[DX12講義第07回]

響け！スプライト・システム

1 スプライト

たくさんの小さな2D画像を重ね合わせて表示し、またそれぞれの画像を個別に移動、回転、拡大縮小を行うことのできるような機能のことを「スプライト」と呼んでいます。

スプライト(Sprite)は「妖精」という意味の英単語ですが、初めてこの機能にスプライトという名前を付けたのは、テキサス・インスツルメンツ社のTMS9918という画像制御チップのマニュアルだと言われています。この用語は、「スプライトが、フレームバッファ内のビットマップデータの一部ではなく、またフレームバッファ内のデータに影響を与えずに、まるで幽霊や妖精のように背景の上に浮かんでいる」ということに由来しているんだそうです。 スプライトという用語が発明される以前は、「ムーバブルオブジェクト」あるいは単に「オブジェクト」と呼ばれていたらしいですね。

2 セル

スプライトとして2D画像を表示する場合、表示したい画像をそれぞれ別のテクスチャとして持つこともできますが、テクスチャの切り替えには時間がかかるため、通常はできるだけ少ない枚数のテクスチャにまとめてしまいます。 そうすると、元の画像の範囲が分かるようなデータが必要になってきます。 このデータは「セル(Cell)」とか「チップ(Chip)」とか呼ばれます。今回はセルと呼ぶことにします。

まずは、`Src`フォルダに`Sprite.h`ファイルを追加してください。 そして、以下の`Cell`構造体のコードを追加します。

```
#include <DirectXMath.h>

 /*
 \* スプライトとして表示する画像の範囲.
 */
 struct Cell {
   DirectX::XMFLOAT2 uv; ///< テクスチャ上の左上座標.
   DirectX::XMFLOAT2 tsize; ///< テクスチャ上の縦横サイズ.
   DirectX::XMFLOAT2 ssize; ///< スクリーン座標上の縦横サイズ
 };
```

セルの構造は比較的単純です。

変数`uv`および`tsize`によって、テクスチャ上のどの範囲を画像とするかを指定しています。 変数`ssize`は、その画像をどれくらいの大きさで画面に表示するかを指定します。このパラメータがあるのは、テクスチャのサイズが一定とは限らないためです。

続けて、スプライト構造体を追加します。

```
/**
 \* スプライト.
 */
 struct Sprite
 {
   const Cell* cell; ///< 表示するCellデータ.
   DirectX::XMFLOAT3 pos; ///< スクリーン座標上のスプライトの位置.
   float rotation; ///< 画像の回転角(ラジアン).
   DirectX::XMFLOAT2 scale; ///< 画像の拡大率.
   DirectX::XMFLOAT4 color; ///< 画像の色.
 };
```

各メンバ変数の説明はコメントにあるとおりです。

3 スプライト描画クラス

3.1 まずは雛形から

DirectX 12において、描画はそれなりに複雑な作業です。こういった場合、複雑な作業をカプセル化して、簡単に扱えるようにしておきたいものです。そこで、スプライトの描画はクラスとして実装することにします。`Sprite`構造体定義の下に、次のコードを追加してください。

```
/**
 \* スプライト描画クラス.
 */
 class SpriteRenderer
 {
 public:
   SpriteRenderer();
   ~SpriteRenderer() = default;
 private:
 };
```

3.2 メンバ変数

このクラスにはどんなメンバ変数が必要でしょうか。まずはコマンドリストに関連する変数が必要ですね。

また、スプライトは長方形なので、一枚ごとに4つの頂点データが必要になります。一枚一枚が3Dゲームのように大量の頂点を持つわけではないため、事前に頂点データを準備して再利用する、というのはあまり効率的ではないように思えます。そこで、毎フレーム、全てのスプライトの頂点データを頂点バッファに詰め込んでいくことにします。描画中の頂点バッファを書き換えるのはまずいので、頂点バッファもフレームバッファと同じ数だけ用意することにします。 頂点データが変化しても、インデックスは常に0から順番に振られます。そのため、インデックスバッファは最初に作ったものを使いまわすことができます。

これらを踏まえて`private`メンバに追加するメンバ変数は以下のとおりです。

```
struct FrameResource
 {
   Microsoft::WRL::ComPtr<ID3D12CommandAllocator> commandAllocator;
   Microsoft::WRL::ComPtr<ID3D12Resource> vertexBuffer;
   D3D12_VERTEX_BUFFER_VIEW vertexBufferView;
   void* vertexBufferGPUAddress;
 };
 std::vector<FrameResource> frameResourceList;
 Microsoft::WRL::ComPtr<ID3D12GraphicsCommandList> commandList;
 Microsoft::WRL::ComPtr<ID3D12Resource> indexBuffer;
 D3D12_INDEX_BUFFER_VIEW indexBufferView;
 size_t maxSpriteCount;
```

今回は`FrameResource`という構造体を作り、フレームバッファごとに必要な変数を格納することにしました。 さらに、変数宣言に必要なヘッダファイルのインクルードも追加しましょう。

`DirectXMath.h`のインクルード文の下に、次のコードを追加してくだい。

```
#include <d3d12.h>
#include <wrl/client.h>
#include <vector>
```

3.3 メンバ関数

次はパブリックメンバ関数について考えていきます。

まずは、コマンドリストや頂点バッファといったデータを初期化する機能が必要です。初期化関数はD3Dデバイスインターフェイス、フレームバッファの数、最大スプライト数(頂点バッファのサイズを決定するため)を受け取ることになるでしょう。そうすると、初期化関数宣言は次のようになります。

```
bool Init(
   Microsoft::WRL::ComPtr<ID3D12Device> device,
   int numFrameBuffer,
   int maxSprite);
```

このクラスはスプライトを描画するものなので、少なくとも描画関数が必要です。描画関数は、スプライトのリストを受け取り、それを頂点データに変換して頂点バッファに格納します。そして、コマンドリストに描画コマンドを追加します。ということは、描画コマンドのための様々なパラメータも引数で受け取ることになります。 これらを考慮した描画関数宣言が次のコードです。

```
bool Draw(
   const std::vector<Sprite>& spriteList,
   const PSO& pso,
   const Texture& texture,
   int frameIndex,
   RenderingInfo& info);
```

スプライトを描画するには、スプライト用の`PSO`とコマンドリストにグラフィックスパイプラインの状態を設定してあげる必要があります。設定項目が多いので、`RenderingInfo`という型にまとめることにしましょう。また、この関数は`PSO`と`Texutre`の参照を使っているので、これらの先行宣言も加えておきましょう。`wrl/client.h`のインクルード文のあとに次のコードを追加します。

```
struct Texture;
struct PSO;
```

さて、`Draw`関数でコマンドリストしたら、それを取り出してコマンドキューに渡すことができなければなりません。 これは特に引数を取る必要はないので、次のような関数宣言になります。

```
ID3D12GraphicsCommandList* GetCommandList();
```

値を取得するだけの関数の場合、`const`を付与することが一般的です。しかし、コマンドリストを渡す先である`ID3D12CommandQueue:: ExecuteCommandLists`関数が非`const`なポインタを要求しているため、この関数宣言にも`const`を付与していません。

3.4 RenderingInfo構造体

`Draw`関数で使う描画情報をまとめた構造体を定義します。`Sprite`構造体定義の下に、次のコードを追加してください。

```
/**
 \* スプライト描画情報.
 */
struct RenderingInfo
{
    D3D12_CPU_DESCRIPTOR_HANDLE rtvHandle; ///< スプライト描画先レンダーターゲット.
    D3D12_CPU_DESCRIPTOR_HANDLE dsvHandle; ///< スプライト描画先深度バッファ.
    D3D12_VIEWPORT viewport; ///< 描画用ビューポート.
    D3D12_RECT scissorRect; ///< 描画用シザリング矩形.
    ID3D12DescriptorHeap* texDescHeap; ///< テクスチャ用のデスクリプタヒープ.
    DirectX::XMFLOAT4X4 matViewProjection; ///< 描画に使用する座標変換行列.
};
```

`Draw`関数の全ての引数をこの構造体に入れてもいいのですが、例えば複数の`SpriteRenderer`を使い、それぞれに異なる`PSO`やテクスチャを割り当てたいと考えるのはありうることです。そこで、そのようなパラメータは単独で引数としておき、その他の変更されにくいパラメータだけを含めることにしました。

4 Main.cppからPSOの作成を分離する

Draw関数は`PSO`を引数にしていますが、現在、`PSO`に関するコードは全て`Main.cpp`にあり、スプライト用のソースからは参照できません。そこで、`Src`フォルダに`PSO.h`と`PSO.cpp`を作成し、そこに`PSO`構造体の定義と作成に関するコードを移動させます。 その際、スプライト用の`PSO`タイプ定義と、作成した`PSO`を取得する関数も追加しておきます。

4.1 ヘッダファイル

`PSO.h`は次のようになります。

```
#pragma once
#include <d3d12.h>
#include <wrl/client.h>

/**
- ルートシグネチャとPSOをまとめた構造体.
*/
struct PSO
{
  Microsoft::WRL::ComPtr<ID3D12RootSignature> rootSignature;
  Microsoft::WRL::ComPtr<ID3D12PipelineState> pso;
};

/**
- PSOの種類.
*/
enum PSOType {
  PSOType_Simple,
  PSOType_CircleTexture,
  PSOType_Sprite,  // ←この値を追加.
  countof_PSOType
};
//                  ↓この部分に引数を追加.
bool CreatePSOList(Microsoft::WRL::ComPtr<ID3D12Device> device, bool warp);
const PSO& GetPSO(PSOType);
```

ヘッダファイルに移動したことにより`using`宣言がされなくなるため、`ComPtr`を名前空間で修飾しています。 また、ファイルを分けたことで`Main.cpp`で定義されている変数にはアクセスできなくなるため、`CreatePSOList`関数に引数を追加しています。

4.2 ソースファイル

ヘッダができたら、次は`PSO.cpp`です。変数`psoList`と`vertexLayout`、関数`LoadShader`、`CreatePSO`、`CreatePSOList`を移動してください。これらは既に定義済みなのでコードを再掲することはしませんが、変更点についていくつか説明しておきます。

- ヘッダと同様に`CreatePSOList`関数に引数を設定します。 同様に、`CretePSO`関数にも`device`と`warp`引数を追加しなければなりません。
- `PSOType_Sprite`用の`PSO`作成コードを追加します。 当面は`PSOType_Simple`と同じシェーダを指定しておけばよいでしょう。
- `GetPSO`関数を追加します。これは次のようなコードになるでしょう。

```
/**
- PSOを取得する.
- @param type 取得するPSOの種類.
- @return typeに対応するPSO.
*/
const PSO& GetPSO(PSOType type)
{
  return psoList[type];
}
```

`PSO`の作成コードの移動が完了したら、`Main.cpp`に`PSO.h`をインクルードして、変更した関数の呼び出しを修正しましょう。

5 SpriteRendererの実装

5.1 頂点データ型

それでは、スプライトの実装に戻りましょう。`Src`フォルダに`Sprite.cpp`を追加し、次のコードを追加してください。

```
#include "Sprite.h"
#include "Texture.h"
#include "PSO.h"
#include "d3dx12.h"

using Microsoft::WRL::ComPtr;
using namespace DirectX;

/**
- スプライト描画用頂点データ型.
*/
struct SpriteVertex {
  XMFLOAT3 position;
  XMFLOAT4 color;
  XMFLOAT2 texcoord;
};
```

最初に、必要となるヘッダファイルをインクルードしています。 その次には、`ComPtr`の`using`宣言とDirectX名前空間の`using`ディレクティブがあります。 ここまではおなじみですね。 `SpriteVertex`構造体の内容は`Main.cpp`の`Vertex`構造体と同じですが、あとで個別に変更できるように、スプライト専用の構造体を定義しています。

5.2 スプライトを頂点データに変換する

最初に実装する関数は、スプライトデータを頂点データに変換する関数です。`SpriteVertex`構造体定義の下に、次のコードを追加してください。

```
/**
- ひとつのスプライトデータを頂点バッファに設定.
- @param sprite スプライトデータ.
- @param v      頂点データを描き込むアドレス.
- @param offset スクリーン左上座標.
*/

void
AddVertex(const Sprite& s, SpriteVertex* v, XMFLOAT2 offset)
{
  const XMFLOAT2 center(offset.x + sprite.pos.x, offset.y - sprite.pos.y);
  const XMFLOAT2 halfSize(
    sprite.cell->ssize.x * 0.5f * sprite.scale.x,
    sprite.cell->ssize.y * 0.5f * sprite.scale.y);

  for (int i = 0; i < 4; ++i) {
    v[i].color = sprite.color;
    v[i].position.z = sprite.pos.z;
  }

  v[0].position.x = center.x - halfSize.x;
  v[0].position.y = center.y + halfSize.y;
  v[0].texcoord.x = sprite.cell->uv.x;
  v[0].texcoord.y = sprite.cell->uv.y;

  v[1].position.x = center.x + halfSize.x;
  v[1].position.y = center.y + halfSize.y;
  v[1].texcoord.x = sprite.cell->uv.x + sprite.cell->tsize.x;
  v[1].texcoord.y = sprite.cell->uv.y;

  v[2].position.x = center.x + halfSize.x;
  v[2].position.y = center.y - halfSize.y;
  v[2].texcoord.x = sprite.cell->uv.x + sprite.cell->tsize.x;
  v[2].texcoord.y = sprite.cell->uv.y + sprite.cell->tsize.y;

  v[3].position.x = center.x - halfSize.x;
  v[3].position.y = center.y - halfSize.y;
  v[3].texcoord.x = sprite.cell->uv.x;
  v[3].texcoord.y = sprite.cell->uv.y + sprite.cell->tsize.y;
}
```

この関数は、スプライトの座標を画像の中心点として、左上から時計回りに4つの頂点データを作成し、頂点データに格納します。

5.3 コンストラクタ

ここからは`SpriteRenderer`クラスを実装していきます。最初はコンストラクタです。 `AddVertex`関数の下に、次のコードを追加してください。

```
/**
 \* コンストラクタ
 */
 SpriteRenderer::SpriteRenderer() :
   maxSpriteCount(0)
 {
 }
```

コンストラクタでは、`maxSpriteCount`を0に初期化しているだけです。

5.4 初期化関数

次は`Init`関数です。 コンストラクタ定義の下に、次のコードを追加してください。

```
/**
- Rendererを初期化する.
*
- @param device           D3Dデバイス.
- @param frameBufferCount フレームバッファの数.
- @param maxSprite        描画できる最大スプライト数.
*
- @retval true  初期化成功.
- @retval false 初期化失敗.
*/

bool SpriteRenderer::Init(
  ComPtr<ID3D12Device> device, int frameBufferCount, int maxSprite)
{
  return true;
}
```

最初は、フレームバッファ毎の変数を初期化します。以下のコードをInit関数の先頭に追加してください。

```
frameResourceList.resize(frameBufferCount);
 for (int i = 0; i < frameBufferCount; ++i) {
   if (FAILED(device->CreateCommandAllocator(
     D3D12_COMMAND_LIST_TYPE_DIRECT,
     IID_PPV_ARGS(&frameResourceList[i].commandAllocator)))) {
     return false;
   }
   if (FAILED(device->CreateCommittedResource(
     &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
     D3D12_HEAP_FLAG_NONE,
     &CD3DX12_RESOURCE_DESC::Buffer(maxSprite * sizeof(SpriteVertex)),
     D3D12_RESOURCE_STATE_GENERIC_READ,
     nullptr,
     IID_PPV_ARGS(&frameResourceList[i].vertexBuffer)))) {
     return false;
```

```
    }
    frameResourceList[i].vertexBuffer->SetName(L"Sprite Vertex Buffer");
    CD3DX12_RANGE range(0, 0);
    if (FAILED(frameResourceList[i].vertexBuffer->Map(
      0,
      &range,
      &frameResourceList[i].vertexBufferGPUAddress))) {
      return false;
    }
    frameResourceList[i].vertexBufferView.BufferLocation =
      frameResourceList[i].vertexBuffer->GetGPUVirtualAddress();
    frameResourceList[i].vertexBufferView.StrideInBytes = sizeof(SpriteVertex);
    frameResourceList[i].vertexBufferView.SizeInBytes =
      static_cast<UINT>(maxSprite * sizeof(SpriteVertex));
  }
```

どのコードも既に実装してきたものなので、難しくはないと思います。 興味深い点としては、`ID3D12Resource::Map`関数を呼び出しだけで、`Unmap`関数がないことが挙げられます。 こうすることで、頂点データ書き込み用のポインタをずっと有効にしておくことができます。

続いて、コマンドリストを作成します。 フレームリソース作成コードの下に、以下のコードを追加してください。

```
if (FAILED(device->CreateCommandList(
    0,
    D3D12_COMMAND_LIST_TYPE_DIRECT,
    frameResourceList[0].commandAllocator.Get(),
    nullptr,
    IID_PPV_ARGS(&commandList)))) {
    return false;
  }
  if (FAILED(commandList->Close())) {
    return false;
  }
```

このコードも、`InitializeD3D`関数で実装したものとほとんど同じです。

最後に、インデックスバッファを作成します。 コマンドリスト作成コードに続けて、次のコードを追加してください。

```
if (FAILED(device->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
    D3D12_HEAP_FLAG_NONE,
    &CD3DX12_RESOURCE_DESC::Buffer(maxSprite * 6 * sizeof(DWORD)),
    D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(&indexBuffer)))) {
    return false;
  }
  indexBuffer->SetName(L"Sprite Index Buffer");
  CD3DX12_RANGE range(0, 0);
  void* tmpIndexBufferAddress;
```

```
  if (FAILED(indexBuffer->Map(0, &range, &tmpIndexBufferAddress))) {
    return false;
  }
  DWORD* pIndexBuffer = static_cast<DWORD*>(tmpIndexBufferAddress);
  for (size_t i = 0; i < maxSprite; ++i) {
    pIndexBuffer[i * 6 + 0] = i * 4 + 0;
    pIndexBuffer[i * 6 + 1] = i * 4 + 1;
    pIndexBuffer[i * 6 + 2] = i * 4 + 2;
    pIndexBuffer[i * 6 + 3] = i * 4 + 2;
    pIndexBuffer[i * 6 + 4] = i * 4 + 3;
    pIndexBuffer[i * 6 + 5] = i * 4 + 0;
  }
  indexBuffer->Unmap(0, nullptr);
  indexBufferView.BufferLocation = indexBuffer->GetGPUVirtualAddress();
  indexBufferView.SizeInBytes = maxSprite * 6 * sizeof(DWORD);
  indexBufferView.Format = DXGI_FORMAT_R32_UINT;
```

インデックスバッファの作成も、`InitializeD3D`関数で実装したものと大きな違いはありません。インデックス配列を自動的に生成しているくらいですね。頂点バッファと違い、インデックスバッファは作成したらもう書き換えることはないので`Unmap`を呼んでいます。

5.5 描画関数

`Init`関数の次は`Draw`関数を実装します。`Init`関数の下に、次のコードを追加してください。

```
/**
- スプライトを描画する.
*
- @param spriteList 描画するスプライトのリスト.
- @param pso        描画に使用するPSO.
- @param texture    描画に使用するテクスチャ.
- @param frameIndex 現在のフレームバッファのインデックス.
- @param info       描画情報.
*
- @retval true  コマンドリスト作成成功.
- @retval false コマンドリスト作成失敗.
*/
bool SpriteRenderer::Draw(
  const std::vector<Sprite>& spriteList,
  const PSO& pso,
  const Resource::Texture& texture,
  int frameIndex,
  RenderingInfo& info)
{
  return true;
}
```

まず最初に、コマンドリストをリセットします。`Draw`関数の先頭に、次のコードを追加してください。

```
FrameResource& fr = frameResourceList[frameIndex];
 if (FAILED(fr.commandAllocator->Reset())) {
   return false;
 }
 if (FAILED(commandList->Reset(fr.commandAllocator.Get(), nullptr))) {
   return false;
 }
 if (spriteList.empty()) {
   return SUCCEEDED(commandList->Close());
 }
```

最初の行では、描画で使用するフレームリソースを選択しています。そして、コマンドアロケータとコマンドリストをリセットしています。また、スプライトリストが空だった場合、描画する必要がないので、即座にコマンドリストをクローズして関数を終了しています。

次に行うのは、グラフィックスパイプラインの設定です。 上記のコードに続けて、次のコードを追加してください。

```
commandList->SetGraphicsRootSignature(pso.rootSignature.Get());
 commandList->SetPipelineState(pso.pso.Get());
 ID3D12DescriptorHeap* heapList[] = { info.texDescHeap };
 commandList->SetDescriptorHeaps(_countof(heapList), heapList);
 commandList->SetGraphicsRootDescriptorTable(0, texture.handle);
 commandList->SetGraphicsRoot32BitConstants(1, 16, &info.matViewProjection, 0);
 commandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
 commandList->IASetVertexBuffers(0, 1, &fr.vertexBufferView);
 commandList->IASetIndexBuffer(&indexBufferView);
 commandList->OMSetRenderTargets(1, &info.rtvHandle, FALSE, &info.dsvHandle);
 commandList->RSSetViewports(1, &info.viewport);
 commandList->RSSetScissorRects(1, &info.scissorRect);
```

個々のコードは`Render`関数の実装過程で説明済みなので、ここでは繰り返しません。

グラフィックスパイプラインの設定が終わったら、頂点データを作成します。 グラフィックスパイプライン設定コードの下に、次のコードを追加してください。

```
const XMFLOAT2 offset(-(info.viewport.Width * 0.5f), info.viewport.Height * 0.5f);
 const int maxSprite =
   fr.vertexBufferView.SizeInBytes / fr.vertexBufferView.StrideInBytes / 4;
 int numSprite = 0;
 SpriteVertex* v = static_cast<SpriteVertex*>(fr.vertexBufferGPUAddress);
 for (const Sprite& sprite : spriteList) {
   AddVertex(sprite, v, offset);
   ++numSprite;
   if (numSprite >= maxSprite) {
     break;
   }
   v += 4;
 }
 commandList->DrawIndexedInstanced(numSprite * 6, 1, 0, 0, 0);
 if (FAILED(commandList->Close())) {
   return false;
```

```
  }
```

変数`offset`には、描画範囲の左上を示す座標が格納されます。スクリーン座標系では右上がプラス方向、左下がマイナス方向であることを思い出してください。

変数`maxSprite`には、描画可能な最大スプライト数が格納されます。このコードでは、頂点バッファビューのパラメータから、`Init`関数で渡された値を計算で復元しています。

変数`numSprite`は描画したスプライトの数が格納されます。最初は0です。

変数`v`は頂点データを設定する位置を示すポインタです。最初は頂点バッファの先頭を指しています。

`for`ループの中では、`AddVertex`関数によってスプライトリストの各スプライトを頂点データに変換し、頂点バッファに格納しています。 頂点データを作成し終えたら、`ID3D12GraphicsCommandList::DrawIndexedInstanced`関数で、作成した頂点データを描画しています。 そして最後に、コマンドリストをクローズして描画関数は終了です。

5.6 コマンドリストの取得

`SpriteRenderer`クラスの最後は、コマンドリストの取得関数の実装です。 `Draw`関数の下に、次のコードを追加してください。

```
/**
 \* コマンドリストを取得する.
 *
 \* @return ID3D12GraphicsCommandListインターフェイスへのポインタ.
 */
 ID3D12GraphicsCommandList* SpriteRenderer::GetCommandList()
 {
   return commandList.Get();
 }
```

見てのとおり、コマンドリストのポインタを返しているだけの単純な関数です。

これで`SpriteRenderer`の実装は完了です。

6 リソースバリアの位置

現在、コマンドリストの先頭と最後には、フレームバッファの状態が変化するのを待つためのリソースバリア命令が追加されていると思います。複数のコマンドリストを実行する場合、2つのリソースバリア命令が、それぞれ最初のコマンドリストと最後のコマンドリストに追加された状態になっている必要があります。 コマンドリストを追加するたびに、これらの命令の追加位置を変更することはできます。しかし、コマンドリストを追加・削除したりの、順序を変更するたびにそんなことをするのは面倒です。

そこで、必ず最初と最後に実行されるコマンドリストを作成し、そこにリソースバリア命令を追加することにしましょう。こうすることで、他のコマンドリストの数や順序がいくら変わっても、リソースバリア命令の位置を気にする必要はなくなります。

`Main.cpp`を開き、`commandList`変数定義の下に次のコードを追加してください。

```
ComPtr<ID3D12GraphicsCommandList> prologueCommandList; // 最初に実行されるコマンドリスト.
 ComPtr<ID3D12GraphicsCommandList> epilogueCommandList; // 最後に実行されるコマンドリスト.
```

次は初期化です。`InitializeD3D`関数へ移動し、`commandList`作成コードの下に、次のコードを追加してください。

```
if (FAILED(device->CreateCommandList(
   0, D3D12_COMMAND_LIST_TYPE_DIRECT, commandAllocator[currentFrameIndex].Get(), nullptr,
   IID_PPV_ARGS(&prologueCommandList)))) {
   return false;
 }
 if (FAILED(prologueCommandList->Close())) {
   return false;
 }
 if (FAILED(device->CreateCommandList(
   0, D3D12_COMMAND_LIST_TYPE_DIRECT, commandAllocator[currentFrameIndex].Get(), nullptr,
   IID_PPV_ARGS(&epilogueCommandList)))) {
   return false;
 }
 if (FAILED(epilogueCommandList->Close())) {
   return false;
 }
```

ここでは直にコマンドリストを作成していますが、作成コードを関数化するほうがいいと思います。

新しいコマンドリストが作成できたので、これを使うコードを実装しましょう。`Render`関数に移動し、コマンドアロケータをリセットするコードの下に、次のコードを追加してください。

```
if (FAILED(prologueCommandList->Reset(
   commandAllocator[currentFrameIndex].Get(), nullptr))) {
   return false;
 }
 prologueCommandList->ResourceBarrier(1,  &CD3DX12_RESOURCE_BARRIER::Transition(
   renderTargetList[currentFrameIndex].Get(),
   D3D12_RESOURCE_STATE_PRESENT,
   D3D12_RESOURCE_STATE_RENDER_TARGET));
 if (FAILED(prologueCommandList->Close())) {
   return false;
 }

 if (FAILED(epilogueCommandList->Reset(
   commandAllocator[currentFrameIndex].Get(), nullptr))) {
   return false;
 }
 epilogueCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
   renderTargetList[currentFrameIndex].Get(),
   D3D12_RESOURCE_STATE_RENDER_TARGET,
   D3D12_RESOURCE_STATE_PRESENT));
 if (FAILED(epilogueCommandList->Close())) {
   return false;
```

```
 }
```

さらに、既存のコマンドリストからリソースバリア命令を取り除いてください。 削除するひとつめのリソースバリア命令は、次のコードでコメントアウトされている部分です。

```
if (FAILED(commandList->Reset(commandAllocator[currentFrameIndex].Get(), nullptr))) {
   return false;
 }

 // ↓この関数呼び出しを削除.
 // commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
 //   renderTargetList[currentFrameIndex].Get(),
 //   D3D12_RESOURCE_STATE_PRESENT,
 //   D3D12_RESOURCE_STATE_RENDER_TARGET));
 D3D12_CPU_DESCRIPTOR_HANDLE dsvHandle =
   dsvDescriptorHeap->GetCPUDescriptorHandleForHeapStart();
 D3D12_CPU_DESCRIPTOR_HANDLE rtvHandle =
   rtvDescriptorHeap->GetCPUDescriptorHandleForHeapStart();
```

削除するふたつめのリソースバリア命令は、次のコードでコメントアウトされている部分です。

```
DrawTriangle();
 DrawRectangles();

 // ↓この関数呼び出しを削除.
 // commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
 //   renderTargetList[currentFrameIndex].Get(),
 //   D3D12_RESOURCE_STATE_RENDER_TARGET,
 //   D3D12_RESOURCE_STATE_PRESENT));

 if (FAILED(commandList->Close())) {
   return false;
 }
```

最後に、コマンドキューに渡すコマンドリストの配列に、今回作成したコマンドリストを追加します。 次のコードを、

```
ID3D12CommandList* ppCommandLists[] = { commandList.Get() };
```

以下のように書き換えてください。

```
ID3D12CommandList* ppCommandLists[] = {
   prologueCommandList.Get(),
   commandList.Get(),
   epilogueCommandList.Get() };
```

7 スプライトを使う

7.1 変数定義

それでは、実際にスプライトを描画してみます。 最初に、スプライトのリストとスプライト描画オブジェクトを定義します。 `Main.cpp`を開き、`Texture`の変数宣言の下に、次の変数宣言を追加してください。

```
std::vector<Sprite> spriteList;
 SpriteRenderer spriteRenderer;
```

7.2 セルの定義

次に、セルデータを準備します。 `triangleVertexCount`変数宣言の下に、次のコードを追加してください。

```
const Cell cellList[] = {
   { XMFLOAT2(0, 0), XMFLOAT2(1, 1), XMFLOAT2(80, 60) },
 };
```

7.3 SpriteRendererの初期化

`InitializeD3D`関数に移動し、`LoadTexture`関数呼び出しの下に、次のコードを追加してください。

```
if (!spriteRenderer.Init(device, frameBufferCount, 10000)) {
   return false;
 }
 spriteList.push_back(
   { &cellList[0], XMFLOAT3(100, 100, 0.1f), 0, XMFLOAT2(1, 1), XMFLOAT4(1, 1, 1, 1) });
```

ここではスプライトの最大数を10000としています(数値は適当です)。 また、スプライトリストに設定しているパラメータも適当です。

7.4 スプライトを描画

`Render`関数にスプライトの描画コードを追加します。 `DrawRectangles`関数呼び出しの下に、次のコードを追加してください。

```
RenderingInfo spriteRenderingInfo;
 spriteRenderingInfo.rtvHandle = rtvHandle;
 spriteRenderingInfo.dsvHandle = dsvHandle;
 spriteRenderingInfo.viewport = viewport;
 spriteRenderingInfo.scissorRect = scissorRect;
 spriteRenderingInfo.texDescHeap = csuDescriptorHeap.Get();
 spriteRenderingInfo.matViewProjection = matViewProjection;
 spriteRenderer.Draw(
   spriteList,
   GetPSO(PSOType_Sprite),
   texImage,
   currentFrameIndex,
   spriteRenderingInfo);
```

また、コマンドキューにスプライト用コマンドリストを渡す必要もあります。 以下のように、コマンドリストの配列にスプライト用コマンドリストを追加してください。

```
ID3D12CommandList* ppCommandLists[] = {
   prologueCommandList.Get(),
   commandList.Get(),
   spriteRenderer.GetCommandList(), // ←この行を追加.
   epilogueCommandList.Get() };
```

これでスプライトを表示できるようになりました。

プロジェクトをビルドして実行してみてください。

8 キー入力

8.1 入力を処理する

スプライトを表示するだけでは面白くないので、キーボードの入力で操作できるようにしてみましょう。 まずは`spriteRenderer`変数定義の下に、次のコードを追加してください。

```
/**
 \* ゲームパッド入力を表す構造体.
 */
 struct GamePad
 {
   enum {
     DPAD_UP = 0x0001,
     DPAD_DOWN = 0x0002,
     DPAD_LEFT = 0x0004,
     DPAD_RIGHT = 0x0008,
     START = 0x0010,
     A = 0x0020,
     B = 0x0040,
     X = 0x0080,
     Y = 0x0100,
     L = 0x0200,
```

```
    R = 0x0400,
  };
  uint32_t buttons;
};
GamePad gamepad;
```

`GaePad`はXInputに似た構造体です。キー入力があったときに`buttons`メンバ変数の対応するビットを立て、入力がなくなったらビットを降ろします。ビットの操作は、ウィンドウプロシージャで行います。ウィンドウプロシージャを以下のように変更してください。

```
HRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wparam, LPARAM lparam)
{
  switch (msg) {
  case WM_KEYDOWN:
    if (wparam == VK_ESCAPE) {
      DestroyWindow(hwnd);
      return 0;
    }
     // ↓このswitch文を追加.
    switch (wparam) {
    case 'W': gamepad.buttons |= GamePad::DPAD_UP; break;
    case 'A': gamepad.buttons |= GamePad::DPAD_LEFT; break;
    case 'S': gamepad.buttons |= GamePad::DPAD_DOWN; break;
    case 'D': gamepad.buttons |= GamePad::DPAD_RIGHT; break;
    case VK_SPACE: gamepad.buttons |= GamePad::A; break;
    }
    break;
  // ↓このcaseを追加.
  case WM_KEYUP:
    switch (wparam) {
    case 'W': gamepad.buttons &= ~GamePad::DPAD_UP; break;
    case 'A': gamepad.buttons &= ~GamePad::DPAD_LEFT; break;
    case 'S': gamepad.buttons &= ~GamePad::DPAD_DOWN; break;
    case 'D': gamepad.buttons &= ~GamePad::DPAD_RIGHT; break;
    case VK_SPACE: gamepad.buttons |= GamePad::A; break;
    }
    break;
  case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
   }
   return DefWindowProc(hwnd, msg, wparam, lparam);
}
```

キー入力は一般的なFPSと同じくWASDに割り振ってみました。 以前説明したように、`WM_KEYDOWN`メッセージでは`wparam`に押されたキーの種類が渡されてきます。 そこで、`wparam`の値を見て、対応する`gamepad.buttons`のビットを立てています。 `WM_KEYUP`はキーが離されたときに送られてくるウィンドウメッセージです。このメッセージも`wparam`に離されたキーの種類が渡されてくるので、`WM_KEYDOWN`と同様の方法で対応するビットを降ろしています。

8.2 状態更新関数を追加する

キー入力を判定して状態を更新する処理は、専用の関数で行うことにします。 まずは関数宣言です。
`WaitForGpu`関数宣言の下に次のコードを追加してください。

```
void Update();
```

この関数は、メッセージループ内で呼び出します。 メッセージループの`Render`関数呼び出しの直前に、次のコードを追加してください。

```
Update();
```

最後に関数定義です。 `WaitForGpu`関数定義の下に、次のコードを追加してください。

```
/**
 \* アプリケーションの状態を更新する.
 */
 void Update()
 {
   if (gamepad.buttons & GamePad::DPAD_LEFT) {
     spriteList[0].pos.x -= 5.0f;
   }  else if (gamepad.buttons & GamePad::DPAD_RIGHT) {
     spriteList[0].pos.x += 5.0f;
   }
   if (gamepad.buttons & GamePad::DPAD_UP) {
     spriteList[0].pos.y -= 5.0f;
   }  else if (gamepad.buttons & GamePad::DPAD_DOWN) {
     spriteList[0].pos.y += 5.0f;
   }
 }
```

このコードでは、キー入力に応じて、前章までに定義したスプライトの座標を変更しています。 画面端のチェック等は行っていませんが、キー入力の雰囲気はつかめると思います。

これでキー入力の実装は完了です。 ビルドして実行してみてください。

9 やってみよう

- セルデータやスプライトを追加してみましょう。
- スプライトの回転や拡大縮小パラメータは未使用です。これらを使ってスプライトの回転拡大縮小を実装してみましょう。
- `GamePad`の未使用ビットを適当なキーに割り当ててみましょう。また、割り当てたキーを使ってスプライトを操作してみましょう。
- 2プレイヤー用の`GamePad`を実装してみましょう。