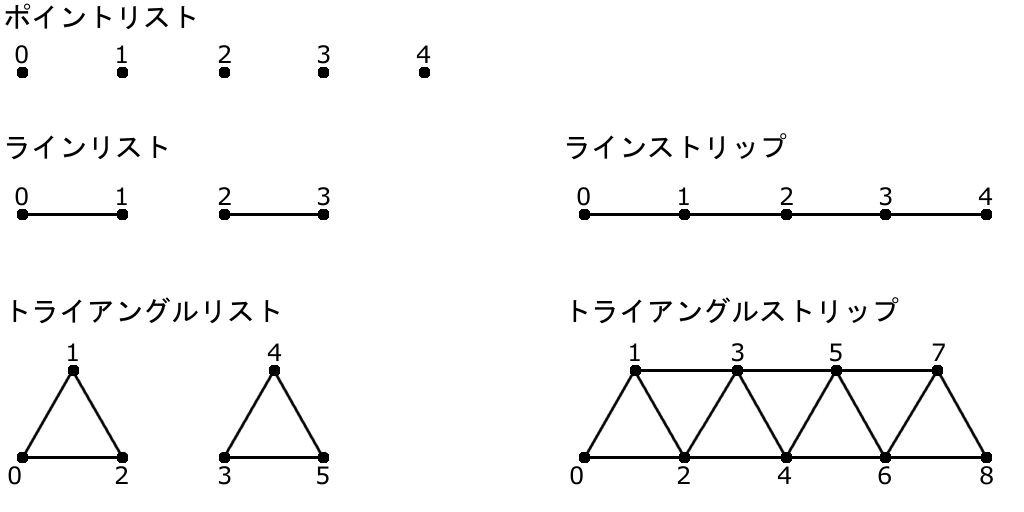
Re:頂点シェーダとピクセルシェーダが  
両方そなわり三角形が見える

# 何が描けるの？

前回はDirectX 12の初期化を行いました。しかし、できることといえば背景色を変える程度…。ゲームを作るなら、何か、もっと意味のあるものを描画できるようにする必要があります。  
とはいえ、意味のあるもの、というのは漠然としすぎていますね。まずは、DirectX 12がどんなものを描画できるのかを見てみましょう。

DirectX 12が描画できるのは、以下の5種類の図形です。



「ポイントリスト」は点の集まりです。1ドットの点を描くのに使います。  
「ラインリスト」は直線の集まりで、「ラインストリップ」は連続した直線です。どちらも幅1ドットの直線を引くことができます。  
「トライアングルリスト」は三角形の集まりで、「トライアングルストリップ」は連続した三角形です。三角形をたくさん描くのに使います。  
これらの図形のことを「プリミティブ」と呼びます(プリミティブというのは「原始的な」という意味の英単語)。なお、四角形より角の多い図形はありません。そのような図形は、三角形を組み合わせて表現することになります。

世にあるゲームに出てくるキャラクターや背景のほとんどは、どんなに複雑な形状であっても、上記のプリミティブを組み合わせて描画しています。

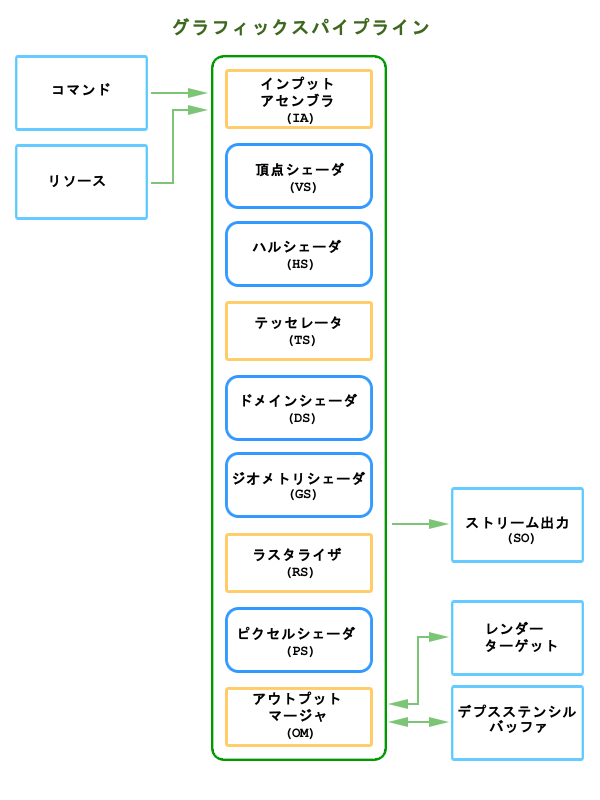
さて、本講義では「トライアングルリスト」を使って描画の手順を説明していくことにします。トライアングルリストを選んだのは、このプリミティブが最も利用範囲が広いからです。  
いずれのプリミティブも、描画方法に大きな違いはないので、どれかひとつのプリミティブを描画できるようになれば、他のプリミティブを描画するのは難しくありません。

この他に「隣接頂点付きのプリミティブ」というものもあるのですが、この講義では扱いません。このタイプのプリミティブは「ジオメトリシェーダ」で使われます。

# グラフィックスパイプライン

DirectX 12では、プリミティブという図形を描画することが分かりました。早速描画しよう！と行きたいところですが、その前に、DirectXがどのような手順でプリミティブを描画するのかを説明します。DirectXの描画は、それぞれが異なる処理を行う「ステージ」という単位に分けられています。ステージが順番に実行されることで、最終的にフレームバッファにプリミティブが描画されることになります。  
この、ステージが順番に実行される仕組みのことを「グラフィックスパイプライン」と呼びます。

次の図はグラフィックスパイプラインがどのようなものかをを示しています。



中央の枠内がグラフィックスパイプラインです。  
左右にあるのは、グラフィックスパイプラインに設定するパラメータです。両者を接続する矢印は、データが送られる向きを示しています。

グラフィックスパイプラインのうち、四角い枠は機能が固定されていて、決められた処理を行うためのステージです。丸い角の枠はシェーダによって処理内容をプログラムできるステージです。シェーダのプログラミングには、High Level Shading Language(HLSL)というMicrosoftが開発した言語を使います。

グラフィックスパイプラインの処理は上から下に向かって流れていきます。  
まず、左側の「コマンド」と「リソース」がパイプラインの先頭にある「インプットアセンブラ」ステージに送られます。「コマンド」というのは、前回出てきたコマンドリストやコマンドキューによってGPUに送られる命令群のことです。「リソース」は頂点データやテクスチャなどです。最後に、「アウトプットマージャ」がレンダーターゲットやデプスステンシルバッファへプリミティブを書き込みます。  
「ハルシェーダ」ステージ～「ドメインシェーダ」ステージは、全体でひとつの機能を実装しているため、まとめて「テッセレーション」ステージと呼ばれます。  
「テッセレーション」ステージと「ジオメトリシェーダ」ステージは、使わない場合は省略することができます。

それでは、各ステージの機能を見ていきましょう。

## インプットアセンブラ(IA)

インプットアセンブラは、グラフィックスパイプラインの最初のステージです。これは固定機能のステージで、与えられたリソースから頂点データを読み取って、後段のステージへと送ります。プログラマは「インプットレイアウト」というパラメータを使って、IAに頂点データの読み取り方を教えます。

## 頂点シェーダ(VS)

頂点シェーダは最初のプログラム可能なステージです。IAで読み取られた全ての頂点データは、最初にこのステージで処理されます。VSでは、プリミティブの回転、拡大縮小、移動のほか、座標系の変換、頂点単位のライティングなど、様々な操作を行うことができます。頂点シェーダステージは省略できません。

## ハルシェーダ(HS)

ハルシェーダは「テッセレーション」ステージの最初のステージです。テッセレーションステージはこのHSを含めて3つのステージで構成され、共同で「テッセレーション」と呼ばれる処理を行います。テッセレーションとは、描画する画像に細かなディテールを加えるために、プリミティブをより多くの小さなプリミティブに分割する操作のことです。  
テッセレーションによって作られた小さなプリミティブは、通常はグラフィックスパイプラインで作られ、またグラフィックスパイプラインの中で使用されます。そのため、リソースとして作られるプリミティブと違い、CPU時間やGPUメモリを消費しません。例えば、リソースとして低ポリゴンモデルを与え、テッセレーションステージで高ポリゴンモデルを生成する、ということができます。  
HSはテッセレーションステージにおいて、新たに作成する小さなプリミティブの頂点データを、どこに、どのように作成するかを指定する作業を担当します。HSもVSと同様にプログラム可能なステージです。  
さて、この講義ではテッセレーションについては扱いません。そのため、HSについてもこれ以上詳細な説明はしませんが、このようなステージがあるということは覚えておいてください。

## テッセレータ(TS)

テッセレータは「テッセレーション」ステージの2番めにあたり、また、IAと同様の固定機能ステージになります。  
TSはHSからの情報を受け取り、実際に小さなプリミティブへと分割し、後段のドメインシェーダへと送ります。

## ドメインシェーダ(DS)

ドメインシェーダは最後の「テッセレーション」ステージです。DSはプログラム可能なステージで、TSから受け取ったプリミティブを変形し、細かなディテールを加える役割を持ちます。TSから送られてきたプリミティブは、元のプリミティブを分割しただけの状態なので、このステージで頂点の座標や色を調整します。

## ジオメトリシェーダ(GS)

ジオメトリシェーダはひとつのプリミティブの全頂点データを受け取り、0個以上のプリミティブの頂点データを生成して後段のステージへ送ります。頂点データの削除ができるのはこのステージだけです。  
プリミティブを増やすという意味では「テッセレーション」ステージと似ていますが、テッセレーションは「元のプリミティブを分割する」という処理に最適化され、大量のプリミティブを生成するのに向いているのに対して、GSは「プリミティブの頂点データから別のプリミティブを生成する、あるいは削除する」という、より小規模で汎用的な処理を目的としています。  
GSはプログラム可能なステージです。また、テッセレーションステージと同様に省略することができます。  
この講義ではGSは扱いませんが、GSはパーティクルの描画や輪郭線の生成など、様々な用途で使われていますので、調べてみると面白いと思います。

## ラスタライザ(RS)

ラスタライザステージは、前段のステージから送られたプリミティブの頂点データからピクセルデータを生成し、後段のピクセルシェーダステージへと送ります。RSは固定機能ステージです。  
各ピクセルデータは頂点データを補間した値を持ちます。RSはまた、裏向きのプリミティブを除去したり(カリングといいます)、プリミティブの画面に表示されない部分を切り捨てたり (クリッピングといいます) します。クリッピングは「ビューポート」というパラメータを元に実行されます。ビューポートについては後のコードで説明します。

## ピクセルシェーダ(PS)

ピクセルシェーダは最後のプログラム可能なステージです。RSから送られたピクセルデータから、実際の色を計算します。頂点シェーダがひとつの頂点データを受けとり、ひとつの頂点最後のを送り出すように、PSもひとつのピクセルデータを受け取り、ひとつのピクセルデータを送り出します。  
PSは主にピクセル単位のライティングを行うために使用されます。実際、PSが後段のステージに送り出すのは4要素の色情報と、前段のステージから送られてきた深度情報のみです。

## アウトプットマージャ(OM)

グラフィックスパイプラインの最後のステージはアウトプットマージャです。OMは固定機能のステージで、PSから送られてきた色情報と深度情報をレンダーターゲットやデプスステンシルバッファに書き込んでくれます。  
OMでは、PSからの色情報とレンダーターゲットの色を混合することもできます。

## ストリーム出力(SO)

ストリーム出力は、ジオメトリシェーダの直後に位置する特別なステージで、グラフィックスパイプラインで計算された頂点データを、メモリに書き戻すことができます。この頂点データは再びグラフィックスパイプラインに送り込むことも、別の用途のためにプログラマが準備したバッファに残しておくこともできます。  
このステージは講義では扱いません。

DirectXにおいて、描画するということは、グラフィックスパイプラインに頂点データやパラメータを送り込むということです。  
本講義では、後ほど実際に頂点シェーダとピクセルシェーダをプログラムしてもらいます。

# インクルードファイルの追加

それでは、新しいコードで使うヘッダファイルをインクルードするところから始めましょう。次のコードをMain.cppの既存のインクルード文のあとに追加してください。

#include <d3dcompiler.h>  
#include <DirectXMath.h>

d3dcompiler.hはシェーダをコンパイルする関数が宣言されているヘッダファイルです。プリミティブの描画にはシェーダを書いてコンパイルする必要があるのでインクルードしています。  
DirectXMath.hは、ベクトルや行列、またそれらを扱う数学関数が宣言されているヘッダファイルです。プリミティブの座標を扱うためにインクルードします。

# using namespace

さきほどインクルードしたDirectXMathは便利なのですが、全てのメンバがDirectXという名前空間の中にあるため、使うときにはいちいち「DirectX::」を書かないといけません。これは面倒だしコードが読みにくくなってしまいます。幸い、C++にはこれを書かなくて済むようにする方法が用意されています。前章のインクルード文の次の行に、以下のコードを追加してください。

using namespace DirectX;

これはusingディレクティブというもので、指定された名前空間のメンバを現在の位置に取り込みます。上のコードはこの機能を使って「これからDirectX名前空間のメンバは「DirectX::」を書かなくても使っていいことにします」とコンパイラに伝えています。

# 変数宣言と関数宣言

次は今回の講義で使用する変数を宣言します。既存の変数宣言の最後に、以下のコードを追加してください。

ComPtr<ID3D12RootSignature> rootSignature;  
ComPtr<ID3D12PipelineState> pso;  
ComPtr<ID3D12Resource> vertexBuffer;  
D3D12\_VERTEX\_BUFFER\_VIEW vertexBufferView;  
D3D12\_VIEWPORT viewport;  
D3D12\_RECT scissorRect;

ID3D12RootSignatureは、シェーダに入力データを結びつけるためのインターフェイスです。ここでいう入力データとは、テクスチャの読み取り方法や、座標変換のための行列、ボーンアニメーションのためのボーン配列、といったものです。

ID3D12PipelineStateは、パイプライン・ステート・オブジェクト(PSO)と呼ばれるオブジェクトのインターフェイスです。PSOは、DirectXのグラフィックスパイプラインのパラメータをまとめたものです。パラメータは、描画に使用するシェーダや、描画するプリミティブの種類、プリミティブのデータ形式、描画先のフォーマットなど多岐にわたります。

ID3D12Resourceはこれまでにもレンダーターゲットなどで使用しましたが、今回はプリミティブのデータを格納するために使用します。プリミティブのデータを格納するオブジェクトを「頂点バッファ」と呼びます。  
D3D12\_VERTEX\_BUFFER\_VIEWは、頂点バッファのパラメータ設定するための構造体です。DirectXへプリミティブのデータを渡すために使います。

D3D12\_VIEWPORTは、レンダーターゲットの実際に描画に使用する範囲を設定する構造体です。左半分にはプレイヤー1視点からの画像、右半分にはプレイヤー2視点からの画像を表示する、などという使い方ができます。

D3D12\_RECTは、範囲を示す汎用の構造体ですが、ここでは描画を許可する範囲を指定するために使います。D3D12\_VIEWPORT構造体と組み合わせることで、さまざまな描画範囲を設定できます。

続けて、今回新しく定義する関数を宣言しましょう。以下のコードを、既存の関数宣言の下に追加してください。

bool LoadShader(const wchar\_t\*, const char\*, ID3DBlob\*\*);  
bool CreatePSO();  
bool CreateVertexBuffer();  
void DrawTriangle();

今回定義する関数は、シェーダを読み込む関数、PSOを作成する関数、プリミティブデータの格納場所を作成する関数、プリミティブを描画する関数、の4つです。

# プリミティブと頂点

全てのプリミティブは頂点の集まりです。1章の図で、番号の振られた点のひとつひとつが頂点です。前述のとおり、DirectXのプリミティブの種類は限定されていますが、頂点にはかなりの自由が認められています。例えば、ひとつの頂点には最大で64個のデータを割り当てることができます。また、それぞれのデータは8bitから64bitまで様々な型を使うことができます。

さて、DirectXでプリミティブを描画するには、頂点データそのものと、データがどのような順序で格納されていて、それぞれ何を表しているのかをDirectXに伝えるパラメータの2つが必要となります(このようなパラメータのことを「レイアウト」と呼びます。)。

また、頂点データは例えばfloat型の配列などでも構わないのですが、読みにくいうえ、様々な型が使えるという利点を生かせません。そこで、頂点データを表す構造体を定義することにします。前章の関数宣言のあとに、以下のコードを追加してください。

// 頂点データ型.  
struct Vertex  
{  
 XMFLOAT3 position; // 座標  
 XMFLOAT4 color; // 色  
};

ここでは、ひとつの頂点データを表すVertexという構造体を定義しています(Vertexは「頂点」という意味の英単語です)。構造体の最初のメンバpositionは、頂点の座標を表しています。2つめのメンバcolorは頂点の色を表します。名前のままですね。  
さて、XMFLOAT3, XMFLOAT4というのはヘッダファイルDirectXMath.hに定義されている構造体で、それぞれ3または4要素のfloat型を格納することができます。座標や色、あるいはベクトルといったものを表現するために使われることが多いです。  
頂点データ型を定義したついでに、レイアウトも定義してしまいましょう。  
上記のコードに続けて、次のコードを追加してください。

// 頂点データのレイアウト情報.  
const D3D12\_INPUT\_ELEMENT\_DESC vertexLayout[] = {{  
 "POSITION",  
 0,  
 DXGI\_FORMAT\_R32G32B32\_FLOAT,  
 0,  
 0,  
 D3D12\_INPUT\_CLASSIFICATION\_PER\_VERTEX\_DATA,  
 0 }, {  
 "COLOR",  
 0,  
 DXGI\_FORMAT\_R32G32B32A32\_FLOAT,  
 0,  
 12,  
 D3D12\_INPUT\_CLASSIFICATION\_PER\_VERTEX\_DATA,  
 0 }};

「レイアウト」はD3D12\_INPUT\_ELEMENT\_DESC構造体の配列です。  
この配列は次のように定義されています。

typedef struct D3D12\_INPUT\_ELEMENT\_DESC  
{  
 LPCSTR SemanticName;  
 UINT SemanticIndex;  
 DXGI\_FORMAT Format;  
 UINT InputSlot;  
 UINT AlignedByteOffset;  
 D3D12\_INPUT\_CLASSIFICATION InputSlotClass;  
 UINT InstanceDataStepRate;  
 } D3D12\_INPUT\_ELEMENT\_DESC;

SemanticName シェーダの入力で使うための名前です。DirectXで定義済みの名前を使うか、自分で好きな名前をつけます。

SemanticIndex 同じSemanticNameを持つデータを識別するためのインデックスです。例えば、頂点に2つの色データがある場合、どちらのSemanticNameも「COLOR」という名前にして、このパラメータには違う番号を設定しておきます。こうすると、シェーダ側では「COLOR0」、「COLOR1」というように、SemanticName+インデックス番号で使い分けることができます。

Format データの型です。例えばVertex::positionはXMFLOAT3なので、3要素のfloat型であることを示す、DXGI\_FORMAT\_R32G32B32\_FLOATを指定しています。

InputSlot シェーダには「入力スロット」と呼ばれる入力の受取口がいくつもあり、それぞれに異なる頂点バッファを割り当てることができます。このパラメータは、どの入力スロットのためのデータなのかを設定するために使います。

AlignedByteOffset データが、頂点データの先頭から何バイト目にあるかを設定します。Vertex構造体の場合、positionは先頭にあるので0です。そして、positionは3つの32bit floatなので、12を設定します。

InputSlotClass データが頂点単位なのかインスタンス単位なのかを設定します。DirectXのインスタンスというのは、同じ頂点データを使って連続して描画を行う機能のことです。

InstanceDataStepRate InputSlotClassをインスタンス単位に設定した場合、ひとつのインスタンスを描画するために必要な頂点データの数を設定します。頂点単位に設定した場合は0を設定します。

# パイプラインステートオブジェクト(PSO)

PSOはDirectX 12で新規に実装された機能のひとつです。変数を定義する際に説明したように、PSOは、DirectXのグラフィックスパイプラインのパラメータを集積したオブジェクトです。  
PSOを作成する前にPSOが要求するいくつかのオブジェクトを作成しなければなりませんが、まずは関数の雛形を作るところから作業を始めましょう。Main.cppの最後の行に移動して、以下のコードを追加してください。

/\*\*  
\* ルートシグネチャとPSOを作成する.  
\*/  
bool CreatePSO()  
{  
 return true;  
}

また、PSOには、描画に使うシェーダを設定しなければなりません。ということは、PSOを作成する前にシェーダを作成しておく必要があります。ということで、先にシェーダを読み込む関数を作ってしまいましょう。  
CreatePSO関数の下に、次の関数を追加してください。

/\*\*  
\* シェーダを読み込む.  
\*/  
bool LoadShader(  
 const wchar\_t\* filename,　// シェーダファイルのパス  
 const char\* target, // 頂点シェーダなら”vs\_5\_0”, ピクセルシェーダなら”ps\_5\_0”  
 ID3DBlob\*\* blob) // シェーダを格納するID3DBlobインターフェイスのポインタ  
{  
 ComPtr<ID3DBlob> errorBuffer;  
 if (FAILED(D3DCompileFromFile(  
 filename,  
 nullptr,  
 nullptr,  
 "main",  
 target,  
 D3DCOMPILE\_DEBUG | D3DCOMPILE\_SKIP\_OPTIMIZATION,  
 0,  
 blob,  
 &errorBuffer))) {  
 if (errorBuffer) {  
 OutputDebugStringA(  
 static\_cast<char\*>(errorBuffer->GetBufferPointer()));  
 }  
 return false;  
 }  
 return true;  
}

この関数はD3DCompileFromFile関数を使ってファイルをコンパイルし、シェーダを作成します。  
D3DCompileFromFile関数の最初の引数は、シェーダファイルのパスです。  
2つめの引数は、コンパイル時に使用するプリプロセッサマクロ定義の配列です。今回は使わないのでnullptrを渡しています。  
3つめの引数は、シェーダで#include文を使えるようにするID3DIncludeインターフェイスのポインタを渡します。これも、今回は使わないのでnullptrを渡しています。  
4つめの引数は、エントリーポイントと呼ばれるシェーダの実行を開始する関数の名前を指定します。ここでは”main”と決め打ちしていますが、特にこうしなければならないというわけではありません。  
5つめの引数は、作成するシェーダの種類を示す文字列です。  
6つめの引数は、シェーダコンパイラに渡すオプションの論理和です。D3DCOMPILE\_DEBUGは、デバッグモードでコンパイルする指示です。D3DCOMPILE\_SKIP\_OPTIMIZATIONは、最適化を行わない指示です。  
7つめの引数は、エフェクトという種類のシェーダをコンパイルする場合に使われる追加のコンパイルオプションです。本講義ではエフェクトファイルを使わないので0を渡しています。  
8つめの引数は、作成されたシェーダを格納するID3DBlobインターフェイスのポインタのアドレスです。  
最後の引数は、コンパイルエラー情報を格納するID3DBlobインターフェイスのポインタのアドレスです。

コンパイルに成功した場合、blob引数にシェーダが格納され、関数はtrueを返します。失敗した場合、Windows APIのOutputDebugStringA関数を使って、エラーログを出力し、falseを返しています。

これでシェーダを読み込む準備はできました。CreatePSO関数の実装作業に戻りましょう。

## ルートシグネチャを作る

まずはルートシグネチャを作成します。ルートシグネチャは、シェーダが使用するデータを宣言します。定義できるデータには、ルート定数、ルートデスクリプタ、ルートテーブルの3つがあります。ルートシグネチャのデータ宣言を「ルートパラメータ」といいます。これは、D3D12\_ROOT\_PARAMETER構造体を使って宣言します。この宣言によって実際にシェーダに渡されたデータは「ルート引数」と呼ばれます。

それでは、コードです。CreatePSO関数の先頭に、以下のコードを追加しましょう。

D3D12\_ROOT\_SIGNATURE\_DESC rsDesc = {  
 0,  
 nullptr,  
 0,  
 nullptr,  
 D3D12\_ROOT\_SIGNATURE\_FLAG\_ALLOW\_INPUT\_ASSEMBLER\_INPUT\_LAYOUT  
};  
ComPtr<ID3DBlob> signatureBlob;  
if (FAILED(D3D12SerializeRootSignature(  
 &rsDesc,  
 D3D\_ROOT\_SIGNATURE\_VERSION\_1\_0,  
 &signatureBlob,  
 nullptr))) {  
 return false;  
}  
if (FAILED(device->CreateRootSignature(  
 0,  
 signatureBlob->GetBufferPointer(),  
 signatureBlob->GetBufferSize(),  
 IID\_PPV\_ARGS(&rootSignature)))) {  
 return false;  
}

ルートシグネチャを作成するには、まずD3D12SerializeRootSignature関数を使って、作成用パラメータからGPUが処理できる中間データに変換します。次にCreateRootSignature関数にそれを渡してルートシグネチャのインターフェイスを得ます。

D3D12SerializeRootSignature関数の最初の引数は、D3D12\_ROOT\_SIGNATURE\_DESC構造体へのポインタです。この構造体の最初の2つのメンバはルートパラメータの数と、ルートパラメータの配列です。次の2つはスタティックサンプラという、テクスチャを読み込むためのパラメータの数と配列です。今回は特にシェーダに渡すパラメータもなく、また、まだテクスチャも使わないため、どちらも0とnullptrを設定しています。  
最後のメンバには、ルートシグネチャの動作を指定するフラグの論理和を設定します。D3D12\_ROOT\_SIGNATURE\_FLAG\_ALLOW\_INPUT\_ASSEMBLER\_INPUT\_LAYOUTフラグは、IAステージがインプットレイアウトを参照できるようにするフラグです。

関数の2つめの引数は、作成するルートシグネチャのバージョンです。現在ver.1.0とver.1.1が存在しますが、今回は全てのDirectX 12環境で使用可能なver.1.0を選択しています。

3つめの引数は、ルートシグネチャの中間データを格納するID3DBlobインターフェイスのポインタのアドレスです。  
ID3DBlobクラスは、一時的にGPUメモリに置かれるデータを操作するためのオブジェクトです。ここでは、ルートシグネチャ作成のための中間バッファとして使用しています。

中間データが作成できたら、ID3D12Device::CreateRootSignature関数でルートシグネチャを作成します。  
最初の引数は、作成したルートシグネチャを使用するGPUを選択するノードマスクです(これまでも何度か出てきましたね)。  
2つめの引数は、D3D12SerializeRootSignature関数で作成した中間データのポインタ、3つめの引数は、中間データのバイト数です。この2つの値は、中間データを格納しているsignatureBlob変数から取得することができます。  
最後はお決まりのIID\_PPV\_ARGSマクロです。

## 頂点シェーダとピクセルシェーダ

次は、

## PSOを作る