「なるほど3Dグラフィック描画の仕組み」

いろんな職業の方が見る資料なので説明を簡単にしてある部分があります。正確には本来の意味と違いますが上記理由のためです。ご了承ください。

まずは基礎知識

■CPUとGPU

CPU：Central Prosessing Unit
なんでもこなすやつ
色んな処理に対応できる

GPU：Graphcs Prosessing Unit
描画処理に特化したやつ
単純な処理しか対応できないが高速

■iPhone/Android端末のそれ

iPhone：**A8**とか**A8X**とか言われているプロセッサ

Android：**Tegra X1**とか**SnapDragon**とか言われ…略

↑↑は「CPUとGPUとメモリをひとまとめ」にしたもの。
小型化、コスト、消費電力の観点からまとめてある。

Tegra X1はNVIDIA
SnapDragonのGPUはAMD系

■で、GPUはどうやって使うの

OpenGLを使う。

OpenGLとはOpen Graphics Libraryの略で、

簡単に言うとGPUを利用して描画を行う為の標準仕様。

各メーカーがこの仕様に沿って実装することで、

プラットフォームに依存せず同じ命令を利用できる。

スマホで利用できるOpenGLのバージョンは、
OpenGL ES1.1、OpenGL ES2.0、OpenGL ES3.0
である。
※主流はOpenGL ES2.0　　※ESはEmbedded Systemsの略で簡略版ということ

■OpenGLはこんな立ち位置

プログラムからOpenGLの命令を通して、
GPUにデータや処理内容を伝える。

そう、つまりGPUに何かさせるには、
CPUからGPUへの通信が発生することになる。
これはCPU内で完結する命令より、**コストが高い**。

■シェーダの話

シェーダとは描画に必要な計算を行う処理部のこと。

一昔前のGPUは決まりきった処理しかできなかった。

プログラマブルシェーダというものが生まれ、

GPUで処理できる内容が、動的に変更できるようになった。

これにより、描画品質を上げることが可能となる。

■シェーダの種類

バーテックスシェーダ
　　頂点に関する処理を行う

フラグメントシェーダ
　　テクスチャなどの物体の表面をゴージャスにする
　　処理を行う。
　　ポストエフェクト。

シェーダーは、プログラム実行時にビルドし利用する。
プログラムから見ると、シェーダーはただの文字列です。
static const char vsh[] = "シェーダーの処理";
こんな感じ。

そろそろ本題ですよ

■まずは座標変換系のお話

3D空間上にある地球と月を、画面に描画する手順を示す。
登場人物

・地球
　モデルデータ（メッシュデータ）
　地球テクスチャ

・月
　モデルデータ（メッシュデータ）
　月テクスチャ

・カメラ

・ライトは今回は省略

描画目標

■座標変換って

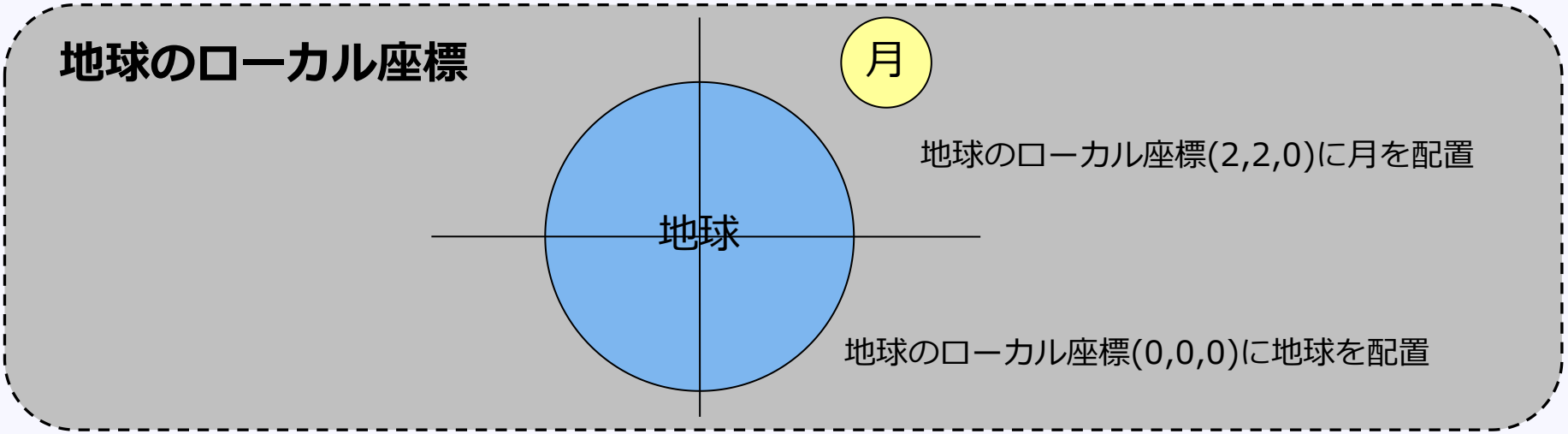
3D座標空間上に配置されたモデルを画面に描画するには、
何度も何度も座標の変換作業が必要となる。
座標変換の流れは下記の通り。

１、ワールド座標変換
２、ビュー座標変換
３、プロジェクション座標変換
４、スクリーン座標変換

■ローカル座標

ローカル座標
　そのモデルの中だけで使われる座標。
　親子関係を相対的に管理するのに向いている。

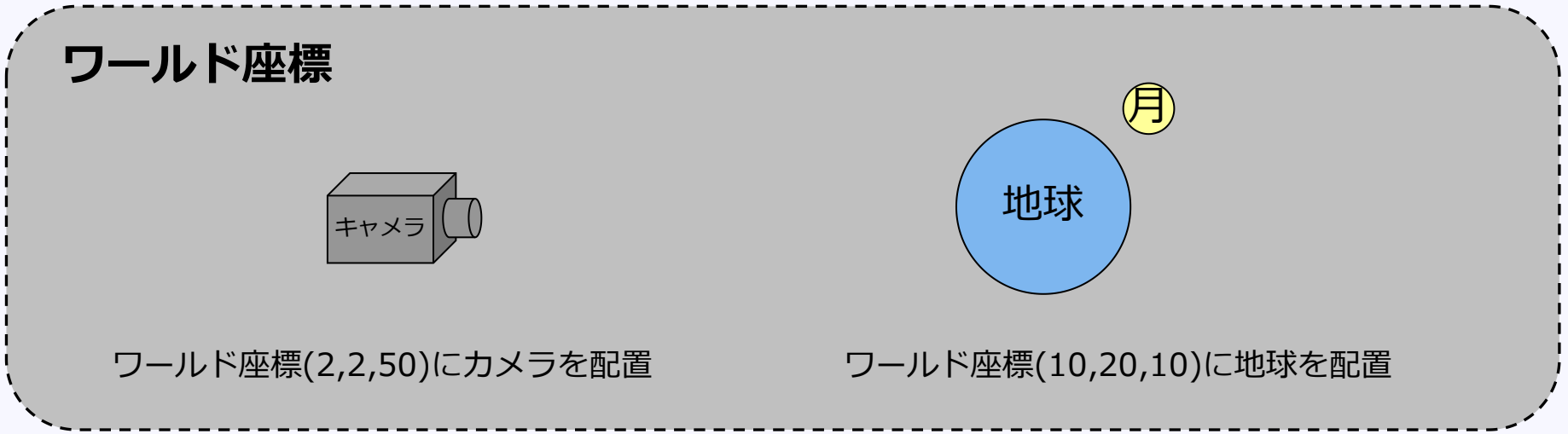
月は地球の周りを廻っているので、地球と親子関係がある。
よって地球のローカル座標で管理すると扱いやすい。



■ワールド座標

ワールド座標
　すべてのモデル、カメラで共通に利用する座標。
　絶対座標。

今回はカメラと地球(+月)をワールド座標で扱う。



■ワールド座標変換

ローカル座標で管理しているモノを、
ワールド座標に変換する工程。

月は地球のローカル座標の値しか持っていないので、
ワールド座標値に変換を行う必要がある。
月の相対座標が(2,2,0)で、
地球のワールド座標が(10,20,10)なので、
月のワールド座標は(12,12,10)ということとなる。

もちろんモデルデータは頂点の集まりなわけだから、
全頂点に対してこの処理を行うこととなる。
ここまでは簡単。

■ビュー座標変換

カメラから見た座標に変換する工程。

ビュー座標変換に必要な情報。
・視点（カメラの位置）
・注視点（焦点、カメラが見ている位置）
・カメラの角度（上はどっちなのかってこと）

変換内容は、
カメラの座標を(0,0,0)に移動し、
それに合わせて地球(+月)も移動する。
合わせてカメラの角度分、地球と月を回転させる。

■プロジェクション座標変換

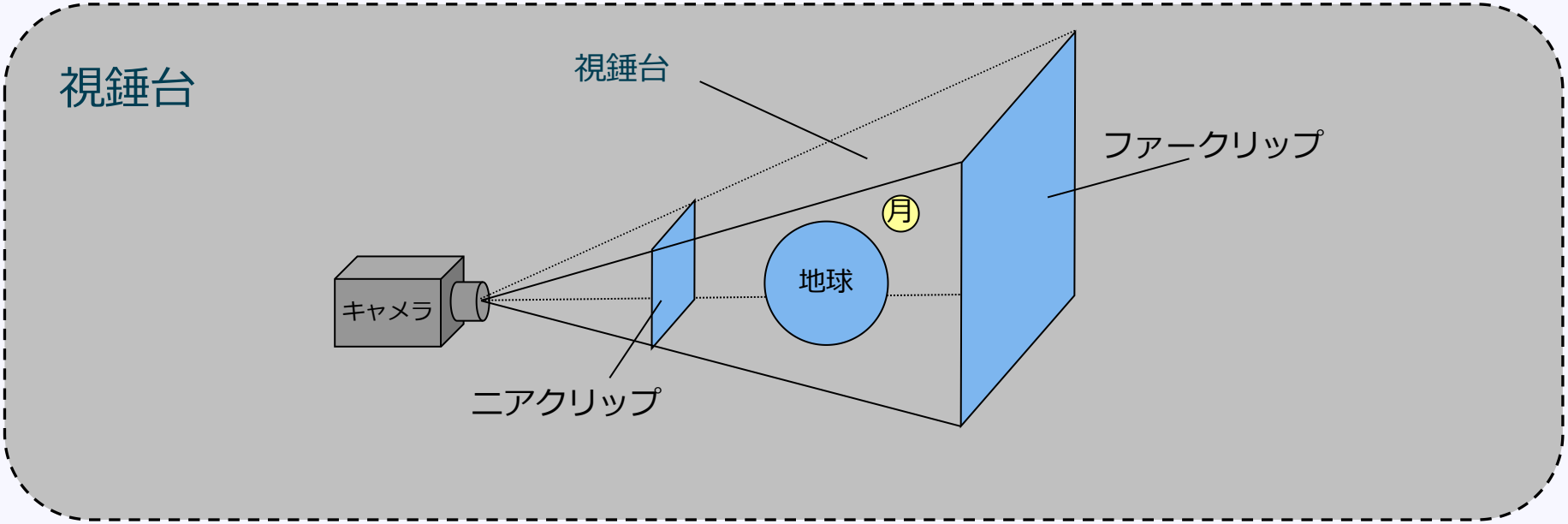
2Dに変換する工程。
透視投影と並行投影の2種類がある。

透視投影は、
　近くのモノは大きく、遠くのモノは小さく変換する。

並行投影は、
　距離に関係なく一定の大きさに変換する。

どちらの変換でも、
遠くのモノほど画面の中央に集まることとなる。

プロジェクション座標変換に必要な情報。
・カメラの画角（FOV、焦点距離、映す広さの事）
・前後の描画範囲（ニアクリップ・ファークリップ）
この描画範囲を視錘台という。これを設定しないと
無限遠まで描画することとなり死亡。
視錘台の外にあるものは描画しない。



■スクリーン座標変換

ディスプレイの座標に変換する工程。

スクリーン座標変換に必要な情報。
・ディスプレイの解像度

変換内容は、
プロジェクション変換により、各座標は-1～1の間に収まる値になっている。
これを実際にのディスプレイの座標に変換する。

■座標変換を終えて

これで地球や月モデルの描画位置を、求める事が出来た。
だけど、これってGPUで全部やってくれる話なのか？
んーNO。

この変換を行う為の変換行列というのを、
プログラム側(CPU側)で作ってやる必要がある。
ちなみに変換行列はプロジェクション座標変換までを行える行列。（スクリーン座標変換は含めない）

この変換行列は各オブジェクトに対して基本的に一つずつ用意する。※同じ変換で済む場合はこの限りではない。

この変換行列とモデルデータをGPUに渡してやると、モデルデータのすべての頂点を、高速に座標変換してくれる。

この作業を行うのがバーテックスシェーダなり。

GPUとのやり取りはOpenGL命令を使う。

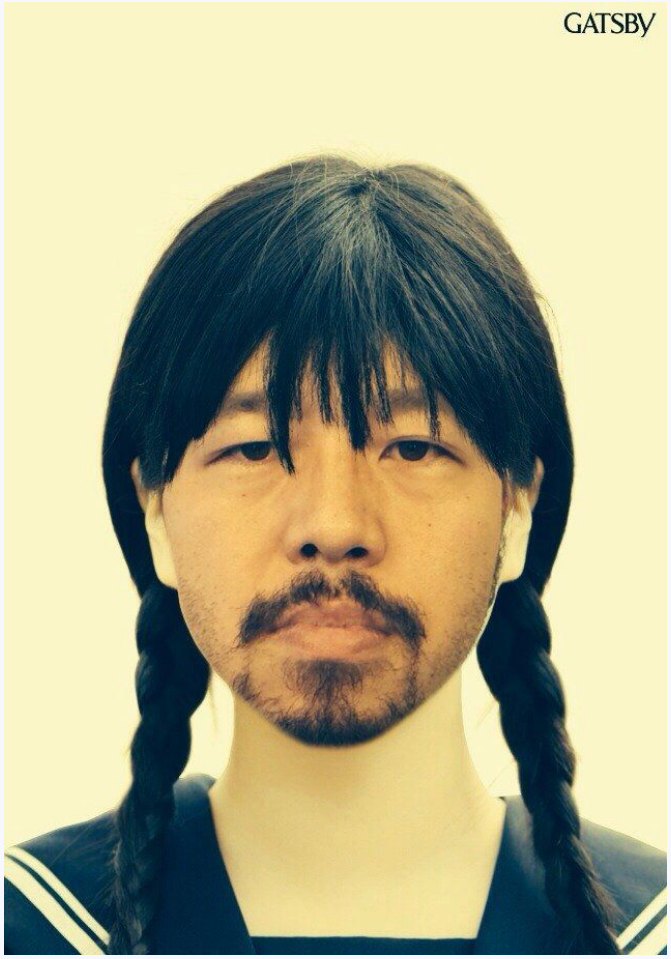
■俺と絵作りしないか？

座標変換で行われたのは、画面上のどこに、どの大きさで
モデルデータを表示するかってことまで。

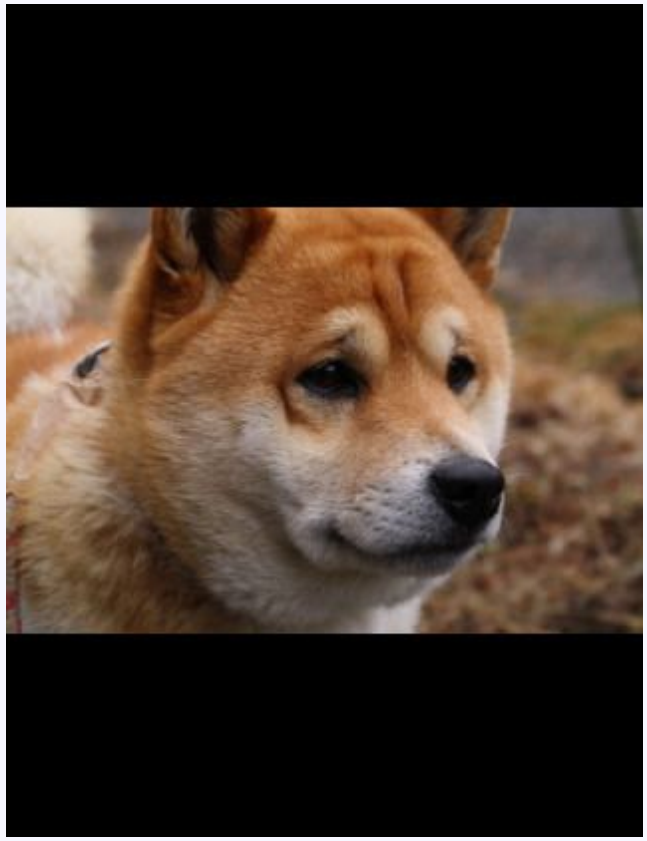
色を付けたりテクスチャ貼ったり、ライトを当てたりとか、
見た目をよくするのはフラグメントシェーダの仕事。
ここがシェーダを使う面白いところなんです！
しかしここだけで大ボリュームなので今回は説明省略…。

ここまでの処理が終わると、めでたく画面に描画されること
となります。

おつかれさまでした。
癒し画像を用意しましたのでご堪能ください。



hosaka



しかしまだ終わらない

■プログラム寄りの話

描画の仕組みを説明してきましたが、
もうちょっとだけ実践寄りの説明をします。

最初の方にGPUを使うためには、CPUからの通信が必要で
コストがかかるという話をしましたね。

それを踏まえての高速化の話をします。

■テクスチャについて

テクスチャサイズ

・２のべき乗にするべし
OpenGL ES 2.0では2のべき乗以外のサイズも扱えるが、
内部的には2のべき乗で扱うため。

テクスチャ枚数

・枚数を減らし、大きなテクスチャにまとめるべし
OpenGLでテクスチャを利用する際に、
バインド（テクスチャの切り替え）する必要があるため。

ついでに話しておくと、描画命令を発行する順番も大切。
同じテクスチャを使う描画命令は、
同じタイミングで呼ぶようにしておけば、
複数の描画命令に対してバインドするのは一回だけで済むこととなる。

さらに話しておくと、テクスチャのバインドに限らず、
同じ座標変換行列を使うもの等、同じ処理内容をさせるオブジェクト群は同じタイミングで描画命令を発行することで、高速化につながる。(Unityで言うところの動的バッチ)

さらにさらに。建物とか動かない静的なオブジェクトは、
VBOという仕組みを利用することで、高速化ができる。
動かないものは再計算する必要が無いから、GPUにキャッシュしておくって方法。(Unityでいうところの静的バッチ)

■テクスチャ転送とGPU

テクスチャをGPUに転送する命令はかなりの高コスト。
・glTexImage2D
・glTexSubImage2D
↑こいつらです。

GPUアーキテクチャよもやま話

iPhoneとAndroidの一部のGPUにはタイルベースレンダリングという仕組みが採用されている。一般的なGPUは描画命令順に描画をこなしていく（単純に奥にあるものから描画していく）。つまり、後から描画されるものに上書きされる部分が多々あり、無駄も多い。一方タイルベースでは、描画命令キャッシュし、ピクセルを描画するのに必要な情報を計算で求めてから描画する。つまり、描画処理は描画されるピクセル数だけで済むので高速である。ただし、一部のOpenGL命令が処理の途中に入ってると、それまでにキャッシュしておいた命令をいったん実行しなくてはならなくなるので、描画の負荷が一気に２倍３倍になったりするので要注意。

■カリングについて

カリングとは画面に描画されないところを事前に調べて、
描画処理を行わなくし高速化をする方法。

視錐台カリング
座標変換の話でも出てきた視錐台。
視錐台に入らないオブジェクトは描画しないことによって
高速化する方法。

オクルージョンカリング
オブジェクトが複数ある場合、手前にあるオブジェクトに
よって後ろのオブジェクトが隠れるなら、そのオブジェク
トは描画しないことで高速化する方法。

ご清聴ありがとうございました