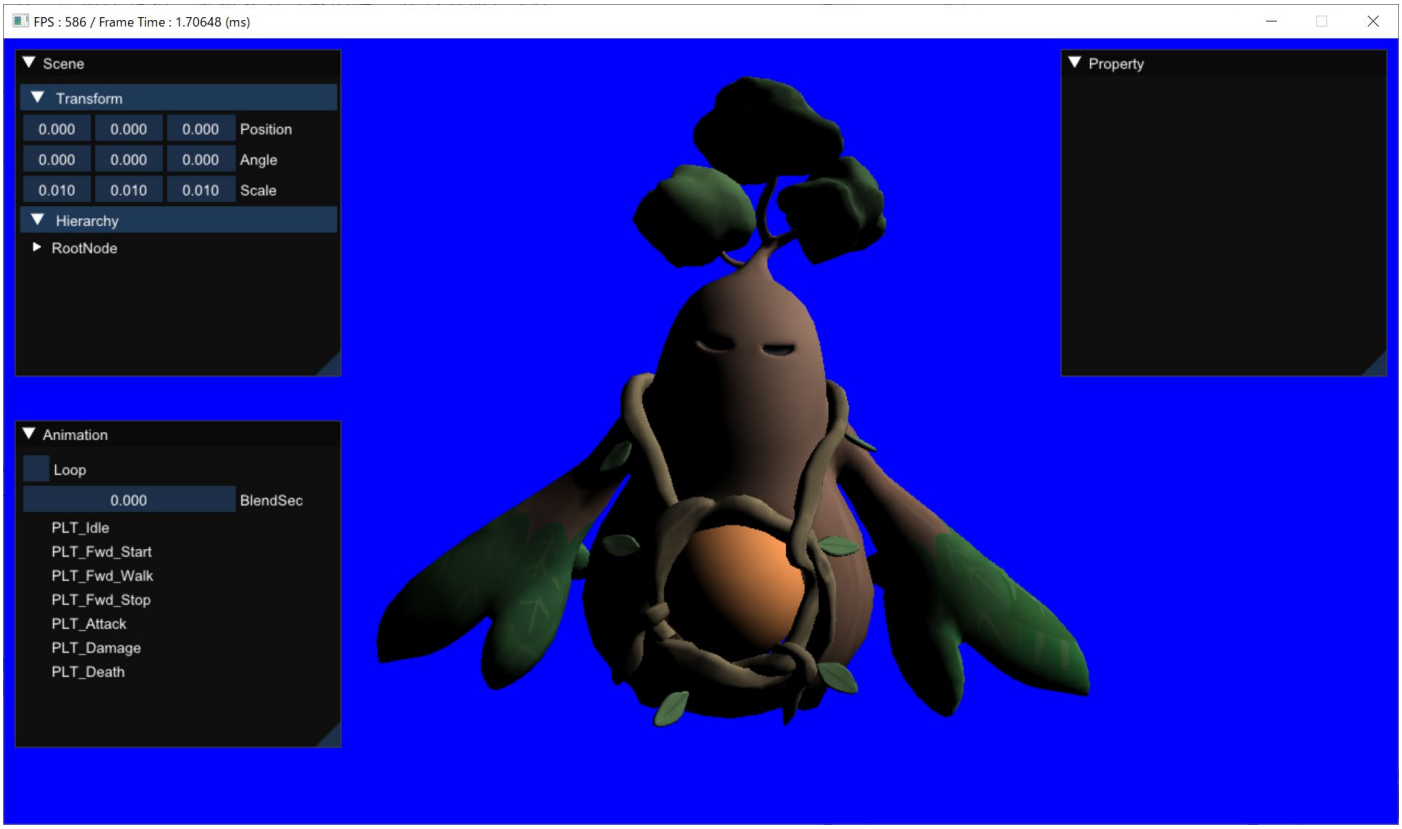
Scene.h

```
---省略---
#include "Light.h"

---省略---

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

private:
    ---省略---
    LightManager                lightManager;
};
```

Scene.cpp

```
---省略---

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

    // ライト設定
    DirectionalLight directionalLight;
    directionalLight.direction = { 1, -1, 0 };
    directionalLight.color = { 1, 1, 1 };
    lightManager.SetDirectionalLight(directionalLight);
}

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

    // 描画コンテキスト設定
    RenderContext rc;
    ---省略---
    rc.lightManager = &lightManager;

    ---省略---
}
```

実行確認してみましょう。
3D モデルに陰影が表現されていれば OK です。



○ハーフランバートシェーディング

通常のランバートシェーディングでは光が当たっていない部分が完全に黒くなってしまいます。
陰影の暗くなりすぎないように明暗の濃淡を和らいでライティングする技術がハーフランバートシェーディングです。

PhongPS.hlsl

```
---省略---

float4 main(VS_OUT pin) : SV_TARGET
```

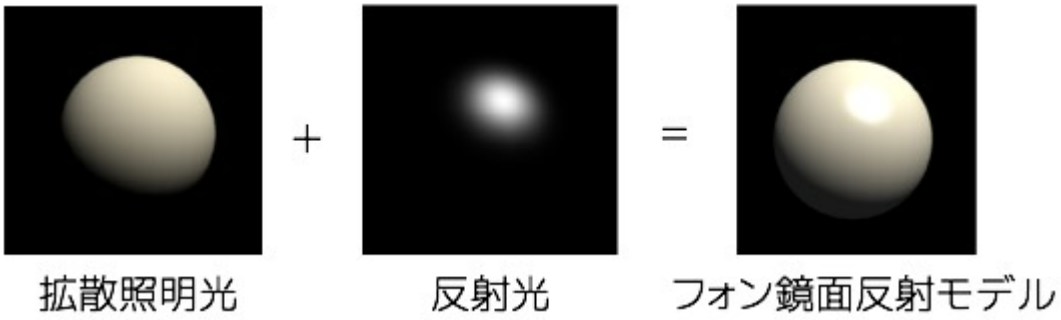
```
{
    ---省略---
    float power = max(0, dot(L, N));

    power = power * 0.7 + 0.3f;

    color.rgb *= lightColor.rgb * power;

    return color;
}
```

陰影をつける強度を 70%にして
30%は光が当たっているようにする

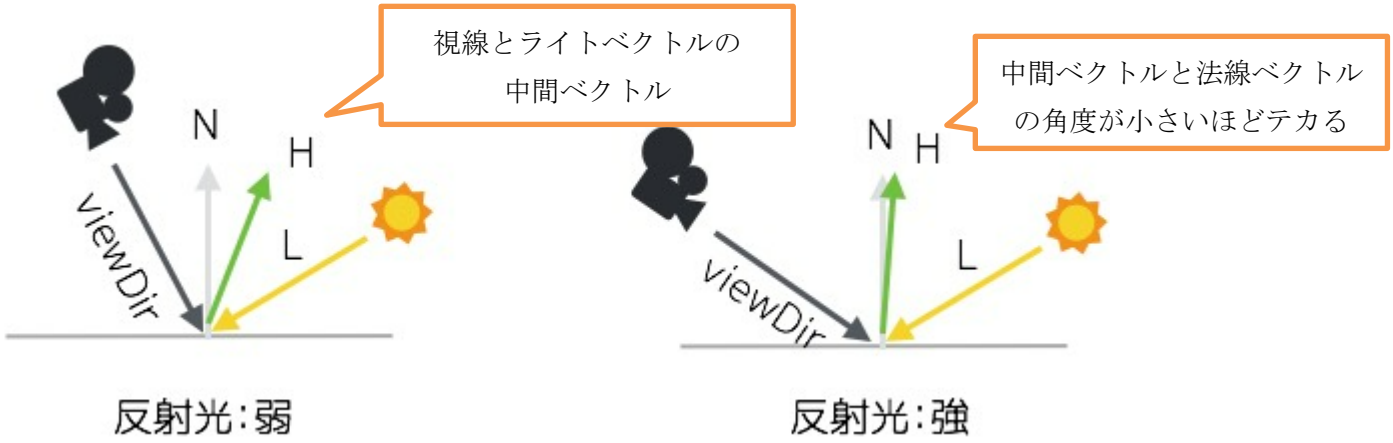
実行確認してみましょう。
ランバートシェーディングより陰影が和らいで表現されていれば OK です。

○フォンシェーディング

ランバートシェーディングの後にスペキュラという「テカり」を追加したシェーディングをフォンシェーディングと呼びます。



カメラの視線とライトの反射ベクトルからテカり具合を計算する。



シェーダーにカメラ位置を渡し、反射光の計算をします。

Phong.hlsli

```
struct VS_OUT
{
    ---省略---
    float3 position : POSITION;
};

cbuffer CbScene : register(b0)
{
    ---省略---
    float4                  cameraPosition;
};
```

PhongVS.hlsl

```
---省略---

VS_OUT main(
    float4 position       : POSITION,
    float4 boneWeights    : BONE_WEIGHTS,
```

```
    uint4  boneIndices    : BONE_INDICES,
    float2 texcoord       : TEXCOORD,
    float3 normal         : NORMAL)
{
    ---省略---
    vout.position = position.xyz;

    return vout;
}
```

視線ベクトルを求めるために
スキニング後のワールド座標を
ピクセルシェーダーに渡す

PhongPS.hlsl

```
---省略---

float4 main(VS_OUT pin) : SV_TARGET
{
    ---省略---

    float3 V = normalize(cameraPosition.xyz - pin.position);
    float3 specular = pow(max(0, dot(N, normalize(V + L))), 128);
    color.rgb += specular;

    return color;
}
```

視線ベクトルを算出

pow()はべき乗する関数。
べき乗することにより、
弱い光をさらに弱く、
強い光をそのままにする

PhongShader.h

```
---省略---

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

private:
    struct CbScene
    {
        ---省略---
        DirectX::XMFLOAT4 cameraPosition;
    };

    ---省略---
};
```

PhongShader.cpp

```
---省略---

// 描画開始
void PhongShader::Begin(const RenderContext& rc)
{
    ---省略---
    // シーン用定数バッファ更新
```

```
    CbScene cbScene{};
    ---省略---
    const DirectX::XMFLOAT3& eye = rc.camera->GetEye();
    cbScene.cameraPosition.x = eye.x;
    cbScene.cameraPosition.y = eye.y;
    cbScene.cameraPosition.z = eye.z;
    dc->UpdateSubresource(sceneConstantBuffer.Get(), 0, 0, &cbScene, 0, 0);
}
```

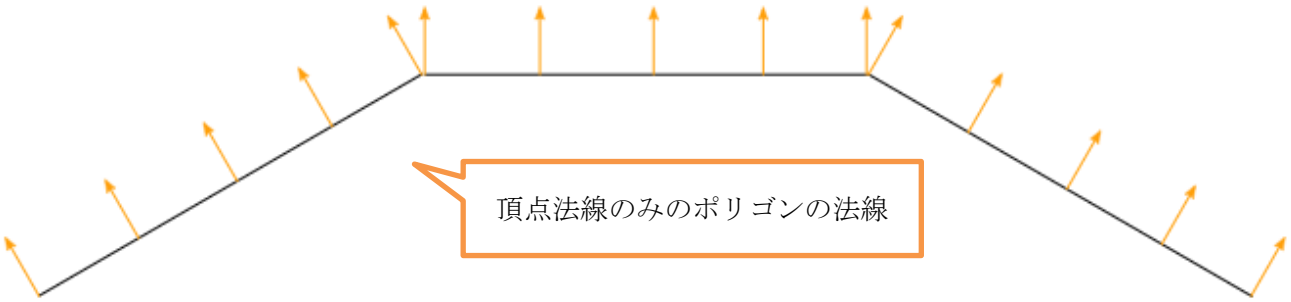
実行確認してみましょう。
3D モデルにテカリが表現されていれば OK です。

○ノーマルマップシェーディング

法線テクスチャを用いることでメッシュの法線データの精度を上げ、ライティングの精度を向上させる技術です。

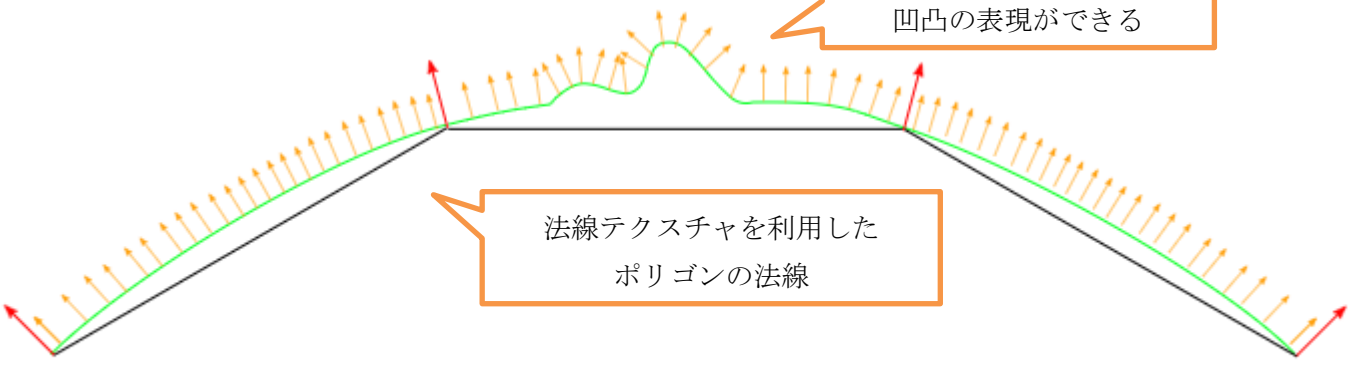
法線テクスチャを
利用した凹凸表現

法線テクスチャは色ではなく、
方向が格納されている

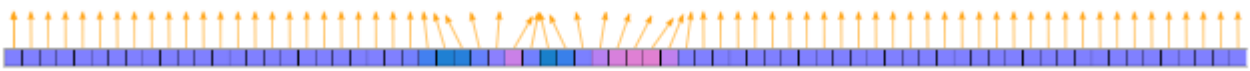


頂点法線のみのポリゴンの法線

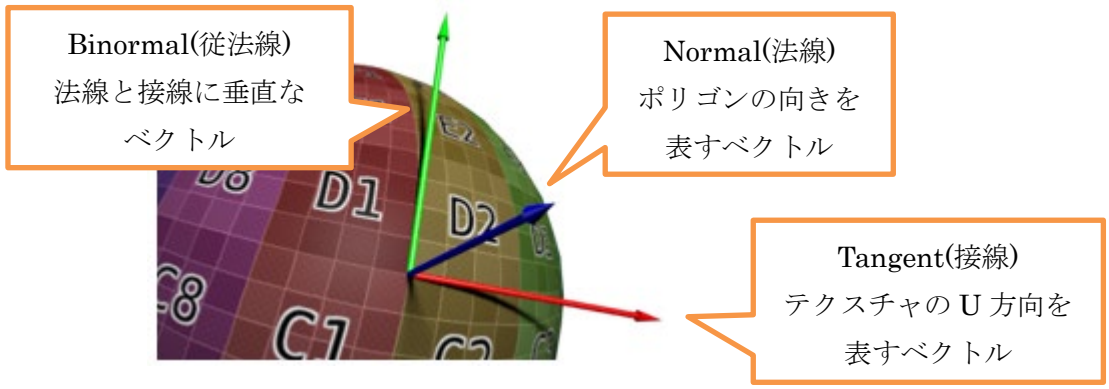
フラットな面でも細かい
凹凸の表現ができる



法線テクスチャを利用した
ポリゴンの法線



法線マップには Tangent（接線）と Binormal（従法線）呼ばれるベクトルデータが必要になります。



Phong.hlsli

```
struct VS_OUT
{
    ---省略---
    float3 tangent    : TANGENT;
};
---省略---
```

PhongVS.hlsl

```
---省略---

VS_OUT main(
    ---省略---
    float3 tangent        : TANGENT)
{
    ---省略---
    vout.tangent = SkinningVector(tangent, boneWeights, boneIndices);

    return vout;
}
```

PhongPS.hlsl

```
---省略---
Texture2D normalMap            : register(t1);

float4 main(VS_OUT pin) : SV_TARGET
{
    float4 color = diffuseMap.Sample(linearSampler, pin.texcoord) * materialColor;

    float3 N = normalize(pin.normal);
    float3 T = normalize(pin.tangent);
    float3 B = normalize(cross(N, T));

    float3 normal = normalMap.Sample(linearSampler, pin.texcoord).xyz;
    normal = (normal * 2.0f) - 1.0f;
    N = normalize((normal.x * T) + (normal.y * B) + (normal.z * N));

    ---省略---
```

Binormal(従法線)は
Normal(法線)と Tangent(接線)との
外積で算出できる

接空間の法線ベクトルを
ワールド空間のベクトルに変換

```
}
```

Model.h

```
---省略---

class Model
{
public:
    ---省略---

    struct Material
    {
        ---省略---
        std::string           normalTextureFileName;
    };

    struct Vertex
    {
        ---省略---
        DirectX::XMFLOAT3     tangent = { 0, 0, 0 };
    };

    ---省略---
};
```

AssimpImporter.cpp

```
---省略---

// マテリアルデータを読み込み
void AssimpImporter::LoadMaterials(MaterialList& materials)
{
    ---省略---
    for (uint32_t aMaterialIndex = 0; aMaterialIndex < aScene->mNumMaterials; ++aMaterialIndex)
    {
        ---省略---

        // ノーマルマップ
        loadTexture(aiTextureType_NORMALS, material.normalTextureFileName);
    }
}

// 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---
```

```
        // 頂点データ
        for (uint32_t aVertexIndex = 0; aVertexIndex < aMesh->mNumVertices; ++aVertexIndex)
        {
            ---省略---
            // 接線
            if (aMesh->HasTangentsAndBitangents())
            {
                vertex.tangent = aiVector3DToXMFLOAT3(aMesh->mTangents[aVertexIndex]);
            }
        }
        ---省略---
    }
    ---省略---
}
```

Model.h

```
---省略---

class Model
{
public:
    ---省略---

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

    ---省略---
};
```

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)
    {
        ---省略---

        if (material.normalTextureFileName.empty())
        {
            // 法線ダミーテクスチャ作成
            HRESULT hr = GpuResourceUtils::CreateDummyTexture(device, 0xFFFF7F7F,
                                                              material.normalMap.GetAddressOf());
            _ASSERT_EXPR(SUCCEEDED(hr), HRTrace(hr));
        }
```

法線ベクトルが(0,0,1)になる
ダミーテクスチャを作成

```
        else
        {
            // 法線テクスチャ読み込み
            std::filesystem::path texturePath(dirpath / material.normalTextureFileName);
            HRESULT hr = GpuResourceUtils::LoadTexture(device, texturePath.string().c_str(),
                                                                material.normalMap.GetAddressOf());
            _ASSERT_EXPR(SUCCEEDED(hr), HRTrace(hr));
        }
    }
    ---省略---
}
```

PhongShader.cpp

```
---省略---

PhongShader::PhongShader(ID3D11Device* device)
{
    // 入力レイアウト
    D3D11_INPUT_ELEMENT_DESC inputElementDesc[] =
    {
        ---省略---
        { "TANGENT",        0, DXGI_FORMAT_R32G32B32_FLOAT,     0, D3D11_APPEND_ALIGNED_ELEMENT,
                                                                    D3D11_INPUT_PER_VERTEX_DATA, 0 },
    };

    ---省略---
}

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

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

        // シェーダーリソースビュー設定
        ID3D11ShaderResourceView* srvs[] =
        {
            mesh.material->diffuseMap.Get(),
            mesh.material->normalMap.Get(),
        };
        dc->PSSetShaderResources(0, _countof(srvs), srvs);

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

実行確認してみましょう。
凹凸が表現されていれば OK です。


FPS : 543 / Frame Time : 1.84162 (ms)
Scene
Transform
0.000
0.000
0.000
Position
0.000
0.000
0.000
Angle
0.010
0.010
0.010
Scale
Hierarchy
RootNode
Property
Animation
Loop
0.000
BlendSec
PLT_Idle
PLT_Fwd_Start
PLT_Fwd_Walk
PLT_Fwd_Stop
PLT_Attack
PLT_Damage
PLT_Death