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_);
```

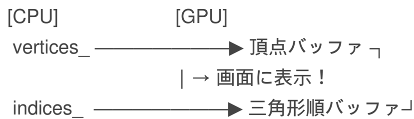
☒ インデックスバッファも同じ感じ

```
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

Revision #4
Created 25 June 2025 06:55:59 by youe2
Updated 2 June 2026 20:23:04 by youe2