目 次

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）。

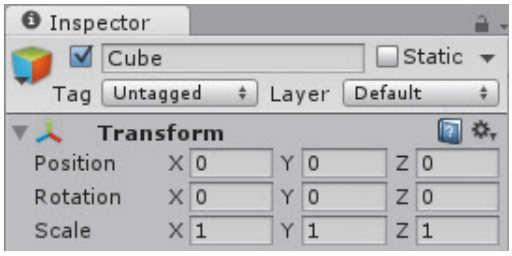

図6.1　TransformコンポーネントのRotation

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。

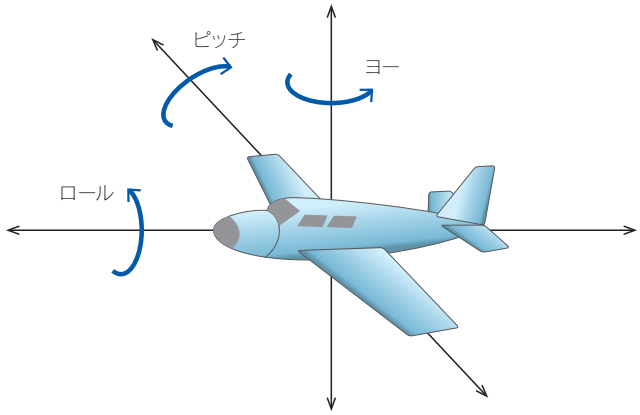

図6.2　ロール、ピッチ、ヨー

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）。

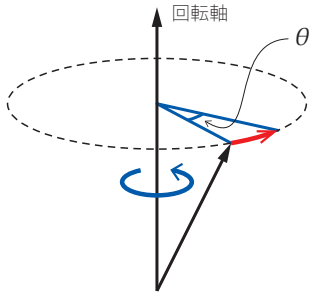

図6.3　ロドリゲスの回転公式

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$\theta$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$\mathbf{e} = \left(u_x \sin\left(\frac{\theta}{2}\right), u_y \sin\left(\frac{\theta}{2}\right), u_z \sin\left(\frac{\theta}{2}\right)\right)$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、ジャイロスコープという機器で、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ー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。

ジャイロスコー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、ロール－ピッチ－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）。

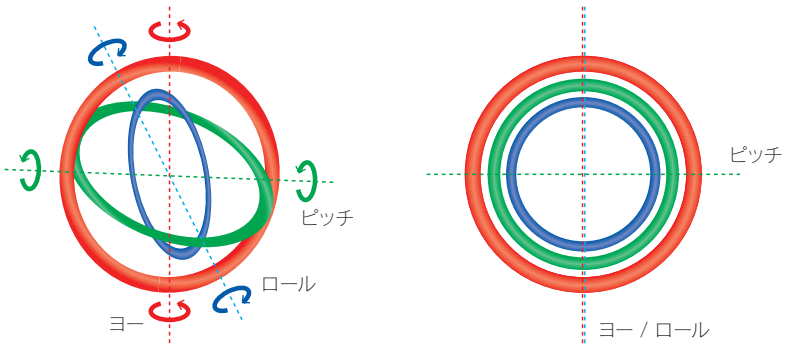

図6.4　ジンバルロック

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ーであるのに対し、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ーで、多数のジオメトリー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するなり、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)。

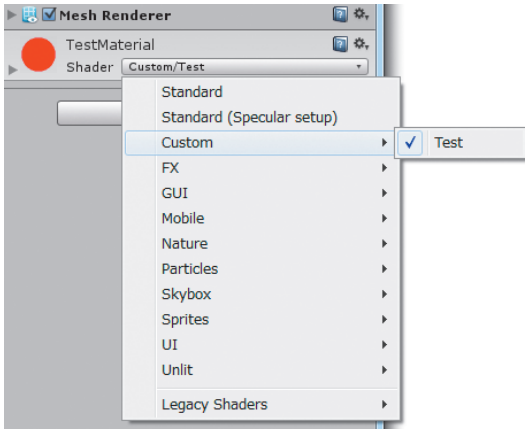

図9.1　シェーダーの名前

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ーのものが多いので、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。各ジオメトリーのバ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ーのコアは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ー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.glslinc"
```

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