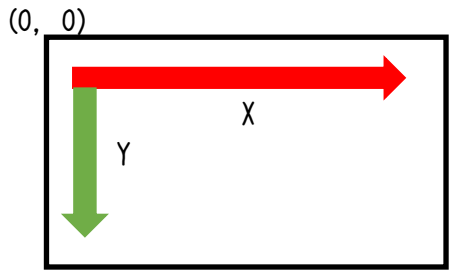
３Ｄゲーム制作のはじめに

覚えることがたくさん出てきますが、１つずつクリアしていきましょう。
演習や制作に時間を使ってあげれば、これまでのように、
勝手に慣れてきますので、辛抱強く取り組んでいきましょう。

３Ｄ基礎①　座標と方向

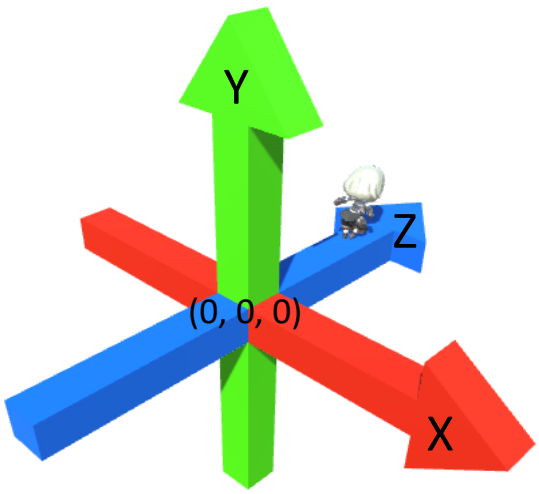
２Ｄゲームは、X(横)とY(縦)で座標管理を行ってきましたが、
３Ｄゲームになると、『 奥行き 』が加わりますので、
X(横)、Y(縦)、Z(奥行き)の３つの数字を使う必要があります。
また、それぞれの軸の正方向も重要になってきますので、
合わせて覚えておきましょう。

【２Ｄ】



左上が(0, 0)
Xの正方向は右。
Yの正方向が下。

【３Ｄ】

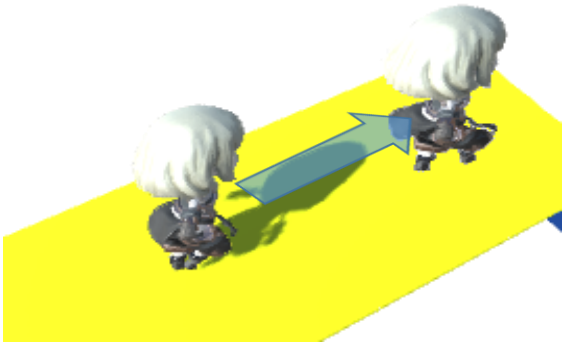


中心が(0, 0, 0)
Xの正方向は右。
Yの正方向が上。
Zの正方向が奥。

３Ｄワールドのルールです。

わかりやすいように、
色も共通化されています。
X(赤)、Y(緑)、Z(青)。

３Ｄキャラクターの Z 座標をプラスしていけば、
(ワールドの)前方に移動していくような動きになります。

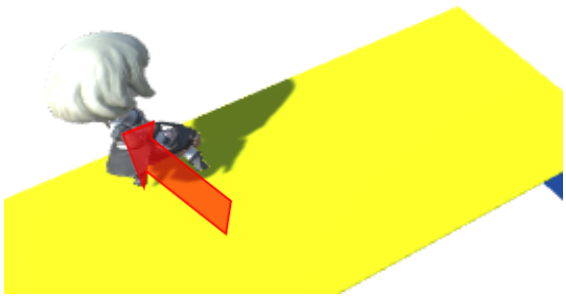


pos_. z++;

３Ｄキャラクターの Y 座標をプラスしていけば、
ジャンプしているような動きになります。

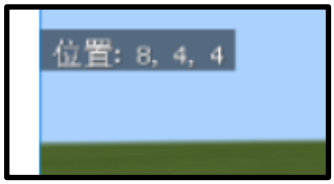
pos_. y++;

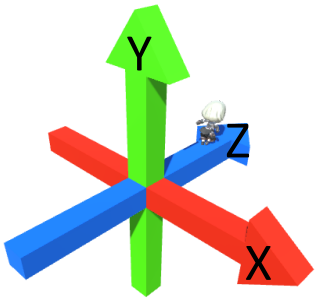
３Ｄキャラクターの X 座標をマイナスしていけば、
(ワールドの)左に移動していくような動きになります。



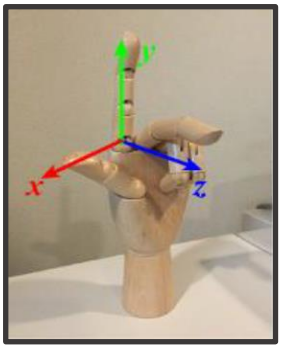
pos_. x--;

普段から座標が画面上に表示されているゲームをプレイしていたら、
馴染みある感覚だと思いますが、そうでない方は、
今すぐマイクラなどの
３Ｄ座標が画面に表示されるゲームをやって、
３Ｄワールドの感覚を掴みましょう。





今回紹介した座標系は、左手系座標と呼ばれ、
DirectXやDxLib、ゲームエンジンの
UnityやUnrealEngineで使用されています。

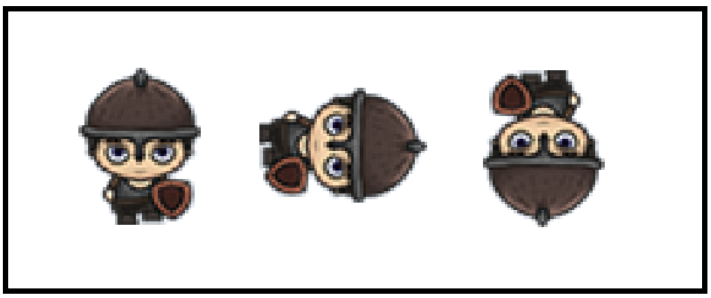


逆の右手系座標というのも存在します。
グラフィックス系のソフト(3Dソフト)やOpenGLなどのグラフィックライブラリ、
一般数学で習う座標系も右手系です。

インターネットや書籍をプログラミングの参考にする際は、
注意しておきましょう。(特に数学系のサイトや本)

３Ｄ基礎②　回転

３Ｄゲームといえば、回転です。
これまでの２Ｄゲーム制作だと、画像の回転がイメージしやすいと思います。

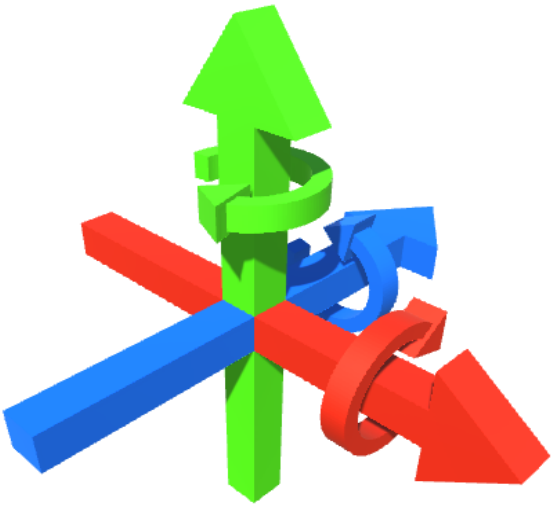


DrawRotaGraph
の引数で、
回転できます。

２Ｄ画像の
Ｙ軸回転。
あまり使わない。

これは、３Ｄでいうところの Z 回転です。
そうなのです。回転にもXYZがあるのです。

左手座標系における回転軸と向き



根本から矢印の先を見た状態で、
時計の反対回りが、正の回転。

正の回転とは、
角度をプラスすると、
回転する回転の向きのこと。
（マイナスすると逆になる）

わかりづらいかと思いますので、下記前提で補足致します。

[前提]

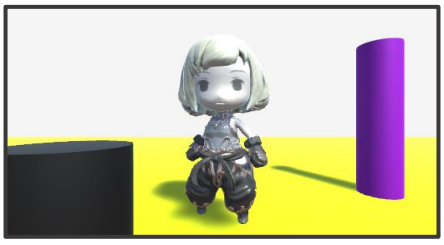
３Ｄキャラクターは、３Ｄワールドの Z の正方向を向いているため、
無回転（X軸0度、Y軸0度、Z軸0度）状態から回転を開始する。
左上画像から、右上、左下、右下の順に+90度ずつ任意軸で回転させる。

■Ｙ軸回転（正方向）

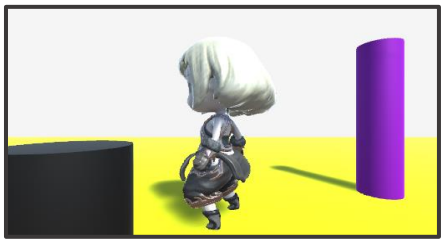
Ｙ軸0度（0 radian）

Ｙ軸90度（1.57 radian）

Ｙ軸180度（3.14 radian）



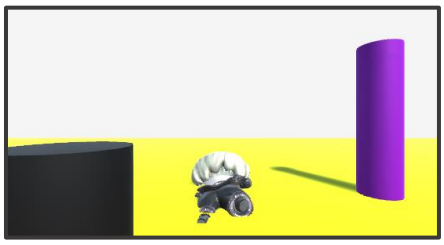
Ｙ軸270度（4.71 radian）



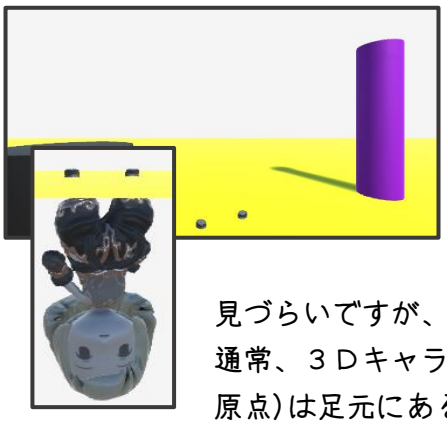
■Ｘ軸回転(正方向)

Ｘ軸0度(0 radian)

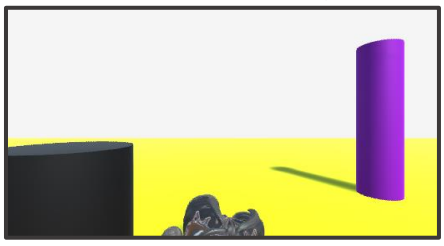
Ｘ軸90度(1.57 radian)



Ｘ軸180度(3.14 radian)



Ｘ軸270度(4.71 radian)



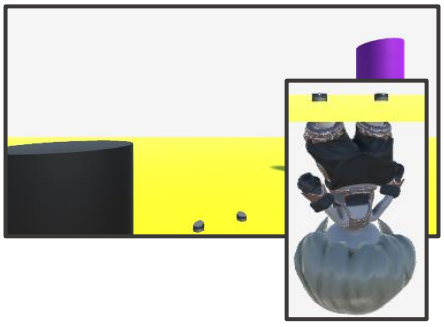
見づらいですが、逆さ吊りになっている状態。
通常、３Ｄキャラクターの Pivot(ローカル座標系の原点)は足元にあるため、足元を中心に回転する。

■Ｚ軸回転(正方向)

Ｚ軸0度(0 radian)

Ｚ軸90度(1.57 radian)

Ｚ軸180度(3.14 radian)



Ｚ軸270度(4.71 radian)

回転のイメージが沸かない場合は、Unityなどのゲームエンジンを
使って、３Ｄモデルをグリグリ回転させてみましょう。

角度の単位

普段私達が使っている角度の単位は、度数法(デグリー degree)と
言いますが、プログラミングで使用する時の単位は、
弧度法(ラジアン radian)が用いられることが多く、
DxLibの関数や３Ｄ制御で使用されている角度もラジアンになっています。

そのため、1度回転させたい！ という時に、

```
angle_.x += 1;                →57度足してますよ！
```

とすると、1 はラジアンからデグリーに変換すると、57度にもなりますので、
想像以上にギュルギュル回転してしまうことでしょう。
回転が早すぎる場合は、単位を見直してみましょう。

度	0	45	90	135	180	225	270	315	360
radian	0.00	0.79	1.57	2.36	3.14	3.93	4.71	5.50	6.28

π = 3.14159265359…

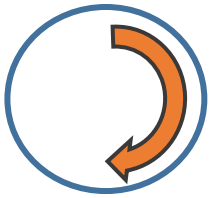
デグリーからラジアンへ変換　：　90度 × π ÷ 180 = 1.57rad
ラジアンからデグリーへ変換　：　1.57rad × 180 ÷ π = 90度

変換したい方の単位を
分子にすると覚えやすい。

$\times \frac{\pi}{180}$　ラジアンへ　　$\times \frac{180}{\pi}$　デグリーへ

どうしてπがラジアンなの？

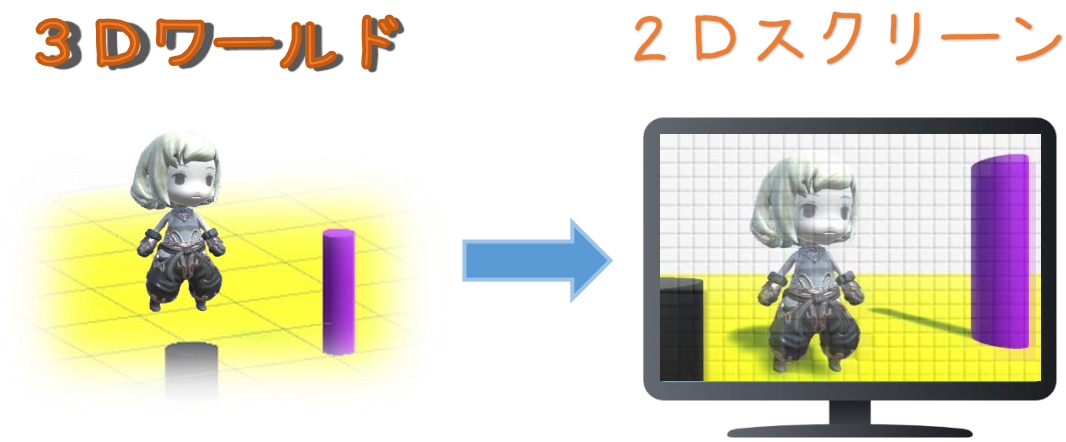
弧度法という名前の通り、円をグルっと１周した円周の長さで、
角度を表しているのがラジアンです。



円周は、2πrで求められる。半径rが1の場合、
円周は、2πとなる。ということは、半周はπです。
半周は、180度の位置にありますので、
比率を計算して上記は変換を行っています。

３Ｄ基礎③　描画の仕組み

現実世界と同じように、縦、横、奥行きのある３Ｄワールドですが、
最終的には、テレビやディスプレイ、スマホの液晶などの薄っぺらい
スクリーン(奥行きのない２Ｄ領域)に映し出さないといけません。

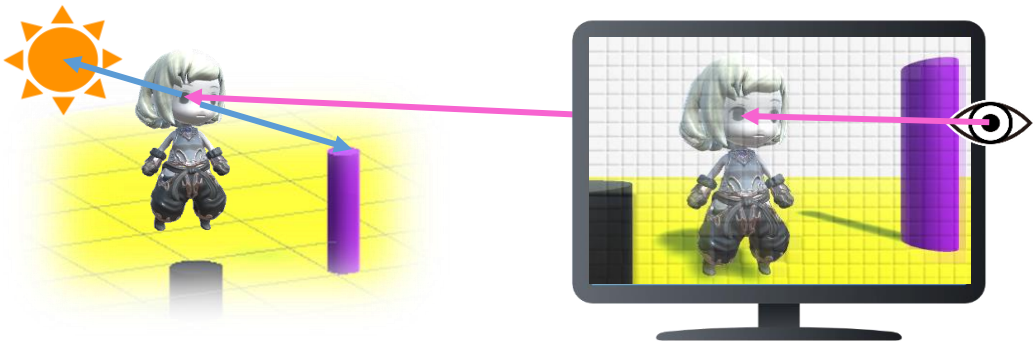


この描画(レンダリング)方式を２種類紹介します。

<u>レイトレーシング</u>

2020年に発売された「サイバーパンク2077」で、
レイトレーシングモードが搭載され、その名前を聞いたことが
ある方もいらっしゃると思います。
光の屈折や反射などを、現実世界の影響をしっかり計算することで、
リアルな表現を行う技術です。
計算量が多く、これまでのパソコンや家庭用ゲーム機では、
スペック的に厳しかったのですが、グラフィックボードの性能が向上
してきたことで、高性能ＰＣやPS5などで実現が可能な時代になって
きました。

なぜ計算量が多いかというと、
仮にフルHDの解像度で画面表示する場合、1920×1080画素になりますので、
約200万画素(ピクセル)が必要になります。
その200万画素１つ１つから光線を出して、その画素に表示する色を
計算して決めていきます。



ゲームでは、映画やＣＧ動画とは異なり、プレイヤー操作が入るため、
キャラクターが動いたり、それに応じて背景が変わったりします。
そのため、200万回の複雑な色計算処理を"リアルタイム"に
行う必要がありました。30FPSとしても、0.033秒でこの処理を完了
させなければいけませんが、それができない時代だったため、
もっと処理速度が早い方式を取るしかありませんでした。
それが、次に紹介するラスタライズ方式で、現在でもゲーム開発の
主流となっています。

※先端技術のレイトレは、就職の武器になると思いますので、
興味がある方は、川野先生の授業を参考にしつつ、
ぜひ、知識・技術の深堀りを行ってください！

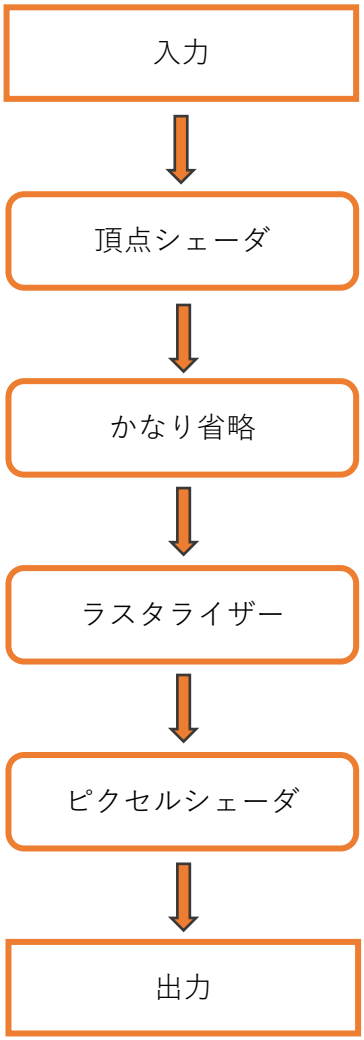
https://www.itmedia.co.jp/pcuser/articles/2202/11/news032.html

ラスタライズ

レイトレは、視点から見て、スクリーンの色はどんな色になるか
計算を行っていましたが、ラスタライズは考え方としては、
視点(カメラ)をゲーム内において、３Ｄワールドの色をスクリーンの
色に割り当てる、というやり方になります。

DxLibの元となっているDirectXの描画処理の流れ、
レンダリングパイプラインを見ながら、処理を追っていきましょう。
https://learn.microsoft.com/ja-jp/windows/win32/direct3d12/pipelines-and-shaders-with-directx-12

Direct3D 12 のレンダリングパイプライン(超簡易版)



３Ｄモデルのポリゴン情報(頂点情報)

３Ｄワールドからスクリーン座標への
変換処理や、３Ｄモデルの整形を行う

※ここでは、省略しますが、
　大手ゲーム会社を狙っている方は、
　しっかり勉強しよう

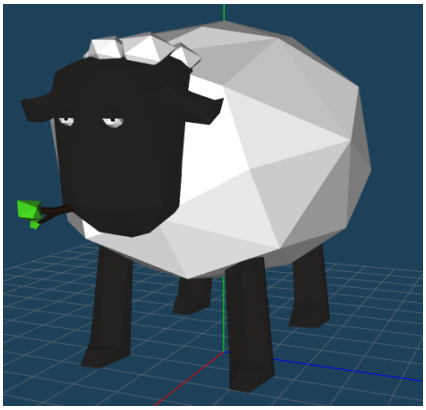
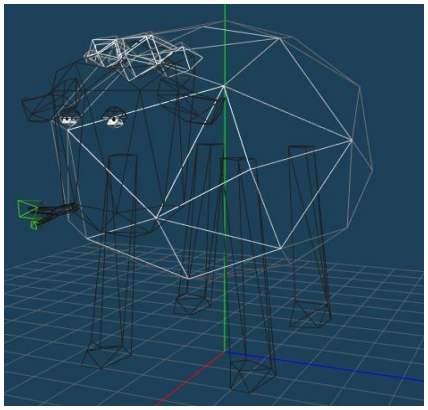
ディスプレイ表示用のピクセルデータに
変換を行う

スクリーン１画素(ピクセル)ごとの色を
決める処理。テクスチャはここで使用する

先ほどのパイプラインをもう少し詳細に解説していきます。

入力

ざっくりにはなりますが、頂点情報を頂点シェーダに送ります。
頂点情報はいくつかありますが、座標が最も重要です。

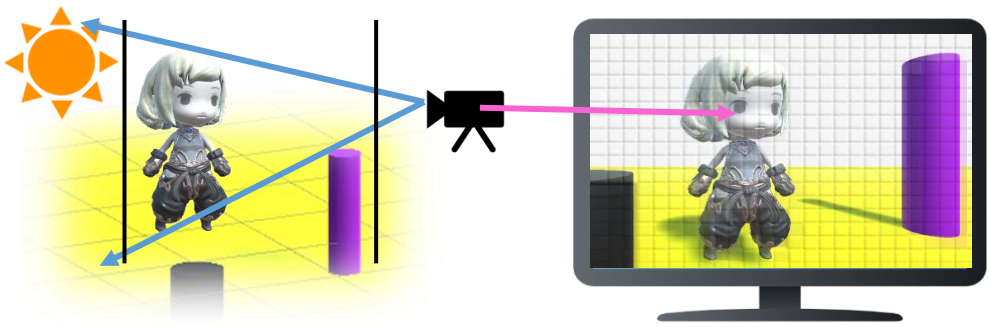
３Ｄモデルは、ポリゴンと呼ばれる三角形で形成されています。
このポリゴン数が多ければ多いほど、きめ細やかになり、
グラフィックが綺麗になります。その分、処理負荷が高くなります。
上図では、ポリゴン数が少ないので、カクカクが目立っています。
これを通称ローポリ（低ポリゴン）などと呼びます。

そして、ここからが本題なのですが、モデルを形成する三角形（ポリゴン）
を作るためには、３つの座標が必ず必要になります。
これを『 頂点座標 』と言います。

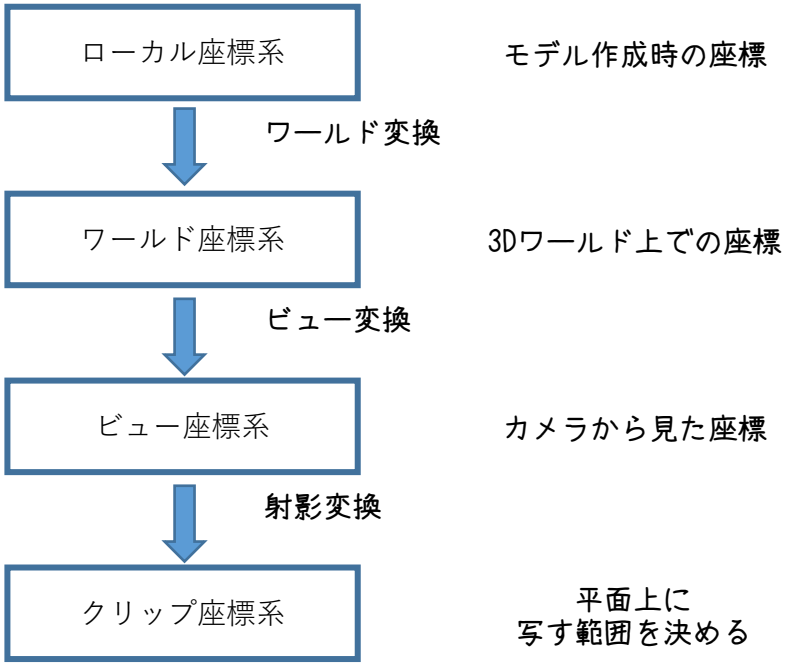
モデルを形成する頂点座標や、色などを頂点シェーダーに渡す役割が、
この入力フェーズになります。

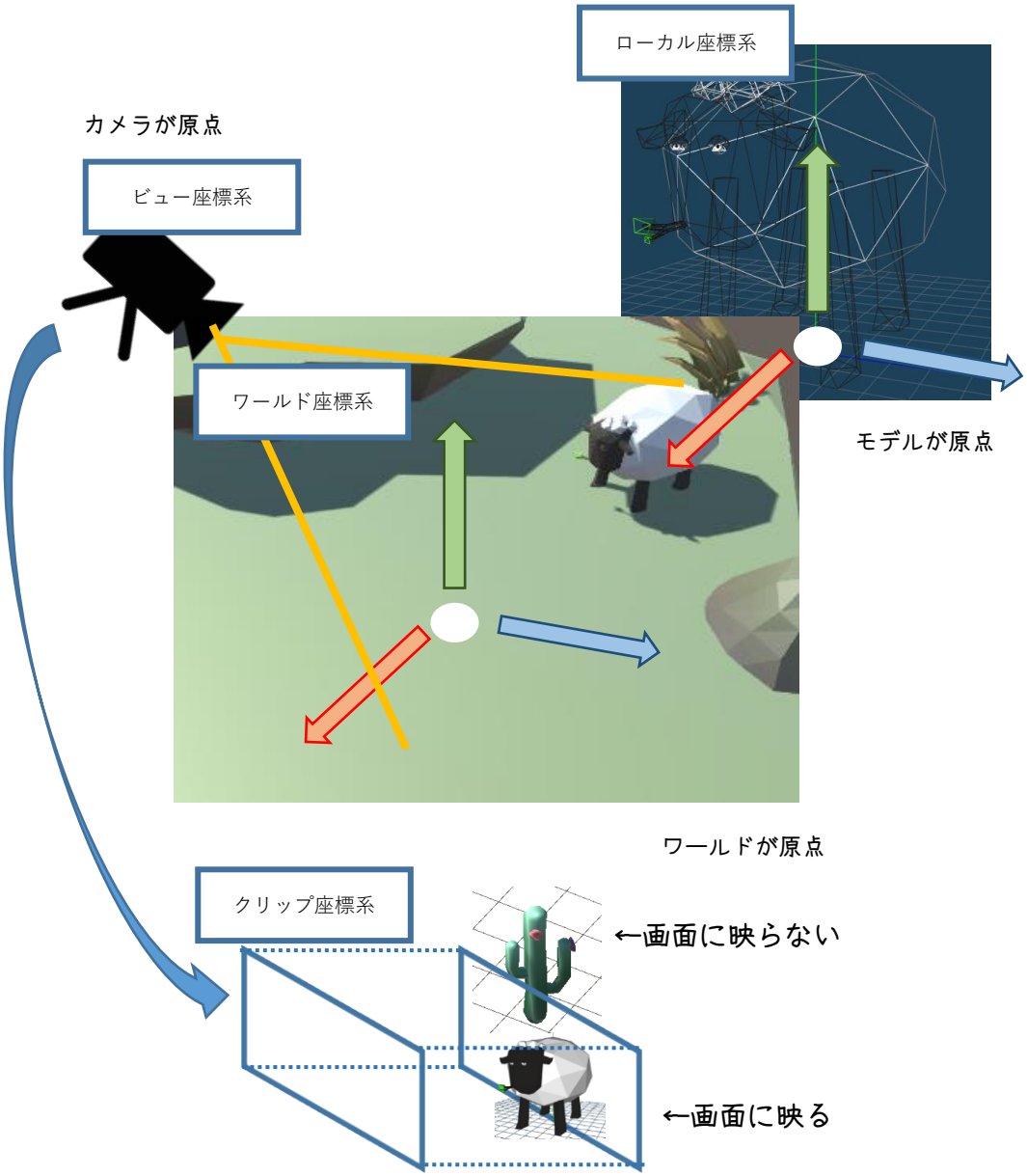
頂点シェーダ

頂点シェーダーは大忙しです。
受け取った頂点座標を何度か変換する必要があるのですが、
なぜ必要かというと、最終的にスクリーン座標に変換するためとなります。



３Ｄワールド内にカメラを置いて、パシャっと１枚写真を撮るような
イメージです。私達の３Ｄ世界もスマホのカメラで撮ると、
１枚の写真（２Ｄ）になりますよね。そのための座標変換です。




ローカル座標系
カメラが原点
ビュー座標系
ワールド座標系
モデルが原点
ワールドが原点
クリップ座標系
←画面に映らない
←画面に映る

ラスタライザー

頂点の位置を元に、スクリーンの
どのピクセルを塗りつぶすか決定する

正規化デバイス
座標系

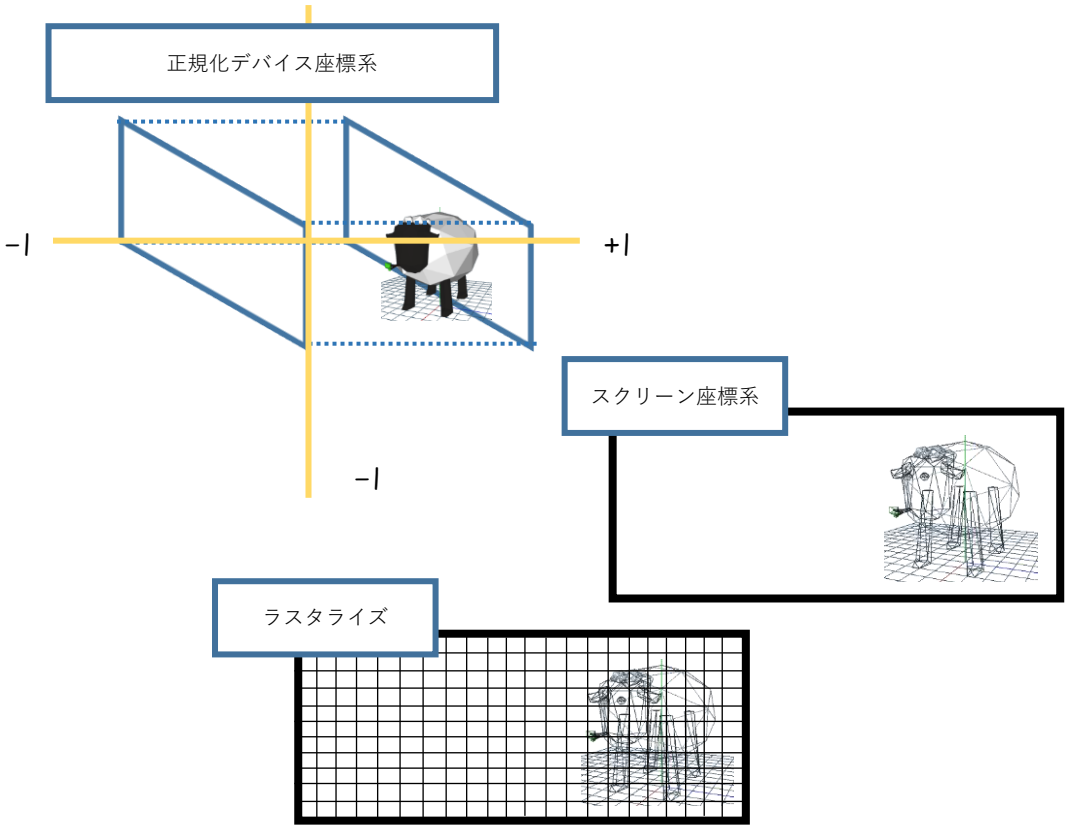
−1 ～ +1の範囲に
正規化

スクリーン座標系

頂点座標を
スクリーン座標に変換

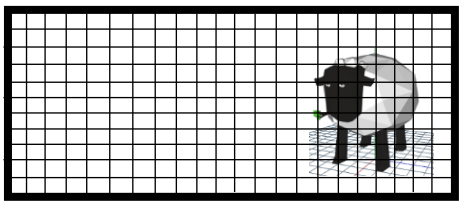
ラスタライズ

ピクセルデータに
変換



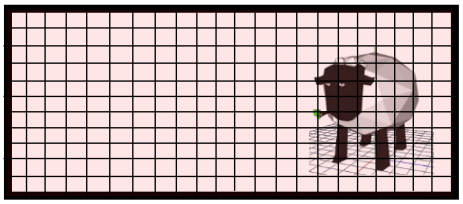
ピクセルシェーダ

ピクセルごとの色を決める。
頂点データに設定されたテクスチャ(画像)の情報(UV)を使って、
色を決めたり、個別の表現を行うための色の計算を行う。



仮に、全てのピクセルに赤色を少し加算すると、

```
color.r += 0.1f;
```



このような色味になる。

シェーダを勉強するなら、ピクセルシェーダが入り易いので、
おすすめです。
これまで、３Ｄゲーム前提のお話でしたが、
２Ｄゲームでもピクセルシェーダ使用して、様々な表現を行うことが
できますので、有効な技術です。

このような過程を経て、３Ｄモデルは画面に表示されていきます。
描画の専門知識が広く必要になりますので、このあたりを習得するのは
大変かと思います。
そのため、グラフィックプログラマーという専門職が存在するくらいです。

描画周りに興味があり、技術を習得されたい方は、
【グラフィックプログラマー】という職種を意識して、
このあたりの知識・技術を深掘りしていきましょう。