

目次

前書きに代えて ― ゲーム開発者の数学	iii
---------------------------	-----

第1章 三角関数 1

1.1 三角形	2
1.2 直角三角形	3
1.3 ピタゴラスの定理	4
1.4 サイン、コサイン、タンジェント	4
1.5 三角関数の周期性	7
単位円	7
余弦定理	8
周期性	8
ラジアン	10
加法定理	11
サイン波、コサイン波	12
1.6 Unityサンプル：クリック位置を向くカプセル・バウンドする球	15
挙動と仕様	15
実装コード	20

第2章 座標系 27

2.1 デカルト座標系	28
2D座標系	28
3D座標系	29
左手系と右手系	32
ローカル座標とワールド座標	34
Center/PivotとLocal/Global	35
スクリーン座標	38
2.2 極座標系	39
2Dの極座標系	39
3Dの極座標系＝球面座標系	41
2.3 Unityサンプル：3人称視点カメラ	43
挙動と仕様	43
実装コード	46

第3章	ベクトル	51
3.1	ベクトルの定義	52
	数ベクトル	52
	幾何ベクトル	53
	スカラー	54
3.2	ベクトルの演算	54
	加法・減法・交換法則・結合法則	54
	スカラー乗法・除法	55
	単位ベクトル	56
	基底と座標系	56
	法線ベクトル	58
	大きさ	58
	内積	59
	ベクトルの正射影	62
	内積の応用	63
	外積	64
	外積の応用	66
3.3	Unityサンプル：簡易当たり判定	67
	挙動と仕様	67
	実装コード	70
第4章	行列	73
4.1	行列の定義	74
	定義	74
	行列の種類	74
4.2	行列の演算	75
	行列と行列の乗算	75
	転置行列	78
	逆行列	79
	行列とベクトルの乗算	80
	スウィズル演算	81
	行優先と列優先	82
	AoSとSoA	84
	行列式	85
	直交行列	88
4.3	Unityサンプル：行列計算のためのInspector拡張	90
	Unity Editorの拡張機能	90
	挙動と仕様	91
	実装コード	93

第5章 座標変換 101

5.1 座標変換とは何か	102
5.2 ジオメトリパイプライン	103
モデル変換	103
ビュー変換	105
プロジェクション変換	106
5.3 同次座標系	110
同次座標系とは何か	110
同次座標系と射影幾何学	111
Unityにおける同次座標系	111
5.4 変換の種類	113
線形変換	113
アフィン変換	114
剛体変換	115
射影変換	115
5.5 座標変換の行列表現	116
平行移動	116
回転	118
スケール	120
モデル変換	121
基底変換	122
ビュー変換	124
透視投影変換	127
正投影変換	133
等角投影変換	135
5.6 Unityサンプル：行列によるアフィン変換とプロジェクション変換	140
挙動と仕様	140
実装コード	142

第6章 クォータニオン 145

6.1 回転表現の種類	146
オイラー角	146
ロドリゲスの回転公式	147
ジンバルロック	149
回転行列の問題	150
クォータニオンの利点	151

6.2	クォータニオンの定義	152
	複素数	152
	二重数	154
	クォータニオンとは何か	154
6.3	クォータニオンの演算	156
	スカラー倍	156
	共役	156
	大きさ	156
	乗算	157
	単位クォータニオン	157
	内積	157
	逆数	158
	行列表現	159
6.4	クォータニオンによる3D回転	160
	回転	160
	補間	165
6.5	デュアルクォータニオン	167
	定義	167
	スキニングへの応用	167
	演算	170
	剛体変換	171
6.6	Unityサンプル：クォータニオンによる回転	172
	挙動と仕様	172
	実装コード	173

第7章 曲線177

7.1	曲線をめぐる概念	178
	補間と近似	178
	パラメトリック関数	179
	多項式	180
	曲線とスプライン	181
	連続性と微分	182
	二重数による自動微分	184
7.2	曲線のアルゴリズム	185
	ベジエ曲線	185
	Catmull-Romスプライン	187
	Bスプライン	190
7.3	Unityサンプル：3種の曲線	194
	挙動と仕様	194
	実装コード	197

第8章 ゲームアプリの環境 203

8.1 ゲームエンジンとしてのUnity	204
シーングラフ	204
コンポーネント指向	205
サンドボックス	208
ゲームループ	209
CPUと並列性	210
GPUフロントエンド	212
8.2 スマートフォンアーキテクチャー	213
グラフィックスドライバー	213
ドローコールバッチング	217
GPUアーキテクチャーの変遷	219
低レベルグラフィックスAPI	223
ヘテロジニアスアーキテクチャー	225
8.3 グラフィックスパイプライン	227
OpenGL ES 2.0	227
CPUによるドローコール生成	228
バーテックスシェーダー	230
プリミティブアセンブリー	231
ラスタライゼーション	232
フラグメントシェーダー	233
ROP処理	233
OpenGL ES 3.0以降	237
遅延シェーディング	237
テッセレーションシェーダー	238
ジオメトリシェーダー	239
コンピュートシェーダー	240

第9章 シェーダー 241

9.1 Unityのシェーダー	242
ShaderLab	242
シェーディング言語	243
GLSL	243
Editor設定	245
シェーダーの基本	246
外部ツールとWebGL	249

目次

9.2 照明	251
拡散反射	251
ランバート反射モデル	252
ハーフランバート拡散	256
フォン反射モデル	257
頂点単位照明	258
ピクセル単位照明	260
リムライティング	262
9.3 テクスチャ処理	263
テクスチャマッピング	263
法線マッピング	265
9.4 物理ベースレンダリング	270
BRDF	270
クックートランス鏡面反射	272
付録 Unityの物理エンジン	277
索引	279

第6章

クォータニオン

Unityでは、スクリプト内でオブジェクトを傾かせる、つまりオブジェクトに回転を加えようとしたとき、カジュアルに Quaternion クラスが登場してくる。一方で、本書で回転を表現するために使ってきたのは回転軸と角度で、回転の座標変換では各軸の回転行列に角度を与えて変換を行っていた。しかし実際には、Unity内部での回転のネイティブ表現には、行列ではなく、クォータニオンが使われている。

クォータニオンは、歴史的には19世紀にハミルトンによって発見され、ベクトル概念の派生元ともなった概念である。ゲーム開発への応用は、1996年の『トゥームレイダー』がクォータニオン採用の走りとして有名だ。

本章では、Unityを含む現代の洗練されたゲームエンジンで3D空間での回転表現のために一般的に利用されるクォータニオンについて解説する。クォータニオンがいったい何なのか知らずともゲーム作りをどうにか進められるのはUnityの長所だが、クォータニオンがなぜ用いられているかを把握することにより水面下で何が行われているか理解が深まるとともに、これまでに学んだ三角関数、ベクトル、行列といった概念の結びつきがより鮮明になるはずだ。

6.1

回転表現の種類

❖ オイラー角

Unity の GameObject に付いている Transform コンポーネントの Rotation は、各軸の角度（ラジアンではない度数）を指定してオブジェクトの回転状態を設定できる（図 6.1）。

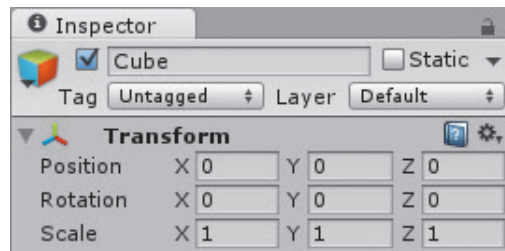


図6.1 TransformコンポーネントのRotation

Transform クラスは、第 5 章で学んだように、GameObject に行われているモデル変換を示す。しかし、前章のサンプルコードの Editor/Chapter05Editor.cs ファイルのコードを見ると、Transform の Rotation に角度を設定するのと同じ効果を回転行列による座標変換で得ようとして、以下のコードが記述されている。

```
rotation = EditorGUILayout.Vector3Field("Rotation", rotation);

if (GUILayout.Button("Apply")) {
    Matrix4x4 m = Matrix4x4.identity;
    m.SetTRS(Vector3.zero,
        Quaternion.Euler(rotation.x, rotation.y, rotation.z), Vector3.one);
    matrix = m * matrix;
}
```

Matrix4x4 クラスの SetTRS メソッドの第 2 引数は、回転行列を作成するために、Quaternion クラスを受け取るようになっている。ここでは、そのために Quaternion クラスの **Euler** メソッドで回転のための **オイラー角** (Euler angles) を z 軸、x 軸、y 軸ごとに与えて回転の表現を生成している。オイラー角とは、3次元ユークリッド空間での剛体 (rigid body) の傾きを、回転軸周りの 3 つの角度の組で表したものだ。オイラー角を用いれば、モデル座標とワールド座標という 2 つのデカルト座標系の位置関係を、3 つの角度の組として直感的に表現できる。

ただし、単に3つの角度を与えるだけでは、回転は一意に決まるわけではない。第5章までに行列について学んだように、行列の乗法では必ずしも交換法則が成立せず、3次元の回転軸周りの回転行列を複数合成する場合、異なる回転軸が含まれていると、掛け合わせる順番によって、できあがる回転は全く異なったものとなる（2次元の回転行列では軸が同じなので交換法則が成立する）。オイラー角でも、回転の順序が同じでなければ異なった回転になってしまうので、APIが異なる環境に回転操作を移植する場合は注意しなければならない。

オイラー角は、航空機やロボットの姿勢制御にも用いられている。図6.2に示すように、前後軸周りの回転を**ロール** (roll)、水平軸周りの回転を**ピッチ** (pitch)、垂直軸周りの回転を**ヨー** (yaw) と呼ぶこともある ($x-y-z$ 軸と、ロール-ピッチ-ヨーの対応関係は任意)。また $z-x-z$ のように、3軸全ての回転を使わない回転順序も広く使われている。この点、Unity の Quaternion クラスの Euler メソッドは、 $z-y-x$ の順序での回転を生成する。

6

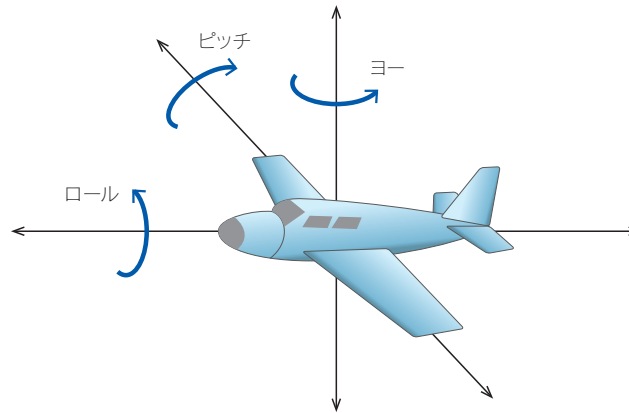


図6.2 ロール、ピッチ、ヨー

❖ ロドリゲスの回転公式

Unityでは直接は用いられないが、**ロドリゲスの回転公式** (Rodrigues' rotation formula) という、任意軸周りの回転を表現するための式が知られている。

ベクトル $\mathbf{v}$ を単位ベクトル $\mathbf{u}$ を軸として角度 θ だけ回転させたベクトル $\mathbf{w}$ は

$$\mathbf{w} = \mathbf{v} \cos \theta + (\mathbf{u} \times \mathbf{v}) \sin \theta + (\mathbf{u} \cdot \mathbf{v})(1 - \cos \theta)\mathbf{u}$$

で表すことができる (図6.3)。

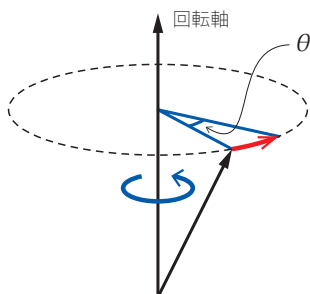


図6.3 ロドリゲスの回転公式

ロドリゲスの回転公式の考え方は、回転対象のベクトルを、回転軸ベクトルに平行な成分と垂直な成分に分解し、垂直成分だけを2次元の部分空間 (subspace) 上で回転させた上で、再度平行成分と合成するというものだ。

また、**オイラーパラメーター** (Euler parameters) という数値の組をロドリゲスの公式に当てはめて回転を求める、**オイラー-ロドリゲスの公式** (Euler-Rodrigues formula) と呼ばれる式も存在する。オイラーパラメーターは4つの**実数** (real number) である a, b, c, d によって

$$a^2 + b^2 + c^2 + d^2 = 1$$

と表現される。実数とは、分数で表される**有理数** (rational number) と無限小数で表される**無理数** (irrational number) の総称で、実数全体の集合を $\mathbb{R}$ と表記する。同じ記号で $\mathbb{R}^3$ のように書くと、3次元ユークリッド空間内の位置全体の集合を示す。

単位ベクトル $\mathbf{u} = (u_x, u_y, u_z)$ を軸として角度 θ だけ回転させるとしたとき、

$$a = \cos\left(\frac{\theta}{2}\right)$$

$$b = \sin\left(\frac{\theta}{2}\right)$$

$$c = \sin\left(\frac{\theta}{2}\right)$$

$$d = \sin\left(\frac{\theta}{2}\right)$$

は、半角の公式から、オイラーパラメーターを満たす。回転軸のベクトルを $\mathbf{e} = (u_x \sin(\frac{\theta}{2}), u_y \sin(\frac{\theta}{2}), u_z \sin(\frac{\theta}{2}))$ にとすると、オイラー-ロドリゲスの回転公式は

$$\mathbf{w} = \mathbf{v} + 2a(\mathbf{e} \times \mathbf{v}) + 2(\mathbf{e} \times (\mathbf{e} \times \mathbf{v}))$$

と書くことができる。

❖ ジンバルロック

オイラー角を用いた回転(=ある順序を持った3回の回転軸周りの回転)は、3つの数字で表せ、シンプルでわかりやすいが、問題もある。第1の問題点は、**ジンバルロック**と呼ばれる現象が起こるというものだ。

航空機や船舶、宇宙船は、ジャイロスコプという機器で、角度や角速度 (angular velocity) を計測して姿勢制御を行っている。スマートフォンにも3軸の自由度 (DOF) を測れるジャイロスコプが搭載されており、加速度センサーの3軸による平行移動加速度測定とあわせ、6軸自由度の移動検出ができる。

ジャイロスコプの内部にはジンバルという部品があり、3つの環が各軸の回転を表現する。内側の環は外側の環の回転に連動して位置を変えることによって、ローラーピッチーヨーのような3軸自由度の回転に対応している。

ところが、通常は3軸の自由度を持って回転を表現できるジンバルの環が重なることにより、1番目の回転と3番目の回転の2つの軸がほぼ一致し、自由度が2軸に限定されてしまう事象が発生する。これがジンバルロック (gimbal lock) と呼ばれる現象だ (図6.4)。

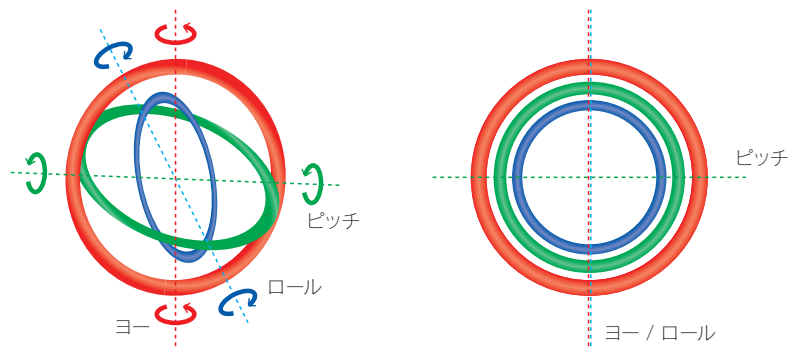


図6.4 ジンバルロック

例えば、 $z-x-z$ の順序でオイラー角を使う場合、 x 軸と z 軸の回転行列は第5章で見たように

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$\mathbf{R}_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

第9章

シェーダー

いよいよシェーダーについて解説する最後の章となった（本章ではシェーダープログラムのことをシェーダーと略記する）。これまで三角関数、ベクトル、行列などを扱ってきたが、全てシェーダーの理解のための準備であったといっても過言ではない。裏返せば、一般的にゲームアプリで用いられるシェーダーの理解には数学的概念の基礎的理解が欠かせず、GPUプログラミングへの入り口のハードルとなりやすい。そのハードルを乗り越えた上で、GPUとは何か、なぜシェーダーを学ぶのかというイメージも掴めているはずだ。前章で学んだように、CPUがせいぜい数個のコアからなるプロセッサであるのに対し、GPUとは数百個のコアを持つ超並列計算のためのプロセッサで、多数のジオメトリやフラグメントを同時並行処理する能力を持つ。ゲームアプリの演算の主役だ。

本章ではシェーディングのための理論と、そのシェーダーによる実装とを交互に眺めながら、シェーダーに慣れ親しむことを目指す。ここまで読み進めてくることができたのであれば、本章を咀嚼するのに要する知識は全て揃っているはずだ。シェーディングに用いられる理論は、ゲームアプリを作る上では100%理解する必要はなく、カスタマイズする際にどうすればよいかという点さえ意識すればよい。そうしてシェーダーに慣れたら、後は世に溢れるシェーダーを参考にしたり、論文を読んで表現に使える理論をシェーダーへ落とし込んでゲームアプリへ応用するなり、自由自在だ。

9.1

Unityのシェーダー

❖ ShaderLab

Unityでのシェーダーは、前章で紹介した**ShaderLab**というUnityの独自形式で外側の殻を書いて主にOpenGL ESの内部状態関連の設定を行い、その内側に、GPUに渡されるシェーダー本文として、各種のシェーディング言語で書いたバーテックスシェーダーやフラグメントシェーダーを埋め込む形態を取る。

例えば、一番シンプルなShaderLabでの指定は以下のようなものだ。

```
Shader "Custom/Test" {  
    Properties {  
  
    }  
    SubShader {  
        Pass {  
            // ここにシェーダーが埋め込まれる  
        }  
    }  
    FallBack "Diffuse"  
}
```

Shaderの後に、スラッシュ(/)の入った名前を入れておくと、マテリアルにシェーダーを設定する際に、カテゴリー分けされて選べるようになる(図9.1)。

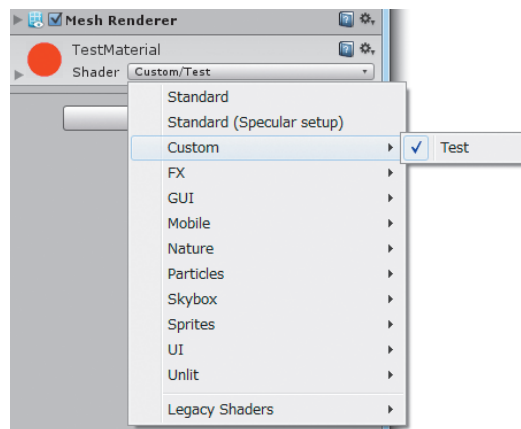


図9.1 シェーダーの名前

PropertiesにはInspector上でユーザーが調整できる、シェーダーへの入力パラメーター（色やテクスチャーなど）をプロパティとして記述する。Subshaderブロックは複数書け、Unityが上から実行してそのプラットフォームで実行可能なものを選択するので、複数プラットフォーム向けのシェーダーを分けて書いておける。どのSubshaderもまともに動かない場合は、Fallbackに指定してあるDiffuseという最も基本的なシェーダーが適用される。Passは1回のレンダリングパイプライン通過を表し、同じオブジェクトを複数回レンダリングするマルチパス（multi-pass）のシェーダーを作りたい場合は、Passブロックを複数書く。

❖ シェーディング言語

GPU向けのシェーディング言語は複数あるが、本書で使うのはOpenGL ESのシェーディング言語である**GLSL**（OpenGL Shading Language）だ。

他に、NVIDIAが開発した最初のシェーディング言語である**Cg**や、MicrosoftがDirectX向けに開発した**HLSL**もUnity上で利用できる（Unity内部で最適化されたGLSLへトランスレートされる）。Unityで空のシェーダーを作成すると、Cgで書かれたシェーダーができる。しかし、Cgそのものは2012年に開発終了となっているため、本書ではGLSLを使ってサンプルを記述する。GLSLはWebGLでも利用されるので、WebGLの理解が進むという利点もある。Cocos2d-xでゲームアプリ開発を行う場合にも、GLSLでシェーダーを書くことになるはずだ。

また、Unityが提供している標準のシェーダーは**サーフェースシェーダー**（surface shader）といい、これは低レベルのシェーダーと違って、Unityのシーンで起こっている高度な照明などの処理をUnity側で適用してくれるもので、ユーザーがGPUを100%制御できない代わりに、ユーザーはUnity側で作った効果との差分のみをCgなどのシェーダーで記述すればよい。本書では、学習目的のためにサーフェースシェーダーは利用せず、生のバーテックスシェーダーやフラグメントシェーダーをGLSLで書く。

❖ GLSL

GLSLはC言語とほぼ同じ文法で、型を指定して変数を宣言する。ただし、ポインター型はない。型には、スウィズル演算をサポートする2/3/4次元ベクトルvec2/vec3/vec4や、2/3/4次元正方行列mat2/mat3/mat4などグラフィックス演算で使えるものが揃っている。近年のモバイルGPUはVLIWではなくスカラープロセッサのものが多く、vec4同士の乗算は無駄を省くため避けたほうがよい。行列クラスの[]演算子での添字アクセスは、列ベクトルを返す。配列や、structで構造体も定義できる。ベクトル同士の*演算子による乗算の演算は、成分同士の演算で結果もベクトルとなり（内積の結果がスカラーとなるのと異なる）、行列でいうアダマール積（Hadamard product）となる。正規化されているベクトル同士を乗算する場合、成分は1以下なので、結果のベクトルの成分は1を超えない。

OpenGL ES向けのGLSLであるGLSL ESにはバージョンがあり、OpenGL ES 2.0ではGLSL ES 1.0、OpenGL ES 3.0ではGLSL ES 3.0が使える。それぞれ、Khronos Groupが「The OpenGL ES ® Shading Language」としてバージョン別に仕様書を配布している。GLSL ES 3.0の場合、先頭に

```
#version 300 es
```

と書いて区別する。GLSL ES 3.0では正方行列でない行列として、例えば 3×4 行列ならmat3x4などの型が追加された。GLSLとGLSL ESの差異は基本的にUnityがシェーダーコンパイル時に吸収するのでUnityを利用するかぎりではあまり意識する必要はない。本書でも、GLSLと書く場合は、原則的にGLSL ESも含めて指すことにする。

第8章で紹介したvaryingやuniformといった変数への指定は、C言語におけるconstのような修飾子(qualifier)で、intやfloatなどの型指定の前に置き、変数の保存場所や振舞いを指定する。各ジオメトリのパーテックスシェーダーから対応したフラグメントシェーダーに渡したい値にはvaryingを、全てのシェーダーで並列に共有する値にはuniformを使えばよい。他に、頂点ごとに渡される組み込み変数としてattributeがある。GLSL ES 3.0ではvaryingやattributeは廃止され、代わりに各ステージの入力をin、出力をoutと指定する。

シェーダーが処理を開始するエントリーポイントはmain関数として書く。ユーザー定義関数も作成でき、引数指定にinを指定すると値渡し、outであれば参照渡しでC#のout同様、inoutであればC#のref同様の既存変数の参照渡しとなる。

GLSLは概ねコンパクトな言語仕様を持つので、より複雑なC#言語を理解しているならば、GLSLの理解はたやすいはずだ。CgやHLSLも、組み込み変数名や型の名前が少々違ったりするくらいで、機能的には大した違いはなく、GLSL同様にC/C++言語に似ているので読む際にはさほどの手間を要しないだろう。細かい差異で言えば、例えばCgならばfixedという12ビット固定小数点数の型があるところ、GLSL ESでは型指定子(type qualifier)のhighpで最低16ビット、mediumpで最低10ビット、lowpで最低8ビットの精度を指定できる。ただし、各GPUがサポートする実際の精度はOpenGL ES APIのglGetShaderPrecisionFormatで取得しないとわからないので、同じシェーダーでGPU毎に描画結果が異なってしまう場合はサポートする精度を調べてみるとよい。低精度で計算したほうが、正確さで劣るものの、シェーダーが高速になる場合が多い(特にフラグメントシェーダーで効果が大きい)。

GLSLではバージョンによっては整数を小数へキャストできないので0や1ではなく0.0や1.0のように書かないとエラーの原因となる。スカラーとベクトルの変換と演算順序も注意すべきだ。

シェーダープロセッサのコアはCPUコアに比べると非常に弱く、またデータの局所性を高めるために、レジスターのようなローカルリソースには厳しい制約がある。最低限保証されている各リソ

ースの最大数は定数で参照でき、OpenGL ES 2.0であればGLSL ES 1.0上で以下の数値となっている。

```
const mediump int gl_MaxVertexAttribs = 8;
const mediump int gl_MaxVertexUniformVectors = 128;
const mediump int gl_MaxVaryingVectors = 8;
const mediump int gl_MaxVertexTextureImageUnits = 0;
const mediump int gl_MaxCombinedTextureImageUnits = 8;
const mediump int gl_MaxTextureImageUnits = 8;
const mediump int gl_MaxFragmentUniformVectors = 16;
const mediump int gl_MaxDrawBuffers = 1;
```

例えばvarying変数の最大数gl_MaxVaryingVectorsは8個であり、gl_MaxCombinedTextureImageUnitsはバーテックスシェーダー／フラグメントシェーダー両方合わせてのテクスチャー参照の最大数で、8個となっている。この定義はGPU依存なので、GPUによってはもっと高い値が設定されている場合もある。

UnityによるGLSLの拡張として、よく使うプリプロセッサマクロや関数はファイルにまとめて再利用することもできる。Unityも小さなライブラリを提供しており、ほとんどはCg言語向けだが、GLSL向けもある。WindowsであればUnityをインストールした場所のEditor¥Data¥CGIncludes以下、MacであればContents/CGIncludes以下に各シェーダー言語での定義済み変数と関数のインクルードファイルがあるので、シェーダープログラム内で

```
#include "UnityCG.glsinc"
```

と書いてインクルードすると、モデル変換行列_Object2Worldなど、インクルードファイルの内部に定義された変数を利用できる。

❖ Editor設定

Microsoft Windowsでは、Unity EditorでGLSLを使うための環境設定が必要となる（Macの場合は、実機のOpenGL ESでの実行結果とは微妙に異なる可能性があるが、PC用のOpenGLでGLSLを使える）。

Unity 5.1からGLレンダラーが一新され、PC上のUnity EditorでもOpenGL ESのシェーダーをAndroidやiOSの実機と同様に実行して結果を確認できるようになった（Unity 5.1.2時点ではWindowsのみ実装されている）。

FileメニューのBuild SettingsからPlayerSettingsを開き、Automatic Graphics APIのチェックを外してGraphics APIのリストの+を押下しOpenGLES3を追加してからOpenGLES3をドラッグ