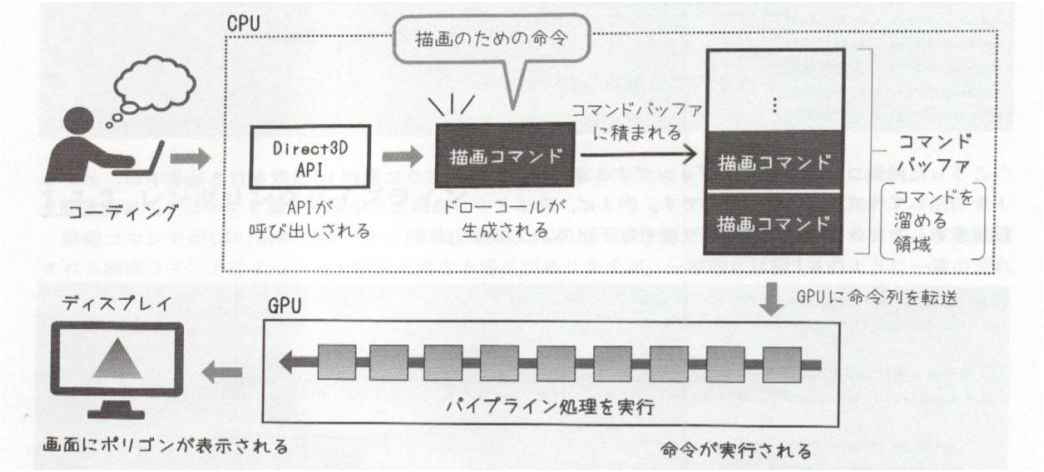
DirectX12

DX12_Vol01

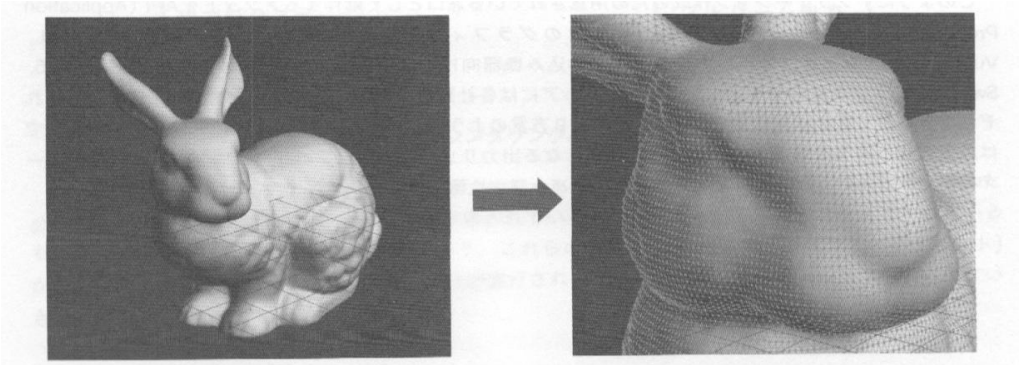
【１】画面にポリゴンが表示されるまでの大まかな流れ

図にすると次の様なイメージになります



まずは、プログラマーが Direct3D 12のメソッドを呼び出す様にコーディングします。
呼び出されたメソッドは、描画コマンドと呼ばれる形でコマンドバッファに一時的に格納され、
ハードウェア(GPU)が解釈できる形の命令列に変換されます。そして、適切な実行タイミングで
GPU側にコマンドバッファからの命令列が転送され、GPU側での処理が実行されます。
このとき、GPU側は工場のベルトコンベアのように決められた一連の処理を実行します。この一連の
処理のことをパイプラインといいます。(具体的なパイプライン処理については後述します)
パイプライン処理を経てPCディスプレイ上のピクセルの色が決まることで、最終的に私たちが
三角形のポリゴンを視認できるようになります。

なお、リアルタイム３Ｄコンピュータグラフィックスでは、描画の構成要素として三角形を使うことが
多いです。これは、平面を構成できる最小の要素が三角形であることに由来します。
複雑な幾何形状も三角形の集合体として表現します。



複雑なモデルも実は三角形ポリゴンとして表現されている

【２】グラフィックスパイラインの概要

簡単に言うと、頂点情報の入力からポリゴン描画までの流れの事です。
ハルシェーダー→テッサレータ→ドメインシェーダー→ジオメトリシェーダーの４つのステージは飛ばす事が可能なので、それらを省いて説明すると次の様になります。

１．頂点情報が頂点シェーダーへ流れる
２．頂点シェーダーが頂点変換などを行って頂点情報を作り、ラスタライザーに流す
３．ラスタライザーが情報を補間してピクセル単位に情報を分割し、ピクセルshーだーに流す
４．ピクセルシェーダーが最終的にピクセルを塗り潰すのに必要な色情報などを計算し Output-Merger に流す

このように、最初はただ単にモデルの頂点情報の塊だったものが、ピクセルの塗り潰し情報になるまでの流れがグラフィックスパイプラインです。
また、それぞれの担当箇所のことを「ステージ」といいます。

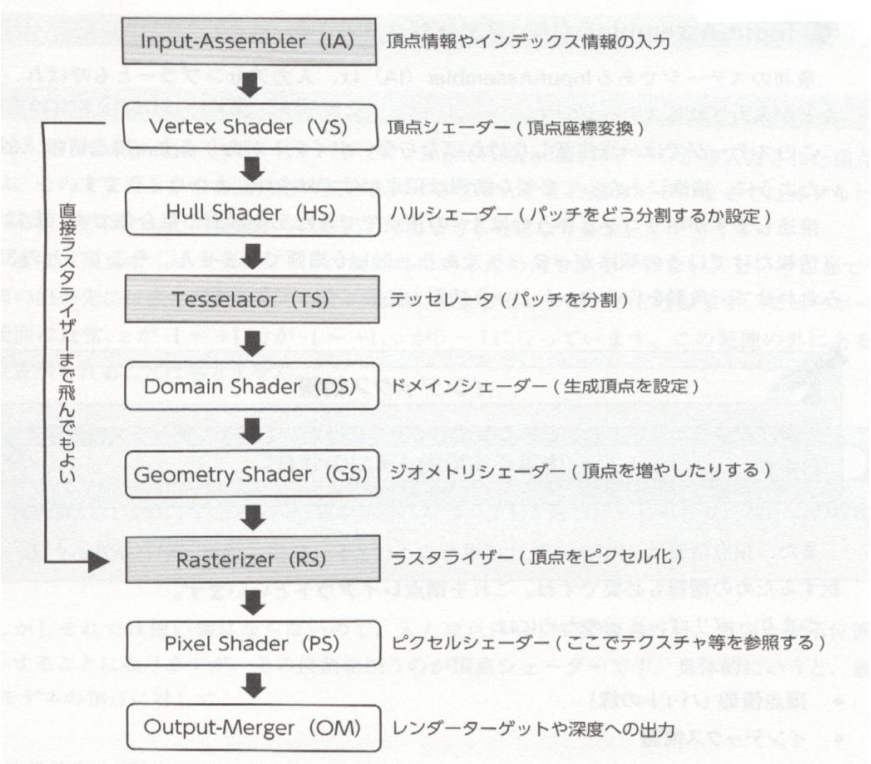


図.1

【３】グラフィックスパイプラインの各ステージ

グラフィックスパイプラインの流れのパターン
１．IA → VS → RS → PS → 出力
２．IA → VS → GS → RS → PS → 出力
３．IA → VS → HS → TS → DS → RS → PS → 出力
４．IA → VS → HS → TS → DS → GS → RS → PS → 出力

上記の４パターンが考えられます。

それぞれのステージの簡単な解説

①Input-Assembler（IA）ステージ　＜入力アセンブラ＞
最初のステージであるInput-Assembler(IA)は、入力アセンブラーとも呼ばれ、頂点情報などが入力されるステージです。
入力リソースからデータを読込み、頂点シェーダーにデータを渡します。

・注意しなければいけないポイント
描画にあたって必要な情報は頂点だけではない、という事。
ポリゴンというのは３つの頂点でできた三角形の集合体です。とはいえ、頂点情報だけではその順序がバラバラであり、正しく描画できません。そこで「どの３頂点を組み合わせて三角形を作るのか」という情報も必要になります。

・頂点情報はGPUから見れば単なる「バイト(数値)の塊」でしかない為、それを解釈する為の情報も必要です。これを頂点レイアウトといいます。

・ポリゴン表示の為に必要なもの
1)頂点情報(バイトの塊)
2)インデックス情報
3)頂点レイアウト
この３つの組み合わせが入力情報として必要になります。
そこで、パイプラインの最初に位置する「情報の入力」の事を Input-Assembler(IA)と呼びます。

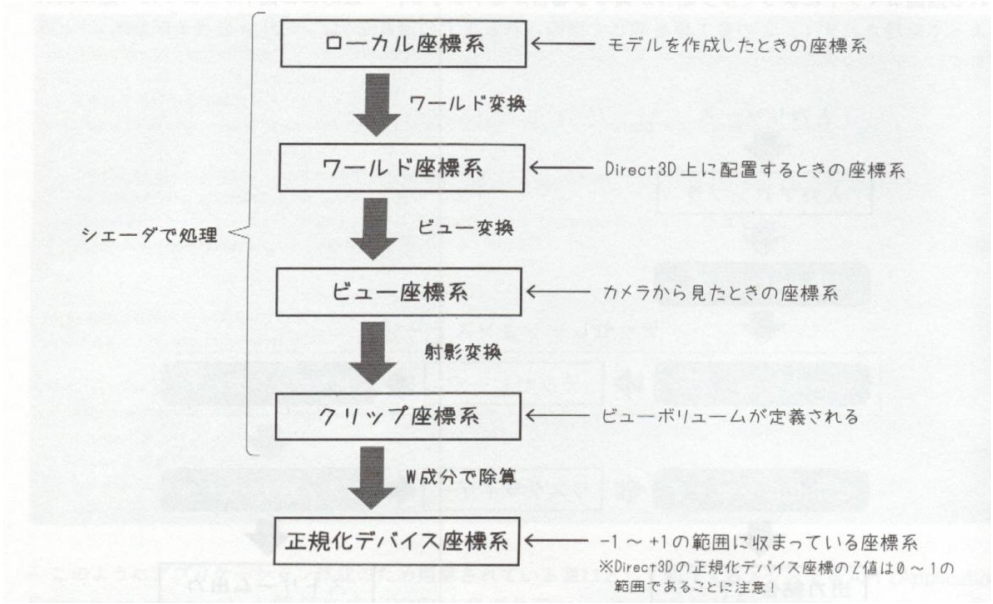
・MMD等で使用されるモデルでは、１つの頂点の頂点情報として「座標」「法線」「uv」に加えて「ボーン番号」等の情報も含まれています。
また、Input-Assembler に入力される頂点情報は、(TポーズやAポーズの)作ったままのモデルそのものの情報です。移動や回転は後述の頂点シェーダーで行われる為、それまでは頂点情報をいじる事はありません。ただ入力するだけです。

Direct3D 12のAPIでは、下記のように入力アセンブラ(Input-Assembler)を示す「IA」が付いたメソッドが対応します。

```
ID3D12GraphicsCommandList::IASetPrimitiveTopology()
ID3D12GraphicsCommandList::IASetIndexBuffer()
ID3D12GraphicsCommandList::IASetVertexBuffer()
```

②Vertex Shader(VS)ステージ　＜頂点シェーダー＞
Input-Assembler ステージから入力された頂点情報は、そのままだと単なる三角形の集合体にすぎません。また通常、そのままではモデルに含まれる頂点のほとんどが画面からはみ出してしまう事になります。
　つまり、頂点が「ビューポート」の範囲外にあるため何も表示されないという事です。
描画の出力先には表示範囲があり、その範囲の事を**ビューポート**と呼びます。
ビューポートの範囲は、x が -1～+1、y が -1～+1、z が 0～1 になっています。
この範囲の外にあるものは表示される事はありません。
　でも、それでは使い物になりませんね。ですので、入力頂点に対して**変換**を行い、適切な表示位置に表示する事になりますが、その役割を担うのが**頂点シェーダー**です。
具体的にいうと、通常３Ｄモデルの頂点に対して
・平行移動や回転の変換
・カメラ用の変換
・スクリーンへ投影する変換
を合わせた変換がかかります。頂点シェーダーの働きによって、３Ｄ空間上でキャラクターがいろんなポーズをとりながらあちこちに動く事ができるのです。

※頂点シェーダーでは、入力アセンブラから渡された頂点情報を受け取って頂点処理を行います。
このとき、与えられた頂点を１つずつ処理を行います。頂点処理とは、主に与えられた頂点に対して座標変換を行い、画面上のどこに描画されるかを計算し、出力することで、頂点ごとに独立して処理されます(頂点シェーダーでは、頂点の増加や削除はできません)。
よって、ある頂点シェーダーで処理されている情報は、別の頂点シェーダーからアクセスすることができません。



ローカル座標系は、モデルの形状を与える座標系(モデリングツールでモデルを作成するときの座標系)です。作成したモデルはCGで描く仮想世界を設けてこの世界に配置する必要があります。このCGで描く仮想世界の基準となる座標系を“ワールド座標系”と呼びます。
モデルをワールド座標系にはいちする際は、移動や回転、拡大縮小といった操作をする事があります。このCGで描く仮想世界の基準となる座標系を“ワールド座標系”と呼びます。
このローカル座標系からワールド座標系に配置するために施す変換処理を“ワールド変換”と呼びます。3DCGでは、移動、回転、、拡大縮小といった変換処理を１つの行列で表現できるように４×４行列で表し、３次元の空間ベクトル(x,y,z)に成分を１つ加えた(x,y,z,w)の同次座標(斉次座標)を用いて空間上の点を表現します。

ワールド座標系においてキャラクターから見た世界や特定カメラからの視界を表現するためには、カメラを原点としたワールド座標系とは別の座標系を用います。カメラ位置を原点とし、カメラの注視する方向をｚ軸として、上向き方向をｙ軸、ｘ軸とする正規直交基底によって定義される座標系を“ビュー座標系”と呼びます。

３Ｄで表現された仮想世界を我々が視認するためには、ディスプレイやスクリーンといった平面上に画像を映し出す処理(射影や投影と呼びます)が必要になります。
このときビュー座標から一平面状に射影する処理を“射影変換”、射影変換後の座標系を“クリッピング座標系”と呼びます。射影変換の際には、横幅、縦幅、奥行を有限にするために範囲を設定します。この設定範囲内で区切られた体積を“ビューボリューム”と呼びます。
クリッピング座標系までは同次座標で表現されますが、画面に表示するためには w成分でx, y, z成分を除算して、正規化処理を行うことで正規化デバイス座標系に変換する必要があります。正規化デバイス座標系は使用するグラフィックスAPIによって定義が異なる場合があり、この項のはじめの方でも記述していますが、Direct3Dの場合はｘ方向は-1から+1、ｙ方向は-1から+1、ｚ方向は0から+1の範囲になります。

ボーン情報を使用してポーズを取らせる(ボーンによる頂点変換)のも頂点シェーダーの役割です。

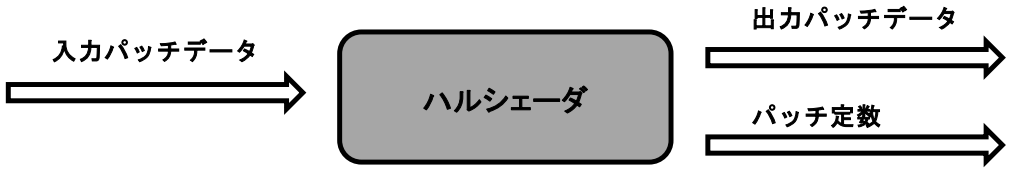
このように頂点シェーダーで変換された頂点情報が、以降の Hull Shader ステージや Geometry Shader ステージ、Rasterizer ステージに流し込まれる事になります。

③Hull Shader(HS)ステージ　＜ハルシェーダー＞
ハルシェーダーは、Direct3D 11から追加されたハードウェアテセレーションを行うために
設けられたテセレーションステージを構成する要素の１つです。

先にテセレーションについて説明しておきます。
・テセレーションとは、三角形や四角形をより細かい三角形に分割する事で、実行時により精密な
表現を可能にする処理です(もちろん線分も分割可能です)。
ただし当然ながら表示するポリゴン数が増える為、処理負荷には注意が必要です。

テセレーションの応用例
テセレーションの応用例としては、サブディビジョンサーフェスやハイトマップ等が
あげられます。凸凹を法線マップなどでごまかすのではなく、実際に細かく頂点を
デコボコさせることで緻密な表現が可能になります。

DirectX12において、テセレーションは
1. Hull Shader(HS)ステージ
2. Tesselator(TS)ステージ
3. Domain Shater(DS)ステージ
という３つのステージが順に連携して働く仕組みになっています。
その中でも Hull Shader(HS)ステージの役割としては、「頂点シェーダーからやってきた
入力パッチ情報から、出力パッチ情報とパッチ定数を生成する」というものです。
そして**ハルシェーダー**には、実際にテッセレータに分割するための「設定」、具体的には
分割数や分割のしかたなどを記述します。



Hull Shader ステージでは分割は行われない
あくまで分割のための準備であって、実際に分割が行われないことには注意しましょう。
実際に分割が行われるのは Tesselator ステージです。

・頂点シェーダーからやってきた頂点データは、３つ一組もしくは４つ一組のパッチという
データとして扱われます。それぞれの頂点はハルシェーダー内において「入力コントロール
ポイント」という、分割の目安になる座標データとして扱われます。

ハルシェーダーには、以下の２つの関数が必要になります。
1. パッチごとに実行される関数
2. パッチ内のコントロールポイントごとに実行される関数
これらが関数の中で分割数を設定したり、属性指定で分割方法を指定したりすることで次の
Tesselator(TS)ステージにおけるテッセレータの分割方法が変わってきます。

④Tesselator(TS)ステージ　＜テセレータ＞
テセレータは、ドメイン(矩形、三角形、線)を多数のより小さいオブジェクト(三角形、点、線)
に分割する**固定機能パインプラインステージ**です。

Tesselator(TS)ステージでは、ハルシェーダーで設定されたパッチデータをもとに、テッセレータが
実際に分割を行います。
分割方法はハルシェーダーの設定によってさまざまです。しかし、Tesselator(TS)ステージは
ブラックボックスであり、特にプログラマーが行えることがないため、さらっと流して次の
Domain Shader(DS)ステージの解説に移ります。

⑤Domain Shader(DS)ステージ　＜ドメインシェーダー＞
テセレーションにより分割された結果、最終的にできた頂点を扱うのが Domain Shader(DS)ステージです。
Domain Shader(DS)ステージでは、ドメインシェーダーが Hull Shader(HS)ステージから出力されたパッチ定数とコントロールポイント、および Tesseleator(TS)ステージから出力された**ドメインロケーション**(分割後の頂点座標を決めるためのパラメーター)を受け取り、実行されます。

ドメインシェーダーでは、テセレータを介して出力された制御点をもとに実際の頂点座標を計算します。計算処理はテセレータの出力点ごとに１回実行され、テセレータの出力テクスチャ座標、ハルシェーダーの出力パッチ、ハルシェーダーの出力パッチ定数にドメインシェーダーのプログラム内からアクセスする処理を(読み取り専用で)記述できます。

最終的な分割後の頂点座標は、コントロールポイントとドメインロケーションを組み合わせて作ることになります。実際には、テッセレータで割った結果の頂点座標を示す重み(各コントロールポイントに対する割合)から、新しい頂点を計算します。そうして、テセレーションありのときの頂点が、ここで「いったん」確定します。

⑥Geometry Shader(GS)ステージ　＜ジオメトリシェーダー＞
上の項で、「いったん」と含みをもたせたのには理由があります。というのも、頂点シェーダーから来た面情報やテセレーションで生成された面情報が、さらにこの Geometry Shader(GS)ステージで加工される可能性があるためです。

ジオメトリシェーダーは、Direct3D 10から導入されたシェーダーです。一つのプリミティブデータを入力として受け取り、プリミティブの増減や変形、変換を行います。

線分も扱える
「面」といっても、テセレーションのときと同様、線分に対する加工も可能です。

Geometry Shader(GS)ステージの役割をひとことでいうと「プリミティブ(ここでは三角ポリゴン)単位でさまざまな操作を行う」処理であり、その方法を記述するのが**ジオメトリシェーダー**です。

頂点シェーダーは頂点ごとに変換を行う為のものでしたが、ジオメトリシェーダーはプリミティブごと、つまり三角面なら頂点３つを一組にして加工します。
加工の内容としては、座標の移動だけでなく、面数を増やすことも可能です。そのためジオメトリシェーダーは、壁などに影を投影するための方法である「シャドウボリューム」や、物体の表面に毛を生やす手法の一つである「フィン法ファー表現」などに活用することができます。

作られる頂点数の最大値の指定
面数を増やす場合、増やした結果作られる頂点数の最大値を指定しておく必要があります。
指定は maxvertexcount というキーワードを使い、

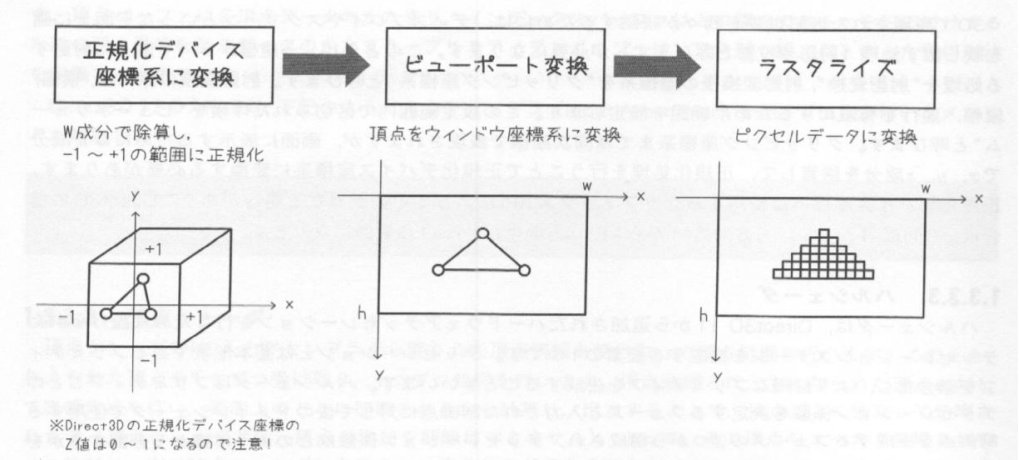
```
[maxvertexcount（最大頂点数）]
void ジオメトリシェーダー名(引数パラメーター, inout 出力ストリーム)
{
    (必要な処理)
}
```

といった感じで行います。

加工後は、ストリーム出力(SO)といって、結果をGPU上のメモリに書き込むことも可能です。
ですが、ここでは通常のパイプラインに沿って、次は Rasterizer(RS)ステージの説明を行います。

⑦Rasterizerステージ(RS)　＜ラスタライザー＞
ここまでに何回か出てきましたが、**ラスタライズ**(ピクセル化)とはプリミティブ(三角ポリゴンや線分など)をピクセルに変換する処理です。

ジオメトリシェーダーを通過した頂点データで構成される幾何形状の情報はベクトルで表現される形式の為、ディスプレイに表示されるようなピクセルデータに変換されていません。私達がモニタを通じて視認できるように表現するためには、データをピクセルの集合であるラスタ表現に変換する必要があります。この辺かをスキャンコンバージョン(または走査変換)と呼びます。ラスタライザーでは、このスキャンコンバージョンが実行され、どのピクセルを塗り潰すかが決定されます。



ラスタライゼーションの前にビューポート変換が実行され、ビューポート上の位置、つまり２Ｄ上に射影したウィンドウ上の位置が決定されます。その頂点の位置をもとに、どのピクセルを塗り潰すかがスキャンコンバージョンによって決定されています。このときにウィンドウの外側にあるものは明らかに塗り潰す必要がないので、塗り潰しの為の処理を省略することができます。シザー矩形と呼ばれる切り取りの為の領域が設定されている場合は、この不要な領域を切り取ることができます。この切り取り処理のことを“クリッピング処理”と呼んだりします。

グラフィックスパイプラインを通りながらここまでさまざまな処理が行われたプリミティブに対して、ラスタライズを行うステージが Rasterizerステージ (RS) です。このステージはTesselator(TS)ステージ同様、プログラマーが手を出せないブラックボックスです。
このステージでは、頂点を渡してピクセル化される(ピクセルシェーダーが呼び出される)のを待つだけです。
ただしラスタライズにもルールがあるため、パイプライン作成時のパラメーターで「どのようにピクセル化するのか」を指定することは可能です。例えば、「ピクセル間の補間を行うのか行わないか」「どちらの面を表と見なしてカリング(裏面を表示しない)するか」などです。

⑧Pixel Shader(PS)ステージ　＜ピクセルシェーダー＞
ある意味描画の主役ともいえる、見た目のコントロールには欠かせないシェーダーがピクセルシェーダーであり、ピクセルシェーダーが動作するのが Pixel Shader(PS)ステージです。ピクセルシェーダーが無ければ、画面上をどの色で塗り潰すかもわかりません。
そのため、ピクセルシェーダーは描画において大変重要な位置を占めている、必須のシェーダーです。
ピクセルシェーダーには、サーフェス情報を細切れにしたデータをもとに、超t年情報を補間したデータが入ってきます。例えば、頂点Aと頂点Bの中点にあたるピクセルでは(A＋B)÷２というデータになる、とイメージするとよいでしょう。このため、頂点シェーダーで頂点に色を付けておけば、勝手に補間されてグラデーションになります。例えば、ランバートの余弦則によるシェーディングの計算を頂点シェーダーで行えば、ピクセルシェーダーはただ入力された色を返すだけで十分かもしれません。しかしたいていの場合、明るさの計算はピクセルシェーダーで行うことになります。
　他にも、テクスチャの適用もピクセルシェーダーで行います。頂点情報にuv情報が設定されていれば補間されたuv座標が得られるため、そのuv座標をもとにテクスチャから色を抽出してポリゴンに色を付けていきます。すると、最終的に「テクスチャを張り付けた」効果を得ることができます。

最も基本的なレンダリングにおけるピクセルシェーダーの役割をまとめると、以下の３つになります。
・マテリアルの色やテクスチャの色を調べる
・シェーディングを行い濃淡を計算する
・調べた色情報と濃淡を合成した色情報を出力する

ピクセルシェーダーのさらなる役割
さらに最近は、マルチパスレンダリングなどで、ポストエフェクトやシャドウマップなど
複雑な処理を行うのにも使用されることがあり、役割はかなり多岐にわたっています。

⑨Output-Merger(OM)ステージ　＜出力結合ステージ＞
グラフィックスパイプラインは Pixel Shader(PS)ステージで終わりではありません。最後に
ステージがもう一つあります。Output-Merger(OM：出力マージャ―)ステージです。
「マージャ―(結合するもの)」という耳慣れない名前ですが、一体何をするものなのでしょうか。
今まで通ってきたステージはあくまでも「入力頂点からプリミティブの表示まで」を行うもの
でした。つまり三角形１個を描画するためにこれらのステージを通ってきたわけです。
しかしゲームというのはポリゴン１つで成り立っているわけではありません。

・どれが手前に来て、どれが奥に来るのか
・半透明ならば、先に描画されているオブジェクトとどのようにブレンドするのか

などを考えなければなりません。
ピクセルシェーダーはあくまで色を決めるだけのものであり、**Ｚ値テスト**(描画しようとしている
ピクセルの奥行き位置を調べ、塗るか塗らないかを判断する)をおこない、すでに描画済みの色との
合成を行うステージが必要なのです。それこそが Output-Merger(OM)ステージです。
とはいえ、ここもシェーダーを書くわけではなく、

・深度テスト(Ｚ値テスト)を行うか否か
・どのようにブレンドするか(αブレンディングや加算など)

といった設定を、各種定数を用いて行うことになります。
ここまでくると、やっと画面上にポリゴンメッシュが出現します。

ピクセルシェーダーの処理が終了したら、出力結合ステージはピクセルシェーダーの出力を正しく
マージして、関連付けられている描画先のリソース(レンダーターゲット)に結果を出力します
(深度ステンシルバッファが関連付けられている場合も同様にリソースに出力されます)。
最終的に出力結合ステージに設定されているリソースがモニタ上の画面に出力され、私達が画面を
通じて確認できるようになります。
Direct3D 12のAPIでは下記のように出力結合ステージ(Output-Merger)を示す「OM」がついた
メソッドが対応します。

```
ID3D12GraphicsCommandList::OMSetBlendFactor()
ID3D12GraphicsCommandList::OMSetStencilRef()
ID3D12GraphicsCommandList::OMSetRenderTargets()
```

全体を見たところで図.1をもう一度眺め、グラフィックスパイプラインのイメージを確実なものに
しておきましょう。

【４】シェーダーとは

ここまで「頂点シェーダー」「ピクセルシェーダー」などという言葉が出てきましたが、それらはすべてGPU上で実行されるプログラムです。以前はこれらの制御はDirectXに任せきりであり、プログラマーはせいぎょできませんでした。しかしDirectX 8から「プログラマブルシェーダー」という名前でプログラマーがレンダリングや座標変換の細かい部分まで制御できるようになりました。

さらにDirectX 10以降は細かい部分まで自分で制御「しなければならなく」なりました。

①シェーダー記述言語HLSL

プログラマーが制御できるということは、シェーダーの制御も「プログラミング言語(シェーダー記述言語)」で記述することになります。初期の頃はアセンブリ言語が使用されていましたが、DirectX 9以降ではHLSL(High Level Shader Language)という言語が使用されています。

他のシェーダー記述言語

ちなみにDirectXとは別のグラフィックスライブラリである OpenGL では GLSL という言語が使われていますし、NVIDIA社からは Cg という独自の言語が提供されています。それぞれ少しずつ違いますが、言語自体はどれもC言語ベースであり、１つ覚えたらシェーダー間の移植は容易だといえます。

②エントリポイント

まずは、シェーダーがどのように呼び出されるかを知っておきましょう。
グラフィックスぱいっぷラインにおける各シェーダーは、パイプラインの流れ(ステージ)に応じてGPUから呼び出されます。CPUから直接呼び出されるわけではありません。

また、よくあるスクリプト言語のように「ファイルの上から下に向かって１行ずつ実行される」わけではなく、C言語と同様に**エントリポイント**(C言語でいうところのmain()関数)が必要になります。どのファイル内のどの関数をエントリポイントとするかをCPU側の処理で指定しておくことで、描画時にそのエントリポイント関数から実行されるというわけです。
DirectX12では、**パイプラインステート**というオブジェクトにより、エントリポイントの指定をはじめ、さまざまなパイプラインにかかわる設定を行うことになります。