GameBaseDx11界隈

学校で使ってるゲームエンジンで、あれやりたいこれやりたい集

- キャラクターの真下にShaderを使って丸い影を描く（投影丸影）
- 地形生成してみるンゴ
  - 自分でランダムな地形？を作って表示する。テクスチャもつける
  - 必要なクラス(Terrain class)を作っていく
  - ランダムな三角形で地形を作っていく
  - GPUリソースを作っていく！→Draw関数
- 地形にプレイヤー立たせて丸影
  - まずは、プレイヤーをつくる
- シャドウマップへの道（険しい）
  - その1
  - その２
  - その３
  - その４
  - その５
  - その６
  - その７
  - その８
  - その９（おまけ）
- DirectXTKを使ったオーディオマネジメント
  - DirectXTKとAudio
  - 前準備その2　スマポ
  - ０１　音を鳴らしてみるだけ
  - 02 Engineに組み込んでゆくぅ
  - 03 SEとBGMを別扱いに（なんで？）
  - 04 サウンドのコントロール機能（簡単なのだけ）
  - 05 応用編　３D音響

キャラクターの真下にShaderを使って丸い影を描く（投影丸影）

☒ この影の作り方のイメージ

> プレイヤーの下に「黒い丸いライト」を当てて、地面を**ちょっとだけ暗く見せる**という工夫です。
> ライトなので、授業でやった点光源がわかっていれば全く同じように実装できます。
> しかも、現在のシェーダーにコンスタントバッファと、影付け部分を足すだけでできます。

こんな感じの影です☒

```
  ← プレイヤー（ジャンプ中でもOK）
  ↓
黒い光を下に照らす
  ↓
――――――――――――――

   ● ← 黒い影（地面に丸く表示される）
――――――――――――――
```

☒ なぜこんな影を作るの？

本物の影は、光を遮ってリアルタイムに計算するので**とても重たい（処理が大変）**です。

でも、このやり方は：

項目	内容
☒ ゲーム性能	とても軽い！（超高速）
☒ 理解しやすさ	仕組みが簡単！
☒ 実装方法	ライトと同じように扱える

☒ どうやって作るの？

丸影（Blob Shadow）導入フロー

---

1. ☒ 目的整理

- 丸影を「地面に貼り付けるような」影として合成する
- 地面用の描画パスに統合せず、「影合成処理」は別パスとして独立させたい
- プレイヤーの現在位置（XZ）と高さを使って「影の位置・サイズ・濃さ」を計算

---

2. ☒ 定数バッファの設計（CBShadow）

```
struct CBShadow {
    XMFLOAT4 casterPos;    // プレイヤーのXZ座標（Yは使わない：お空の方向を表すから）
    XMFLOAT4 shadowParams; // (softness, alphaScale, unused, playerHeightY)
};
```

- `.w` に高さを埋めることで1スロットで完結
- `b2` スロットなど空いてる定数バッファにバインド

---

3. ☒ HLSL シェーダー統合 (3Dのhlslの後ろに追加するなど？)

⚙ 丸影のアルファ合成ロジックをピクセルシェーダー末尾に追加

```
float2 casterXZ = casterPos.xz;
float2 pixelXZ = inData.wpos.xz;

float2 diff = pixelXZ - casterXZ;
float distSq = dot(diff, diff);

float softness = shadowParams.x;
float alphaScale = shadowParams.y;
float heightY = shadowParams.w;

float heightRatio = saturate(heightY / 2.0f); // 最大ジャンプ2.0f想定
float radius = lerp(0.4f, 1.0f, heightRatio);
float alpha  = lerp(0.6f, 0.1f, heightRatio);

float shadowAlpha = saturate((radius * radius - distSq) * softness) * alpha;

// 丸影を黒で合成（必要に応じて色も乗せられる）
float4 shadowColor = float4(0, 0, 0, shadowAlpha);
return lerp(resultColor, shadowColor, shadowAlpha);
```

---

4. ☒ C++側：描画前にプレイヤーの情報を渡す

Stage::Draw や Stage::Update にて：

```
CBShadow shadowCB;
shadowCB.casterPos = XMFLOAT4(playerPos.x, 0.0f, playerPos.z, 1.0f);
shadowCB.shadowParams = XMFLOAT4(softness, alphaScale, 0.0f, playerHeightY);

// 書き込み → バッファ更新
context->UpdateSubresource(pCBShadow, 0, nullptr, &shadowCB, 0, 0);
context->PSSetConstantBuffers(2, 1, &pCBShadow);
```

- `playerHeightY = playerPos.y - Stage::GetTerrainHeight(playerPos.x, playerPos.z)`
- プレイヤーの高さを計算して `.w` に渡すのがポイント

---

5. ☒ 描画順序

1. 通常の地面 + モデル描画（`Model::Draw()`）
2. その後 `Stage.hlsl` のピクセルシェーダー内で影を合成（クワッド描画不要）

---

6. ☒ 確認と調整

テスト項目	備考
影が正しい位置に出るか	XZが正しく渡っているか
高さでサイズ変化するか	`.w` の補間式が効いているか
透明度が変化しているか	`alpha` の計算式の係数調整
カメラを回しても違和感ないか	視差が出ないか、影が地面に貼りついて見えるか

☒ 最終的な関数と構造の一覧

要素	内容	追加すべき場所 / クラス
`CBShadow`	丸影用の定数バッファ構造体	`Engine/ShaderStruct.h` など定数バッファ定義系ヘッダ
`UpdateShadowCB()`	プレイヤー情報から `CBShadow` を更新・送信する関数	`Stage` クラス、または `ShadowManager` を作ってもOK
`Stage.hlsl`	丸影の合成コードを含む HLSL	`Assets/Shader/Stage.hlsl`（または新規に丸影対応シェーダ）
`GetTerrainHeight(x, z)`	指定座標の地面高さを返す	`Stage` クラスに追加（地形データを持っているなら）

☒ より具体的に解説

☒ 1. `CBShadow`

```
// ShaderStruct.h または ShadowStruct.h など
struct CBShadow {
    DirectX::XMFLOAT4 casterPos;    // プレイヤーのXZ座標
    DirectX::XMFLOAT4 shadowParams;  // (softness, alphaScale, -, playerHeightY)
};
```

- すでに `CBGlobal` や `CBLight` がある場所に追加すると整理しやすい

☒ 2. `UpdateShadowCB()` のような関数

追加先: `Stage.cpp` 内のメンバ関数 or `ShadowManager` クラスとして独立化もOK

```
void Stage::UpdateShadowCB(const Player& player)
{
    CBShadow cb;
    auto pos = player.GetTransform().position_;
    float terrainY = GetTerrainHeight(pos.x, pos.z);

    cb.casterPos = XMFLOAT4(pos.x, 0.0f, pos.z, 1.0f);
    cb.shadowParams = XMFLOAT4(softness, alphaScale, 0.0f, pos.y - terrainY);

    D3D11_MAPPED_SUBRESOURCE mapped;
    context->Map(pCBShadow_, 0, D3D11_MAP_WRITE_DISCARD, 0, &mapped);
    memcpy(mapped.pData, &cb, sizeof(cb));
    context->Unmap(pCBShadow_, 0);

    context->PSSetConstantBuffers(2, 1, &pCBShadow_);
}
```

☒ 3. `Stage.hlsl` 内のピクセルシェーダー後半に合成コード

追加先: `Assets/Shader/Stage.hlsl`

```
cbuffer ShadowParam : register(b2)
{
    float4 casterPos;
    float4 shadowParams; // (softness, alphaScale, -, playerHeightY)
}

...

// ピクセルシェーダー末尾
```

```
float2 delta = inData.wpos.xz - casterPos.xz;
float distSq = dot(delta, delta);

float softness = shadowParams.x;
float alphaScale = shadowParams.y;
float playerHeightY = shadowParams.w;
float heightRatio = saturate(playerHeightY / 2.0f);
float radius = lerp(0.4f, 1.0f, heightRatio);
float alpha  = lerp(0.6f, 0.1f, heightRatio);

float shadowAlpha = saturate((radius * radius - distSq) * softness) * alpha;
float4 shadowColor = float4(0, 0, 0, shadowAlpha);

return lerp(resultColor, shadowColor, shadowAlpha);
```

☒ 4. `Stage::GetTerrainHeight(x, z)` 関数

追加先: `Stage.h` / `Stage.cpp`

```
float Stage::GetTerrainHeight(float x, float z) const
{
    // 例えば地形がグリッドなら、高さマップや地面モデルから高さを補間して返す
    return heightMap.SampleAt(x, z);
}
```

☒ 補足：必要な DirectX11 の初期化項目

名前	概要
`ID3D11Buffer* pCBShadow_`	丸影用定数バッファ
`CreateBuffer()` で生成	初期化時に `sizeof(CBShadow)` を渡す

地形生成してみるンゴ

いつものエンジンで、地形を生成して自分のゲームに読みこんで、キャラクターを絶たせるまでの軌跡

自分でランダムな地形？を作って表示する。テクスチャもつける

☒ ステップ①：データの設計（どんな情報を使うか？）

地形を作るには「たくさんの点（てん）」が必要です。
この点は「**頂点（ちょうてん）**」と呼ばれていて、1つ1つの頂点には次のような情報があります：

名前	何の情報？
position	その点がどこにあるか（x, y, z）
normal	光の当たり方（かたむき）
uv	絵（テクスチャ）の貼り方

これをたくさん集めて、「地面の形」を作っていきます。
地面の形は三角形をたくさん並べてできていて、三角形のつなぎ方を `indices` という番号のデータで管理します。

☒ ステップ②：データの準備（どうやってデータを持っておく？）

C++のクラスで、`Terrain`（テレイン）という「地面クラス」を作ります。
中にはこんな変数があります：

```
std::vector<Vertex> vertices_; // 点の集まり（地面の形）
std::vector<uint32_t> indices_; // 三角形のつなぎ方（番号）

int width, height; // 地面の横と縦のマスの数
float scale;       // 1マスの大きさ（例：1.0f = 1メートル）
```

このクラスで、地面を作ったり、表示したりできるようになります！

☒ ステップ③：ランダムに地形を作る（でこぼこをつけよう！）

`MakeTerrain()` という関数で、地面をランダムにでこぼこさせます。
ここでは「**サイコロのような乱数（ランスウ）**」を使って、山や谷を作ります：

```
float y = ランダムな数 * 高さの倍率;
```

そして、点を1マスずつ作っていきます：

```
for (int z = 0; z < 高さのマス数; z++) {
    for (int x = 0; x < 横のマス数; x++) {
        Vertex v;
        v.position = { xの位置, yの高さ, zの位置 };
        ...
        vertices_.push_back(v);
    }
}
```

そのあと、「マス」を三角形2枚にわけて、`indices_` に三角形を作ります。

例として、横5×縦4の頂点グリッド（幅5×高さ4）を使います。

■ 地形の頂点の並び（`vertices_`のインデックス）

```
z方向（奥行き）
↑
|
```

```
|  (0,0)  (1,0)  (2,0)  (3,0)  (4,0)
|    0      1      2      3      4
|
|  (0,1)  (1,1)  (2,1)  (3,1)  (4,1)
|    5      6      7      8      9
|
|  (0,2)  (1,2)  (2,2)  (3,2)  (4,2)
|   10     11     12     13     14
|
|  (0,3)  (1,3)  (2,3)  (3,3)  (4,3)
|   15     16     17     18     19
+---------------------------------------→ x方向（横）
```

■ 説明

- `vertices_` という動的配列には、**上から下、左から右の順番**で頂点が入っています。
- `vertices_[0]` は左上、`vertices_[19]` は右下の頂点です。
- 各頂点には `x, y, z` の位置や、`法線（normal）`、`UV` などの情報が入っています。

■ 使いどころ

この並び順は次の処理で使われます：

- `インデックス`の生成（3つで三角形を作る）
- `GetHeight(x, z)` で高さを調べる
- テクスチャのUVを計算する

☒ ステップ④：地形を画面に表示する（ゲームの絵にする！）

できあがった `vertices_` と `indices_` を「GPU（じーぴーゆー）」に送って、DirectXで描きます。

```
context->IASetVertexBuffers(...); // 頂点（点）の情報をセット
context->IASetIndexBuffer(...);  // 三角形のつなぎ方をセット
context->DrawIndexed(...);       // 実際に画面に描く！
```

描くときには、光の当たり方や、テクスチャ（地面の絵）も設定して、見た目をよくします。

☒ まとめ

ステップ	やることの意味
データの設計	地面に必要な情報を決める（点や三角形）
データの準備	地面の形を覚えるための変数を作る
ランダム地形生成	高さをランダムに決めて地形を作る
地形の表示	地形のデータをGPUに送って画面に描く

ここまでくれば、あとはQuadクラスを作ってテクスチャ張った時とほぼ同じだよ。
それが、地形の大きさで並んでるだけだと思えばいいです。

必要なクラス(Terrain class)を作っていく

☒ ステップ⑤：Terrain クラスを作ろう！

ゲームに出てくる「地面」や「山」をつくるために、`Terrain`（テレイン）という **地形クラス** を作ります。

これは「**地形を作ったり、描いたり、調べたりする便利な道具**」だと思ってください。

☒ 1. 必要な情報（メンバ変数）

地形を作るには、「点」や「三角形」、サイズなどの情報が必要です。

```
class Terrain {
public:
    Terrain() = default; // 何も設定しない初期状態のコンストラクタ

    void MakeTerrain(); // ← ランダムに地形を作る関数
    void Update();      // 地形の更新（今は何もしない）
    void Draw(Transform& t); // 地形を画面に描く関数

    void SetParams(const TerrainParams& params) { params_ = params; }

private:
    std::vector<Vertex> vertices_; // 点のリスト
    std::vector<uint32_t> indices_; // 三角形のリスト

    ID3D11Buffer* vertexBuffer_ = nullptr; // GPU用の頂点バッファ
    ID3D11Buffer* indexBuffer_ = nullptr;  // GPU用のインデックスバッファ
    ID3D11Buffer* globalCB = nullptr;      // 定数バッファ（カメラなどの情報）

    TerrainParams params_; // 地形のサイズ・スケール情報など

    void CreateBuffers();    // GPUにデータを送る関数
    void ComputeNormals();   // 法線（光の方向）を計算する関数
};
```

☒ 2. 地形の設定データ `TerrainParams`

`params_` に入れるデータはこういう構造になっています：

```
struct TerrainParams {
    int width = 128;        // 横マス数（点の数）
    int height = 128;       // 縦マス数
    float scale = 1.0f;     // 1マスの大きさ（メートル）
    float heightScale = 10.0f; // 高さの最大値
};
```

☒ 3. `Vertex` とは？

点（頂点）は `Vertex` という名前で、こう定義されています：

```
struct Vertex {
    DirectX::XMFLOAT3 position; // 座標（どこにあるか）
    DirectX::XMFLOAT3 normal;   // 法線（光の方向）
    DirectX::XMFLOAT2 uv;       // UV（テクスチャの貼る場所）
```

```
};
```

☒ 補足：DirectXに必要なもの

地形を表示するには、DirectXの以下の機能を使います：

- 頂点バッファ（点の情報）
- インデックスバッファ（三角形のつなぎ方）
- テクスチャ（見た目の模様）
- 定数バッファ（カメラやライトの情報）

☒ ここまでのまとめ

名前	役割
`Terrain`	地形を作る・表示するクラス
`Vertex`	1つの点の情報（位置など）
`TerrainParams`	地形全体のサイズやスケール
`vertices_`	点の集まり
`indices_`	三角形のつなぎ方
`Draw()`	地形を画面に表示する関数

ランダムな三角形で地形を作っていく

☒ 地形の高さをランダムに決めて、頂点を作ろう！

☒目的：

地面の形を作るには、まず「高さのある点（＝頂点）」をたくさん並べます。
そしてこの点の高さをランダムに決めることで、でこぼこした地形になります。

☒ 使うデータ

前回までに作った `TerrainParams` を使います。

```
struct TerrainParams {
    int width = 128;      // 横に何個頂点を並べるか
    int height = 128;     // 奥に何個頂点を並べるか
    float scale = 1.0f;   // 1マスの広さ（距離）
    float heightScale = 10.0f; // 高さの最大値
};
```

☒ 乱数で高さを作る

C++では乱数を使って、毎回ちがう地形にすることができます。

```
std::mt19937 rng(std::random_device{}()); // ランダムの元
std::uniform_real_distribution<float> heightDist(0.0f, 1.0f); // 0〜1の高さ
```

☒ 頂点を作るコード

ここで、地面の「点（Vertex）」を作ります。

```
for (int z = 0; z < params_.height; ++z) {
    for (int x = 0; x < params_.width; ++x) {
        float y = heightDist(rng) * params_.heightScale; // ランダムな高さ！

        Vertex v;
        v.position = {
            x * params_.scale - halfWidth,  // X座標（左から右）
            y,                              // Y座標（高さ）
            z * params_.scale - halfHeight  // Z座標（手前から奥）
        };
        v.normal = { 0, 1, 0 }; // 法線（とりあえず上）
        v.uv = {
            static_cast<float>(x) / (params_.width - 1),
            static_cast<float>(z) / (params_.height - 1)
        };

        vertices_.push_back(v); // 頂点リストに追加！
    }
}
```

☒ なにが起こってるの？

1. `for` でグリッド状に頂点を並べる
2. 1つ1つに、ランダムな高さ `y` をつける
3. 地面の中心が (0, 0, 0) に来るように位置を調整
4. 頂点データを `vertices_` に入れる

☒ 実行するとどうなる？

こんな感じの地形ができます☒（イメージ）

```
高い ■
   ■■  ■■
■■ ■■ ■■  ■■
  ■■■■■■■■ ← ランダムな高さででこぼこ
```

頂点作ったら、インデックスを考える

☒ 1. まずは四角を考えよう

たとえば、次のような 2×2 の四角形（セル）があります：

```
点の番号（vertices_ のインデックス）

↑ z方向

0———1
| ／ |
| ／ |
2———3    → x方向
```

この 4つの頂点から、2枚の三角形を作ります。

☒ 2. 三角形の作り方（インデックス配列）

1. 左下の三角形 → 点 0, 2, 1
2. 右上の三角形 → 点 2, 3, 1

※この順番（左回り／反時計回り）が**「表面」になるためのルール**です！

```
// C++ではこう書く
indices_.push_back(i0); // = 左上の頂点
indices_.push_back(i2); // = 左下の頂点
indices_.push_back(i1); // = 右上の頂点

indices_.push_back(i2); // = 左下
indices_.push_back(i3); // = 右下
indices_.push_back(i1); // = 右上
```

☒ 3. 地形全体のループでこれを繰り返す！

```
for (int z = 0; z < height - 1; ++z) {
    for (int x = 0; x < width - 1; ++x) {
        int i0 = z * width + x;       // 左上
        int i1 = i0 + 1;              // 右上
        int i2 = i0 + width;          // 左下
        int i3 = i2 + 1;              // 右下

        // 三角形①: 左上 → 左下 → 右上
        indices_.push_back(i0);
        indices_.push_back(i2);
        indices_.push_back(i1);
```

```
        // 三角形②: 左下 → 右下 → 右上
        indices_.push_back(i2);
        indices_.push_back(i3);
        indices_.push_back(i1);
    }
}
```

☒ 補足：なぜこうするの？

GPUに渡すときには、**「三角形の頂点3つ」のセット**として教える必要があるからです。
たとえば「地面を描きたい」と思ったら、こうして三角形の集合（メッシュ）として組み立てます。
順番逆にしちゃったりして、ポリゴンが表示されたりされなかったりするときは、ラスタライズステート（Direct3D.cpp）の設定をワイヤーフレーム＋カリングなし、にしてみよう。

GPUリソースを作っていく！→Draw関数

☒ GPUリソースを作るってどういうこと？

☒まずはイメージ！

- 「頂点の情報（場所や高さなど）」を作るだけでは、画面に出せません。
- 作った情報を **GPU** に渡して、「これを描いて！」とお願いしないといけません。

そのために、**「バッファ」**という入れ物を作って、GPUに渡す必要があります。

☒ GPUに送るデータは2つある！

名前	説明
頂点バッファ	点（場所・高さ・向き・UV）の情報
インデックスバッファ	どの点とどの点をつないで三角形にするか

☒ 具体的にどうやるの？

☒ 頂点バッファを作る部分（簡略）

```
D3D11_BUFFER_DESC vbDesc = {};
vbDesc.Usage = D3D11_USAGE_DEFAULT; // 普通の使い方
vbDesc.ByteWidth = sizeof(Vertex) * vertices_.size(); // 頂点のサイズぶん
vbDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER; // 頂点バッファだよ！

D3D11_SUBRESOURCE_DATA vbData = {};
vbData.pSysMem = vertices_.data(); // これが中身！

device->CreateBuffer(&vbDesc, &vbData, &vertexBuffer_);
```

☒ インデックスバッファも同じ感じ

```
D3D11_BUFFER_DESC ibDesc = {};
ibDesc.Usage = D3D11_USAGE_DEFAULT;
ibDesc.ByteWidth = sizeof(uint32_t) * indices_.size();
ibDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;

D3D11_SUBRESOURCE_DATA ibData = {};
ibData.pSysMem = indices_.data();

device->CreateBuffer(&ibDesc, &ibData, &indexBuffer_);
```

☒ わかりやすく言うと…

1. `vbDesc` や `ibDesc` に「バッファの情報（サイズとか）」を伝える
2. `vbData` や `ibData` に「実際の中身（点のデータなど）」を入れる
3. `CreateBuffer()` で「バッファを作ってGPUに渡す」！

☒ 絵にすると…

```
[CPU]                [GPU]
 vertices_ ──────────▶ 頂点バッファ ┐
                      | → 画面に表示！
 indices_  ──────────▶ 三角形順バッファ┘
```

☒ 古いバッファはちゃんと片付けよう！

```
if (vertexBuffer_) vertexBuffer_->Release();
if (indexBuffer_) indexBuffer_->Release();
```

これは「前のバッファがまだ残ってたら、片付けてから作ろうね！」というお片付けの処理です。

☒Simple3Dシェーダを使ってレンダリングする

Simple3Dシェーダへの入力に合わせた、インプットレイアウトと（頂点の構造体）と、毎フレーム変更される情報を送るためのコンスタントバッファを作ります。

☒ 1. 頂点の並び順（インプットレイアウト）

☒ そもそも「インプットレイアウト」ってなに？

GPUは「1つの頂点に何が入ってるのか」を知らないと、正しく使えません。
そこで「この順番でデータが入ってるよ！」と教えるための設定が、**インプットレイアウト**です。

☒ このコードがその設定：

```
D3D11_INPUT_ELEMENT_DESC layout[] = {
 { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },   // 座標（x, y, z）
 { "NORMAL",   0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },  // 法線（x, y, z）
 { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT,    0, 24, D3D11_INPUT_PER_VERTEX_DATA, 0 },  // UV座標（u, v）
};
```

☒ つまり1頂点はこう：

バイト位置	内容	サイズ
0〜11	Position	12バイト（`float x,y,z`）
12〜23	Normal	12バイト（`float x,y,z`）
24〜31	UV	8バイト（`float u,v`）

これを `CreateInputLayout()` でGPUに登録して、「これに従って読み込んでね」と指示します。

☒ 2. 定数バッファ（CBGlobal）

☒ 頂点シェーダに渡す「カメラや光の情報」

描画時に使いたい「世界の情報（カメラ、ライティング、マトリクスなど）」は、毎フレーム変わるので、**定数バッファ**にまとめて送ります。

☒ `CBGlobal` 構造体の中身

```
struct CBGlobal {
    XMMATRIX g_matWVP;        // モデル→ビュー→プロジェクションの行列（最終位置）
    XMMATRIX g_matNormalTrans; // 法線用の変換行列
    XMMATRIX g_matWorld;      // モデルのワールド変換
    XMFLOAT4 g_vecLightDir;   // 光の向き
```

```
    XMFLOAT4 g_vecDiffuse;    // 拡散光の色
    XMFLOAT4 g_vecAmbient;    // 環境光の色
    XMFLOAT4 g_vecSpeculer;   // 鏡面反射の色
    XMFLOAT4 g_vecCameraPosition; // カメラの位置
    float g_shuniness;        // 鏡面反射の強さ
    BOOL g_isTexture;         // テクスチャありかなしか（フラグ）
    float pad[2];             // 16バイトにそろえるためのパディング
};
```

☒ どう使われるの？

描画のときに、C++ 側で値をセットして GPU に送ります：

```
Direct3D::pContext_->UpdateSubresource(globalCB, 0, nullptr, &cb, 0, 0);
Direct3D::pContext_->VSSetConstantBuffers(0, 1, &globalCB);
Direct3D::pContext_->PSSetConstantBuffers(0, 1, &globalCB);
```

このようにすると、HLSLのシェーダー側で次のように受け取れます：

```
cbuffer CBGlobal : register(b0)
{
    float4x4 g_matWVP;
    float4x4 g_matNormalTrans;
    float4x4 g_matWorld;
    float4 g_vecLightDir;
    ...
}
```

☒ 最後にまとめると

パーツ名	役割
インプットレイアウト	頂点が「どういう順番で並んでるか」をGPUに教える
CBGlobal構造体	カメラ・光・変換マトリクスなど、描画に必要な「毎回変わる情報」をまとめる

☒ Draw関数を作って描画していく

地形（Terrain）が画面に出るようにする！
基本は、今までやったQuadクラスとかの描画と一緒。

☒ ステップで説明

☒ ① 頂点バッファとインデックスバッファをGPUに渡しておく（もうやった）

これは `CreateBuffers()` の中でやりました。
「地形の形（点と三角形）」を GPU に教えてある状態です。

☒ ② シェーダーの準備（もうやった）

- 頂点シェーダ（VS）
- ピクセルシェーダ（PS）
- 入力レイアウト（頂点データの並び方）

これは `InitShaderBundle()` で設定済みです。

🆕 ③ 定数バッファにデータを入れて送る

`CBGlobal` に、プレイヤーの位置・カメラ・光の向きなどを詰めて送ります。

```
CBGlobal cb = {};
cb.g_matWVP = ...; // カメラを使った行列を計算して代入
cb.g_vecLightDir = { 0, -1, 1, 0 }; // 斜め上から光
...
context->UpdateSubresource(globalCB, 0, nullptr, &cb, 0, 0);
context->VSSetConstantBuffers(0, 1, &globalCB);
context->PSSetConstantBuffers(0, 1, &globalCB);
```

🆕 ④ GPU に地形データをセットする

地形の「点の情報」や「三角形のつなぎ方」を GPU に渡します。

```
UINT stride = sizeof(Vertex); // 1つの頂点の大きさ
UINT offset = 0;
context->IASetVertexBuffers(0, 1, &vertexBuffer_, &stride, &offset);
context->IASetIndexBuffer(indexBuffer_, DXGI_FORMAT_R32_UINT, 0);
context->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST); // 三角形で描くよ！
```

🆕 ⑤ テクスチャをGPUに渡す（画像つきの場合）

```
ID3D11ShaderResourceView* srv = texture_->GetSRV();
context->PSSetShaderResources(0, 1, &srv);
```

🆕 ⑥ 実際に描く命令を出す！

ここで「GPUよ！描けー！」と命令します。

```
context->DrawIndexed(static_cast<UINT>(indices_.size()), 0, 0);
```

これで、GPUが全部の三角形を使って地形を画面に出します。

☒ 最終的な `Draw()` の形

```
void Terrain::Draw(Transform& t)
{
    // ① 定数バッファを埋める
    CBGlobal cb = {};
    cb.g_matWVP = ...;
    ...
    context->UpdateSubresource(globalCB, 0, nullptr, &cb, 0, 0);
    context->VSSetConstantBuffers(0, 1, &globalCB);
    context->PSSetConstantBuffers(0, 1, &globalCB);

    // ② シェーダーを使う
    Direct3D::SetShader(Direct3D::SHADER_3D);

    // ③ 頂点とインデックスを渡す
    context->IASetVertexBuffers(...);
    context->IASetIndexBuffer(...);
    context->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

    // ④ テクスチャを渡す
    context->PSSetShaderResources(0, 1, &texture_->GetSRV());

    // ⑤ 描画命令
    context->DrawIndexed(static_cast<UINT>(indices_.size()), 0, 0);
}
```

☒ まとめ

ステップ	やること	状態
① 頂点を作る	`MakeTerrain()` などで作成済み	
② バッファ作る	`CreateBuffers()` 済み	
③ シェーダーセット	`InitShaderBundle()` 済み	
④ 定数バッファに情報入れる	`Draw()` 内でやる	
⑤ 頂点・インデックス・テクスチャを渡す	`Draw()` 内でやる	
⑥ 描画命令を出す	`Draw()` 内でやる	

おまけ

☒ 目的

地形の見た目を決める「シェーダー」の中身を作る！
使うのは：

- 頂点シェーダ（VS）→ 三角形の位置を変える（カメラの向きなど）
- ピクセルシェーダ（PS）→ 色や明るさを決める（光やテクスチャ）

☒ 1. 共通で使う定数バッファ（C++と同じ構造）

```
cbuffer CBGlobal : register(b0)
{
    matrix g_matWVP;        // ワールド×ビュー×プロジェクション
    matrix g_matNormalTrans; // 法線の変換行列
    matrix g_matWorld;      // ワールド行列（モデル座標→ワールド）
    float4 g_vecLightDir;
    float4 g_vecDiffuse;
    float4 g_vecAmbient;
    float4 g_vecSpeculer;
    float4 g_vecCameraPosition;
    float g_shuniness;
    bool g_isTexture;
    float2 pad;
};
```

これは C++ 側の `CBGlobal` とペアになります。カメラや光の情報をGPUに渡すための箱です。

☒ 2. 頂点シェーダ VS

```
struct VS_IN
{
    float3 pos : POSITION;
    float3 normal : NORMAL;
    float2 uv : TEXCOORD;
};

struct VS_OUT
{
    float4 pos : SV_POSITION;
    float3 worldPos : POSITION1;
    float3 normal : NORMAL;
    float2 uv : TEXCOORD;
};

VS_OUT VS(VS_IN input)
{
    VS_OUT output;

    float4 worldPos = mul(float4(input.pos, 1.0f), g_matWorld);
```

```
    output.pos = mul(worldPos, g_matWVP); // 画面に変換
    output.worldPos = worldPos.xyz;

    // 法線ベクトルの変換（回転だけ反映）
    output.normal = normalize(mul(float4(input.normal, 0.0f), g_matNormalTrans).xyz);

    output.uv = input.uv;
    return output;
}
```

☒ 何をやってる？

入力	やってること	出力
頂点の位置	カメラ視点の座標に変換	`output.pos`（画面用）
法線	ライト計算できるように変換	`output.normal`（ライト用）
UV座標	テクスチャの模様位置を受け渡す	`output.uv`

☒ 3. ピクセルシェーダ PS

```
Texture2D tex0 : register(t0);
SamplerState smp : register(s0);

float4 PS(VS_OUT input) : SV_TARGET
{
    float3 normal = normalize(input.normal);
    float3 lightDir = normalize(-g_vecLightDir.xyz);

    // ランバート拡散
    float diff = max(dot(normal, lightDir), 0.0f);

    float3 ambient = g_vecAmbient.rgb;
    float3 diffuse = g_vecDiffuse.rgb * diff;

    // 鏡面反射（スペキュラ）
    float3 viewDir = normalize(g_vecCameraPosition.xyz - input.worldPos);
    float3 halfVec = normalize(lightDir + viewDir);
    float spec = pow(max(dot(normal, halfVec), 0.0f), g_shuniness);
    float3 specular = g_vecSpeculer.rgb * spec;

    float4 texColor = tex0.Sample(smp, input.uv);
    float3 finalColor = (ambient + diffuse + specular);

    if (g_isTexture)
        return float4(finalColor, 1.0f) * texColor;
    else
        return float4(finalColor, 1.0f);
}
```

☒ 何をしてる？

ステップ	内容
光の向きと法線の角度	明るさ（影の強さ）を計算
カメラと光の反射	ピカっと光る所（ハイライト）を計算
テクスチャと合成	模様のある色と光の色を合成する

☒ まとめ

シェーダーの流れ図

```
頂点データ（位置・法線・UV）
        ↓
  [ 頂点シェーダ (VS) ]
        ↓
VS_OUT（画面座標・法線・UVなど）
        ↓
  [ ピクセルシェーダ (PS) ]
        ↓
画面に出す最終の色（テクスチャ＋光）
```

☒ 最後に

このシェーダーは「リアルな明るさ＋テクスチャ模様」が出るようになっていて、
以下のような構成をすべて活かしています：

- カメラ行列（WVP）
- ライト方向と色
- 法線の変換と補間
- テクスチャのUV座標

地形にプレイヤー立たせて丸影

地形にプレイヤー立たせて丸影を落とします。
毎フレーム、レイキャストとかやってるとしんどいので、地形データを使って高さを補間で算出してプレイヤーを立たせます。
（よく考えるとレイキャストとあんまり計算量変わらないかな。。。）

まずは、プレイヤーをつくる

これは、いつものモデル読んでGameObject継承したPlayerクラス作る感じでいいと思う。
必要な情報はあとで追加していこう。
最低限必要なところで考えると。以下のようなクラスになりそう。

☒ 例：クラス構成の改善案（関数分割）

```
class Player : public GameObject {
public:
    void Initialize() override;
    void Update() override;
    void Draw() override;
    void Release() override;

private:
    void HandleInput();          // 入力処理まとめ
    void ApplyGroundAdjustment(); // 地面に沿って傾き・高さ補正
    void SetTPSCamera();         // TPS視点カメラ制御

    int hPlayer = -1;
    const float moveSpeed = 5.0f;
    const float rotateSpeed = 90.0f;
    XMFLOAT3 forward_ = {0, 0, -1, 1.0};
};
```

☒ ~~小学生でもわかる~~要約文（ここまでの内容）

- プレイヤーは「矢印キーで動く人形」です。
- 地面に合わせてちゃんと立つようにしてます。
- 地形の高さを調べて、プレイヤーが浮いたり埋まったりしないようにします。
- プレイヤーの後ろにカメラがくっついて、いつも後ろから見えるようにしてます。
- そのためのいろんな処理を、関数にわけてスッキリさせたいです。

☒ 1. `Initialize()`：プレイヤーの初期化

☒ 目的：

プレイヤーのモデルを読み込み、初期位置と向きを設定します。また、TPS（後ろから追いかけるカメラ）の初期位置もここでセットします。

☒ 実装と説明：

```
void Player::Initialize()
{
    hPlayer = Model::Load("Player.fbx"); // モデル読み込み
    Model::SetAnimFrame(hPlayer, 0, 42, 1.0); // アニメーション範囲設定（0〜42）

    // 初期向き（Z+方向を向くようにするために180度回転）
    transform_.rotate_ = { 0.0f, 180.0f, 0.0f };

    // 初期位置（地形の中心あたりに置く想定）
    transform_.position_ = { 0.0f, 0.0f, 0.0f };

    // 初期の前向きベクトルもZ+方向（右手系の場合、奥に向かって前進）
    forward_ = { 0.0f, 0.0f, 1.0f };

    // カメラ初期設定
    SetTPSCamera();
}
```

シャドウマップへの道（険しい）

自作のDx11ベースエンジンにシャドウマップを組み込む

その1

第1章：シャドウマップの考え方

この章の目的

コードを書く前に、なぜ「ライトから見たZ値」を比べると影がわかるのかを理解する。

この章は説明だけ。コード変更はしない。

---

1. 普通のライト計算だけでは影は出ない

今のライト計算では、だいたい次のことをしている。

```
面の向きとライト方向を比べる
↓
ライトの方を向いていれば明るい
ライトと逆を向いていれば暗い
```

これは「その面がライト方向を向いているか」を見ているだけ。
しかし、影を出すには次のことも調べる必要がある。

```
ライトとその場所の間に、別の物体があるか
```

手前に別の物体があれば、光はそこで遮られる。
その奥にある場所は影になる。

---

2. 影とは「ライトから見えない場所」

カメラから床が見えていても、ライトから見えなければ光は届かない。

```
ライト
 \
  \
   \ 距離3：ドーナツ
    \■
     \
      \ 距離7：床
       □
```

この場合、床 `□` はカメラからは見えているかもしれない。
でも、ライトから見ると距離3のドーナツ `■` に隠れている。

だから床 `□` は影になる。

---

3. シャドウマップとは何か

シャドウマップは、ライトから見たときの「一番手前のZ値」を保存した画像。

普通の画像は色を保存する。

```
赤、緑、青、透明度
```

シャドウマップは色ではなく、奥行きを保存する。

```
ライトから見て、この場所の一番手前は Z=0.35
ライトから見て、別の場所の一番手前は Z=0.70
```

つまり、シャドウマップは「ライトから見た奥行きメモ」。

---

4. どうやって影判定するか

通常描画中に、今描いているピクセルをライト視点に変換する。

例：今描いている床のピクセル

```
ライト画面上の位置：UV = (0.42, 0.61)
ライト視点Z値：0.70
```

次に、シャドウマップの同じUV位置を見る。

```
シャドウマップの UV = (0.42, 0.61)
保存されていたZ値：0.35
```

比較する。

```
今のピクセルのZ値 0.70
シャドウマップのZ値 0.35
```

今のピクセルの方が奥にある。

```
0.70 > 0.35
```

つまり、ライトから見ると、手前の `0.35` の位置に別の物体がある。
今描いている `0.70` のピクセルには光が届かない。

だから影。

---

5. 判定式

```
今描いているピクセルのライト視点Z値 > シャドウマップのZ値
  → 手前に別の物体がある
  → 光が届かない
  → 影
```

```
今描いているピクセルのライト視点Z値 <= シャドウマップのZ値
  → ライトから直接見えている
  → 光が届く
  → 影ではない
```

---

6. 要点まとめ

```
シャドウマップは、ライトから見た一番手前のZ値メモです。

通常描画中に、今描いているピクセルをライト視点に変換します。
そして、そのピクセルのZ値と、シャドウマップに保存されたZ値を比べます。

今描いているピクセルの方が奥なら、手前に別の物体があるということです。
つまり光が遮られているので、そのピクセルは影になります。
```

この章の確認ポイント

確認問題：

1. 影とは、カメラから見えない場所のことか？
2. 影とは、ライトから見えない場所のことか？
3. シャドウマップには色が保存されるのか？

4. シャドウマップには何が保存されるのか？

答え：

1. 違う
2. そう
3. 違う
4. ライトから見た一番手前のZ値

その２

第2章：ライトを仮想カメラとして扱う

この章の目的

シャドウマップを作るには、ライトから見た画面が必要になる。
そのために、ライト用のビュー行列と射影行列を追加する。

この章では、まだ画面は変わらない。

考え方

普通の描画では、カメラから見た画面を作っている。

```
カメラ位置
 ↓見る
シーン
```

シャドウマップでは、ライトから見た画面を作る。

```
ライト視点の仮想カメラ位置
 ↓見る
シーン
```

今回のライトは平行光源として扱う。
平行光源には本来「位置」はない。

しかし、シャドウマップを作るには「ライトから見た画面」が必要なので、ライト方向の反対側に仮想カメラを置く。

```
lightEye = ライト方向の反対側に離した仮想カメラ位置
lightAt  = 原点
```

ここでの原点はライト位置ではない。
仮想ライトカメラの注視点。

変更ファイル

- `Engine/Direct3D.h`
- `Engine/Direct3D.cpp`

実装指示

```
ステンシルは使わないシャドウマップ実装の第2章です。

Beforeプロジェクトを基準に、Direct3Dにライト視点用のビュー行列と射影行列を追加してください。

変更内容：
1. Engine/Direct3D.h に次の関数宣言を追加する。
   - DirectX::XMMATRIX GetLightViewMatrix();
   - DirectX::XMMATRIX GetLightProjectionMatrix();
```

2. Engine/Direct3D.cpp に上記2関数を実装する。

3. GetLightViewMatrix() は、既存の lightPosition を方向ベクトルとして使う。
  - lightDir = normalize(lightPosition)
  - lightEye = -lightDir * 10.0f
  - lightAt = 原点
  - XMMatrixLookAtLH(lightEye, lightAt, lightUp) を返す

4. lightDir がY軸にほぼ平行な場合は LookAt が壊れないように up をZ軸に切り替える。

5. GetLightProjectionMatrix() は平行光源用なので XMMatrixOrthographicLH を使う。
  - width = 20.0f
  - height = 20.0f
  - nearZ = 1.0f
  - farZ = 50.0f

既存の関数名・型名に合わせて実装し、不要な新規クラスは作らないでください。

実装イメージ

Engine/Direct3D.h

```
DirectX::XMMATRIX GetLightViewMatrix();
DirectX::XMMATRIX GetLightProjectionMatrix();
```

Engine/Direct3D.cpp

```
DirectX::XMMATRIX Direct3D::GetLightViewMatrix()
{
    XMVECTOR lightDir  = XMVector3Normalize(XMLoadFloat4(&lightPosition));
    XMVECTOR lightEye  = -lightDir * 10.0f;
    XMVECTOR lightAt   = XMVectorSet(0, 0, 0, 0);

    XMVECTOR upY = XMVectorSet(0, 1, 0, 0);
    float dotY   = fabsf(XMVectorGetX(XMVector3Dot(lightDir, upY)));
    XMVECTOR lightUp = (dotY > 0.99f) ? XMVectorSet(0, 0, 1, 0) : upY;

    return XMMatrixLookAtLH(lightEye, lightAt, lightUp);
}

DirectX::XMMATRIX Direct3D::GetLightProjectionMatrix()
{
    float width  = 20.0f;
    float height = 20.0f;
    float nearZ  = 1.0f;
    float farZ   = 50.0f;

    return XMMatrixOrthographicLH(width, height, nearZ, farZ);
}
```

この章でのコード変更点

変更の概要

ファイル	変更内容
Engine/Direct3D.h	関数宣言を2つ追加
Engine/Direct3D.cpp	関数の実装を2つ追加

画面は変わらない。まだどこからも呼ばれていないため。

Engine/Direct3D.h の変更

Before（変更前）

```
namespace Direct3D
{
    // ...既存の関数宣言...
    DirectX::XMFLOAT4 GetLightPos();
    void SetLightPos(DirectX::XMFLOAT4 pos);
};
```

After（変更後）

```
namespace Direct3D
{
    // ...既存の関数宣言...
    DirectX::XMFLOAT4 GetLightPos();
    void SetLightPos(DirectX::XMFLOAT4 pos);

    DirectX::XMMATRIX GetLightViewMatrix();       // ← 追加
    DirectX::XMMATRIX GetLightProjectionMatrix(); // ← 追加
};
```

追加した宣言の意味：

関数名	意味
`GetLightViewMatrix()`	ライトを仮想カメラとして「どこから・どこを見るか」を表す行列を返す
`GetLightProjectionMatrix()`	ライト視点の「画面の映し方（正射影）」を表す行列を返す

`Engine/Direct3D.cpp` の変更

追加した関数① `GetLightViewMatrix()`

```
XMMATRIX Direct3D::GetLightViewMatrix()
{
    // lightPosition はライト方向ベクトル（平行光源のため位置ではなく向き）
    XMVECTOR lightDir = XMVector3Normalize(XMLoadFloat4(&lightPosition));

    // ライト方向の延長線上（10倍先）に仮想カメラを置く
    XMVECTOR lightEye = -lightDir * 10.0f;

    // 仮想カメラはシーンの原点（0,0,0）を見る
    XMVECTOR lightAt = XMVectorSet(0, 0, 0, 0);

    // 通常は「上方向 = Y軸」でよい
    XMVECTOR upY = XMVectorSet(0, 1, 0, 0);
    float dotY   = fabsf(XMVectorGetX(XMVector3Dot(lightDir, upY)));

    // ライト方向がY軸とほぼ一致するとき（真上/真下）は、
    // LookAt の計算が壊れるため、上方向をZ軸に切り替える
    XMVECTOR lightUp = (dotY > 0.99f) ? XMVectorSet(0, 0, 1, 0) : upY;

    // ライト視点の View 行列を作って返す
    return XMMatrixLookAtLH(lightEye, lightAt, lightUp);
}
```

ポイント：

- `lightPosition` は平行光源なので「位置」ではなく「方向」として使う
- 方向を正規化して `-10.0f` を掛け、ライト方向の反対側に仮想カメラを置く
- Y軸に平行なとき（`dotY > 0.99f`）だけ up をZ軸にする。これをしないと `LookAt` の計算が破綻する

追加した関数② `GetLightProjectionMatrix()`

```
XMMATRIX Direct3D::GetLightProjectionMatrix()
{
    // 平行光源は遠近感がないので「正射影（Orthographic）」を使う
    // width=20, height=20 : ライトが照らす範囲（ワールド単位）
    // nearZ=1, farZ=50 ：ライト視点の手前・奥のクリップ距離
    return XMMatrixOrthographicLH(20.0f, 20.0f, 1.0f, 50.0f);
}
```

ポイント：

- 通常のカメラは `XMMatrixPerspectiveFovLH`（遠近感あり）を使う
- ライト視点は平行光源なので `XMMatrixOrthographicLH`（遠近感なし）を使う
- `width / height` はライトが影を作れる範囲。狭すぎると影が切れる

変更のイメージ図

```
【変更前】
Direct3D には、カメラ用の行列しかなかった

 SetLightPos() / GetLightPos()  ← ライトの位置を持つだけ

【変更後】
ライトを「仮想カメラ」として扱う2つの行列が追加された

 GetLightViewMatrix()       ← ライトはどこから・どこを見るか
 GetLightProjectionMatrix() ← ライト視点の映し方（正射影）

 この2つを掛け合わせると「ライト視点のVP行列」になる
 → 3章以降でシャドウマップ作成に使う
```

ビルド確認

- ビルドが通れば成功。
- 画面は変わらない。

その３

第3章：シャドウマップ用テクスチャを作る

この章の目的

ライトから見たZ値を保存するためのテクスチャを作る。
まだそのテクスチャには何も描かない。

この章でも画面は変わらない。

考え方

シャドウマップは普通の色画像ではない。

> 色を保存する画像ではなく、ライトから見たZ値を保存する画像

DirectXでは、テクスチャ本体と、その使い道を分けて考える。

```
ID3D11Texture2D
  ├── DepthStencilView   : 深度を書き込む口
  └── ShaderResourceView : シェーダーで読む口
```

同じテクスチャを、

> パス1では DSV として使う
> パス2では SRV として使う

という使い方にする。

変更ファイル

- `Engine/Direct3D.h`
- `Engine/Direct3D.cpp`

実装指示

> ステンシルは使わないシャドウマップ実装の第3章です。
>
> Direct3Dに、シャドウマップ用の深度テクスチャを作成する処理を追加してください。
>
> 変更内容：
> 1. Engine/Direct3D.h に次の関数宣言を追加する。
> - HRESULT InitShadowMap(int width, int height);
> - ID3D11ShaderResourceView* GetShadowMapSRV();
>
> 2. Engine/Direct3D.cpp の namespace Direct3D 内に、次の変数を追加する。
> - int screenWidth
> - int screenHeight
> - ID3D11Texture2D* pShadowMapTexture
> - ID3D11DepthStencilView* pShadowMapDSV
> - ID3D11ShaderResourceView* pShadowMapSRV

3. Initialize(int winW, int winH, HWND hWnd) の冒頭で screenWidth と screenHeight を保存する。

4. Initialize() 内の InitShader() 後に InitShadowMap(1024, 1024) を呼ぶ。

5. Release() に pShadowMapSRV / pShadowMapDSV / pShadowMapTexture の SAFE_RELEASE を追加する。

6. InitShadowMap() を実装する。
  - Texture2D本体は DXGI_FORMAT_R32_TYPELESS
  - BindFlags は D3D11_BIND_DEPTH_STENCIL | D3D11_BIND_SHADER_RESOURCE
  - DSV は DXGI_FORMAT_D32_FLOAT
  - SRV は DXGI_FORMAT_R32_FLOAT

7. GetShadowMapSRV() は pShadowMapSRV を返す。

既存の Direct3D の書き方に合わせ、不要な新規クラスは作らないでください。

実装の重要ポイント

テクスチャ本体

```
texDesc.Format    = DXGI_FORMAT_R32_TYPELESS;
texDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL | D3D11_BIND_SHADER_RESOURCE;
```

`TYPELESS` は「あとから用途を決める」という意味。

書き込み口 DSV

```
dsvDesc.Format = DXGI_FORMAT_D32_FLOAT;
```

深度として書き込む。

読み込み口 SRV

```
srvDesc.Format = DXGI_FORMAT_R32_FLOAT;
```

シェーダーで浮動小数として読む。

この章でのコード変更点

変更の概要

ファイル	変更内容
`Engine/Direct3D.h`	関数宣言を2つ追加
`Engine/Direct3D.cpp`	変数5つ追加・Initialize/Release修正・関数2つ追加

画面は変わらない。まだ `InitShadowMap()` は呼ばれているが、描画には使われていないため。

`Engine/Direct3D.h` の変更

Before（変更前）

```
DirectX::XMMATRIX GetLightViewMatrix();
DirectX::XMMATRIX GetLightProjectionMatrix();
```

After（変更後）

```
DirectX::XMMATRIX GetLightViewMatrix();
DirectX::XMMATRIX GetLightProjectionMatrix();

HRESULT InitShadowMap(int width, int height);    // ← 追加：シャドウマップ用テクスチャ作成
ID3D11ShaderResourceView* GetShadowMapSRV();     // ← 追加：シェーダーで読む口を返す
```

Engine/Direct3D.cpp の変更①：変数の追加

Before（変更前）

```
namespace Direct3D
{
    // ...
    SHADER_BUNDLE shaderBundle[SHADER_MAX];
    XMFLOAT4 lightPosition{ 0.5f, -1.0f, 0.7f, 0.0f };
}
```

After（変更後）

```
namespace Direct3D
{
    // ...
    SHADER_BUNDLE shaderBundle[SHADER_MAX];
    XMFLOAT4 lightPosition{ 0.5f, -1.0f, 0.7f, 0.0f };

    int screenWidth  = 0;  // ← 追加：画面幅（EndShadowPassでビューポートを戻すために使う）
    int screenHeight = 0;  // ← 追加：画面高さ

    ID3D11Texture2D*          pShadowMapTexture = nullptr;  // ← 追加：深度テクスチャ本体
    ID3D11DepthStencilView*   pShadowMapDSV     = nullptr;  // ← 追加：書き込み口（パス1用）
    ID3D11ShaderResourceView* pShadowMapSRV     = nullptr;  // ← 追加：読み込み口（パス2用）
}
```

Engine/Direct3D.cpp の変更②：Initialize() への追加

Before（変更前）

```
HRESULT Direct3D::Initialize(int winW, int winH, HWND hWnd)
{
    // ...
    hr = InitShader();
    if (FAILED(hr)) return hr;

    return S_OK;
}
```

After（変更後）

```
HRESULT Direct3D::Initialize(int winW, int winH, HWND hWnd)
{
    screenWidth  = winW;  // ← 追加：画面サイズを保存
    screenHeight = winH;  // ← 追加

    // ...
    hr = InitShader();
    if (FAILED(hr)) return hr;

    hr = InitShadowMap(1024, 1024);  // ← 追加：シャドウマップ用テクスチャを1024x1024で作成
    if (FAILED(hr)) return hr;

    return S_OK;
```

```
}
```

1024×1024 の意味：
シャドウマップの解像度。大きいほど影が細かくなるが、メモリを多く使う。
この教材では 1024 を基準にする。

Engine/Direct3D.cpp の変更③：Release() への追加

Before（変更前）

```
void Direct3D::Release()
{
    // ...
    SAFE_RELEASE(pRenderTargetView);
}
```

After（変更後）

```
void Direct3D::Release()
{
    // ...
    SAFE_RELEASE(pShadowMapSRV);     // ← 追加
    SAFE_RELEASE(pShadowMapDSV);     // ← 追加
    SAFE_RELEASE(pShadowMapTexture); // ← 追加
    SAFE_RELEASE(pRenderTargetView);
}
```

解放の順番： SRV → DSV → Texture の順。使う側から先に解放する。

Engine/Direct3D.cpp の変更④：InitShadowMap() の実装（最重要）

```
HRESULT Direct3D::InitShadowMap(int width, int height)
{
    HRESULT hr;

    // ========== ① テクスチャ本体を作る ==========
    D3D11_TEXTURE2D_DESC texDesc = {};
    texDesc.Width          = width;
    texDesc.Height         = height;
    texDesc.MipLevels      = 1;
    texDesc.ArraySize      = 1;
    texDesc.Format         = DXGI_FORMAT_R32_TYPELESS;  // ← あとから用途を決める
    texDesc.SampleDesc     = { 1, 0 };
    texDesc.Usage          = D3D11_USAGE_DEFAULT;
    texDesc.BindFlags      = D3D11_BIND_DEPTH_STENCIL | D3D11_BIND_SHADER_RESOURCE; // ← 2つの口をつける
    texDesc.CPUAccessFlags = 0;
    texDesc.MiscFlags      = 0;

    hr = pDevice->CreateTexture2D(&texDesc, nullptr, &pShadowMapTexture);
    if (FAILED(hr)) { MessageBox(nullptr, L"ShadowMap Texture の作成に失敗しました", L"エラー", MB_OK); return hr; }

    // ========== ② 書き込み口（DSV）を作る ==========
    // パス1でライト視点から深度を書き込む口
    D3D11_DEPTH_STENCIL_VIEW_DESC dsvDesc = {};
    dsvDesc.Format             = DXGI_FORMAT_D32_FLOAT;  // ← 深度として書き込む
    dsvDesc.ViewDimension      = D3D11_DSV_DIMENSION_TEXTURE2D;
    dsvDesc.Texture2D.MipSlice = 0;

    hr = pDevice->CreateDepthStencilView(pShadowMapTexture, &dsvDesc, &pShadowMapDSV);
    if (FAILED(hr)) { MessageBox(nullptr, L"ShadowMap DSV の作成に失敗しました", L"エラー", MB_OK); return hr; }

    // ========== ③ 読み込み口（SRV）を作る ==========
    // パス2でシェーダーがサンプリングする口
    D3D11_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
```

```
    srvDesc.Format                    = DXGI_FORMAT_R32_FLOAT;  // ← 浮動小数として読む
    srvDesc.ViewDimension             = D3D11_SRV_DIMENSION_TEXTURE2D;
    srvDesc.Texture2D.MostDetailedMip = 0;
    srvDesc.Texture2D.MipLevels       = 1;

    hr = pDevice->CreateShaderResourceView(pShadowMapTexture, &srvDesc, &pShadowMapSRV);
    if (FAILED(hr)) { MessageBox(nullptr, L"ShadowMap SRV の作成に失敗しました", L"エラー", MB_OK); return hr; }

    return S_OK;
}
```

なぜ TYPELESS か：

```
DXGI_FORMAT_D32_FLOAT は DSV 専用で、SRV には使えない。
DXGI_FORMAT_R32_FLOAT は SRV 専用で、DSV には使えない。

→ どちらにも使えるように、テクスチャ本体を「用途未定（TYPELESS）」にしておく。
  用途は DSV と SRV それぞれを作るときに決める。
```

BindFlags を2つ立てる意味：

```
D3D11_BIND_DEPTH_STENCIL | D3D11_BIND_SHADER_RESOURCE
```

```
「このテクスチャは DSV としても SRV としても使いますよ」と DirectX に伝える。
片方しか指定しないと、もう片方のビューが作れない。
```

変更のイメージ図

```
【変更前】
通常の深度バッファしかない

  pDepthStencil     : 画面描画用の深度テクスチャ
  pDepthStencilView : 画面描画用の DSV

【変更後】
シャドウマップ専用の深度テクスチャが追加された

  pDepthStencil     : 画面描画用の深度テクスチャ（変更なし）
  pDepthStencilView : 画面描画用の DSV（変更なし）

  pShadowMapTexture : シャドウマップ用の深度テクスチャ（追加）
  pShadowMapDSV     : パス1で深度を書き込む口（追加）
  pShadowMapSRV     : パス2でシェーダーが読む口（追加）
```

ビルド確認

- ビルドが通れば成功。
- 画面は変わらない。

その４

第4章：深度だけを書く ShadowMap.hlsl を作る

この章の目的

ライト視点で深度だけを書くための専用シェーダーを作る。
通常描画用の `Simple3D.hlsl` とは別に、`ShadowMap.hlsl` を追加する。

この章でも画面は変わらない。

考え方

通常描画では、色・テクスチャ・ライト計算を使う。

しかしシャドウマップ作成では、必要なのはライトから見たZ値だけ。

```
必要：頂点をライト視点に変換する
不要：色、テクスチャ、ライティング
```

ピクセルシェーダーは何もしなくてよい。
`SV_POSITION` のZ値が、自動的に深度バッファへ書き込まれる。

変更ファイル

- `ShadowMap.hlsl` 新規追加
- `Engine/Direct3D.h`
- `Engine/Direct3D.cpp`
- `MyFirstGame.vcxproj` 必要に応じて追加

実装指示

```
ステンシルは使わないシャドウマップ実装の第4章です。

シャドウマップ作成用の専用HLSLと、その初期化処理を追加してください。

変更内容：
1. プロジェクトルートに ShadowMap.hlsl を新規作成する。
  - cbuffer cbShadow : register(b0) に row_major float4x4 matLightWVP を持たせる。
  - VS は POSITION を受け取り、mul(pos, matLightWVP) を SV_POSITION として返す。
  - PS は void でよい。色は出さない。

2. Engine/Direct3D.h の SHADER_TYPE に SHADER_SHADOWMAP を追加する。
  - SHADER_MAX の前に追加する。

3. Engine/Direct3D.h に次の関数宣言を追加する。
  - HRESULT InitShadowShader();
  - void BeginShadowPass();
  - void EndShadowPass();

4. Engine/Direct3D.cpp の InitShader() から InitShadowShader() を呼ぶ。
```

5. InitShadowShader() を実装する。
   - ShadowMap.hlsl の VS / PS をコンパイルする。
   - InputLayout は POSITION のみ。
   - ラスタライザーは D3D11_CULL_NONE にする。

6. BeginShadowPass() を実装する。
   - pShadowMapDSV を ClearDepthStencilView で 1.0f にクリアする。
   - OMSetRenderTargets で RTV を nullptr、DSV を pShadowMapDSV にする。
   - Viewport をシャドウマップサイズにする。
   - SetShader(SHADER_SHADOWMAP) を呼ぶ。

7. EndShadowPass() を実装する。
   - OMSetRenderTargets を通常の pRenderTargetView / pDepthStencilView に戻す。
   - Viewport を screenWidth / screenHeight に戻す。

ステンシル処理は追加しないでください。

`ShadowMap.hlsl` の内容

```
cbuffer cbShadow : register(b0)
{
    row_major float4x4 matLightWVP;
};

float4 VS(float4 pos : POSITION) : SV_POSITION
{
    return mul(pos, matLightWVP);
}

void PS(float4 pos : SV_POSITION)
{
    // 何もしない。
    // GPU が SV_POSITION の Z 値を深度バッファに書き込む。
}
```

注意点

InputLayoutはPOSITIONだけ

シャドウマップではUVや法線は使わない。

```
D3D11_INPUT_ELEMENT_DESC layout[] = {
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
      D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
```

色を書かない

`BeginShadowPass()` ではレンダーターゲットビューを `nullptr` にする。

```
ID3D11RenderTargetView* nullRTV = nullptr;
pContext->OMSetRenderTargets(1, &nullRTV, pShadowMapDSV);
```

ビルド確認

- ビルドが通れば成功。
- 画面は変わらない。

その５

第5章：モデルをシャドウマップに描けるようにする

この章の目的

通常描画用の `Draw()` とは別に、シャドウマップ作成用の `DrawShadow()` を追加する。

この章でも画面は変わらない。

考え方

普通の `Draw()` は、画面に表示するための描画。

```
Draw()
  色を出す
  テクスチャを読む
  ライト計算をする
  カメラ視点で描く
```

シャドウマップ用の描画では、色はいらない。

```
DrawShadow()
  色を出さない
  テクスチャを読まない
  ライト計算もしない
  ライト視点のWVPだけ使う
```

変更ファイル

- `Engine/Fbx.h`
- `Engine/Fbx.cpp`
- `Engine/Model.h`
- `Engine/Model.cpp`

実装指示

```
ステンシルは使わないシャドウマップ実装の第5章です。

FbxとModelに、シャドウマップ生成用の DrawShadow を追加してください。

変更内容：
1. Engine/Fbx.h に void DrawShadow(Transform& transform); を追加する。

2. Engine/Fbx.h の private に、シャドウ用コンスタントバッファ構造体を追加する。
   - struct CB_SHADOW { XMMATRIX matLightWVP; };

3. Engine/Fbx.h に ID3D11Buffer* pShadowConstantBuffer_; を追加する。

4. Engine/Fbx.cpp の Fbx コンストラクタで pShadowConstantBuffer_ を nullptr 初期化する。
```

5. Engine/Fbx.cpp の InitConstantBuffer() で、pShadowConstantBuffer_ を作成する。
   - ByteWidth = sizeof(CB_SHADOW)
   - Usage = D3D11_USAGE_DYNAMIC
   - BindFlags = D3D11_BIND_CONSTANT_BUFFER
   - CPUAccessFlags = D3D11_CPU_ACCESS_WRITE

6. Engine/Fbx.cpp に DrawShadow(Transform& transform) を実装する。
   - transform.Calculation() を呼ぶ。
   - 頂点バッファをセットする。
   - matLightWVP = World * Direct3D::GetLightViewMatrix() * Direct3D::GetLightProjectionMatrix() を計算する。
   - pShadowConstantBuffer_ に CB_SHADOW を Map / memcpy_s / Unmap で送る。
   - VSSetConstantBuffers(0, 1, &pShadowConstantBuffer_) でセットする。
   - マテリアルごとのインデックスバッファをセットして DrawIndexed する。
   - テクスチャやマテリアル色は使わない。

7. Engine/Model.h に void DrawShadow(int hModel); を追加する。

8. Engine/Model.cpp に Model::DrawShadow(int hModel) を追加し、内部で Fbx::DrawShadow を呼ぶ。

既存の Draw() を壊さないでください。

実装の中心

```
XMMATRIX matLightWVP = transform.GetWorldMatrix()
                     * Direct3D::GetLightViewMatrix()
                     * Direct3D::GetLightProjectionMatrix();
```

これは、モデルの頂点をライト視点に変換するための行列。

```
モデル座標
↓ World
ワールド座標
↓ Light View
ライトから見た座標
↓ Light Projection
ライト画面上の座標
```

この章でのコード変更点

この章でやること

シャドウマップにモデルを描くための関数 `DrawShadow()` を追加する。
既存の `Draw()` は一切変更しない。新しい関数を追加するだけ。

変更の概要

ファイル	変更内容
`Engine/Fbx.h`	`DrawShadow()` 宣言・`CB_SHADOW` 構造体・`pShadowConstantBuffer_` 追加
`Engine/Fbx.cpp`	コンストラクタ初期化・`InitConstantBuffer()` 修正・`DrawShadow()` 実装
`Engine/Model.h`	`DrawShadow(int hModel)` 宣言追加
`Engine/Model.cpp`	`DrawShadow()` 実装

画面は変わらない。まだ `DrawShadow()` はどこからも呼ばれていないため。

`Engine/Fbx.h` の変更

ここでやること

`DrawShadow()` 関数の宣言と、それに必要なバッファの定義を追加する。

`CB_SHADOW` 構造体をなぜ別に作るか

既存の `CONSTANT_BUFFER` はこれだけのデータを持っている：

```
struct CONSTANT_BUFFER
{
    XMMATRIX matWVP;     // カメラ視点のWVP
    XMMATRIX matWorld;   // ワールド行列
    XMMATRIX matNormal;  // 法線変換行列
    XMFLOAT4 diffuse;    // 色
    // ...（たくさん）
};
```

シャドウマップでは**ライト視点のWVPしか要らない**。
色も法線も何もいらない。

```
struct CB_SHADOW
{
    XMMATRIX matLightWVP;  // これだけ
};
```

色やテクスチャを送らない分、軽くて速い。

`pShadowConstantBuffer_` をなぜ別に持つか

`pConstantBuffer_`（通常描画用）を使い回せばいいと思うかもしれないが、それはできない。

```
理由：バッファのサイズが違う

pConstantBuffer_     → sizeof(CONSTANT_BUFFER)  大きい
pShadowConstantBuffer_ → sizeof(CB_SHADOW)      小さい

サイズが違うバッファを使い回すと、シェーダー側が期待するデータ配置と合わなくなる。
```

だから**シャドウ用のバッファを別に作る**。

After（変更後）

```
class Fbx
{
public:
    // ...
    void DrawShadow(Transform& transform);  // ← 追加

private:
    // ...
    struct CB_SHADOW
    {
        XMMATRIX matLightWVP;  // ← 追加：ライト視点のWVP行列
    };

    ID3D11Buffer* pShadowConstantBuffer_;  // ← 追加：シャドウ用バッファ
};
```

`Engine/Fbx.cpp` の変更

ここでやること

① コンストラクタで `pShadowConstantBuffer_` を nullptr 初期化する。
② `InitConstantBuffer()` でシャドウ用バッファを作成する。
③ `DrawShadow()` を実装する。

① コンストラクタの初期化

```
Fbx::Fbx()
{
    // ...既存の初期化...
    pShadowConstantBuffer_ = nullptr;  // ← 追加
}
```

② `InitConstantBuffer()` への追加

```
// シャドウ用コンスタントバッファを作成する
D3D11_BUFFER_DESC cbd = {};
cbd.ByteWidth      = sizeof(CB_SHADOW);          // CB_SHADOWのサイズ
cbd.Usage          = D3D11_USAGE_DYNAMIC;        // CPUから毎フレーム書き換える
cbd.BindFlags      = D3D11_BIND_CONSTANT_BUFFER;  // コンスタントバッファとして使う
cbd.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;     // CPUが書き込める
Direct3D::pDevice->CreateBuffer(&cbd, nullptr, &pShadowConstantBuffer_);
```

③ `DrawShadow()` の実装

```
void Fbx::DrawShadow(Transform& transform)
{
    // ワールド行列を計算する
    transform.Calculation();

    // 頂点バッファをセットする（通常のDrawと同じ）
    UINT stride = sizeof(VERTEX);
    UINT offset = 0;
    Direct3D::pContext->IASetVertexBuffers(0, 1, &pVertexBuffer_, &stride, &offset);

    // ライト視点のWVP行列を作る
    // モデル座標 → ワールド → ライト視点 → ライト画面
    XMMATRIX matLightWVP = transform.GetWorldMatrix()
                         * Direct3D::GetLightViewMatrix()
                         * Direct3D::GetLightProjectionMatrix();

    // CB_SHADOW にデータを詰めてGPUに送る
    CB_SHADOW cb;
    cb.matLightWVP = matLightWVP;

    // Map：GPUのバッファをCPUから書き込めるように開く
    D3D11_MAPPED_SUBRESOURCE pdata;
    Direct3D::pContext->Map(pShadowConstantBuffer_, 0, D3D11_MAP_WRITE_DISCARD, 0, &pdata);
    // memcpy_s：データをコピーする
    memcpy_s(pdata.pData, pdata.RowPitch, &cb, sizeof(cb));
    // Unmap：書き込みを終了してGPUに返す
    Direct3D::pContext->Unmap(pShadowConstantBuffer_, 0);

    // バッファを頂点シェーダーの b0 スロットにセット
    Direct3D::pContext->VSSetConstantBuffers(0, 1, &pShadowConstantBuffer_);

    // マテリアルごとに描画する（色・テクスチャは使わない）
    for (int i = 0; i < materialCount_; i++)
    {
        Direct3D::pContext->IASetIndexBuffer(pIndexBuffer_[i], DXGI_FORMAT_R32_UINT, 0);
        Direct3D::pContext->DrawIndexed(indexCount_[i], 0, 0);
    }
}
```

`Engine/Model.h` / `Engine/Model.cpp` の変更

ここでやること

`Fbx::DrawShadow()` を外から呼べるように `Model::DrawShadow()` を追加する。

```
// Model.h に追加
void DrawShadow(int hModel);

// Model.cpp に追加
void Model::DrawShadow(int hModel)
{
    modelList[hModel]->pfbx_->DrawShadow(modelList[hModel]->transform_);
}
```

`Stage.cpp` からは `Model::DrawShadow(hDonut_)` のように呼ぶ（6章で追加）。

ビルド確認

- ビルドが通れば成功。
- 画面は変わらない。
- まだ `DrawShadow()` は呼ばれていないため、見た目に変化はない。

その６

第6章：描画を2パス構成にする

この章でやること

`Stage::Draw()` の描画を2回に分けます。

```
パス1：ライト視点で深度だけ描く（シャドウパス）
パス2：カメラ視点で普通に描く（メインパス）
```

あわせて、第7章でシェーダーがライトVP行列を使えるよう、
`CONSTANTBUFFER_STAGE` に `matLightVP` を追加して毎フレーム送ります。

この章が終わっても**画面の見た目は変わりません。**
影が出るのは第7章からです。

この章でのコード変更点

ファイル	変更内容
`Stage.h`	`CONSTANTBUFFER_STAGE` に `matLightVP` を追加
`Stage.cpp`	`Update()` でライトVP行列を計算してCBに入れる
`Stage.cpp`	`Draw()` を2パス構成に変える

---

① `matLightVP` を追加する

ここでは何をするか

`CONSTANTBUFFER_STAGE`（シェーダーの `cbuffer gStage` に対応）に、
ライト視点のVP行列 `matLightVP` を追加します。

なぜ追加するのか

第7章でシェーダーが影を判定するとき、「このピクセルはシャドウマップ上のどこにあるか」を計算するためにライトVP行列が必要です。
今章ではまだ使いませんが、**データだけ先に送れる状態にしておきます。**

```
// Before（Stage.h）
struct CONSTANTBUFFER_STAGE
{
    XMFLOAT4 lightPosition;
    XMFLOAT4 eyePosition;
    int      lightType;
    XMFLOAT3 _pad;
    // matLightVP がない
};

// After（Stage.h）
struct CONSTANTBUFFER_STAGE
{
    XMFLOAT4   lightPosition;
    XMFLOAT4   eyePosition;
    int        lightType;
```

```
    XMFLOAT3   _pad;
    XMFLOAT4X4 matLightVP;  // ← 追加
};
```

---

② Update() でライトVP行列を計算する

ここでは何をするか

毎フレーム、ライトのビュー行列とプロジェクション行列を掛け合わせて `matLightVP` を作り、コンスタントバッファに入れます。

```
// After（Stage.cpp の Update() 内、cbに値をセットする部分）
XMMATRIX lightV  = Direct3D::GetLightViewMatrix();
XMMATRIX lightP  = Direct3D::GetLightProjectionMatrix();
XMMATRIX lightVP = lightV * lightP;
XMStoreFloat4x4(&cb.matLightVP, lightVP);
// row_major 指定のため、転置（XMMatrixTranspose）は不要
```

---

③ Draw() を2パスに分ける

ここでは何をするか

今まで1回だった `Draw()` を、シャドウパスとメインパスの2回に分けます。

なぜ部屋（hRoom_）はシャドウパスに入れないのか

部屋の外壁がライトを遮ってしまい、室内全体が影になる可能性があるためです。
影を落とすのはドーナツだけにします。

```
影を落とすもの：ドーナツ（hDonut_）
影を受けるもの：部屋・床（hRoom_）
```

```
// Before（Stage.cpp）
void Stage::Draw()
{
    Model::Draw(hball_);
    Model::Draw(hRoom_);
    Model::Draw(hDonut_);
}

// After（Stage.cpp）
void Stage::Draw()
{
    // ===== パス1：シャドウパス =====
    // ライト視点でドーナツの深度だけ書く
    Direct3D::BeginShadowPass();
    Model::DrawShadow(hDonut_);
    Direct3D::EndShadowPass();

    // ===== パス2：メインパス =====
    // 普通にカメラ視点で全部描く
    // hball_ はライト位置を確認するための表示用モデル
    Model::Draw(hball_);
    Model::Draw(hRoom_);
    Model::Draw(hDonut_);
}
```

考え方

今までは1フレームに1回だけ描いていた。

```
カメラから見て描く
```

シャドウマップでは、先にライトから見たZ値を作る必要がある。

> 1回目：ライトから見て、深度（Z値）だけ保存する
> 2回目：カメラから見て、普通に描く

この「1フレームに2回描く」ことを**2パス描画**という。

ビルド確認

- ビルドが通れば成功。
- 画面はほぼ変わらない。
- まだ影が出なくても正常。

その７

第7章：Simple3D.hlslで深度比較して影を出す

この章でやること

Simple3D.hlsl のピクセルシェーダーで、シャドウマップを読んで影を判定します。

この章で初めて画面に影が出ます。 ただし影のエッジはジャギー（ギザギザ）になります。 なめらかにするのは第8章で行います。

この章でのコード変更点

ファイル	変更内容
Stage.cpp	メインパスの前後にシャドウマップSRVをセット／解除
Simple3D.hlsl	g_shadowMap・g_shadowSampler の宣言を追加
Simple3D.hlsl	cbuffer gStage に matLightVP を追加
Simple3D.hlsl	PS() に影判定コードを追加

---

1 Stage.cpp：シャドウマップSRVをセットする

ここでは何をするか

メインパスの描画前に、第3章で作ったシャドウマップのテクスチャをピクセルシェーダーの t1 に渡します。 描画後は必ず解除します。

なぜ解除が必要か

解除しないと次のフレームで同じテクスチャを書き込み用（DSV）と読み込み用（SRV）の両方にバインドしようとして DirectX が警告を出し、正常に動作しなくなります。

Before（Stage.cpp の Draw() メインパス部分）

```
Model::Draw(hball_);
Model::Draw(hRoom_);
Model::Draw(hDonut_);
```

After

```
// シャドウマップを t1 にセット
ID3D11ShaderResourceView* pShadowSRV = Direct3D::GetShadowMapSRV();
Direct3D::pContext->PSSetShaderResources(1, 1, &pShadowSRV);

Model::Draw(hball_);
Model::Draw(hRoom_);
Model::Draw(hDonut_);

// 描画後は必ず解除する
ID3D11ShaderResourceView* nullSRV = nullptr;
Direct3D::pContext->PSSetShaderResources(1, 1, &nullSRV);
```

---

2 Simple3D.hlsl：宣言を追加する

ここでは何をするか

シャドウマップを読むためのテクスチャ変数とサンプラーを宣言します。 あわせて cbuffer gStage に matLightVP を追加します。
C++ 側は第6章ですでに送っています。シェーダー側の受け口を追加するだけです。

Before（宣言部）

```
Texture2D    g_texture : register(t0);
SamplerState g_sampler : register(s0);
```

After

```
Texture2D    g_texture       : register(t0);
SamplerState g_sampler       : register(s0);
Texture2D    g_shadowMap     : register(t1);
SamplerState g_shadowSampler : register(s1);
```

Before（cbuffer gStage）

```
cbuffer gStage : register(b1)
{
    float4 lightPosition;
    float4 eyePosition;
    int    lightType;
    float3 _pad;
};
```

After

```
cbuffer gStage : register(b1)
{
    float4            lightPosition;
    float4            eyePosition;
    int               lightType;
    float3            _pad;
    row_major float4x4 matLightVP;
};
```

---

3 Simple3D.hlsl：PS() に影判定を追加する

ここでは何をするか

今描いているピクセルが「ライトから見えているか（明るい）」「見えていないか（影）」を判定して、色に掛けます。

判定の手順

1. ワールド座標をライト視点のクリップ座標に変換する
2. クリップ座標をシャドウマップのUV座標に変換する
3. シャドウマップから「ライトに一番近いZ値」を読む
4. 今のピクセルのZ値と比べる 今のピクセルの方が奥 → 手前に何かある → 影

After（PS() の return color の直前に追加）

```
float shadow = 1.0;

float4 lightClipPos = mul(inData.wpos, matLightVP);

float2 shadowUV;
shadowUV.x =  lightClipPos.x / lightClipPos.w * 0.5 + 0.5;
shadowUV.y = -lightClipPos.y / lightClipPos.w * 0.5 + 0.5;

if (shadowUV.x >= 0.0 && shadowUV.x <= 1.0 &&
    shadowUV.y >= 0.0 && shadowUV.y <= 1.0)
```

```
{
    float currentDepth = lightClipPos.z / lightClipPos.w;
    float bias = 0.015;

    float shadowDepth = g_shadowMap.Sample(g_shadowSampler, shadowUV).r;
    shadow = (currentDepth - bias <= shadowDepth) ? 1.0 : 0.0;
}

color *= (0.3 + 0.7 * shadow);
```

bias（バイアス）とは

シャドウマップのZ値には浮動小数点の誤差があります。 バイアスなしだと、自分自身の表面が「自分より奥にある」と誤判定されてノイズのような影が全面に出ます（セルフシャドウ）。 bias = 0.015 は「少しだけ手前のものは影にしない」という調整値です。

ビルド確認

- ビルドが通る
- ドーナツが床や部屋に影を落とす
- WASD / 上下キーでライト方向を変えると影も動く
- 影のエッジがジャギー（ギザギザ）になる → 第8章で改善する

その８

第8章：比較サンプラーで影のエッジをなめらかにする

この章でやること

影のエッジのジャギー（ギザギザ）を改善します。 第7章では Sample() で1点だけ読んで影を判定していました。 この章では SampleCmpLevelZero() に差し替えて、近隣の複数点を平均した判定にします。

この章が終わると影のエッジが少しなめらかになります。

この章でのコード変更点

ファイル	変更内容
Simple3D.hlsl	SamplerState を SamplerComparisonState に変更（1行）
Simple3D.hlsl	Sample() を SampleCmpLevelZero() に変更（2行）
Stage.cpp	Initialize() に比較サンプラーの作成・s1 へのセットを追加

---

1 なぜ比較サンプラーを使うとなめらかになるか

ここでは何をするか

サンプラーの種類を「普通のサンプラー」から「比較サンプラー」に変えます。

違い

```
Sample()           : 影 or 明るい（0か1）      → ジャギー
SampleCmpLevelZero() : 0.0〜1.0 の間の値で返す   → なめらか
```

SampleCmpLevelZero() は近隣の複数点を読んで比較し、その平均を返します（PCFフィルタリング）。

---

2 Simple3D.hlsl の変更

ここでは何をするか

サンプラーの型を変えて、影判定の命令を差し替えます。

Before（宣言部）

```
SamplerState g_shadowSampler : register(s1);
```

After

```
SamplerComparisonState g_shadowSampler : register(s1);
```

Before（PS() 内の影判定）

```
float shadowDepth = g_shadowMap.Sample(g_shadowSampler, shadowUV).r;
```

```
shadow = (currentDepth - bias <= shadowDepth) ? 1.0 : 0.0;
```

After

```
shadow = g_shadowMap.SampleCmpLevelZero(g_shadowSampler, shadowUV, currentDepth - bias);
// 比較まで自動でやってくれる。自分で if を書かなくてよい。
```

---

3 Stage.cpp：比較サンプラーを作成して s1 にセットする

ここでは何をするか

Initialize() に比較サンプラーの作成と PSSetSamplers の呼び出しを追加します。 シェーダー側が SamplerComparisonState を使うようになったので、 C++ 側も対応したサンプラーを渡す必要があります。

Before（Initialize() にサンプラー作成コードがない）

After

```
D3D11_SAMPLER_DESC sd = {};
sd.Filter         = D3D11_FILTER_COMPARISON_MIN_MAG_LINEAR_MIP_POINT;
sd.AddressU       = D3D11_TEXTURE_ADDRESS_BORDER;
sd.AddressV       = D3D11_TEXTURE_ADDRESS_BORDER;
sd.AddressW       = D3D11_TEXTURE_ADDRESS_BORDER;
sd.BorderColor[0] = 1.0f;  // 範囲外は影なし（明るい）扱い
sd.BorderColor[1] = 1.0f;
sd.BorderColor[2] = 1.0f;
sd.BorderColor[3] = 1.0f;
sd.ComparisonFunc = D3D11_COMPARISON_LESS_EQUAL;
ID3D11SamplerState* pShadowSampler = nullptr;
Direct3D::pDevice->CreateSamplerState(&sd, &pShadowSampler);
Direct3D::pContext->PSSetSamplers(1, 1, &pShadowSampler);
SAFE_RELEASE(pShadowSampler);
```

BorderColor を 1.0f にしている理由： シャドウマップの範囲外に出たときに「ライトから見える（影なし）」と判定させるためです。

ビルド確認

- ビルドが通る
- 影のエッジが第7章よりなめらかになる
- ライト方向を変えても影がなめらかについてくる

その９（おまけ）

第9章：影が出ないときの確認ポイント

目的

つまずきやすいところを、症状別に確認できるようにする。

---

1. 影がまったく出ない

確認すること

- `Stage::Draw()` で `Direct3D::BeginShadowPass()` を呼んでいるか。
- `Model::DrawShadow(hDonut_)` を呼んでいるか。
- `Direct3D::EndShadowPass()` を呼んでいるか。
- `Stage::Draw()` のメインパス前に `PSSetShaderResources(1, 1, &pShadowSRV)` を呼んでいるか。
- `Simple3D.hlsl` に `g_shadowMap : register(t1)` があるか。
- `Simple3D.hlsl` に `g_shadowSampler : register(s1)` があるか。
- `Stage::Update()` で `matLightVP` を送っているか。

---

2. 画面全体が暗くなる

原因候補

`hRoom_` を `DrawShadow()` している可能性が高い。

理由

部屋全体をシャドウパスに描くと、部屋の壁や天井がライトを遮って、室内全体が影になることがある。

対応

最初は、シャドウパスにはドーナツだけを描く。

```
Model::DrawShadow(hDonut_);
// Model::DrawShadow(hRoom_); は呼ばない
```

---

3. ドーナツの表面がザラザラする

原因候補

自分自身に影を落としている。
これをシャドウアクネという。

対応

`Simple3D.hlsl` の bias を大きくする。

この教材では現在この値を使っている。

```
float bias = 0.015;
```

それでも出る場合は少しずつ大きくする。

```
float bias = 0.02;
```

ただし、大きくしすぎると影が浮いたり消えたりする。

---

4. 影が消える、または薄すぎる

確認すること

- `bias` が大きすぎないか。
- `GetLightProjectionMatrix()` の正射影サイズが広すぎないか。
- シャドウマップ解像度が低すぎないか。

この教材ではまず次の値を使う。

```
XMMatrixOrthographicLH(20.0f, 20.0f, 1.0f, 50.0f);
```

---

5. 影の位置がライト方向と合わない

確認すること

- `GetLightViewMatrix()` の `lightEye` の向き。
- `Simple3D.hlsl` のライト方向 `lightPosition` の扱い。
- `matLightVP = lightV * lightP` の順番。
- `DrawShadow()` の `matLightWVP = World * LightView * LightProjection` の順番。

今回の配布プログラムでは、`lightPosition` を「光が来る方向ベクトル」として扱うため、`lightEye` は `-lightDir * 10.0f` とする。

```
XMVECTOR lightDir = XMVector3Normalize(XMLoadFloat4(&lightPosition));
XMVECTOR lightEye = -lightDir * 10.0f;
```

---

6. 次フレームで警告や描画崩れが出る

原因候補

シャドウマップをSRVにセットしたまま、次のフレームでDSVとして使おうとしている。

対応

メインパスの後で必ずSRVを解除する。

```
ID3D11ShaderResourceView* nullSRV = nullptr;
Direct3D::pContext->PSSetShaderResources(1, 1, &nullSRV);
```

---

7. この教材で使わないもの

この教材ではステンシルは使わない。

```
ステンシルで影の領域を作る
```

のではなく、

```
Simple3D.hlsl のピクセルシェーダーで深度比較する
```

方式に統一する。

DirectXTKを使ったオーディオマネジメント

DirectXTKを使ったオーディオマネジメント
XAudio2を楽に使うためのライブラリを使って、ちょろっとSEとBGMを管理するマネージャ的なものを作ってみよう。の会

DirectXTKとAudio

Step 00 DirectXTK Audio の基本

この回の目的

この回では、DirectXTK Audio で使う基本的なクラスの役割を整理します。

音を鳴らすプログラムでは、いきなりコードを書く前に、次の3つを分けて考えることが大切です。

```
AudioEngine          … 音を鳴らす仕組み全体
SoundEffect          … 音声ファイル1個分のデータ
SoundEffectInstance  … 実際に再生中の音
```

この3つの違いが分かると、DirectXTK Audio のコードがかなり読みやすくなります。

---

1. DirectXTK Audio で音を鳴らす流れ

DirectXTK Audio では、おおまかに次の流れで音を鳴らします。

```
AudioEngine を作る
↓
SoundEffect で wav ファイルを読み込む
↓
SoundEffect::Play() で音を鳴らす
```

一番単純な考え方は、これです。

```
AudioEngine  … 音を鳴らす機械本体
SoundEffect  … wav ファイルを読み込んだ音の素材
Play()       … その音を鳴らす命令
```

---

2. AudioEngine

`AudioEngine` は、DirectXTK Audio の中心になるクラスです。

```
std::unique_ptr<DirectX::AudioEngine> audioEngine_;
```

`AudioEngine` は、音を鳴らすための仕組み全体を管理します。

簡単に言うと、

```
AudioEngine = 音を鳴らす機械本体
```

です。

これがないと、音声ファイルを読み込んでも音を鳴らすことができません。

---

3. SoundEffect

`SoundEffect` は、音声ファイル1個分のデータを表します。

```
std::unique_ptr<DirectX::SoundEffect> sound_;
```

例えば、次のような音声ファイルを読み込んだものです。

```
shot.wav
jump.wav
damage.wav
```

実際のコードでは、次のように作ります。

```
sound_ = std::make_unique<DirectX::SoundEffect>(
    audioEngine_.get(),
    L"Assets/Audio/shot.wav"
);
```

イメージとしては、

```
SoundEffect = 音の素材
```

です。

短い効果音であれば、`SoundEffect` の `Play()` を呼ぶだけで再生できます。

```
sound_->Play();
```

---

4. SoundEffectInstance

`SoundEffectInstance` は、実際に再生中の音を表します。

```
std::unique_ptr<DirectX::SoundEffectInstance> instance_;
```

`SoundEffectInstance` は、`SoundEffect` から作ります。

```
instance_ = sound_->CreateInstance();
```

イメージとしては、次のようになります。

```
SoundEffect       = 音の素材
SoundEffectInstance = 実際に鳴っている音
```

例えば、`engine.wav` というエンジン音があったとします。

```
engine.wav
```

これは音の素材なので、`SoundEffect` として読み込みます。

その音を実際に鳴らしながら、あとから音量を変えたり、止めたり、ループさせたりしたい場合は、`SoundEffectInstance` を使います。

```
instance_->Play(true);
instance_->SetVolume(0.5f);
instance_->Stop();
```

---

5. SoundEffect と SoundEffectInstance の違い

ここが一番大切です。

クラス	役割	向いている使い方
`SoundEffect`	音声ファイルのデータ	短いSEを鳴らす
`SoundEffectInstance`	再生中の音	BGM、ループ音、音量変更、停止、一時停止

---

6. 短い効果音なら SoundEffect::Play()

例えば、弾を撃つ音です。

```
shotSound_->Play();
```

これは、

```
鳴らす
↓
終わったら自動で終わる
```

という使い方です。

この方法に向いている音は、次のようなものです。

```
ジャンプ音
攻撃音
決定音
ダメージ音
爆発音
```

何度も重なって鳴ってもよい音に向いています。

---

7. 止めたり音量を変えたいなら SoundEffectInstance

例えば、BGMです。

```
bgmInstance_ = bgmSound_->CreateInstance();
bgmInstance_->Play(true);
```

BGMは、あとから操作したくなることが多いです。

```
bgmInstance_->SetVolume(0.5f);
bgmInstance_->Pause();
bgmInstance_->Resume();
bgmInstance_->Stop();
```

この方法に向いている音は、次のようなものです。

```
BGM
足音ループ
エンジン音
環境音
3D空間で鳴らす音
```

---

8. たとえで考える

次のように考えると分かりやすいです。

```
AudioEngine
→ 音を鳴らすゲーム機本体

SoundEffect
→ wav ファイルを読み込んだ音の素材

SoundEffectInstance
→ 今、実際に鳴っている音
```

または、音楽プレイヤーにたとえると、こうなります。

```
AudioEngine
→ 音楽プレイヤー本体

SoundEffect
→ 音楽ファイル

SoundEffectInstance
```

→ 実際に再生中の曲

9. なぜ SoundEffect だけではだめなのか

短い効果音だけなら、`SoundEffect::Play()` で十分です。

```
shotSound_->Play();
```

しかし、次のようなことをしたい場合は、`SoundEffect` だけでは扱いにくくなります。

```
ループさせたい
途中で止めたい
一時停止したい
再開したい
音量を少しずつ変えたい
3D空間の位置を反映したい
```

このような場合は、`SoundEffectInstance` を使います。

10. SoundEffectInstance を使うときの注意

`SoundEffectInstance` は、元になった `SoundEffect` の音データを使います。

つまり、次のような関係です。

```
SoundEffect
  └── SoundEffectInstance
```

そのため、`SoundEffectInstance` を使っている間に、元の `SoundEffect` を先に消してはいけません。

危険な順番はこれです。

```
SoundEffect を先に消す
↓
SoundEffectInstance が使う音データがなくなる
↓
正しく動かない可能性がある
```

安全な考え方は、次の順番です。

```
作る順番
AudioEngine
↓
SoundEffect
↓
SoundEffectInstance

消す順番
SoundEffectInstance
↓
SoundEffect
↓
AudioEngine
```

11. 今回作るオーディオエンジンでの使い分け

今回のオーディオエンジンでは、次のように使い分けます。

SE

SEは `SoundEffect` を使います。

```
Audio::LoadSE("shot", shotPath);
Audio::PlaySE("shot");
```

内部では、次のような処理になります。

```
sounds_["shot"]->Play();
```

SEは短く、鳴らしたら終わるものが多いからです。

BGM

BGMは `SoundEffectInstance` を使います。

```
Audio::LoadBGM("stage", bgmPath);
Audio::PlayBGM("stage", true);
Audio::StopBGM();
```

内部では、次のような処理になります。

```
bgmInstance_ = bgmSound_->CreateInstance();
bgmInstance_->Play(true);
```

BGMはループさせたり、止めたり、音量を変えたりしたいからです。

---

12. 最初に覚えること

最初は、これだけ覚えれば大丈夫です。

```
AudioEngine は、音を鳴らす仕組み本体。

SoundEffect は、wav ファイルを読み込んだ音データ。

SoundEffectInstance は、再生中の音を操作するためのもの。
```

使い分けは、次のように考えます。

```
短いSE
→ SoundEffect::Play()

BGM、ループ音、3D音、音量操作したい音
→ SoundEffectInstance
```

確認問題

問題1

`AudioEngine` は何をするためのクラスですか。

```
答え：
```

問題2

`SoundEffect` は何を表しますか。

```
答え：
```

問題3

BGMのように、ループさせたり止めたりしたい音には、`SoundEffect` と `SoundEffectInstance` のどちらを使うとよいですか。

> 答え：

問題4

短いジャンプ音を1回鳴らすだけなら、次のどちらが向いていますか。

> A. SoundEffect::Play()
> B. SoundEffectInstance を作って管理する

> 答え：

前準備その2　スマポ

スマートポインタ入門

目的

この資料では、C++ の `std::unique_ptr` の基本的な使い方を学びます。

今回の目標は、スマートポインタを深く理解することではありません。

まずは、次のことが分かれば十分です。

```
new で作ったものは delete で消す必要がある
std::unique_ptr を使うと delete を自動でやってくれる
std::make_unique は unique_ptr 用の new のようなもの
```

この考え方を使って、あとで DirectXTK Audio の `AudioEngine` や `SoundEffect` を安全に管理します。

この資料の位置づけ

この資料では、オーディオエンジンを作る前に必要になる `std::unique_ptr` を中心に扱います。

`std::shared_ptr` や `std::weak_ptr` は、複数の場所で同じデータを共有するときの発展内容です。今回のオーディオエンジン本体では、まず `std::unique_ptr` を使います。

```
Audioが作る
Audioが持つ
Audioが消す
```

この流れをはっきりさせるためです。

---

1. 生ポインタ版：new と delete

まずは、スマートポインタを使わない書き方を確認します。

例：Player クラス

```
#include <iostream>

class Player
{
public:
    Player()
    {
        std::cout << "Playerを作成しました\n";
    }

    ~Player()
    {
        std::cout << "Playerを削除しました\n";
    }
```

```
    void Attack()
    {
        std::cout << "攻撃しました\n";
    }
};
```

この `Player` を `new` で作ると、次のようになります。

```
int main()
{
    Player* player = new Player();

    player->Attack();

    delete player;
    player = nullptr;

    return 0;
}
```

ポイント

```
Player* player = new Player();
```

これは、メモリ上に `Player` を作っています。

作ったものは、最後に消す必要があります。

```
delete player;
player = nullptr;
```

この `delete` を忘れると、作ったものがメモリに残ったままになります。

---

2. delete を忘れると困る

次のコードはよくありません。

```
int main()
{
    Player* player = new Player();

    player->Attack();

    return 0;
}
```

`new Player()` で作ったのに、`delete` していません。

```
Player* player = new Player();
```

したら、基本的には、あとで

```
delete player;
```

が必要です。

小さいプログラムではすぐ問題が見えないこともあります。 しかし、ゲームエンジンのように長く動くプログラムでは、消し忘れが増えると危険です。

---

3. std::unique_ptr 版

`std::unique_ptr` を使うと、`delete` を自動で行えます。

必要な include

```
#include <memory>
```

`std::unique_ptr` や `std::make_unique` を使うには、`<memory>` が必要です。

---

unique_ptr で Player を作る

```
#include <iostream>
#include <memory>

class Player
{
public:
    Player()
    {
        std::cout << "Playerを作成しました\n";
    }

    ~Player()
    {
        std::cout << "Playerを削除しました\n";
    }

    void Attack()
    {
        std::cout << "攻撃しました\n";
    }
};

int main()
{
    std::unique_ptr<Player> player = std::make_unique<Player>();

    player->Attack();

    return 0;
}
```

このコードには `delete` がありません。

```
std::unique_ptr<Player> player = std::make_unique<Player>();
```

と書くと、`Player` を作って、`unique_ptr` が管理してくれます。

`main` 関数が終わると、`player` も自動的に片付けられます。

そのため、手動で

```
delete player;
```

を書く必要がありません。

---

4. 生ポインタ版と unique_ptr 版の比較

生ポインタ版

```
Player* player = new Player();

player->Attack();

delete player;
player = nullptr;
```

unique_ptr 版

```
std::unique_ptr<Player> player = std::make_unique<Player>();

player->Attack();
```

違い

書き方	作り方	消し方
生ポインタ	`new`	`delete` が必要
`unique_ptr`	`std::make_unique`	自動で消える

---

5. まず覚えること

`std::unique_ptr` は、次のように考えると分かりやすいです。

```
自動で delete してくれる箱
```

例えば、次のコードでは、`player` が `Player` を管理しています。

```
std::unique_ptr<Player> player = std::make_unique<Player>();
```

`player` がなくなると、中に入っている `Player` も自動で削除されます。

---

6. 関数を呼ぶときは普通のポインタに近い

`unique_ptr` で管理していても、使い方は普通のポインタに似ています。

```
std::unique_ptr<Player> player = std::make_unique<Player>();

player->Attack();
```

`player` は `Player` を指しているので、メンバ関数を呼ぶときは `->` を使います。

```
player->Attack();
```

これは、生ポインタ版と同じ感覚です。

```
Player* player = new Player();
player->Attack();
```

---

7. reset で手動削除もできる

基本的には自動で消えるので、毎回使う必要はありません。

ただし、途中で明示的に消したい場合は `reset()` を使えます。

```
std::unique_ptr<Player> player = std::make_unique<Player>();

player->Attack();

player.reset();
```

`reset()` を呼ぶと、中の `Player` が削除されます。

生ポインタ版で言うと、次の処理に近いです。

```
delete player;
player = nullptr;
```

---

8. 空かどうかを確認する

`unique_ptr` が何も持っていない場合もあります。

```
std::unique_ptr<Player> player;
```

この状態では、まだ `Player` は作られていません。

使う前に確認できます。

```
if (player != nullptr)
{
    player->Attack();
}
```

または、次のようにも書けます。

```
if (player)
{
    player->Attack();
}
```

---

9. get() で中のポインタを取り出す

`unique_ptr` が管理している中身を、普通のポインタとして渡したい場合があります。

そのときは `get()` を使います。

```
std::unique_ptr<Player> player = std::make_unique<Player>();

Player* rawPlayer = player.get();
```

`get()` は、中に入っているポインタを取り出します。

ただし、重要な注意があります。

```
Player* rawPlayer = player.get();
```

で取り出した `rawPlayer` を、次のように `delete` してはいけません。

```
// これはダメ
delete rawPlayer;
```

理由は、`Player` は `unique_ptr` が管理しているからです。

消す仕事は `unique_ptr` に任せます。

---

10. get() は DirectXTK Audio で使う

DirectXTK Audio では、`SoundEffect` を作るときに `AudioEngine` のポインタを渡します。

生ポインタ版では、こうでした。

```
DirectX::AudioEngine* audioEngine = new DirectX::AudioEngine();

DirectX::SoundEffect* sound = new DirectX::SoundEffect(
    audioEngine,
    L"Assets/Audio/test.wav"
);
```

`SoundEffect` の作成には、`audioEngine` が必要です。

`unique_ptr` 版では、`audioEngine` は次のように管理します。

```
std::unique_ptr<DirectX::AudioEngine> audioEngine;
```

そして、作成します。

```
audioEngine = std::make_unique<DirectX::AudioEngine>();
```

このままだと、`SoundEffect` に渡す普通のポインタがありません。

そこで `get()` を使います。

```
std::unique_ptr<DirectX::SoundEffect> sound;
```

```
sound = std::make_unique<DirectX::SoundEffect>(
    audioEngine.get(),
    L"Assets/Audio/test.wav"
);
```

ここで使っている

```
audioEngine.get()
```

は、`audioEngine` の中にある普通のポインタを取り出して渡しているだけです。

`delete` はしません。

---

11. オーディオエンジンでの使い方

今回のオーディオエンジンでは、次のような変数を使います。

```
#include <memory>
#include <unordered_map>
#include <string>
#include <filesystem>
#include <Audio.h>

using file_path = std::filesystem::path;

namespace
{
    std::unique_ptr<DirectX::AudioEngine> audioEngine_;
    std::unordered_map<std::string, std::unique_ptr<DirectX::SoundEffect>> sounds_;
}
```

AudioEngine を作る

```
bool Audio::Initialize()
{
    audioEngine_ = std::make_unique<DirectX::AudioEngine>();
    return true;
}
```

これは、生ポインタ版で言うと次の処理に近いです。

```
audioEngine_ = new DirectX::AudioEngine();
```

ただし、`unique_ptr` 版では `delete` を自動で行います。

---

12. SoundEffect を作る

音声ファイルを読み込む処理は、次のようになります。

```
bool Audio::Load(const std::string& name, file_path& filepath)
{
    if (audioEngine_ == nullptr)
```

```
    {
        return false;
    }

    sounds_[name] = std::make_unique<DirectX::SoundEffect>(
        audioEngine_.get(),
        filepath.c_str()
    );

    return true;
}
```

ここで大事なところ

```
std::make_unique<DirectX::SoundEffect>( ... )
```

で `SoundEffect` を作っています。

```
sounds_[name] = ...
```

で、作った音を `sounds_` に保存しています。

```
audioEngine_.get()
```

で、`SoundEffect` に `AudioEngine` のポインタを渡しています。

```
filepath.c_str()
```

で、`std::filesystem::path` から DirectXTK Audio に渡すファイルパスを取り出しています。

---

13. 音を鳴らす

保存した音を鳴らす処理は、次のようになります。

```
void Audio::Play(const std::string& name)
{
    auto itr = sounds_.find(name);

    if (itr == sounds_.end())
    {
        return;
    }

    itr->second->Play();
}
```

`sounds_` には、名前と音がセットで保存されています。

```
sounds_["shot"]
```

のように、名前で音を探せます。

見つかったら、次のように再生します。

```
itr->second->Play();
```

`itr->second` は `std::unique_ptr<DirectX::SoundEffect>` です。

`unique_ptr` でも、メンバ関数を呼ぶときは `->` を使えます。

---

14. 終了処理

`unique_ptr` を使っている場合、ひとつずつ `delete` する必要はありません。

```
void Audio::Release()
{
    sounds_.clear();
    audioEngine_.reset();
}
```

sounds_.clear()

```
sounds_.clear();
```

`sounds_` の中に入っている `SoundEffect` をまとめて消します。

`std::unique_ptr` が入っているので、中の `SoundEffect` も自動で削除されます。

audioEngine_.reset()

```
audioEngine_.reset();
```

`AudioEngine` を削除します。

生ポインタ版で言うと、次に近いです。

```
delete audioEngine_;
audioEngine_ = nullptr;
```

---

15. 作る順番と消す順番

オーディオでは、作る順番が大事です。

```
AudioEngine を作る
↓
SoundEffect を作る
```

`SoundEffect` は `AudioEngine` を使って作ります。

そのため、消すときは逆順にします。

```
SoundEffect を消す
↓
AudioEngine を消す
```

今回の `Release()` では、この順番にしています。

```
void Audio::Release()
{
    sounds_.clear();
    audioEngine_.reset();
}
```

先に `sounds_` を消してから、`audioEngine_` を消しています。

---

16. 今回覚えるまとめ

生ポインタ

```
Player* player = new Player();
player->Attack();
delete player;
player = nullptr;
```

unique_ptr

```
std::unique_ptr<Player> player = std::make_unique<Player>();
player->Attack();
```

AudioEngine

```
audioEngine_ = std::make_unique<DirectX::AudioEngine>();
```

SoundEffect

```
sounds_[name] = std::make_unique<DirectX::SoundEffect>(
    audioEngine_.get(),
    filepath.c_str()
);
```

get()

```
audioEngine_.get()
```

`unique_ptr` の中にあるポインタを取り出して、DirectXTK Audio に渡すために使います。

reset()

```
audioEngine_.reset();
```

中身を削除します。

clear()

```
sounds_.clear();
```

`unordered_map` の中身を空にします。

中身が `unique_ptr` なので、管理している `SoundEffect` も自動で削除されます。

---

17. 確認問題

問題1

次の生ポインタ版で、最後に必要な処理を書いてください。

```
Player* player = new Player();
```

```
player->Attack();

// ここに必要な処理を書く
```

問題2

次の空欄を埋めてください。

```
std::unique_ptr<Player> player = std::_____________<Player>();
```

問題3

`unique_ptr` で管理しているオブジェクトのメンバ関数を呼ぶとき、どちらを使いますか。

```
player.Attack();
player->Attack();
```

問題4

次の `get()` の説明として正しいものを選んでください。

```
audioEngine_.get()
```

```
A. unique_ptr の中のポインタを取り出す
B. unique_ptr を削除する
C. 音を再生する
```

問題5

次のコードで、`delete rawPlayer;` をしてはいけない理由を説明してください。

```
std::unique_ptr<Player> player = std::make_unique<Player>();
Player* rawPlayer = player.get();

// delete rawPlayer; はしない
```

18. 解答例

問題1

```
delete player;
player = nullptr;
```

問題2

```
std::make_unique
```

問題3

```
player->Attack();
```

問題4

A. unique_ptr の中のポインタを取り出す

問題5

`rawPlayer` は、 `unique_ptr` が管理している中身を一時的に取り出しただけです。

削除は `unique_ptr` が行うので、 `delete rawPlayer;` はしません。

０１　音を鳴らしてみるだけ

第1回 DirectXTK Audioで音を1回鳴らす

この回に入る前に

先にDirectXTKとAudio を読み、次の3つの役割を確認しておきます。

```
AudioEngine　　: 音を鳴らす仕組み本体
SoundEffect　　: wavファイルを読み込んだ音データ
SoundEffectInstance : 再生中の音を操作するもの
```

この回では、まず `AudioEngine` と `SoundEffect` だけを使います。

目標

この回では、DirectXTK Audioを使って、WAVファイルを1回再生します。 まだオーディオエンジン化はしません。 まずは「DirectXTK Audioで音が鳴る」ことを確認します。

前提

DirectXTKのNuGet設定は完了しているものとします。 この資料では、NuGet設定の手順は扱いません。

使用する音声ファイルを次の場所に用意してください。

```
Assets/Audio/test.wav
```

WAVファイル名は、必ず `test.wav` にしてください。 ファイル名やフォルダ名が違うと読み込みに失敗します。

今回変更するファイル

```
TestScene.h
TestScene.cpp
```

今回は `Main.cpp` にはまだ手を入れません。 DirectXTK Audioの最小コードを `TestScene` に直接書いて確認します。

---

1. TestScene.h にAudio用の変数を追加する

`TestScene.h` を開きます。

先頭付近に、次の include を追加します。

```
#include <memory>
#include <filesystem>
#include <Audio.h>
```

次に、`TestScene` クラスの private メンバとして、次の2つを追加します。

```
private:
    std::unique_ptr<DirectX::AudioEngine> audioEngine_;
```

```
    std::unique_ptr<DirectX::SoundEffect> testSound_;
```

`TestScene.h` 全体の形は、だいたい次のようになります。

```
#pragma once
#include "Engine/GameObject.h"
#include <memory>
#include <filesystem>
#include <Audio.h>

class TestScene : public GameObject
{
public:
    TestScene(GameObject* parent);
    ~TestScene();

    void Initialize() override;
    void Update() override;
    void Draw() override;
    void Release() override;

private:
    std::unique_ptr<DirectX::AudioEngine> audioEngine_;
    std::unique_ptr<DirectX::SoundEffect> testSound_;
};
```

---

2. TestScene.cpp の Initialize で音声を読み込む

`TestScene.cpp` の `Initialize()` に処理を追加します。

変更前は次のようになっています。

```
void TestScene::Initialize()
{
    Instantiate<Stage>(this);
}
```

次のように変更します。

```
void TestScene::Initialize()
{
    Instantiate<Stage>(this);

    std::filesystem::path filepath = "Assets/Audio/test.wav";

    audioEngine_ = std::make_unique<DirectX::AudioEngine>();
    testSound_ = std::make_unique<DirectX::SoundEffect>(
        audioEngine_.get(),
        filepath.c_str()
    );

    testSound_->Play();
}
```

ここで行っていることは、次の3つです。

```
AudioEngineを作る
WAVファイルを読み込む
Playで再生する
```

`AudioEngine` は音を鳴らす仕組み本体です。 `SoundEffect` は音声ファイル1個分のデータです。

---

3. ファイルパスの型を確認する

今回のコードでは、音声ファイルの場所を `std::filesystem::path` に入れています。

```
std::filesystem::path filepath = "Assets/Audio/test.wav";
```

`std::filesystem::path` は、ファイルやフォルダの場所を表すための型です。 文字列のまま扱うよりも、「これはファイルパスである」という意味が分かりやすくなります。

DirectXTK Audio の `SoundEffect` に渡すときは、`filepath.c_str()` を使います。

```
testSound_ = std::make_unique<DirectX::SoundEffect>(
    audioEngine_.get(),
    filepath.c_str()
);
```

第2回から作る `Audio.h` では、次のように短い名前を付けて使います。

```
using file_path = std::filesystem::path;
```

---

4. スマートポインタの意味を確認する

今回のコードでは、次のような書き方をしています。

```
audioEngine_ = std::make_unique<DirectX::AudioEngine>();
```

これは、簡単に言うと次の意味です。

```
AudioEngineをnewで作る
作ったものをunique_ptrに持たせる
使い終わったら自動でdeleteしてもらう
```

`std::unique_ptr` は、作ったオブジェクトを1か所だけで管理するためのスマートポインタです。 普通のポインタと違って、使い終わったときに自動で解放してくれます。

生ポインタで書くと、次のようになります。

```
DirectX::AudioEngine* audioEngine_ = nullptr;
DirectX::SoundEffect* testSound_ = nullptr;

audioEngine_ = new DirectX::AudioEngine();

testSound_ = new DirectX::SoundEffect(
    audioEngine_,
    filepath.c_str()
);
```

この場合、終了時に自分で `delete` する必要があります。

```
delete testSound_;
testSound_ = nullptr;

delete audioEngine_;
audioEngine_ = nullptr;
```

消す順番は、作った順番の逆です。

```
作る順番：AudioEngine → SoundEffect
消す順番：SoundEffect → AudioEngine
```

今回の授業では、解放忘れを防ぐために `std::unique_ptr` を使います。 `std::make_unique` は、`new` と `unique_ptr` への代入をまとめて行う便利な書き方です。

`SoundEffect` を作るときの `audioEngine_.get()` は、`unique_ptr` の中に入っているポインタを一時的に取り出すための書き方です。

```
testSound_ = std::make_unique<DirectX::SoundEffect>(
    audioEngine_.get(),
    filepath.c_str()
);
```

ここでは、`SoundEffect` を作るために `AudioEngine` の場所を渡しています。 `get()` で取り出したポインタを `delete` してはいけません。解放は `unique_ptr` に任せます。

---

5. UpdateでAudioEngineを更新する

`TestScene.cpp` の `Update()` に、次の処理を追加します。

```
void TestScene::Update()
{
    if (audioEngine_ != nullptr)
    {
        audioEngine_->Update();
    }
}
```

`AudioEngine` は毎フレーム更新します。 今後、Audio処理をエンジン側に移したあとも、`Update()` は必要になります。

---

6. Releaseで解放する

`TestScene.cpp` の `Release()` に次の処理を追加します。

```
void TestScene::Release()
{
    testSound_.reset();
    audioEngine_.reset();
}
```

`unique_ptr` を使っているので、`reset()` で解放できます。

---

7. ビルドして実行する

実行したときに、起動直後に `test.wav` が1回鳴れば成功です。

確認すること：

```
ビルドが通る
起動直後に音が鳴る
終了時にエラーが出ない
```

よくある失敗

Audio.h が見つからない

DirectXTKのNuGet設定ができていない可能性があります。

```
#include <Audio.h>
```

でエラーが出る場合は、DirectXTKがプロジェクトに追加されているか確認してください。

音が鳴らない

まずファイルパスを確認してください。

```
"Assets/Audio/test.wav"
```

実行時の作業フォルダから見て、この場所にファイルが必要です。

mp3やoggを指定している

今回使うファイルはWAVです。 まずはWAVファイルで確認してください。

今回の完成状態

この回では、音声処理をまだ `TestScene` に直接書いています。 これは最終形ではありません。

次回は、音声処理を `Engine/Audio.h` と `Engine/Audio.cpp` に移して、ゲームエンジンの機能として使える形にします。

02 Engineに組み込んでゆくぅ

第2回 Audio機能をEngineに組み込んでゆくぅ

目標

この回では、第1回で `TestScene` に直接書いた音声処理を、エンジン側の機能として分離します。

最終的に、ゲーム側から次のように使える形にします。

```
file_path testPath = "Assets/Audio/test.wav";
Audio::Load("test", testPath);
Audio::Play("test");
```

DirectXTK Audioの細かい処理は、`Audio` の中に隠します。

今回変更するファイル

```
Engine/Audio.h      新規作成
Engine/Audio.cpp    新規作成
Main.cpp            追加
TestScene.h         第1回のAudio変数を削除
TestScene.cpp       Audio::Load / Audio::Play を使う
```

Visual Studioのソリューションエクスプローラーで、`Engine/Audio.h` と `Engine/Audio.cpp` をプロジェクトに追加してください。

---

1. Engine/Audio.h を作成する

`Engine` フォルダに `Audio.h` を作成します。

```
#pragma once
#include <string>
#include <filesystem>

using file_path = std::filesystem::path;

namespace Audio
{
    bool Initialize();
    void Update();
    void Release();

    bool Load(const std::string& name, file_path& filepath);
    void Play(const std::string& name);
}
```

`file_path` は、ファイルパスを表す型として `std::filesystem::path` に別名を付けたものです。

```
using file_path = std::filesystem::path;
```

これにより、呼び出し側では `L"Assets/..."` のようにワイド文字列を直接書かず、通常の文字列からパスを作って渡せます。

```
file_path testPath = "Assets/Audio/test.wav";
Audio::Load("test", testPath);
```

今回の `Load` は `file_path&` なので、文字列を直接渡すのではなく、いったん `file_path` 変数を作ってから渡します。

この段階では、効果音とBGMはまだ分けません。 まずは「名前を付けて読み込む」「名前で鳴らす」だけを作ります。

---

2. Engine/Audio.cpp を作成する

`Engine` フォルダに `Audio.cpp` を作成します。

```
#include "Audio.h"
#include <Audio.h>
#include <memory>
#include <unordered_map>

namespace
{
    std::unique_ptr<DirectX::AudioEngine> audioEngine_;
    std::unordered_map<std::string, std::unique_ptr<DirectX::SoundEffect>> sounds_;
}

bool Audio::Initialize()
{
    audioEngine_ = std::make_unique<DirectX::AudioEngine>();
    return true;
}

void Audio::Update()
{
    if (audioEngine_ != nullptr)
    {
        audioEngine_->Update();
    }
}

void Audio::Release()
{
    sounds_.clear();
    audioEngine_.reset();
}

bool Audio::Load(const std::string& name, file_path& filepath)
{
    if (audioEngine_ == nullptr)
    {
        return false;
    }

    sounds_[name] = std::make_unique<DirectX::SoundEffect>(
        audioEngine_.get(),
        filepath.c_str()
    );

    return true;
}

void Audio::Play(const std::string& name)
{
    auto it = sounds_.find(name);
    if (it == sounds_.end())
    {
        return;
    }

    it->second->Play();
}
```

3. Main.cpp に Audio を組み込む

`Main.cpp` の include 部分に、次を追加します。

```
#include "Engine\\Audio.h"
```

`Input::Initialize(hWnd);` の後に、Audioの初期化を追加します。

```
Input::Initialize(hWnd); // 入力の初期化
Audio::Initialize();     // オーディオの初期化
```

メインループ内で、`pRootJob->UpdateSub();` の後に Audio の更新を追加します。

```
pRootJob->UpdateSub();
Audio::Update();
```

終了処理に Audio の解放を追加します。

```
Model::Release();
pRootJob->ReleaseSub();
Audio::Release();
Input::Release();
Direct3D::Release();
```

4. TestScene.h から第1回のAudio変数を削除する

第1回で `TestScene.h` に追加した次の include と変数は削除します。

```
#include <memory>
#include <Audio.h>
```

```
std::unique_ptr<DirectX::AudioEngine> audioEngine_;
std::unique_ptr<DirectX::SoundEffect> testSound_;
```

`TestScene` はDirectXTK Audioを直接知らない形に戻します。

5. TestScene.cpp で Audio を使う

`TestScene.cpp` の include に、次を追加します。

```
#include "Engine/Audio.h"
```

スペースキーで鳴らす確認を行うため、`Engine/Input.h` が入っていない場合は追加します。

```
#include "Engine/Input.h"
```

`Initialize()` を次のように変更します。

```
void TestScene::Initialize()
{
    Instantiate<Stage>(this);

    file_path testPath = "Assets/Audio/test.wav";
    Audio::Load("test", testPath);
    Audio::Play("test");
}
```

`Update()` は、いったん空で構いません。

```
void TestScene::Update()
```

```
{
}
```

`Release()` も、いったん空で構いません。

```
void TestScene::Release()
{
}
```

---

6. ビルドして実行する

起動直後に `test.wav` が1回鳴れば成功です。

第1回と違う点は、`TestScene` がDirectXTK Audioを直接使っていないことです。

```
第1回：TestSceneがDirectXTK Audioを直接使う
第2回：Audio機能の中でDirectXTK Audioを使う
```

---

7. スペースキーで音を鳴らす

`TestScene.cpp` の `Update()` を次のように変更します。

```
void TestScene::Update()
{
    if (Input::IsKeyDown(DIK_SPACE))
    {
        Audio::Play("test");
    }
}
```

この変更で、スペースキーを押した瞬間に音が鳴ります。

`IsKeyDown` は押した瞬間だけ true になります。 押しっぱなしで何度も鳴らしたくないときに使います。

今回の完成状態

この回で、Audio機能が `Engine` 側に移動しました。

現在の使い方は次の形です。

```
file_path testPath = "Assets/Audio/test.wav";
Audio::Load("test", testPath);
Audio::Play("test");
```

次回は、効果音とBGMを分けて管理できるようにします。

03 SEとBGMを別扱いに（なんで？）

第3回 効果音とBGMを分ける

目標

この回では、Audio機能を次の2種類に分けます。

```
SE  : 効果音。短い音。何度も鳴る。
BGM : 背景音楽。長い音。基本的に1曲を流し続ける。
```

最終的に、次のように使える形にします。

```
file_path shotPath = "Assets/Audio/SE/shot.wav";
file_path stagePath = "Assets/Audio/BGM/stage.wav";

Audio::LoadSE("shot", shotPath);
Audio::LoadBGM("stage", stagePath);

Audio::PlaySE("shot");
Audio::PlayBGM("stage", true);
```

今回変更するファイル

```
Engine/Audio.h
Engine/Audio.cpp
TestScene.cpp
```

使用する音声ファイルを次の場所に用意してください。

```
Assets/Audio/SE/shot.wav
Assets/Audio/SE/damage.wav
Assets/Audio/BGM/stage.wav
```

ファイル名は、資料内のコードと合わせてください。

---

1. Engine/Audio.h を変更する

`Engine/Audio.h` を次のように変更します。

```
#pragma once
#include <string>
#include <filesystem>

using file_path = std::filesystem::path;

namespace Audio
{
    bool Initialize();
    void Update();
    void Release();
```

```
    bool LoadSE(const std::string& name, file_path& filepath);
    void PlaySE(const std::string& name);

    bool LoadBGM(const std::string& name, file_path& filepath);
    void PlayBGM(const std::string& name, bool loop = true);
    void StopBGM();
}
```

`file_path` の使い方は第2回と同じです。

```
file_path shotPath = "Assets/Audio/SE/shot.wav";
Audio::LoadSE("shot", shotPath);
```

`LoadSE("shot", "Assets/Audio/SE/shot.wav")` のように直接文字列を渡すのではなく、いったん `file_path` 変数に入れてから渡します。

第2回で作った `Load` と `Play` は、今回から使いません。

```
Load   → LoadSE / LoadBGM に分ける
Play   → PlaySE / PlayBGM に分ける
```

---

2. Engine/Audio.cpp を変更する

`Engine/Audio.cpp` を次のように変更します。

```
#include "Audio.h"
#include <Audio.h>
#include <memory>
#include <unordered_map>

namespace
{
    std::unique_ptr<DirectX::AudioEngine> audioEngine_;

    std::unordered_map<std::string, std::unique_ptr<DirectX::SoundEffect>> seSounds_;
    std::unordered_map<std::string, std::unique_ptr<DirectX::SoundEffect>> bgmSounds_;

    std::unique_ptr<DirectX::SoundEffectInstance> currentBGM_;
    std::string currentBGMName_;
}

bool Audio::Initialize()
{
    audioEngine_ = std::make_unique<DirectX::AudioEngine>();
    return true;
}

void Audio::Update()
{
    if (audioEngine_ != nullptr)
    {
        audioEngine_->Update();
    }
}

void Audio::Release()
{
    currentBGM_.reset();
    bgmSounds_.clear();
    seSounds_.clear();
    audioEngine_.reset();
}

bool Audio::LoadSE(const std::string& name, file_path& filepath)
{
    if (audioEngine_ == nullptr)
    {
```

```
        return false;
    }

    seSounds_[name] = std::make_unique<DirectX::SoundEffect>(
        audioEngine_.get(),
        filepath.c_str()
    );

    return true;
}

void Audio::PlaySE(const std::string& name)
{
    auto it = seSounds_.find(name);
    if (it == seSounds_.end())
    {
        return;
    }

    it->second->Play();
}

bool Audio::LoadBGM(const std::string& name, file_path& filepath)
{
    if (audioEngine_ == nullptr)
    {
        return false;
    }

    bgmSounds_[name] = std::make_unique<DirectX::SoundEffect>(
        audioEngine_.get(),
        filepath.c_str()
    );

    return true;
}

void Audio::PlayBGM(const std::string& name, bool loop)
{
    auto it = bgmSounds_.find(name);
    if (it == bgmSounds_.end())
    {
        return;
    }

    if (currentBGM_ != nullptr)
    {
        currentBGM_->Stop(true);
        currentBGM_.reset();
    }

    currentBGM_ = it->second->CreateInstance();
    currentBGM_->Play(loop);
    currentBGMName_ = name;
}

void Audio::StopBGM()
{
    if (currentBGM_ != nullptr)
    {
        currentBGM_->Stop(true);
        currentBGM_.reset();
        currentBGMName_.clear();
    }
}
```

3. SoundEffect と SoundEffectInstance の違い

今回から `SoundEffectInstance` が出てきます。

```
SoundEffect         : 音声ファイルのデータ
SoundEffectInstance : 実際に再生している音
```

効果音は、単純に `PlaySE` で鳴らします。

```
Audio::PlaySE("shot");
```

BGMは、再生中の音を止めたり、ループしたりする必要があります。 そのため、`SoundEffectInstance` を使います。

```
currentBGM_ = it->second->CreateInstance();
currentBGM_->Play(loop);
```

---

4. TestScene.cpp で使う

`TestScene.cpp` の `Initialize()` を次のように変更します。

```
void TestScene::Initialize()
{
    Instantiate<Stage>(this);

    file_path shotPath = "Assets/Audio/SE/shot.wav";
    file_path damagePath = "Assets/Audio/SE/damage.wav";
    file_path stagePath = "Assets/Audio/BGM/stage.wav";

    Audio::LoadSE("shot", shotPath);
    Audio::LoadSE("damage", damagePath);
    Audio::LoadBGM("stage", stagePath);

    Audio::PlayBGM("stage", true);
}
```

`Update()` を次のように変更します。

```
void TestScene::Update()
{
    if (Input::IsKeyDown(DIK_SPACE))
    {
        Audio::PlaySE("shot");
    }

    if (Input::IsKeyDown(DIK_D))
    {
        Audio::PlaySE("damage");
    }

    if (Input::IsKeyDown(DIK_B))
    {
        Audio::StopBGM();
    }
}
```

---

5. ビルドして実行する

確認すること：

```
起動するとBGMがループ再生される
スペースキーでshot.wavが鳴る
Dキーでdamage.wavが鳴る
BキーでBGMが止まる
```

よくある失敗

BGMが鳴らない

ファイルパスを確認してください。

```
"Assets/Audio/BGM/stage.wav"
```

BGMが一瞬で止まる

`currentBGM_` をローカル変数にしていると、関数終了時に消えてしまいます。 今回のコードでは、ファイル上部の `namespace` 内に置いています。

```
std::unique_ptr<DirectX::SoundEffectInstance> currentBGM_;
```

効果音を連打すると音が重なる

これは正常です。 効果音は、短い音を何度も鳴らす用途なので、同じ音が重なって鳴っても問題ありません。

今回の完成状態

この回で、Audio機能は次の形になりました。

```
file_path shotPath = "Assets/Audio/SE/shot.wav";
file_path stagePath = "Assets/Audio/BGM/stage.wav";

Audio::LoadSE("shot", shotPath);
Audio::PlaySE("shot");

Audio::LoadBGM("stage", stagePath);
Audio::PlayBGM("stage", true);
Audio::StopBGM();
```

次回は、マスター音量、SE音量、BGM音量を分けて管理します。

04 サウンドのコントロール機能（簡単なのだけ）

第4回 音量管理とBGMフェード

目標

この回では、オーディオエンジンに音量管理を追加します。

次の3種類の音量を扱います。

```
MasterVolume : 全体の音量
SEVolume     : 効果音の音量
BGMVolume    : BGMの音量
```

さらに、BGMのフェードアウトも追加します。 シーン切り替え時に、音を急に止めず、自然に小さくできます。

今回変更するファイル

```
Engine/Audio.h
Engine/Audio.cpp
TestScene.cpp
```

今回のフェード処理について

この資料では、分かりやすさを優先して `BGMVolume` そのものを少しずつ変化させます。

本格的なエンジンでは、次のように分けて管理する方法もあります。

```
BGMの基本音量
フェード用の音量倍率
```

今回は、まず動くものを作ることを優先します。

---

1. Engine/Audio.h に音量とフェードの関数を追加する

`Engine/Audio.h` を次のように変更します。

```
#pragma once
#include <string>
#include <filesystem>

using file_path = std::filesystem::path;

namespace Audio
{
    bool Initialize();
    void Update();
    void Release();

    bool LoadSE(const std::string& name, file_path& filepath);
    void PlaySE(const std::string& name);
```

```
    bool LoadBGM(const std::string& name, file_path& filepath);
    void PlayBGM(const std::string& name, bool loop = true);
    void StopBGM();

    void SetMasterVolume(float volume);
    void SetSEVolume(float volume);
    void SetBGMVolume(float volume);

    void FadeOutBGM(float fadeTime);
}
```

2. Engine/Audio.cpp に音量用の変数を追加する

`Engine/Audio.cpp` の `namespace` 内に、音量用の変数を追加します。

```
float masterVolume_ = 1.0f;
float seVolume_ = 1.0f;
float bgmVolume_ = 1.0f;
```

さらに、フェード用の変数も追加します。

```
bool isBGMFadeOut_ = false;
float bgmFadeTimer_ = 0.0f;
float bgmFadeTime_ = 0.0f;
float bgmFadeStartVolume_ = 1.0f;
```

`namespace` 内は、最終的に次のような形になります。

```
namespace
{
    std::unique_ptr<DirectX::AudioEngine> audioEngine_;

    std::unordered_map<std::string, std::unique_ptr<DirectX::SoundEffect>> seSounds_;
    std::unordered_map<std::string, std::unique_ptr<DirectX::SoundEffect>> bgmSounds_;

    std::unique_ptr<DirectX::SoundEffectInstance> currentBGM_;
    std::string currentBGMName_;

    float masterVolume_ = 1.0f;
    float seVolume_ = 1.0f;
    float bgmVolume_ = 1.0f;

    bool isBGMFadeOut_ = false;
    float bgmFadeTimer_ = 0.0f;
    float bgmFadeTime_ = 0.0f;
    float bgmFadeStartVolume_ = 1.0f;
}
```

3. 音量を0.0f〜1.0fに収める関数を作る

`Audio.cpp` の `namespace` 内に、次の補助関数を追加します。

```
float ClampVolume(float volume)
{
    if (volume < 0.0f)
    {
        return 0.0f;
    }
    if (volume > 1.0f)
    {
        return 1.0f;
    }
    return volume;
}
```

```
float GetBGMFinalVolume()
{
    return masterVolume_ * bgmVolume_;
}

float GetSEFinalVolume()
{
    return masterVolume_ * seVolume_;
}
```

音量は、基本的に `0.0f` から `1.0f` の範囲で扱います。

```
0.0f : 無音
0.5f : 半分くらい
1.0f : 標準音量
```

---

4. PlaySE にSE音量を反映する

`PlaySE` を次のように変更します。

```
void Audio::PlaySE(const std::string& name)
{
    auto it = seSounds_.find(name);
    if (it == seSounds_.end())
    {
        return;
    }

    it->second->Play(GetSEFinalVolume());
}
```

これで、効果音は `MasterVolume × SEVolume` の音量で鳴ります。

---

5. PlayBGM にBGM音量を反映する

`PlayBGM` の最後に、音量設定を追加します。

```
void Audio::PlayBGM(const std::string& name, bool loop)
{
    auto it = bgmSounds_.find(name);
    if (it == bgmSounds_.end())
    {
        return;
    }

    if (currentBGM_ != nullptr)
    {
        currentBGM_->Stop(true);
        currentBGM_.reset();
    }

    currentBGM_ = it->second->CreateInstance();
    currentBGM_->SetVolume(GetBGMFinalVolume());
    currentBGM_->Play(loop);
    currentBGMName_ = name;

    isBGMFadeOut_ = false;
}
```

---

6. 音量設定関数を追加する

`Audio.cpp` に次の関数を追加します。

```
void Audio::SetMasterVolume(float volume)
{
    masterVolume_ = ClampVolume(volume);

    if (currentBGM_ != nullptr)
    {
        currentBGM_->SetVolume(GetBGMFinalVolume());
    }
}

void Audio::SetSEVolume(float volume)
{
    seVolume_ = ClampVolume(volume);
}

void Audio::SetBGMVolume(float volume)
{
    bgmVolume_ = ClampVolume(volume);

    if (currentBGM_ != nullptr)
    {
        currentBGM_->SetVolume(GetBGMFinalVolume());
    }
}
```

再生中のBGMは、音量を変更した瞬間に反映します。 効果音は、次に鳴らす音から新しい音量になります。

---

7. フェードアウト開始関数を追加する

`Audio.cpp` に次の関数を追加します。

```
void Audio::FadeOutBGM(float fadeTime)
{
    if (currentBGM_ == nullptr)
    {
        return;
    }

    if (fadeTime <= 0.0f)
    {
        StopBGM();
        return;
    }

    isBGMFadeOut_ = true;
    bgmFadeTimer_ = 0.0f;
    bgmFadeTime_ = fadeTime;
    bgmFadeStartVolume_ = bgmVolume_;
}
```

`fadeTime` は、何秒かけて音を小さくするかです。

```
Audio::FadeOutBGM(1.0f); // 1秒で小さくする
Audio::FadeOutBGM(2.0f); // 2秒で小さくする
```

---

8. Updateでフェード処理を進める

`Audio::Update()` を次のように変更します。

```
void Audio::Update()
{
    if (audioEngine_ != nullptr)
    {
        audioEngine_->Update();
```

```
    }

    if (isBGMFadeOut_ && currentBGM_ != nullptr)
    {
        const float DELTA_TIME = 1.0f / 60.0f;

        bgmFadeTimer_ += DELTA_TIME;
        float rate = bgmFadeTimer_ / bgmFadeTime_;

        if (rate >= 1.0f)
        {
            SetBGMVolume(0.0f);
            StopBGM();
            bgmVolume_ = bgmFadeStartVolume_;
            isBGMFadeOut_ = false;
            return;
        }

        float volume = bgmFadeStartVolume_ * (1.0f - rate);
        SetBGMVolume(volume);
    }
}
```

このエンジンはメインループを約60FPSで回しているため、ここでは `1.0f / 60.0f` を使います。 将来的に正確な `deltaTime` を作った場合は、その値を使う形に変更できます。

---

9. TestScene.cpp で確認する

`TestScene.cpp` の `Update()` を次のように変更します。

```
void TestScene::Update()
{
    if (Input::IsKeyDown(DIK_SPACE))
    {
        Audio::PlaySE("shot");
    }

    if (Input::IsKeyDown(DIK_D))
    {
        Audio::PlaySE("damage");
    }

    if (Input::IsKeyDown(DIK_1))
    {
        Audio::SetMasterVolume(1.0f);
    }

    if (Input::IsKeyDown(DIK_2))
    {
        Audio::SetMasterVolume(0.5f);
    }

    if (Input::IsKeyDown(DIK_3))
    {
        Audio::SetMasterVolume(0.0f);
    }

    if (Input::IsKeyDown(DIK_F))
    {
        Audio::FadeOutBGM(2.0f);
    }
}
```

確認すること：

```
1キーで音量が通常になる
2キーで音量が小さくなる
```

```
3キーで無音になる
FキーでBGMが2秒かけて小さくなる
```

今回の完成状態

この回で、Audio機能は次の形になりました。

```
Audio::SetMasterVolume(0.8f);
Audio::SetSEVolume(0.7f);
Audio::SetBGMVolume(0.5f);
Audio::FadeOutBGM(2.0f);
```

次回は、ゲームオブジェクトの位置と音を結びつけるために、AudioSourceと3D Audioを追加します。

05 応用編　３D音響

第5回 AudioSourceと3D Audio

目標

この回では、音に位置を持たせます。

通常の効果音は、画面全体に同じように聞こえます。 3D Audioでは、音源の位置とカメラの位置を使って、左右の聞こえ方や距離による音量変化を作ります。

この回の最終形は次の使い方です。

```
Audio::SetListener(cameraPosition, cameraForward, cameraUp);
Audio::PlaySE3D("explosion", soundPosition);
```

さらに、音を鳴らす部品として `AudioSource` クラスを作ります。

今回変更するファイル

```
Engine/Audio.h
Engine/Audio.cpp
Engine/AudioSource.h　　新規作成
Engine/AudioSource.cpp　新規作成
TestScene.h
TestScene.cpp
```

使用する音声ファイルを次の場所に用意してください。

```
Assets/Audio/SE/explosion.wav
```

---

1. Engine/Audio.h に3D Audio用の関数を追加する

`Engine/Audio.h` に `DirectXMath.h` を追加します。

```
#include <DirectXMath.h>
```

次に、`namespace Audio` の中に、次の関数を追加します。

```
void SetListener(
    const DirectX::XMFLOAT3& position,
    const DirectX::XMFLOAT3& forward,
    const DirectX::XMFLOAT3& up
);

void PlaySE3D(
    const std::string& name,
    const DirectX::XMFLOAT3& position
);
```

`Audio.h` 全体は、だいたい次のようになります。

```
#pragma once
#include <string>
#include <filesystem>
```

```
#include <DirectXMath.h>

using file_path = std::filesystem::path;

namespace Audio
{
    bool Initialize();
    void Update();
    void Release();

    bool LoadSE(const std::string& name, file_path& filepath);
    void PlaySE(const std::string& name);

    bool LoadBGM(const std::string& name, file_path& filepath);
    void PlayBGM(const std::string& name, bool loop = true);
    void StopBGM();

    void SetMasterVolume(float volume);
    void SetSEVolume(float volume);
    void SetBGMVolume(float volume);

    void FadeOutBGM(float fadeTime);

    void SetListener(
        const DirectX::XMFLOAT3& position,
        const DirectX::XMFLOAT3& forward,
        const DirectX::XMFLOAT3& up
    );

    void PlaySE3D(
        const std::string& name,
        const DirectX::XMFLOAT3& position
    );
}
```

---

2. Engine/Audio.cpp にListenerと3D再生処理を追加する

`Engine/Audio.cpp` の include に、`vector` と `algorithm` を追加します。

```
#include <vector>
#include <algorithm>
```

`namespace` 内に、3D Audio用の変数を追加します。

```
DirectX::AudioListener listener_;
std::vector<std::unique_ptr<DirectX::SoundEffectInstance>> active3DSE_;
```

`active3DSE_` は、3D効果音の `SoundEffectInstance` を保持するための配列です。

3D効果音では `SoundEffectInstance` を作って再生します。 しかし、作った `SoundEffectInstance` をどこにも保持しないと、関数が終わったときに消えてしまいます。 そのため、鳴っている間は `active3DSE_` に入れておきます。

`SetListener` を追加します。

```
void Audio::SetListener(
    const DirectX::XMFLOAT3& position,
    const DirectX::XMFLOAT3& forward,
    const DirectX::XMFLOAT3& up
)
{
    listener_.SetPosition(position);
    listener_.SetOrientation(forward, up);
}
```

`PlaySE3D` を追加します。

```
void Audio::PlaySE3D(
    const std::string& name,
    const DirectX::XMFLOAT3& position
)
{
    auto it = seSounds_.find(name);
    if (it == seSounds_.end())
    {
        return;
    }

    DirectX::AudioEmitter emitter;
    emitter.SetPosition(position);

    auto instance = it->second->CreateInstance(
        DirectX::SoundEffectInstance_Use3D
    );

    instance->Apply3D(listener_, emitter);
    instance->SetVolume(GetSEFinalVolume());
    instance->Play();

    active3DSE_.push_back(std::move(instance));
}
```

3D Audioでは、次の2つを使います。

```
AudioListener : 聞く側
AudioEmitter  : 音を出す側
```

このエンジンでは、最初はカメラを `AudioListener` として扱います。

---

3. Updateで鳴り終わった3D効果音を削除する

3D効果音は、再生するたびに `active3DSE_` に追加されます。

```
active3DSE_.push_back(std::move(instance));
```

このままだと、効果音を鳴らすたびに配列の中身が増え続けます。 そこで、 `Audio::Update()` の中で、鳴り終わったものを削除します。

`Audio::Update()` を次のように変更します。

```
void Audio::Update()
{
    if (audioEngine_ != nullptr)
    {
        audioEngine_->Update();
    }

    active3DSE_.erase(
        std::remove_if(
            active3DSE_.begin(),
            active3DSE_.end(),
            [](const std::unique_ptr<DirectX::SoundEffectInstance>& instance)
            {
                return instance == nullptr || instance->GetState() == DirectX::SoundState::STOPPED;
            }
        ),
        active3DSE_.end()
    );
}
```

これで、再生が終わった3D効果音は自動で消えます。

```
鳴っている3D効果音　→ active3DSE_ に残る
鳴り終わった3D効果音　→ Updateで削除される
```

4. Releaseで3D効果音を解放する

`Audio::Release()` の中に、次の処理を追加します。

```
active3DSE_.clear();
```

`Release()` は、だいたい次のようになります。

```
void Audio::Release()
{
    active3DSE_.clear();
    currentBGM_.reset();
    bgmSounds_.clear();
    seSounds_.clear();
    audioEngine_.reset();
}
```

5. Engine/AudioSource.h を作成する

`Engine` フォルダに `AudioSource.h` を作成します。

```
#pragma once
#include <string>
#include <filesystem>
#include <DirectXMath.h>

using file_path = std::filesystem::path;

class AudioSource
{
public:
    AudioSource();

    void SetClip(const std::string& name);
    void SetPosition(const DirectX::XMFLOAT3& position);
    void Set3D(bool is3D);
    void Play();

private:
    std::string clipName_;
    DirectX::XMFLOAT3 position_;
    bool is3D_;
};
```

`AudioSource` は、音を鳴らすための小さな部品です。

6. Engine/AudioSource.cpp を作成する

`Engine` フォルダに `AudioSource.cpp` を作成します。

```
#include "AudioSource.h"
#include "Audio.h"

AudioSource::AudioSource()
    : position_{ 0.0f, 0.0f, 0.0f }
    , is3D_(false)
{
}

void AudioSource::SetClip(const std::string& name)
{
    clipName_ = name;
}
```

```
void AudioSource::SetPosition(const DirectX::XMFLOAT3& position)
{
    position_ = position;
}

void AudioSource::Set3D(bool is3D)
{
    is3D_ = is3D;
}

void AudioSource::Play()
{
    if (clipName_.empty())
    {
        return;
    }

    if (is3D_)
    {
        Audio::PlaySE3D(clipName_, position_);
    }
    else
    {
        Audio::PlaySE(clipName_);
    }
}
```

Visual Studioのソリューションエクスプローラーで、`AudioSource.h` と `AudioSource.cpp` をプロジェクトに追加してください。

---

7. TestScene.h にAudioSourceを追加する

`TestScene.h` に include を追加します。

```
#include "Engine/AudioSource.h"
```

`TestScene` クラスにメンバ変数を追加します。

```
private:
    AudioSource explosionSound_;
```

---

8. TestScene.cpp で3D Audioを確認する

`TestScene.cpp` の include に、次を追加します。

```
#include "Engine/Camera.h"
#include <DirectXMath.h>
```

`Initialize()` に、3D用の効果音読み込みを追加します。

```
void TestScene::Initialize()
{
    Instantiate<Stage>(this);

    file_path shotPath = "Assets/Audio/SE/shot.wav";
    file_path damagePath = "Assets/Audio/SE/damage.wav";
    file_path explosionPath = "Assets/Audio/SE/explosion.wav";
    file_path stagePath = "Assets/Audio/BGM/stage.wav";

    Audio::LoadSE("shot", shotPath);
    Audio::LoadSE("damage", damagePath);
    Audio::LoadSE("explosion", explosionPath);
    Audio::LoadBGM("stage", stagePath);
```

```
    Audio::PlayBGM("stage", true);

    explosionSound_.SetClip("explosion");
    explosionSound_.Set3D(true);
    explosionSound_.SetPosition({ 3.0f, 0.0f, 0.0f });
}
```

`Update()` の先頭に、聞く側の情報を設定します。

```
void TestScene::Update()
{
    DirectX::XMFLOAT3 cameraPosition;
    DirectX::XMStoreFloat3(&cameraPosition, Camera::GetPosition());

    Audio::SetListener(
        cameraPosition,
        { 0.0f, 0.0f, 1.0f },
        { 0.0f, 1.0f, 0.0f }
    );

    if (Input::IsKeyDown(DIK_SPACE))
    {
        Audio::PlaySE("shot");
    }

    if (Input::IsKeyDown(DIK_D))
    {
        Audio::PlaySE("damage");
    }

    if (Input::IsKeyDown(DIK_E))
    {
        explosionSound_.Play();
    }

    if (Input::IsKeyDown(DIK_F))
    {
        Audio::FadeOutBGM(2.0f);
    }
}
```

この確認では、音源位置を次に固定しています。

```
{ 3.0f, 0.0f, 0.0f }
```

カメラから見て右側に音源があるため、左右の聞こえ方が変われば成功です。

---

9. ビルドして実行する

確認すること：

```
Eキーでexplosion.wavが鳴る
通常のPlaySEとは違い、左右の聞こえ方が変わる
FキーでBGMがフェードアウトする
```

ヘッドホンか左右が分かるスピーカーで確認すると分かりやすいです。

注意点

3D Audioは反響や壁判定までは行わない

今回の3D Audioで扱うのは、主に次の内容です。

```
左右の聞こえ方
```

> 距離による音量変化
> 音源位置と聞く側の位置

壁で音がこもる、洞窟で反響する、といった処理は別の発展内容です。

Listenerの向きは固定している

今回の確認では、聞く向きを次のように固定しています。

```
{ 0.0f, 0.0f, 1.0f }
```

カメラが自由に回転するゲームでは、カメラの向きから forward を計算して設定する必要があります。

今回の完成状態

この回で、Audio機能は次の形まで拡張されました。

```
Audio::PlaySE("shot");
Audio::PlayBGM("stage", true);
Audio::SetMasterVolume(0.8f);
Audio::FadeOutBGM(2.0f);

Audio::SetListener(cameraPosition, cameraForward, cameraUp);
Audio::PlaySE3D("explosion", soundPosition);
```

さらに、GameObject側で使いやすい部品として `AudioSource` を追加しました。

```
AudioSource sound;
sound.SetClip("explosion");
sound.Set3D(true);
sound.SetPosition({ 3.0f, 0.0f, 0.0f });
sound.Play();
```

これで、DirectXTK Audioを使った基本的なオーディオエンジンは完成です。