# 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
- 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

# 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_);
```

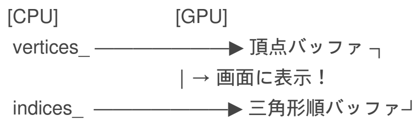
---

## ☒ 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…




## ☒ 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