

# 第 16 章

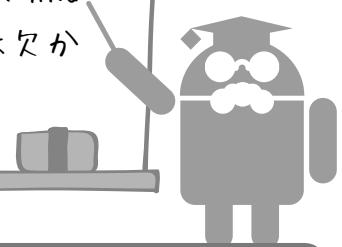
## グラフィック

著：山下武志

# 16-1 Android での グラフィック処理

著：山下武志

モバイル端末は「電話」としての役割の他に、ゲーム等のマルチメディア用途でも多く使用されています。この節では、Androidプラットフォームでの高速なグラフィック処理には欠かすことができない「OpenGL ES」について取り上げます。



## この節で学ぶこと

- ・CPUとGPUの違い
- ・Android標準のViewを使用しない描画
- ・OpenGL ESの基礎
- ・シェーダーの記述方法
- ・多角形の描画
- ・テクスチャ

## この節で出てくるキーワード一覧

GLSurfaceView	プリミティブ
OpenGL ES 2.0	テクスチャ
GLSL ES	
static import	
gl_Position	
gl_FragColor	
コンパイル	
リンク	
ビルド	
クライアント	
サーバー	
RAM	
VRAM	



## 16-1-1 モバイル端末でのグラフィック処理

Androidを搭載している端末には、ほぼ間違いなくディスプレイが搭載されています。もしディスプレイがないとしても、HDMI等へ画面が出力できない機種はほぼ一般に流通していないと言って過言ではありません。

この「ディスプレイに何かが表示されている」というのは、最終的にはAndroid端末が「グラフィックを出力する」という処理を行っているということです。この処理が無ければどんなアプリが起動しているかもわかりませんし、文鎮と大差ありません。習字にでも使いましょう。

このグラフィックを司っている機器(ハードウェア)は、大きく2種類に分けることができます。1つはCPU(Central Processing Unit)で、もう1つはGPU(Graphics Processing Unit)です。グラフィック処理の歴史はCPUのほうが古く、GPUは比較的新しい「新参者」といえます。どちらもAndroidには古くから搭載されており、現在においても現役で活躍しています。また、「端末のディスプレイに図柄を出力できる」という機能ではどちらも変わりありません。しかし、それらにはまったく違う特徴があり、明確な使い分けを行わなければなりません。

### CPUでのグラフィック処理

CPUでのグラフィック処理はAndroidではCanvasクラスやViewでの描画に取り入れられています。本章では便宜上「CPU描画」と呼びます。CPU描画の特徴は描画のほぼすべてがソフトウェアによって記述されているということです。ソフトウェア制御であることから柔軟性が高く、多くの端末で同様に動作します。モバイル用途とは少し離ますが、ハリウッド映画等で多用される3DCGも、最終的にはソフトウェア描画によって出力が行われます。特に、2014年現在のGPUが得意とする「曲線描画」は、CPU描画の得意分野といえます。たとえば多くの曲線によってでき上がる「文字」の描画がそれにあたります。

この描画方法の弱点は、「CPUで描画を行うこと」自体にあります。「高負荷な処理」の代表例と言ってもいいグラフィック描画をCPUで行わせるということは、それだけOSやアプリ自体の処理に割く余力が減ることであり、アプリの動作を妨げてしまう恐れがあります。

### GPUでのグラフィック処理

Androidでは「OpenGL ES」を使うことにより、GPUでのグラフィック処理を行えます。本章ではこれを便宜上「GPU描画」と呼びます。GPU描画の特徴はGPUという「描画に特化したハードウェア」を使用して、高速な描画が行えるということです。「GPU」という「CPU」から独立したハードウェアを使用することで、CPUによるアプ

リ側の処理を妨げること無く描画処理を行えます。昨今の3DゲームはGPUを使用することにより、美麗なゲーム画面を構築しています。

一方で、GPU描画の弱点は、やはり「GPUという専用ハードウェアを使用すること」です。専用ハードウェアを使用することで高速な描画が行える一方、ハードウェアの限界を超えた処理は行うことができません。

## なぜGPUが必要なのか

CPUというハードウェアは、C言語やJava言語で実装可能なすべての演算は実行できます。本書を読む読者が思いつく、ほぼすべての処理が行えると思ってください。むしろ、GPUよりも細やかな制御が行える、非常に万能な道具です。

さて、では万能が正義かといえば、必ずしもそれだけではありません。GPUというハードウェアが搭載されているのは、それ相応の理由があります。CPUは、例えるなら軽自動車です。それさえあれば日本中の大抵の場所を走ることができます。買い物に困ることは無いでしょう。舗装されていない農道だって走れちゃいますね。ですが、F1カーに匹敵する速度は出せません。想像してみてください。サーキットを走るF1カーに、軽自動車が追いつけるでしょうか。F1カーの最高時速は400km/h超です。どんなに性能の高い軽自動車でも、F1カーには勝てないでしょう。

さて、ではF1カーは農道を走れるでしょうか？ 軽自動車のように長い距離を走れるでしょうか？ たくさんの荷物を運べるでしょうか？ どんなに速いF1カーも、それは「サーキットで最高の性能を引き出す」という前提で造られています。そのためには悪路走行性能も、燃費も、積載性も必要ありませんね。

CPUが軽自動車だとしたら、GPUはF1カーです。「グラフィック処理を行うため」に進化した(進化している)、最強のF1カーです。「どこでも低燃費で行ってやるぜ」という日本の軽自動車のような万能さはありません。とにかく、自分の得意分野=グラフィック処理さえ高速に行えれば良いのです。

逆にGPUは、細やかな制御処理を非常に苦手としています。例えば、GPUはif文が苦手です。for文もちょっと苦手です。関数(メソッド)呼び出しあは行えません。再帰処理等のモダンで複雑な機能も使えません。Androidアプリのように大きなプログラムを実行することもできません。スレッドの細かい制御もできません。

一方のCPUは、グラフィック処理を「やってやれないことはない」です。ですが、サーキットで軽自動車を走らせるように、とてもとてもゆっくりとしたスピードでしか処理することができません。

GPUはグラフィック処理ただ一点に的を絞った高性能化を行い、Androidの基本的なView描画や、ゲームの描画処理を支えてくれています。

GPUは、正確には超並列的な浮動小数演算に特化しています。2014年現在の最新GPUは、192ものコアが並列処理を行っています。



## 16-1-2 OpenGL ES 2.0 の概要

単刀直入に言えば、「OpenGL ES」は主に3D描画を行うためのAPIです。それを応用することで2D描画を行うこともできますし、CPU描画やその他の技術を組み合わせることで文字列の描画もこなすことができます。ゲームエンジンとして有名な「Unity」や「Unreal Engine」も、描画部分はOpenGL ESを使用しています。

OpenGL ESには、2014年現在いくつかのバージョンがあり、Android端末では(OSやGPUによる制限はありますが)すべてのバージョンを使用することができます(表1)。

バージョン名	使用できるAndroidバージョン	特徴
1.0	すべて	OpenGL 1.3のサブセット
1.1	1.6以上	OpenGL 1.5のサブセット
2.0	2.0以上	プログラマブルパイプラインの導入
3.0	4.3以上	OpenGL ES 2.0との互換性を保つアップデート
3.1	L以上	コンピュートシェーダーの導入

表1:OpenGL ESのバージョンごとの機能比較

本章では2014年現在に普及しているほぼすべてのAndroid端末で動作可能なOpenGL ES 2.0を用いて解説します。



## 16-1-3 Hello OpenGL ES 2.0!

本章では「見本」となるプロジェクトと、読者が「演習」として実装を行うSkelt onプロジェクトの2種類を用意しています。Skeltonを加工し、各節ごとの課題を進めていきましょう。

また、基本となる演習と、その練習課題となる「TRY」、さらに難しく応用的な知識を試される「CHALLENGE」が用意されています。プログラムの方法は多数あるため、どのような方法で実現しても構いません。

### 演習:OpenGL ES 2.0の初期化を行う

各サンプルはFragmentを用いて実装します(リスト1)。Fragmentとは、Androidのアプリを構成する断片(フラグメント)を示すクラスで、複雑なアプリを開発する際には欠かせないものです。ですが、ここではFragment自体の機能は殆ど使いません。

リスト1:Chapter01\_01.java

```
public class Chapter01_01 extends Fragment implements
    GLSurfaceView.Renderer {

    protected GLSurfaceView glSurfaceView;

    @Override
    public View onCreateView(LayoutInflater inflater,
        ViewGroup container, Bundle savedInstanceState) {
        glSurfaceView = new GLSurfaceView(getActivity());
        glSurfaceView.setEGLContextClientVersion(2);
        glSurfaceView.setRenderer(this);

        // SurfaceViewを使う場合のおまじない
        // ※ これはWindowSystemの都合で、詳細は割愛
        glSurfaceView.setZOrderOnTop(false);

        return glSurfaceView;
    }

    @Override
    public void onPause() {
        super.onPause();
        glSurfaceView.onPause(); ②

        Log.i(getClass().getSimpleName(), "onPause");
    }

    @Override
    public void onResume() {
        super.onResume();
        glSurfaceView.onResume(); ③

        Log.i(getClass().getSimpleName(), "onResume");
    }

    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    }

    @Override
    public void onSurfaceChanged(GL10 gl, int width, int height) { ④
    }

    @Override
    public void onDrawFrame(GL10 gl) {
        GLES20.glClearColor(0.0f, 1.0f, 1.0f, 1.0f); ⑤
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
    }
}
```

このプログラムを実行すると、図1のような画面が表示されます。



図1:初期化して塗りつぶしたOpenGL ES 2.0の出力画面

ここからはリスト1の内容を順に追っていきます。まず、①はFragmentの描画用のViewを作成します。ここではGLSurfaceViewの生成と初期化を行い、描画用のViewとして返しています。

AndroidでOpenGL ESの描画を行うためには、**SurfaceHolder**か**SurfaceTexture**というオブジェクトを作成しなければなりません。Androidの初期から存在し、SurfaceHolderを内包しているのが**SurfaceView**です。多くのゲームはこのSurfaceViewを使用して開発されています。

現在はNativeActivityというC/C++で直接扱え、かつウインドウシステムと直結した仕組みを使用するケースも増えています

Androidには「EGL」という細やかな制御を行えるAPIと、「GLSurfaceView」というそれらをラッピングした簡易的なAPIが用意されています。本章はOpenGL ESの基本的な部分を学ぶという内容であるため、「GLSurfaceView」を使用し、初期化部分の詳細については言及しません。

「GLSurfaceView」は、初期状態ではOpenGL ES 1.1用に初期化を行います。今回はOpenGL ES 2.0を対象としますので、初期化バージョンを2へと切り替えます(リスト2)。

リスト2:setEGLContextClientVersion(OpenGL ESのバージョンを指定する)

```
glSurfaceView.setEGLContextClientVersion(2);
```

「GLSurfaceView」はGLSurfaceView.Rendererインターフェースの適切なメソッドを適切なタイミングで呼び出してくれます。逆にいえば、このインターフェースをセットしない限り「GLSurfaceView」は描画を行ってくれません。

Chapter01\_01クラスはFragmentを継承すると共に、「GLSurfaceView.Renderer」インターフェースの実装を行っています。「GLSurfaceView.Renderer」インターフェースで実装すべきメソッドは3つあり、それらについては後述します。

最後に、「SurfaceView」のZオーダーを変更します。これはAndroidのウィンドウシステムと密接に関わっており、他のViewとの位置関係を示します。ここでは「false」を指定することで、他のViewよりも後ろに表示されるようにします(**リスト3**)。

リスト3:setZOrderOnTop(SurfaceViewのZオーダーを指定する)

```
glSurfaceView.setZOrderOnTop(false);
```

初期値は端末により異なるため、必ず「true」または「false」の指定は行うようにしてください。

Androidのライフサイクルに合わせ、「GLSurfaceView」もライフサイクル処理を行わなければなりません。必要なある箇所は2つで、「Activity#onPause」と「Activity#onResume」です。呼び出すべきメソッドも同じ名前であり、**リスト1**の②③の部分で呼び出しを行っています。

**リスト1**の④の範囲にあるのが、「GLSurfaceView.Renderer」インターフェースで実装すべきメソッドです。「GLSurfaceView」自体はAndroidの最初期から存在するAPIということもあり、メソッドにはすべてGL11インターフェースが第1引数として渡されます。GL11インターフェースはOpenGL ES 1.1を使用するためのクラスです。ですが今回はOpenGL ES 2.0を使用するため、GL11インターフェースの出番はありません。

「onSurfaceCreated」は、「GLSurfaceView」用の描画メモリが確保されたタイミングで呼び出されます。後述する「画像の読み込み」等はここで行う必要があります。

「onSurfaceChanged」は、「onSurfaceCreated」の直後の他、Android端末の縦横を切り替えたタイミングや「SurfaceView」の大きさが変わったタイミングで呼び出されます。

前置きが長くなりましたが、**リスト1**の⑤がOpenGL ESのコマンド呼び出しだす。OpenGL ESのAPIはコマンドと呼ばれ、C言語やJava言語等、さまざまな言語で

実装されています。例えば、「glClearColor」コマンドは「画面を塗りつぶす色はコレだ」と指定するコマンド、「glClear」は「事前に指示してあった色で塗りつぶす」というコマンドです(図2)。

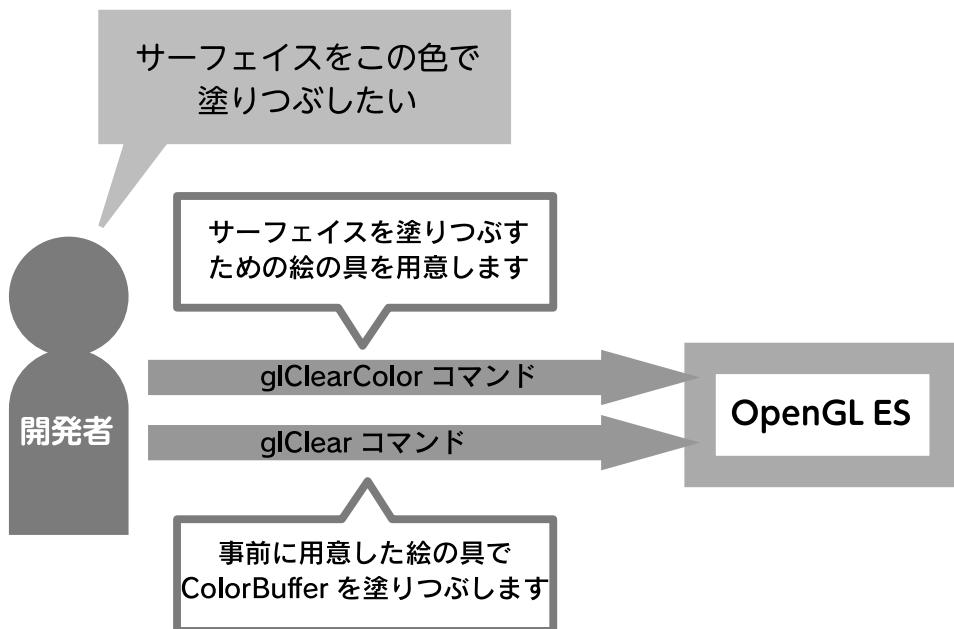


図2:OpenGL ESのコマンド

AndroidではJava言語とC/C++言語で呼び出すことができ、どちらもAPIはほぼ同じです。ですが、言語ごとの仕様に基づいた多少の差異があります。

OpenGL ES 2.0のAPIはすべてGLES20クラスのstaticメソッドとして実装されています。

最初のサンプルは「画面を単色で塗りつぶす」というものです。単色で塗りつぶすためには、OpenGL ESに対して「どんな色で塗りつぶすか」を伝えなければなりません。「glClearColor」コマンドはR(赤)、G(緑)、B(青)、A(不透明度)を引数に指定します。引数の順番はRGBAで、OpenGL ESのAPIで「色」を扱う場合は必ずこの順番となります。ここでの注意点は、色の範囲はPCやゲームの世界では一般的になっている0~255の値ではなく、0.0~1.0の浮動小数で現さなければならないということです(リスト4)。

#### リスト4:glClearColor(画面の塗りつぶし色を指定)

```
GLES20.glClearColor(0.0f, 1.0f, 1.0f, 1.0f);
```

画面を実際に塗りつぶすのが「glClear」コマンドです(リスト5)。このコマンドの引数には「OpenGL ESが管理するどのバッファをクリアするか」を指定します。バッファとは、OpenGL ESが持つキャンバスのようなものだと考えてください。実際に画面に反映される「色」を管理しているのがカラー・バッファで、それを示す引数が「GL\_COLOR\_BUFFER\_BIT」です。

リスト5:glClear(画面を実際に塗りつぶす)

```
GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
```

このコマンドを呼び出すことで、実際に画面の塗りつぶしが行われます。

## TRY:画面を任意の色で描画する

コマンドの引数を変更し、画面を任意の色で塗りつぶしてみましょう。

## TRY:画面をランダムな色で描画する

GLSurfaceViewは毎秒60回ほど、「onDrawFrame」メソッドを呼び出します。ですがずっと同じ色を描画していたら、「本当に60回も描画してるの?」と思うかもしれません。そこで、乱数を使用して描画する毎に画面の色を変更してみましょう。Javaでの乱数生成は、下記のメソッドを使用します(リスト6)。

リスト6:random(0.0~1.0の乱数を生成する)

```
Math.random()
```



## 16-1-4 プリミティブの描画を行う

OpenGL ES 2.0は、「プログラマブルパイプライン」という仕組みを導入しています。これはプログラマが描画の仕組みを「プログラマブル」にする、つまりプログラムによって柔軟に書き換えが行えるということです。ですがそれと同時に、プログラマに対してその「パイプラインを構築する」という義務を課しているという側面もあります。簡単にいえば、「やれること」が増えた代わりに、「やらなければいけないこと」もまた増えたわけです。「Hello World」のプログラムから多くの変更がありますが、ひとつひとつはあまり大きくありません。

### 演習1:三角形を描画する

サンプルコードは次のとおりです(リスト7)。

## リスト7:Chapter01\_02.java

```

import static android.opengl.GLES20.*;

public class Chapter01_02 extends Chapter01_01 {
    /**
     * プログラムオブジェクト
     */
    protected int program = 0;

    /**
     * attr_pos
     */
    protected int attr_pos;

    /**
     * ポリゴン色
     */
    protected int unif_color;

    /**
     * Surfaceが生成されたタイミングの処理
     */
    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        {
            final String vertexShaderSource =
                "attribute mediump vec4 attr_pos;" +
                "void main() {" +
                "    gl_Position = attr_pos;" +
                "}";

            final String fragmentShaderSource =
                "uniform lowp vec4 unif_color;" +
                "void main() {" +
                "    gl_FragColor = unif_color;" +
                "}";
        }

        final int vertexShader =
            ES20Util.compileShader(GL_VERTEX_SHADER,
            vertexShaderSource);
        final int fragmentShader =
            ES20Util.compileShader(GL_FRAGMENT_SHADER,
            fragmentShaderSource);

        this.program =
            ES20Util.linkShader(vertexShader, fragmentShader);
    }
    // locationを取得する
    {
        attr_pos = glGetAttribLocation(program, "attr_pos");
        assert attr_pos >= 0;

        unif_color = glGetUniformLocation(program, "unif_color");
        assert unif_color >= 0;
    }
    glUseProgram(program); ⑦
}

```

```
}

@Override
public void onSurfaceChanged(GL10 gl, int width, int height) { ⑧
    glViewport(0, 0, width, height);
}

@Override
public void onDrawFrame(GL10 gl) {
    glClearColor(0.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL10.GL_COLOR_BUFFER_BIT);

    // ポリゴン色をアップロードする
    // 色はRGBAでアップロードする
    glUniform4f(unif_color, 1.0f, 0.0f, 0.0f, 1.0f); ⑨

    // attr_posを有効にする
    glEnableVertexAttribArray(attr_pos); ⑩

    // 画面中央へ描画する
    final float[] position = {
        // v0
        0.0f, 1.0f,
        // v1
        1.0f, -1.0f,
        // v2
        -1.0f, -1.0f}; ⑪

    glVertexAttribPointer(attr_pos, 2, GL_FLOAT,
        false, 0, ES20Util.wrap(position));

    glDrawArrays(GL_TRIANGLES, 0, 3); ⑫
}
}
```

## リスト8:ES20Utilクラス 抜粋

```
/***
 * 頂点シェーダー/フラグメントシェーダーのコンパイルを行う
 */
public static int compileShader(int GL_XXXX_SHADER, String source) {
    final int shader = glCreateShader(GL_XXXX_SHADER);

    glShaderSource(shader, source);
    glCompileShader(shader);

    // コンパイルエラーをチェックする
    {
        int[] compileSuccess = new int[]{0};
        glGetShaderiv(shader, GL_COMPILE_STATUS, compileSuccess, 0);
        if (compileSuccess[0] == GL_FALSE) {
            throw new RuntimeException(glGetShaderInfoLog(shader));
        }
    }

    return shader;
}

/***
 * 頂点シェーダとフラグメントシェーダをリンクさせる
 */
public static int linkShader(int vertexShader, int fragmentShader) {
    final int program = glCreateProgram();
    glAttachShader(program, vertexShader);
    glAttachShader(program, fragmentShader);

    glLinkProgram(program);

    // リンクエラーをチェックする
    {
        int[] linkSuccess = new int[]{0};
        glGetProgramiv(program, GL_LINK_STATUS, linkSuccess, 0);
        if (linkSuccess[0] == GL_FALSE) {
            throw new RuntimeException(glGetProgramInfoLog(program));
        }
    }

    // delete
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);

    return program;
}
```

これらのプログラムの実行結果は図3のようになります。

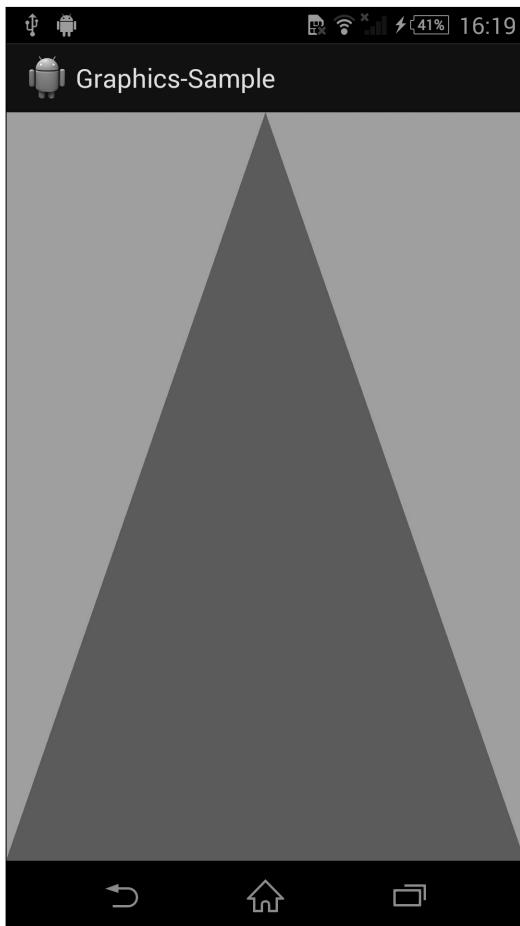


図3:三角形を描いたOpenGL ES 2.0の出力画面

## importを工夫してコーディングをしやすくする

このサンプルコードを見て気づいたかもしれません、今回のサンプルではOpenGL ES 2.0コマンドの呼び出しから「GLES20.」の部分が省略されています。これはJavaの機能で、次のように記述することでstaticメソッドやstatic変数へのアクセスの記述を簡略化できるというものです(リスト9～11)。

リスト9:static(GLES20クラス内の全staticメソッドとフィールドにアクセスする)

```
import static android.opengl.GLES20.*;
```

リスト10:static\_before(変更前のコード)

```
GLES20.glClearColor(0.0f, 1.0f, 1.0f, 1.0f);
GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
```

リスト11:static\_after(変更後のコード)

```
glClearColor(0.0f, 1.0f, 1.0f, 1.0f);
glClear(GL10.GL_COLOR_BUFFER_BIT);
```

「GLES20.」を省略することで、多くのOpenGL ES関連書籍と同様の構文で記述できるようになります。今後の理解もより深まるでしょう。

## 初期化を行う

OpenGL ES 2.0では、「GPUで実行される、描画専用のプログラム」を最低2種類記述する必要があります。それが「頂点シェーダー」と「フラグメントシェーダー」です。フラグメントシェーダーは、「ピクセルシェーダー」と言い換えればご存知の方もいるかもしれません。フラグメントシェーダーとピクセルシェーダーはどちらも特に違いはありませんが、OpenGL ESではピクセルを「フラグメント」と呼ぶため、フラグメントシェーダーと名付けられています。

ただし、このフラグメントとはAndroidプログラミング的な意味の「`android.app.Fragment`」クラスとはまったく意味合いが異なるため、注意してください。グラフィックの章内でカタカナ表記の「フラグメント」とはOpenGL ESのピクセルを示し、英語表記の「Fragment」とはAndroid SDKに含まれる「`android.app.Fragment`」クラスを示します。

**リスト7**の①③④⑤の箇所が、シェーダーの生成とプログラム内への保持を行っている箇所です。まずは頂点シェーダー・フラグメントシェーダーのプログラムを見てみましょう(**リスト12～13**)。

リスト12:vertexShaderSource(頂点シェーダー)

```
final String vertexShaderSource =
    "attribute mediump vec4 attr_pos;" +
    "void main() {" +
        "    gl_Position = attr_pos;" +
    "};";
```

リスト13:fragmentShaderSource(フラグメントシェーダー)

```
final String fragmentShaderSource =
    "uniform lowp vec4 unif_color;" +
    "void main() {" +
        "    gl_FragColor = unif_color;" +
    "};";
```

これらはすべてString型の変数、つまりただの文字列としてJava言語のプログラム内に保持されています。もちろん、Javaがコンパイルされてアプリとなっても文字列は文字列のままで、このままで実行できません。

OpenGL ES 2.0のシェーダーはGPUで動作させる必要がありますが、GPUは無数に種類があり、なおかつ機械語単位で見ると互換性がありません。そのため、GPUで実行するプログラムをコンパイルするためには、そのGPUが載っている端末でコンパイルを行わなければならないのです。厳密にいえば、事前にコンパイルすること

もできますが、機種の違いによって互換性が失われるため、Androidではあまり使用されていません。

シェーダーは「GLSL ES」というC言語に似た言語で記述することができ、エントリーポイントとなるのは「main()」関数です。このプログラムはC言語に似ていますが、GPUにより同時並列的な実行が行われるため、さまざまな制約があります。ですが、まずは細かいことを考えずに少しづつ理解していってください。

シェーダーで最初に覚えるべきは2つあります。1つは、「頂点シェーダーでは必ず「gl\_Position」に頂点座標を書き込むこと」。もう1つは、「gl\_FragColorに最終的なフラグメント色を書き込むこと」です。

サンプルの頂点シェーダーは「gl\_Positionに対し、Java側のプログラムから与えられたattr\_posを代入する」ということを行い、フラグメントシェーダーでは「gl\_FragColorに対し、Java側のプログラムから与えられたunif\_colorを代入する」という処理を行います。

**リスト7**の④の部分では頂点シェーダーとフラグメントシェーダーのコンパイルを行っています(図4)。「ES20Util.compileShader」は筆者が用意した補助メソッドです。内部ではコンパイルとエラーチェックを行っています。詳細な解説を行うためにはページ数が足りませんので、興味がある方は実装を覗いてみるとよいでしょう。

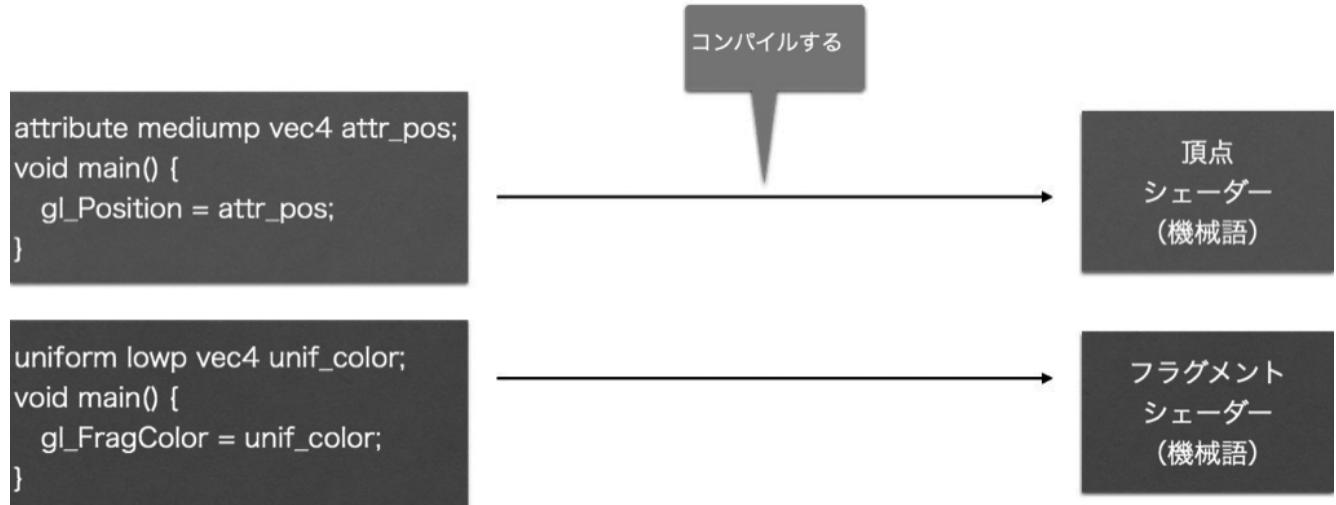


図4:シェーダーをコンパイルして機械語にする

シェーダーのコンパイル結果はそれぞれint型の変数として取得できます。OpenGL ESはオブジェクト指向のAPIではなく、OpenGL ESが管理するメモリはすべて「ID」が割り振られ、そのIDによって管理する必要があります。

まだこの状態では実行することができません。たとえばAndroidアプリは「.java」拡張子のプログラムを記述し、コンパイルによって「.class」ファイルができ上がります。実際にAndroid端末上で実行するためにはそれらを結合して「.dex」ファイルに変換しなければなりません。これらは、「コンパイル」と「リンク」という別々の作業であり(厳密にいえば少し違いますが)、そのどちらも実行には欠かすことができません。

Eclipseのような統合開発環境はそれらを隠ぺいしているため、「コンパイル=実行できる状態にする」ように見えますが、実際には複数の処理を行っているのです。このような「コンパイル～実際に実行可能な状態にする」という一連の流れを、一般的に「ビルド(build)」と呼びます。

シェーダーも同じように、リンクを行わなければ実行することができません(図5)。リンクを行っているのがリスト7の⑤の「ES20Util.linkShader」メソッドです。「ES20Util.compileShader」と「ES20Util.linkShader」はどちらも定型的な処理で、エラー処理を除けばほぼ書き換えることはありません。

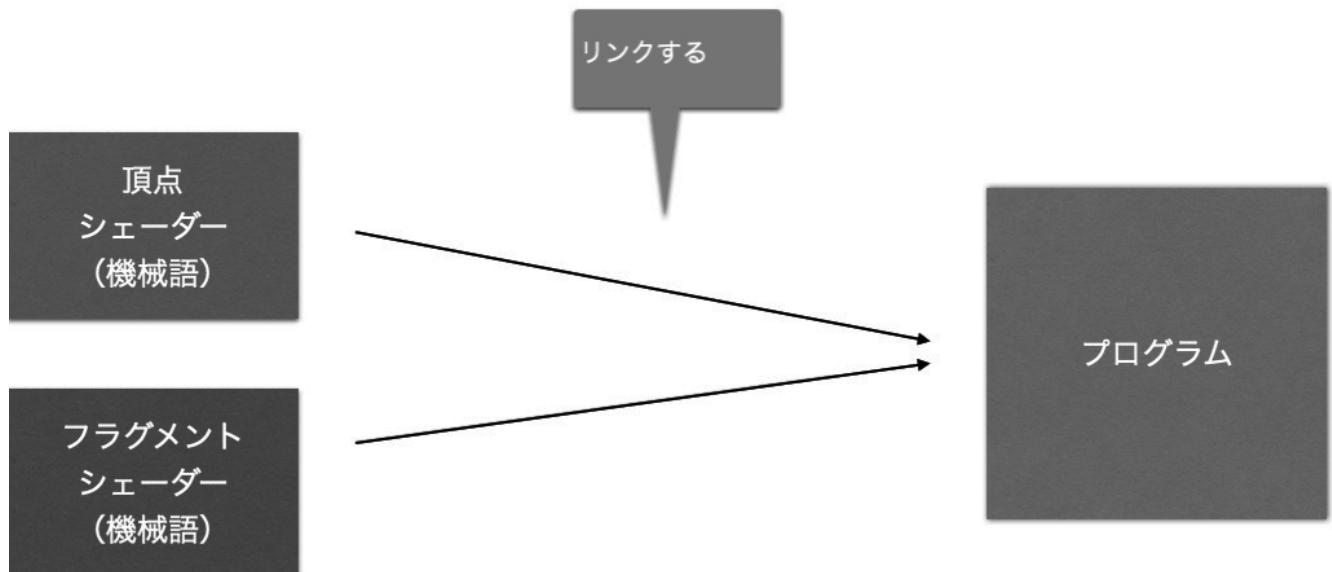


図5:リンクすることで初めて「プログラム」として動作する

リスト7の②⑥ではシェーダーから変数の「Location(番号)」を取得します。シェーダーはGPUという特殊な環境下で動作する特殊なプログラムです。そのため、Java側のコードから値をアップロードするためには、変数の管理番号を取得して「何番の変数にこの値を入力してください」という命令を実行しなければなりません。このサンプルでシェーダーで宣言している変数は2つです(リスト14～15)。

リスト14:attr\_pos(頂点シェーダーで定義している変数)

```
attribute mediump vec4 attr_pos;
```

リスト15:unif\_color(フラグメントシェーダーで定義している変数)

```
uniform lowp vec4 unif_color;
```

「attr\_pos」と「unif\_color」にそれぞれ値をアップロードしなければ、シェーダーは正常に計算を行うことができません。そしてアップロードを行うためには「Location」を取得するしかなく、「Location」を取得したら、どこかに保持して置かなければ

ば、使用することができません。取得した「Location」を保持しているのがメンバ変数「attr\_pos」と「unif\_color」です。今回はそれぞれシェーダー内の変数名と同じ変数名をそれぞれ名付けていますが、この名前はプログラマがわかりやすければ何でも構いません。

「Location」を取得するためにはそれぞれ「glGetAttribLocation」と「glGetUniformLocation」を使用する必要があります。どちらも第1引数は取得を行うプログラムオブジェクト、第2引数は変数名です。正常に「Location」が取得できると0以上の値が返却されます。もし変数名が間違っていたり、使用されていない変数の場合は「-1」がエラーとして返却されます。

最後にリスト7の⑦で「glUseProgram」コマンドを使用し、「このシェーダーで描画を行いますよ」とGPUに対して宣言します。

以上で初期化は完了です。

## Viewportの変更を行う

「onSurfaceChanged」では、「glViewport」コマンドを呼び出しています。OpenGL ESは、「画面内のどの領域(Viewport)を描画に使用するか」という設定を自由に行うことができます。Androidではデフォルトで「画面全体を描画に使用する」としています。

ただし、Android端末によっては初期値が「0」であったり、端末の縦横を切り替えた際に問題が発生したります。そのため、「サーフェイスの大きさが変更された」というメッセージである「onSurfaceChanged」で「Viewport」を変更しています。

この「Viewport」の注意点は、一般的なディスプレイの座標系(左上が原点0,0)ではなく、左下を原点(0, 0)とする座標系となることです(図6)。

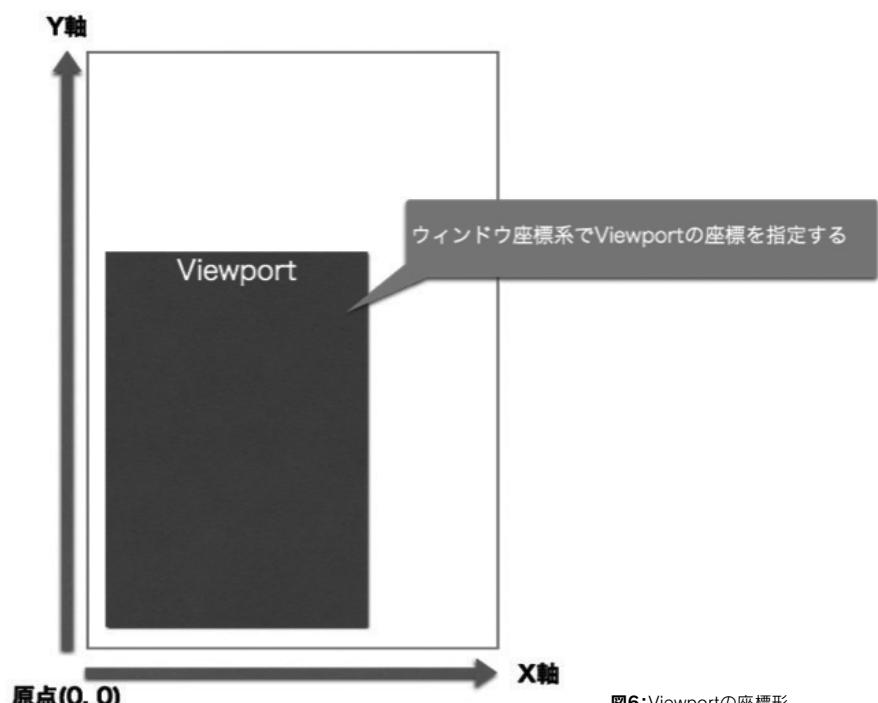


図6:Viewportの座標系

## 三角形描画を行う

**リスト7の⑨**の部分では、「uniform変数」へ「プリミティブの色」をアップロードしています。サンプルでフラグメントシェーダーで定義しているのが、「uniform変数」です。「uniform変数」へ値をアップロードするためには、「glUniform4f」コマンドを使用します。OpenGL ESでは「CPU=クライアント」、「GPU=サーバー」として考えるため、値の設定や取得を「アップロード」／「ダウンロード」と呼んでいます。

アップロードは第1引数が「uniform変数」の「Location」、第2～第5引数がそれぞれX/Y/Z/W要素に渡される値です。XYZWは「gl\_FragColor」に書き込まれるとRGBAそれぞれの色として認識されます。ここではRGBAそれぞれR(1.0f)、G(1.0f)、B(0.0f)、A(1.0f)として、赤い色をアップロードしています(**リスト16**)。

### リスト16:glUniform4f(アップロード部分)

```
glUniform4f(unif_color, 1.0f, 0.0f, 0.0f, 1.0f);
```

「glUniformXXXX」コマンドは変数の型ごとに用意されており、今回宣言している「unif\_color」変数は「vec4」型であるため、「glUniform4f」コマンドを使用します(**リスト17**)。

### リスト17:uniform(uniform変数の宣言)

```
uniform lowp vec4 unif_color;
```

この他に「vec3」型の変数をアップロードする「glUniform3f」コマンドや、後述する「mat4」型の変数をアップロードするための「glUniformMatrix4fv」コマンド等があります。

**リスト7の⑩**の部分で行っているのは、「glEnableVertexAttribArray」コマンドによる変数の有効化です。「attribute」変数には、GPUによって「使用する」「使用しない」という2つの状態があります。これは不要な変数は使用しないほうがGPUにとって負荷が減り、描画速度の向上やバッテリー消費を抑えることに役立つからです。すべての変数は初期状態で「使用しない」状態となるため、必要に応じてこのコマンドを呼び出す必要があります。

頂点シェーダー内で使用される「attribute」は、アプリが用意した「頂点」と「GPUの変数」とを結びつける重要な変数です。このサンプルでは三角形の各頂点の座標を「attribute」変数としてGPUへアップロードします。また、「attribute」変数の内容は頂点シェーダーの「main()」関数が呼び出されるごとに変化します。

初心者にとってわかりにくいのが、「uniform変数」と「attribute変数」の違いです。「uniform変数」は頂点シェーダーとフラグメントシェーダーの両方で使用することができます。また、描画処理(後述の「glDrawArrays」コマンド)実行中は

一切変数の値が変化しません。それに対し、「attribute」変数は頂点シェーダーでしか使用することができません。「attribute変数」は「頂点データ専用」の変数と考えてください。

GPUでは1頂点に付き、1回の「main()」関数が実行され、さらに頂点数だけ繰り返されます。三角形は3つの頂点で構成されるため、頂点シェーダーの呼び出し回数は3回です。「attribute」変数の内容は「main()」関数呼び出しの度に「頂点0のデータ」「頂点1のデータ」「頂点2のデータ」...という具合に変化します。

リスト7の⑪では、実際に使用する頂点データを定義しています。「頂点データ」といっても、専用のクラスは用意されていません。なぜなら、GPUにとって「頂点」とはメモリの塊でしかないからです。

OpenGL ES 2.0では、頂点の内容をプログラマが自由に決めることができます。サンプルでの頂点データは「1つの頂点は浮動小数点型のX座標とY座標を持つ」という内容になっています。三角形ですので頂点は3つなければなりません。合計すると、2要素(XY)×3頂点=6つのデータがあるということです。

それを定義しているのが⑪の「position」配列です。配列のindex0とindex1が頂点0、index2とindex3が頂点1、index4とindex5が頂点2をそれぞれ構成します。この時、頂点データは「1つの配列」でなければなりません(リスト18)。

#### リスト18:defined\_data(定義している頂点データ)

```
final float[] position = {  
    // v0  
    0.0f, 1.0f,  
    // v1  
    1.0f, -1.0f,  
    // v2  
    -1.0f, -1.0f};
```

この時、座標の値は-1.0～1.0になるということに注目してください。GPUが描画に使用できる空間は-1.0～1.0であり、ピクセル座標(現代のモバイル端末の多くはHD以上の解像度を持ちます)ではありません。「Viewport」内が-1.0～1.0として扱われ、最終的な画面座標として出力されます。今回の三角形は「画面(Viewport)いっぱいに表示する」という内容のため、座標系の目一杯端まで使っていることがわかります。

次に行うのが、「attribute変数」と頂点データの関連付けです。頂点データは非常に大きいデータ(たとえば、最近のゲームでは1キャラクターに付き数万の頂点が使用されます)であるため、一括ではGPUが処理できません。そのため、「読み出し場所」をGPUに教えておくことで、必要なデータを逐次処理(ストリーミング)することができます。

その「読み出し場所」を指定するのが「glVertexAttribPointer」コマンドです。第1引数には関連付ける「attribute変数」を指定します。第2引数は頂点の

要素数を指定します。ここではXYの2要素ですので、「2」を指定します。第3引数は頂点の型を指定します。今回はfloat型ですので、それを示す定数の「GL\_FLOAT」を指定します。第4引数は頂点データの正規化の有無を指定します。これは省メモリ・速度重視の設計では重要になりますが、本章では説明しません。「GL\_FLOAT」を使う限り、この引数は「false」を指定すれば問題ありません。第5引数には頂点間のオフセット(byte単位)を指定しますが、現在は0を指定すれば問題ありません。この引数についての詳細は後述します。最後の引数でようやくデータを渡します。この時、引数の型は「java.nio.Buffer」であり、配列型ではありません。

## java.nio.Buffer型

Android SDKによって記述されたアプリは、「Dalvik VM」(もしくはART)という仮想マシンの上で動作します。配列に格納されたデータは、仮想マシン上で読み込むために最適化された状態で保存されますが、これらはGPUから読み込むためには適した状態ではありません。

GCによってRAM上のアドレスが変更されたり、そもそもCPUやGPUが求めるエンディアンに従っていない場合もあります。

「Buffer」型はGPU(や、C/C++言語のコード)から読みやすい場所へメモリを配置します。配列から「Buffer」への変換は画一的な動作です。注意するのは、配列から「Buffer」を生成すると「コピー処理」が発生するため、メモリの使用量が2倍になってしまうということです。

サンプルでは可読性を上げるため効率的な「Buffer」の生成を行っていません。サンプルを実行すると、毎秒60回のペースで配列と「Buffer」オブジェクトが生成／廃棄を繰り返すことになります。

Bufferオブジェクトの生成自体は次のようにになります(**リスト19**)。

リスト19:ByteBuffer(float配列からBufferオブジェクトの生成)

```
public static FloatBuffer wrap(float[] buffer) {
    return (FloatBuffer) ByteBuffer
        .allocateDirect(buffer.length * 4)
        .order(ByteOrder.nativeOrder())
        .asFloatBuffer().put(buffer).position(0);
}
```

ここまでたらもう一息です。あとは描画コマンドを発行すれば完了です(**図7**)。描画を行うためには、**リスト7**の⑫のように「glDrawArrays」コマンドを呼び出します。第1引数は頂点をどのようにつなげて三角形を構築するかを指定します。これは2つ以上の三角形を効率的なメモリ量、速度で描画するのに役立ちますが、サンプルでは最も基本的な形である「GL\_TRIANGLES」となります。第2引数と第3

GCとはGarbage Collectionの略で、不要なオブジェクトを削除や最適化することで、残メモリを多くします。GCは便利な機能ですが、時間がかかる処理であるため、頻繁に行うことできません。

引数は、「何番目の頂点から何個分の頂点を使用するか」を指定します。今回は0番目の頂点から3個分の頂点で描画を行わせますので、それぞれ「0」、「3」となります。この時、第3引数が3未満だと描画が正常に行えません。なぜなら、「三角形を構築するには最低でも3つの頂点が必要」だからです。

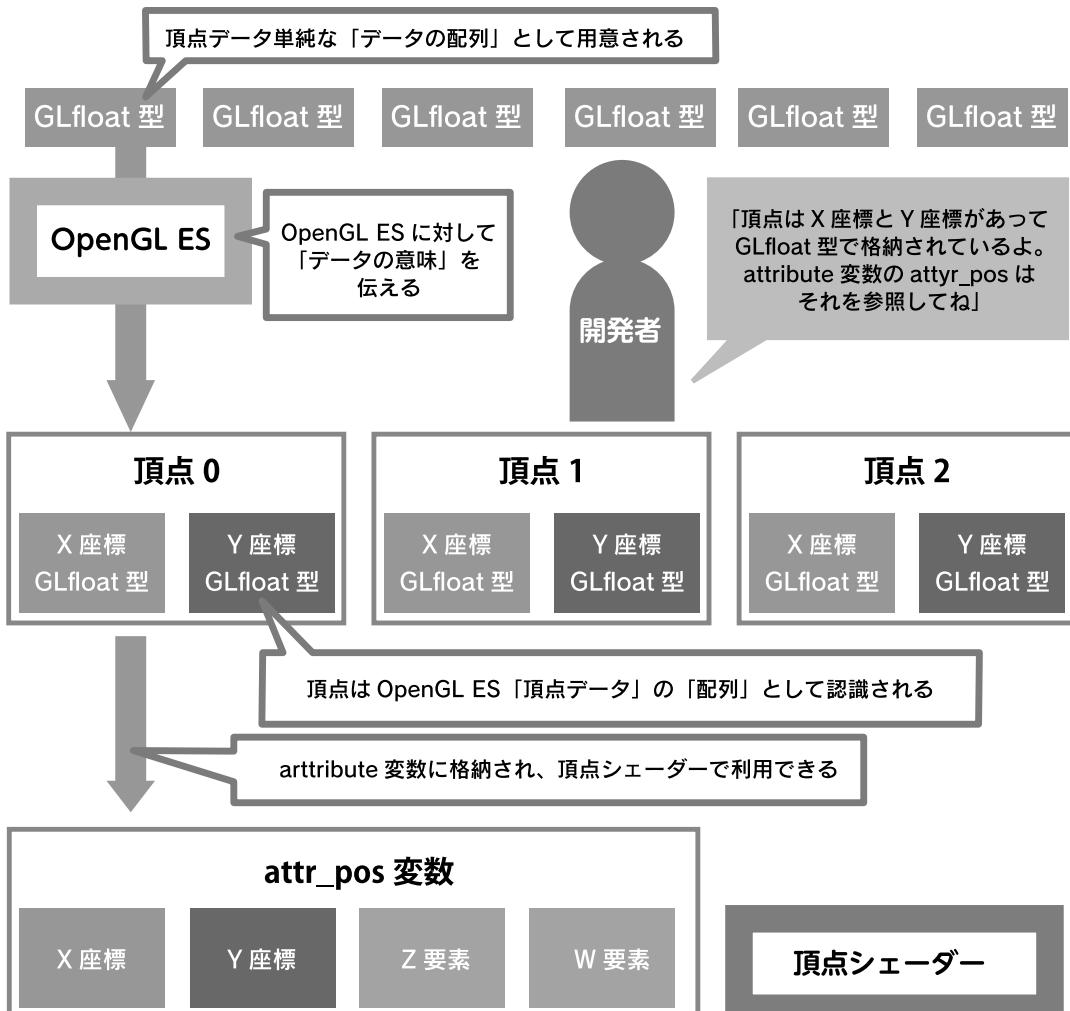


図7:頂点データと頂点シェーダーの関係

## TRY: 端末の縦横を切り替えてみよう

OpenGL ESの3D描画は「ピクセル座標に依存しない」座標系です。そのため、「縦長」「横長」の「Viewport」に関係なく、座標系は-1.0～1.0となります。端末の縦横を切り替え、どのように描画内容が変化するか確かめてみましょう。

## TRY: 「Viewport」を変更しよう

画面の縦横切替と同じく、「Viewport」を自由に変更してどのような変化があるか確認してみましょう。

## TRY:三角形の色を変えてみよう

サンプルでは赤い色で三角形を描画していました。これを修正し、任意の色で三角形を描画できるようにしてみましょう。修正箇所はサンプル内のどこでも構いません。

## TRY:三角形の大きさや位置を変更しよう

サンプルでは「Viewport」いっぱいに三角形が表示されています。頂点位置を自由に修正し、どのように三角形の見え方が変わるか確認してみてください。

## TRY:「ES20Util.wrap」の呼び出し回数を最小限にしよう

前述のように、サンプルでは毎秒60回程度「Util.wrap」メソッドが呼び出されています。呼び出し回数が増えればそれだけGC対象が増えますし、バッテリーへの負担も大きくなります。そこで、「Util.wrap」の実行回数がなるべく少なくなるように修正してみてください。

## CHALLENGE:「glUniform4f」や「assert」の部分を変更せずに、シェーダーだけで任意の色のポリゴンを表示させてみよう

「glUniform4f」でアップロードする値を変更すれば、任意の色の描画を行うことができます。そこで、「glUniform4f(unif\_color, 1.0f, 0.0f, 0.0f, 1.0f)」の部分を変更せず、「uniform変数」も使用した上で「緑」「青」の三角形を描画してみましょう。

## 演習2:四角形を描画する

サンプルコードは次のとおりです(リスト20)

リスト20:Chapter01\_03.java

```
public class Chapter01_03 extends Chapter01_02 {
    @Override
    public void onDrawFrame(GL10 gl) {
        中略...
        {
            final float[] position = {
                // triangle 0
                // v0(left top)
                -0.75f, 0.75f,
                // v1(left bottom)
```

```

        -0.75f, -0.75f,
        // v2(right top)
        0.75f, 0.75f,

        // triangle 1
        // v3(right top)
        0.75f, 0.75f,
        // v4(left bottom)
        -0.75f, -0.75f,
        // v5(right bottom)
        0.75f, -0.75f,
    };
    GLES20.glVertexAttribPointer(attr_pos, 2, GLES20.GL_FLOAT,
        false, 0, ES20Util.wrap(position));
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, 6);
}
}
}

```

このプログラムを実行すると、図8のような画面が表示されます。

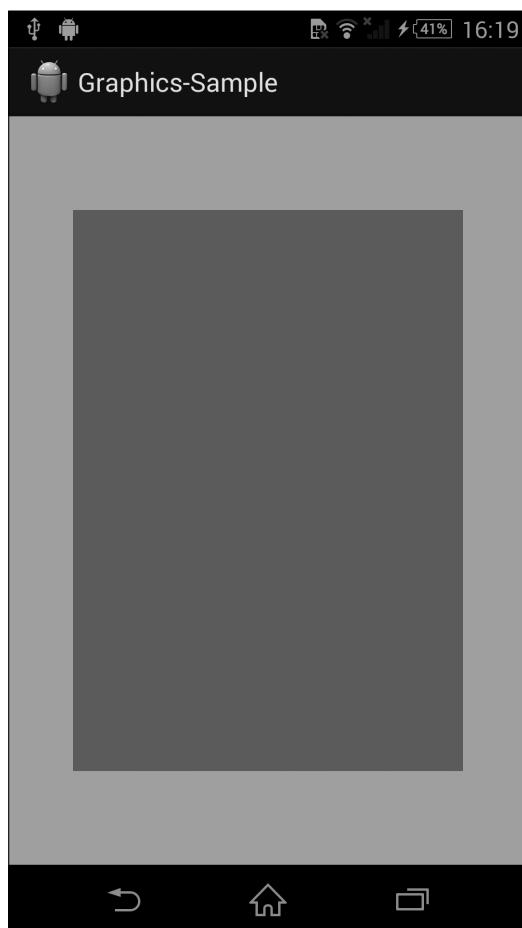


図8:四角形を描画した結果

OpenGL ESは四角形以上のn角形の描画も行うことができます。たとえば、四角形は2つの三角形に分割できますし、五角形は3つの三角形に分割することができます。このサンプルでは三角形を2つ描画することで、四角形を構築しています。

そのため、2つの三角形を描画するため、頂点データを6(三角形)×2=12個用意

しています。「glVertexAttribPointer」はまったく変わりありません。なぜなら、「glVertexAttribPointer」コマンドには「いくつの頂点を渡すか」という情報は付随しないからです。そのため、頂点数が6個でも、12個でも、さらに増えて数万あったとしてもこのコマンドには影響がありません。

「glDrawArrays」コマンドの第3引数は「6」に変更されています。これは四角形=三角形2つで構築される形状であるため、6つの頂点を描画に使用することを示しています。

### TRY:右側の三角形、左側の三角形だけをそれぞれ描画してみよう

「glDrawArrays」の引数を修正して、「前半の三角形」だけを描画してみましょう。また、同じく「glDrawArrays」の引数を修正して「後半の三角形」だけを描画してみましょう。

### TRY:五角形や六角形を描画してみよう

三角形を組み合わせることで、複雑な図形を描画することが可能ですが。三角形の配置や頂点座標を工夫し、五角形や六角形を構築してみましょう。



## 16-1-5 VRAM の利用

GPUは描画専用のメモリ=VRAMを持ちます。VRAMにあるデータはGPUから効率的にアクセスできるため、描画速度を重視する場合は非常に重要です。特に描画時に使用する2次元画像=「Texture」はVRAMに配置した状態でしかアクセスすることができません。

VRAMはJavaの仮想マシンとは別に確保されるため、たとえば数百MBもの量を使用しても仮想マシンが「OutOfMemory」によって強制終了される心配はありません。

### 演習3:テクスチャを読み込む

サンプルコードは次のとおりです(リスト21)。

リスト21:Chapter01\_04.java

```
public class Chapter01_04 extends Chapter01_01 {  
  
    中略...  
  
    /**  
     * UV座標  
     */  
    protected int attr_uv;  
  
    /**  
     * テクスチャUniform  
     */  
    protected int unif_texture;  
  
    /**  
     * テクスチャオブジェクト  
     */  
    protected int texture;  
  
    @Override  
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {  
        {  
            final String vertexShaderSource =  
                "attribute mediump vec4 attr_pos;" +  
                "attribute mediump vec2 attr_uv;" +  
                "varying mediump vec2 vary_uv;" +  
                "void main() {" +  
                "    gl_Position = attr_pos;" +  
                "    vary_uv = attr_uv;" +  
                "}" +  
  
            final String fragmentShaderSource =  
                "uniform sampler2D unif_texture;" +  
                "varying mediump vec2 vary_uv;" +  
                "void main() {" +  
                "    gl_FragColor = texture2D(unif_texture, vary_uv);" +  
                "}" +  
  
            // コンパイルとリンクを行う  
            this.program =  
                ES20Util.compileAndLinkShader(vertexShaderSource,  
                    fragmentShaderSource);  
        }  
  
        // locationを取り出す  
        {  
            attr_pos = glGetAttribLocation(program, "attr_pos");  
            assert attr_pos >= 0;  
            attr_uv = glGetAttribLocation(program, "attr_uv");  
            assert attr_uv >= 0;  
            unif_texture = glGetUniformLocation(program, "unif_texture");  
            assert unif_texture >= 0;  
        }  
  
        // テクスチャを読み込む  
    }  
}
```

```

{
    Bitmap bitmap =
        ES20Util.decodeBitmapFromAssets(getActivity(),
            "sample512x512.png");

    int[] textureId = {0};
    glGenTextures(1, textureId, 0); 4
    this.texture = textureId[0];
    assert this.texture != 0;

    glBindTexture(GL_TEXTURE_2D, texture);
    {
        GLUtils.texImage2D(GL_TEXTURE_2D, 0, bitmap, 0); // テクスチャをバインド
        glTexParameteri(GL_TEXTURE_2D,
            GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D,
            GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D,
            GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D,
            GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    }
    glBindTexture(GL_TEXTURE_2D, 0); // テクスチャをアンバインド 5

    ES20Util.assertGL();
}

glUseProgram(program);
ES20Util.assertGL(); 6
}

中略...

@Override
public void onDrawFrame(GL10 gl) {
    glClearColor(0.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // attr_posを有効にする
    glEnableVertexAttribArray(attr_pos);
    glEnableVertexAttribArray(attr_uv);

    // 使用するテクスチャをバインドする
    glBindTexture(GL_TEXTURE_2D, texture); 6

    {
        final float[] position = {
            // triangle 0
            // v0(left top)
            -0.75f, 0.75f,
            // v1(left bottom)
            -0.75f, -0.75f,
            // v2(right top)
            0.75f, 0.75f,
            // triangle 1
            // v3(right bottom)
            0.75f, -0.75f,
            // v4(left middle)
            -0.5f, 0.0f,
            // v5(right middle)
            0.5f, 0.0f
        };
        ...
    }
}

```

```
// v3(right top)
0.75f, 0.75f,
// v4(left bottom)
-0.75f, -0.75f,
// v5(right bottom)
0.75f, -0.75f,
};

glVertexAttribPointer(attr_pos, 2, GL_FLOAT, false, 0,
ES20Util.wrap(position));

final float[] uv = {
    // triangle 0
    // v0(left top)
    0, 0,
    // v1(left bottom)
    0, 1,
    // v2(right top)
    1, 0,

    // triangle 1
    // v3(right top)
    1, 0,
    // v4(left bottom)
    0, 1,
    // v5(right bottom)
    1, 1,
};

glVertexAttribPointer(attr_uv, 2, GL_FLOAT, false, 0,
ES20Util.wrap(uv));
glDrawArrays(GL_TRIANGLES, 0, 6);
}

}
```

7

このプログラムを実行すると、図9のような画面が表示されます。

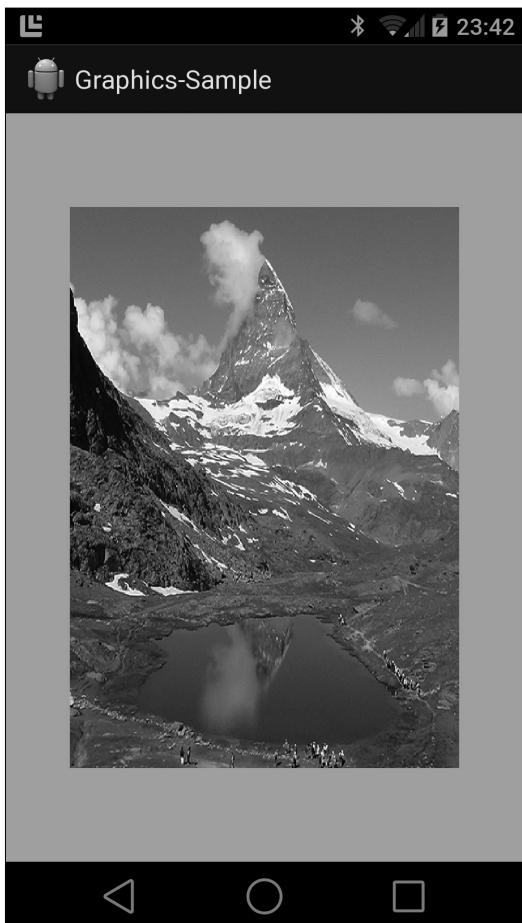


図9:VRAMを利用して描画した結果

## テクスチャオブジェクトの生成

今回のシェーダーでは今まで使用していた「色」を保存した「uniform変数」「unif\_color」に変わり、テクスチャを示す型である「sampler2D型」が使用されています。OpenGL ESによって生成されたテクスチャオブジェクトは、シェーダーのようにint型の変数によって「ID」が管理されています(リスト22)。

リスト22:TextureObject(テクスチャオブジェクトはint型変数で管理される)

```
/**  
 * テクスチャオブジェクト  
 */  
protected int texture;
```

テクスチャの読み込み部分はリスト21の④⑤です。Androidでは画像を表示するためのクラスとして「Bitmap」クラスがありますが、「Bitmap」クラスだけではテクスチャとして使用することができません。「Bitmap」クラスからテクスチャオブジェクトへアップロードを行う必要があります。

まず、テクスチャオブジェクトの生成です(リスト23)。

### リスト23:glGenTextures(テクスチャオブジェクト生成)

```
int[] textureId = {0};  
glGenTextures(1, textureId, 0);  
this.texture = textureId[0];
```

テクスチャオブジェクトの生成は「glGenTextures」コマンドで行います。このコマンドは第1引数に「何枚のテクスチャを確保するか」を指定し、第2引数には戻り値の格納先となる配列を指定します。最後に、第3引数で「配列のどのindexから使用するか」を指定します。特別な理由がない限り、第3引数は常に0で問題ありません。

Android SDKに組み込まれたOpenGL ESのコマンドは、多くの場合この「配列のどのindexから使用するか」を指定することになります。これはJava言語がポインタ(メモリ)を直接扱えないための救済措置ともいえますが、本章では特別なことをしないため、すべて"0"にしています。

サンプルでは1枚のテクスチャを「textureId配列」のindex0以降に格納するため、「glGenTextures」内部では「textureId[0] = テクスチャオブジェクトのID」のような処理が行われます。「textureId」は一時変数であるため、「this.texture」に対して返却値を保存して、テクスチャオブジェクトの生成処理は完了です。

テクスチャオブジェクトは生成しただけでは使用することができません。OpenGL ESに対して「これから使用を開始するよ」ということを宣言しなければなりません。テクスチャオブジェクトの使用を開始するコマンドが「glBindTexture」です。「glBindTexture」の第1引数には「GL\_TEXTURE\_2D」を指定します。第2引数には、使用を開始するテクスチャオブジェクトを指定します。また、第2引数に「0」を指定することで「使用を終了するよ」というのを明示できます。

テクスチャの使用開始を「バインド」、使用終了を「アンバインド」とそれぞれ呼びます。

テクスチャは最大32個存在するテクスチャユニットにバインドができ、どれを操作(Active)するかの切り替えも行います。また、シェーダーに関連付けるテクスチャオブジェクトは「glUniform1i」コマンドとテクスチャユニット番号によって制御できますが、ページの都合上ここでは割愛しています。

バインド処理は描画直前であるリスト21の[6]でも行っています。なぜなら、「これらの描画でこのテクスチャを使用するよ」ということをGPUに伝えなければならないからです。バインド中に行われるのが次の処理です(リスト24)。

## リスト24:texImage2D(バインド中に行われる処理)

```
GLUtils.texImage2D(GL_TEXTURE_2D, 0, bitmap, 0); // テクスチャをアップロード
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

「GLUtils.texImage2D」メソッドは、Androidが用意している補助メソッドです。このメソッドを呼び出すと、「Bitmap」にある画像情報をテクスチャオブジェクトへアップロードすることができます。

「glTexParameteri」コマンドでは、テクスチャがGPUによってどのようにアクセスされるかを指定することができます。

WRAPはUVが0.0～1.0を超えた場合の補完方法、「FILTER」は拡大縮小時のピクセル色の補完方法をそれぞれ指定します。

詳細の解説は割愛しますが、サンプルにある4つは「お約束」として呼び出すようにしてください。もし「真っ黒なテクスチャ」が描画されてしまった場合、この初期設定が適切に行われなかった可能性があります。

これでテクスチャオブジェクトの初期化は完了です。「GLUtils.texImage2D」メソッドを呼び出した時点で、「Bitmap」画像はすべてVRAMというGPU専用メモリにコピーされています(図10)。そのため、「Bitmap」クラスは捨ててしまって構いません。

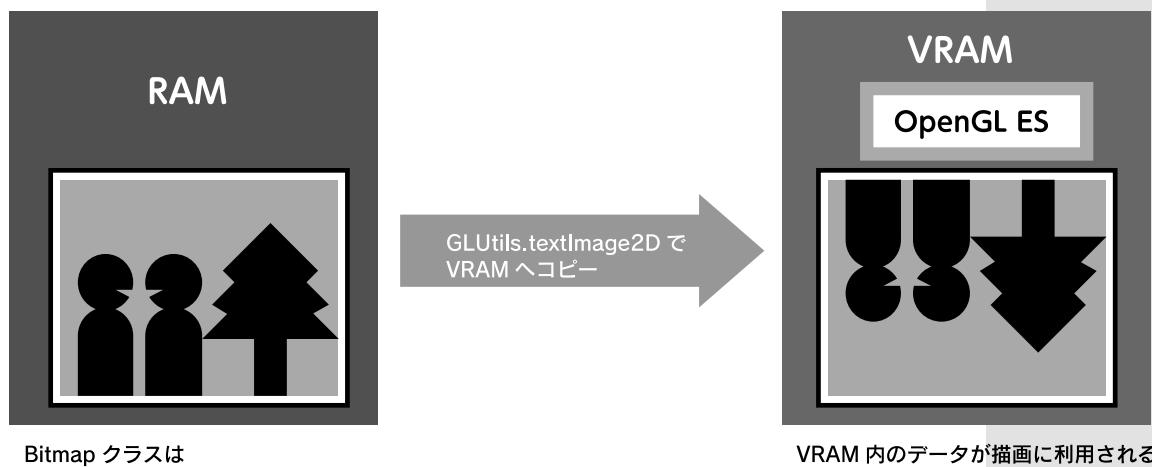


図10:テクスチャはBitmapからVRAMにコピーされる

## UV座標

今回のシェーダーでは新たな種類の変数である「varying変数」が登場しました。「varying変数」は「頂点シェーダーからフラグメントシェーダーに対して値を渡す」場合に使用される変数です。「varying変数」を使用すると、頂点間の値が自動的に補完されます。

サンプルでは「vec2型」の「varying変数」の「vary\_uv」が頂点シェーダーとフラグメントシェーダーの両方に定義されており、頂点シェーダーは「書き込み用」、フラグメントシェーダーは「読み込み用」としてそれぞれ使用しています。

テクスチャを使用するためには「UV座標」について知らなければなりません。「UV座標」とは、「テクスチャ内の位置」を示す情報で、横方向の座標を「U」、縦方向の座標を「V」としてそれぞれ扱います。座標値は0.0～1.0で、テクスチャ解像度に依存しない仕組みとなっています(図11)。

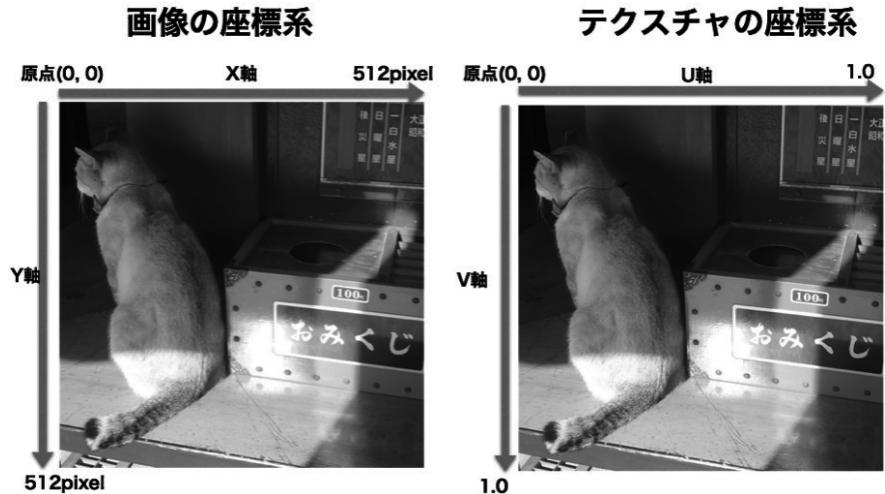


図11:画像の座標とUV座標の関係

OpenGL ESで「ポリゴンにテクスチャを貼り付ける」というのは、「1つ1つの頂点に対して、どの頂点をどのUV座標に関連付けるか」を指定するということです(図12)。

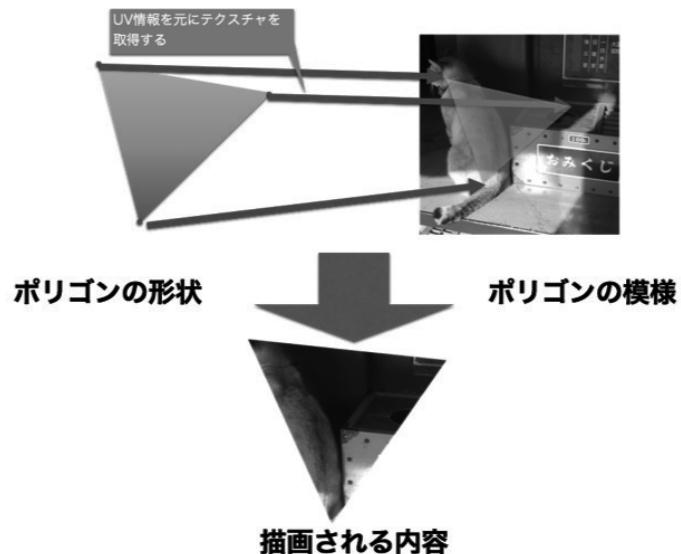


図12:ポリゴンに画像を貼り付ける概念

頂点単位で渡された「UV座標」は、「varying変数」を通してフラグメントシェーダーへと渡されます。フラグメントシェーダーはその「UV座標」から、実際に「テクスチャの色」を取得しなければなりません。「UV座標」からテクスチャの色を取得する関数が「texture2D」です。第1引数にはアクセスするテクスチャを、第2

引数には「UV」座標をそれぞれ指定します(リスト25～26)。

#### リスト25:vary\_uvVertex(頂点シェーダー)

```
final String vertexShaderSource =
    "attribute mediump vec4 attr_pos;" +
    "attribute mediump vec2 attr_uv;" +
    "varying mediump vec2 vary_uv;" +
    "void main() {" +
        "    gl_Position = attr_pos;" +
        "    vary_uv = attr_uv;" +
    "}";
}
```

#### リスト26:vary\_uvFragment(フラグメントシェーダー)

```
final String fragmentShaderSource =
    "uniform sampler2D unif_texture;" +
    "varying mediump vec2 vary_uv;" +
    "void main() {" +
        "    gl_FragColor = texture2D(unif_texture,
vary_uv);"
    };
}
```

## 頂点にUV座標を追加する

頂点に「UV座標」を追加するため、リスト21の[7]では新たなデータであるfloat型変数「uv」を追加しています。「UV座標」は「U」と「V」の2要素で構成されるため、2要素×3頂点×2ポリゴンで、合計12要素の配列となります。

直後にある「glVertexAttribPointer」コマンドでは、「attr\_uv」に対する設定を行っています。頂点の要素(位置、UV、etc)が増えても、基本的には「配列を用意する→glVertexAttribPointerでattribute変数と関連付ける」という流れは変わりません。

最後に、「glDrawArrays」で描画を行います。これも、「三角形の数」自体は同じであるため、呼び出しに変化はありません。

## TRY:UV座標を操作してテクスチャの一部を描画してみよう

「UV座標」を変更することで、描画されるテクスチャの範囲を自由に決めることができます。歪んだ形状にするなど、「UV座標」によるテクスチャ表示のされ方の変化を確かめてみましょう。

また、UVの範囲が0.0～1.0を超えた場合にどのような描画が行われるか確かめてみましょう。

## TRY:シェーダーを使ってテクスチャの「ネガ」を描画してみよう

「texture2D関数」によって読み出した値はシェーダー内の計算処理で自由に変化させることができます。そこで、計算処理によって写真の「ネガ」のような画像を描画してみましょう。

# 16-2 3D 描画への基礎知識

著：山下武志

第16章

グラフィック

この節では、「OpenGL ES」の3D描画に欠かすことのできない「行列」について取り上げます。行列を理解することで、頂点を3D空間内の任意の座標に移動できるようになります。



## この節で学ぶこと

- ・行列(Matrix)を理解する
- ・行列を使って、平行移動を行う
- ・ポリゴンを行列で拡大縮小、回転させる
- ・カメラの位置を設定してポリゴンを相対的に移動する

## この節で出てくるキーワード一覧

行列、行列演算

単位行列

mat4、mat3、mat2

glUniformMatrix4fv

カメラ

画角

ニアクリップ／ファークリップ



## 16-2-1 シェーダーでのシンプルな計算

### 演習: ポリゴンを平行移動させる

このサンプルコードはポリゴンを平行移動させます。順番に見ていきましょう(リスト27)。

リスト27: ポリゴンの平行移動

中略...

```
/**  
 * 平行移動  
 * unif_translate  
 */  
protected int unif_translate;
```

中略...

```
/**  
 * X方向の平行移動量  
 */  
protected float translateX;  
  
/**  
 * Y方向の平行移動量  
 */  
protected float translateY;
```

```
/**  
 * Surfaceが生成されたタイミングの処理  
 */  
@Override  
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
```

{

```
    final String vertexShaderSource =  
        "uniform mediump vec4 unif_translate;" +  
        "attribute mediump vec4 attr_pos;" +  
        "void main() {" +  
        "    gl_Position = attr_pos + unif_translate;" +  
        "}" ;
```

①

```
    final String fragmentShaderSource =  
        "uniform lowp vec4 unif_color;" +  
        "void main() {" +  
        "    gl_FragColor = unif_color;" +  
        "}" ;
```

```
    // コンパイルとリンクを行う  
    this.program = ES20Util.compileAndLinkShader(vertexShaderSource, fragmentShaderSource);
```

}

```
// attributeを取得する
{
    attr_pos = glGetUniformLocation(program, "attr_pos");
    assert attr_pos >= 0;

    unif_color = glGetUniformLocation(program, "unif_color");
    assert unif_color >= 0;

    unif_translate = glGetUniformLocation(program, "unif_translate");
    assert unif_translate >= 0;
}

glUseProgram(program);
ES20Util.assertGL();
}

@Override
public void onSurfaceChanged(GL10 gl, int width, int height) {
    glViewport(0, 0, width, height);
}

/**
 * 毎描画時の処理
 */
@Override
public void onDrawFrame(GL10 gl) {
    glClearColor(0.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL10.GL_COLOR_BUFFER_BIT);

    // attr_posを有効にする
    glEnableVertexAttribArray(attr_pos);

    // ポリゴン色をアップロードする
    // 色はRGBAでアップロードする
    glUniform4f(unif_color, 1.0f, 0.0f, 0.0f, 1.0f);

    // 平行移動を行う
    {
        glUniform4f(unif_translate, translateX, translateY, 0, 0);

        translateX += 0.01f;
        translateY += 0.005f;

        // 適当なところで元に戻す
        if (translateX > 1) {
            translateX = 0;
        }
        if (translateY > 1) {
            translateY = 0;
        }
    }

    final float[] position = {
```

②

```

    // triangle 0
    // v0(left top)
    -0.75f, 0.75f,
    // v1(left bottom)
    -0.75f, -0.75f,
    // v2(right top)
    0.75f, 0.75f,

    // triangle 1
    // v3(right top)
    0.75f, 0.75f,
    // v4(left bottom)
    -0.75f, -0.75f,
    // v5(right bottom)
    0.75f, -0.75f,
};

glVertexAttribPointer(attr_pos, 2, GL_FLOAT, false, 0, ES20Util.wrap(position));
glDrawArrays(GL_TRIANGLES, 0, 6);
// デバッグ用メッセージを表示する
SampleUtil.setDebugText(getActivity(), String.format("translate(%.2f, %.2f)", translateX, ❸
translateY));
}
}

```

このプログラムを実行すると図13のような画面が表示されます。

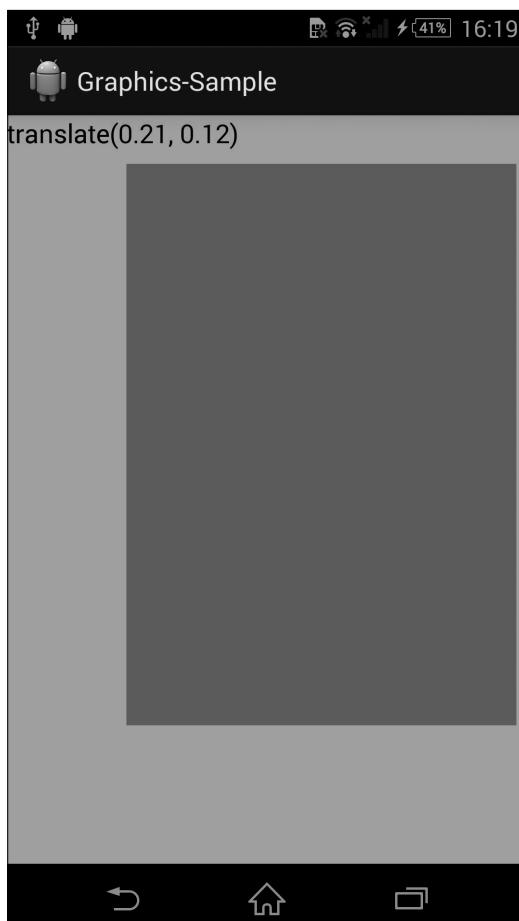


図13:リスト27のプログラムの実行結果の画面

## シンプルな平行移動を行う

このサンプルでは、頂点シェーダーに「頂点の座標を移動させるためのuniform変数」を追加しています。静的なデータである「頂点情報」を移動させる手段はいくつもあります。その中で最もシンプルな方法は、「任意の値だけ座標を加算する」というものです。

具体的には、「頂点シェーダー」では「gl\_Position」に頂点位置を書き込む際、「uniform変数」「unif\_translate」と加算を行っています(リスト28)。

リスト28:gl\_PositionWrite(gl\_Position書き込み部分)

```
gl_Position = attr_pos + unif_translate;
```

リスト27の②では描画ごとに「unif\_translate」に対してアプリが持つメンバ変数「translateX」と「translateY」の値をアップロードしています。それと同時に、「translateX」と「translateY」の値を少しづつ変化させています。「少しづつ値を変化して連続描画する」というのは、「アニメーション」を実現するために必要となる技術です。なお、サンプルでは少しづつX座標とY座標を加算していますが、それを続けているといつしか画面外へ四角形が消えてしまいます(リスト29)。

リスト30:reset(値のリセット処理)

```
if (translateX > 1) {
    translateX = 0;
}

if (translateY > 1) {
    translateY = 0;
}
```

最後に、このサンプル以後は「アニメーション」として毎秒60回も変換する変数が登場します。そのような場合、ブレークポイントで値を追いかけるのは非常に効率が悪い場合があります。ブレークポイントというのはある「瞬間」値を捉えますが、アニメーションは「瞬間の繰り返し」によって完成します。そのため、リスト27の③では常に値を画面に出力することで、値の変化をわかりやすくしています。

「SampleUtil.setDebugText」メソッドは、Activity内にあるデバッグ用TextViewに対して「setText」を呼び出すだけのシンプルなメソッドです(リスト31)。この表示を行うことで、「translate」の値が加算されるにしたがって少しづつ四角形が動いているということがわかりやすくなるのではないかでしょうか。

リスト31:setDebugText(デバッグ用メッセージ表示部分)

```
SampleUtil.setDebugText(getActivity(), String.format("translate(.2f, %.2f)", translateX, translateY));
```



## 16-2-2 行列演算を行う

3D描画にとって「行列」(Matrix)は欠かせない存在です。高校数学では学科や選択科目によっては学習しますが、そうでない場合もあります。そのため、もし現時点で「行列」を知らなかったとしても恥ずかしいことではありません。行列とは、図14のように「縦横に数字を並べたもの」とまずは考えてください。行列の基本的な形は、「斜めの要素がすべて1.0、それ以外は0.0」です。この状態を「単位行列」(Identity Matrix)と呼びます。

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

図14:単位行列

OpenGL ESでは、多くの場合、横4つ、縦4つに数字を並べた $4\times 4$ 行列を使用します。その他にも縦3つ、横3つ並べた $3\times 3$ 行列、同様に $2\times 2$ 行列も使用できます。この時、縦横の数は一致していなければなりません。プログラム上、これらはfloat型数値の配列として表されることが多いです。シェーダーの処理で特に重要なため、それらの行列に対応する「mat4」、「mat3」、「mat2」といった型が用意されています。

行列の特徴として、「頂点を3D空間内で任意座標に簡単に移動できる」というものがあります。移動というとシンプルなのは前述のような「加算処理」ですが、それ以外にも「回転」「拡大縮小」という処理も「頂点を任意座標に移動させる」ものと言えます。行列はたとえるなら「便利な計算機」です。頂点座標をその「計算機」に渡す(「乗算する」、「適用する」と言います)と、自動的に任意の座標に移動した状態の行列が取り出せます。計算機の内部構造は知らなくてもプログラミングはできます。

Androidには、行列演算を簡単に行うためのメソッドが大量に用意されているため、Androidで開発を行う限りは行列の複雑な処理を覚える必要はありません。単に「行列の使い方」に注力して覚えてください。

## 演習：行列で平行移動を行う

移動(Translate)を行うための行列は次のようにになります（リスト32）。

リスト32：移動(Translate)を行うための行列

```
public class Chapter02_02 extends Chapter01_01 {
    中略...

    /**
     * 頂点に適用する行列
     * unif_matrix
     */
    protected int unif_matrix;

    中略...

    /**
     * Surfaceが生成されたタイミングの処理
     */
    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        {
            final String vertexShaderSource =
                "uniform mediump mat4 unif_matrix;" +
                "attribute mediump vec4 attr_pos;" +
                "void main() {" +
                "    gl_Position = unif_matrix * attr_pos;" +
                "}";

            final String fragmentShaderSource =
                "uniform lowp vec4 unif_color;" +
                "void main() {" +
                "    gl_FragColor = unif_color;" +
                "}";

            // コンパイルとリンクを行う
            this.program = ES20Util.compileAndLinkShader(vertexShaderSource, fragmentShaderSource);
        }

        // attributeを取得する
        {
            attr_pos = glGetAttribLocation(program, "attr_pos");
            assert attr_pos >= 0;

            unif_color = glGetUniformLocation(program, "unif_color");
            assert unif_color >= 0;

            unif_matrix = glGetUniformLocation(program, "unif_matrix");
            assert unif_matrix >= 0;
        }

        glUseProgram(program);
        ES20Util.assertGL();
    }
}
```

中略...

```
@Override  
public void onDrawFrame(GL10 gl) {  
  
    中略...  
  
    // 平行移動を行う  
    {  
        float[] matrix = ES20Util.createMatrixIdentity();  
        matrix[4 * 3 + 0] = translateX;  
        matrix[4 * 3 + 1] = translateY;  
        glUniformMatrix4fv(unif_matrix, 1, false, matrix, 0);  
  
        translateX += 0.01f;  
        translateY += 0.005f;  
  
        // 適当なところで元に戻す  
        if (translateX > 1) {  
            translateX = 0;  
        }  
        if (translateY > 1) {  
            translateY = 0;  
        }  
    }  
  
    中略...  
}  
}
```

2

このプログラムを実行すると、図15のような画面が表示されます。



図15:Translate(移動)処理の実行結果

**リスト32の①**の部分では、頂点シェーダーで「mat4」型変数「unif\_matrix」を追加しています。行列は通常、「頂点ごとに違う内容」ではなく、「頂点すべてに同じ行列を適用」しなければなりません。そのため、「uniform変数」として「頂点シェーダー」に渡すのが基本となります。

「unif\_matrix」は次のように「attr\_pos」と乗算されて、「gl\_Position」に渡されます。このとき、「unif\_matrix」を「前方」、「attr\_pos」を「後方」に配置して計算してください(リスト33)。

リスト33:unif\_matrix(行列と頂点座標を乗算する)

```
gl_Position = unif_matrix * attr_pos;
```

この計算は「掛け算」ですが、行列計算に交換法則( $A \times B = B \times A$ )は成り立たず、掛け算の順番によって計算結果が全く異なってしまいます。行列は便利ですが、順番を間違えただけであっさりと想定外の動作をするため、計算順には常に細心の注意が必要です。行列の計算は必ず「先に適用したいものほど右側に配置する」と覚えてください。

**リスト32の②**では実際の行列生成とアップロードを行っています。Android SDKでプログラムを組む際、行列はfloat型の1次元配列となります。行列は見た目は2次元配列ですが、GPUにアップロードする際には1次元配列でないと問題が発生するためです。一方、C言語の場合はメモリの配置がコントロールできるため、2次元配列を使用できます。

単位行列の生成処理は非常にシンプルなため、サンプルではユーティリティメソッドを提供しています(リスト33)。

リスト33:Identify(単位行列の生成)

```
public static float[] createMatrixIdentity() {
    return new float[]{
        1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1,
    };
}
```

移動行列の生成は非常に簡単なため、サンプルではユーティリティメソッドを使用せずに直接配列に書き込んでいます。図16のように、行列の指定箇所へ「X移動量」「Y移動量」「Z移動量」を書き込むだけです。

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ \text{X移動量} & \text{Y移動量} & \text{Z移動量} & 1.0 \end{pmatrix}$$

図16:移動行列

図は2次元ですが、Java言語上では1次元配列となるため、配列のindexを指定する際に少しだけ工夫が必要です。とはいえ、非常に簡単なため、覚えることは容易でしょう。

今回は2D描画ですので、「translateX」と「translateY」をそれぞれ所定のindexへ書き込んでいます(リスト34～35)。

リスト34:matrix(一次元配列を使用する際の書き込み位置の計算方法)

```
matrix[4 * 縦のインデックス + 横のインデックス];
```

リスト35:matrix2(2D移動行列の作成)

```
matrix[4 * 3 + 0] = translateX;
matrix[4 * 3 + 1] = translateY;
```

最後に、行列をシェーダーへアップロードすれば完了です。行列のアップロードは「glUniformMatrix4fv」コマンドで行います。第1引数はアップロード先の「Location」、第2引数はアップロードする行列の個数(今回は1つなので「1」)です。第3引数は転置有無を指定しますが、必ず「false」を指定します。第4引数は行列を示した配列を指定します。第5引数は例によって配列のオフセットを指定しますが、ここでは0を指定します。

## 演習:ポリゴンを行列で拡大縮小させる

次のサンプルでは、ポリゴンを拡大縮小させています(リスト36)。注目すべきは、シェーダー部分に一切の変更を加えないということです。行列の内容を変化させるだけで、「移動」ではなく「拡大縮小」を行わせています。

## リスト36:Chapter02\_03.java

```
public class Chapter02_03 extends Chapter01_01 {  
  
    中略...  
  
    /**  
     * X方向の拡大縮小  
     */  
    protected float scaleX = 1;  
  
    /**  
     * Y方向の拡大縮小  
     */  
    protected float scaleY = 1;  
  
    中略...  
  
    /**  
     * 毎描画時の処理  
     */  
    @Override  
    public void onDrawFrame(GL10 gl) {  
  
        中略...  
  
        // 平行移動を行う  
        {  
            float[] matrix = ES20Util.createMatrixIdentity();  
            matrix[4 * 0 + 0] = scaleX; ①  
            matrix[4 * 1 + 1] = scaleY;  
            glUniformMatrix4fv(unif_matrix, 1, false, matrix, 0);  
  
            scaleX -= 0.01f; ②  
            scaleY -= 0.005f;  
        }  
  
        中略...  
  
    }  
}
```

このプログラムを実行すると図17のような画面が表示されます。

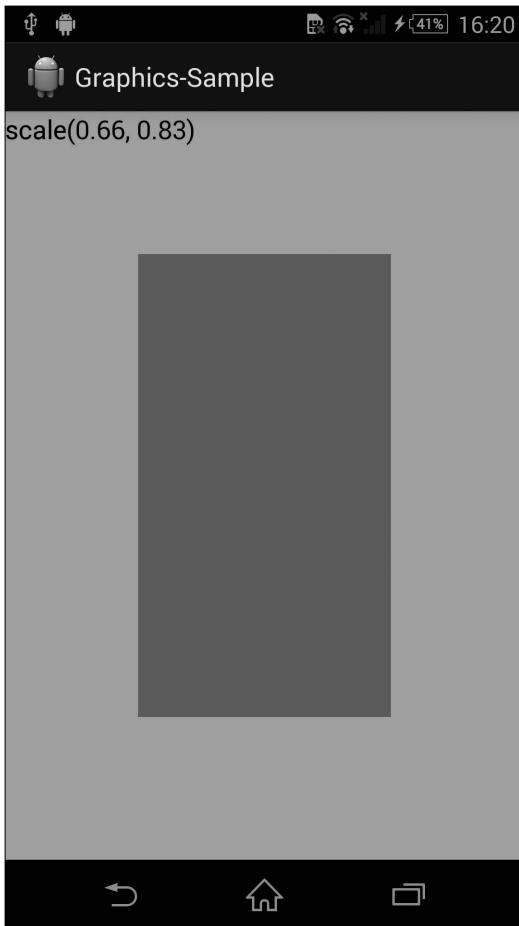


図17:拡大縮小処理の実行結果

拡大縮小(Scale)を行うための行列は図18のようになります。これも移動行列と同じく非常にシンプルで、斜め方向に拡大縮小の「倍率」を指定するだけです。普段(Identity)は1.0のため、拡大縮小が行われません(数値に1.0を乗算しても元の数字のままですね)。

$$\begin{pmatrix} \text{X倍率} & 0.0 & 0.0 & 0.0 \\ 0.0 & \text{Y倍率} & 0.0 & 0.0 \\ 0.0 & 0.0 & \text{Z倍率} & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

図18:拡大縮小行列

リスト36の①の部分では、拡大縮小用の行列を作成しています。これは前述のとおり、行列の所定位置に拡大率を指定するだけです。1.0より大きければ拡大、1.0より小さければ縮小となります。

リスト36の②の部分では、拡大率を少しずつ小さくしています。当然、この計算

を続ければ0よりも小さくなります、計算してもエラーにはなりません。なぜなら、負の値を乗算したら「座標の正負が反転する」だけで計算エラーにはならないからです。また、行列は内部的に「乗算」と「加算」しか行わないため、計算結果が不正になることは基本的にありません。

NaNやInfinity等を指定した場合、不正な行列が生成される場合がありますが、これは例外的な動作と言えるでしょう。そもそもそんな数値が演算に含まれないようにならなければなりません。

## 演習：ポリゴンを行列で回転させる

ポリゴンの回転(Rotate)は、前2つの行列に比べて非常に複雑です。初見では行列の式を見ただけで「関わりたくない」と思うかもしれません(図19)。

$XX(1.0 - \cos \theta) + \cos \theta$	$XY(1 - \cos \theta) - Z\sin \theta$	$XZ(1 - \cos \theta) + Y\sin \theta$	0.0
$XY(1 - \cos \theta) - Z\sin \theta$	$YY(1 - \cos \theta) + \cos \theta$	$YZ(1 - \cos \theta) - X\sin \theta$	0.0
$XZ(1 - \cos \theta) - Y\sin \theta$	$ZY(1 - \cos \theta) + X\sin \theta$	$ZZ(1 - \cos \theta) + \cos \theta$	0.0
0.0	0.0	0.0	1.0

図19:回転行列

実際のところ、Androidにはサポート用のメソッドが用意されているため、この式を覚える必要もなく、そのメソッドを呼び出すだけで済みます(リスト37)。そのため、

図19の行列は参考程度に考えてください。

### リスト37:chapter02\_04.java

```
public class Chapter02_04 extends Chapter01_01 {
    中略...
    /**
     * 回転量
     */
    protected float rotate;
    中略...
    @Override
    public void onDrawFrame(GL10 gl) {
        中略...
    }
}
```

```

// 回転を行う
{
    float[] matrix = ES20Util.createMatrixIdentity();
    Matrix.setRotateM(matrix, 0, rotate, 0, 0, 1.0f); ①
    glUniformMatrix4fv(unif_matrix, 1, false, matrix, 0);

    rotate += 1;
}

中略...

}

```

このプログラムを動作させると、図20のような画面が表示されます。

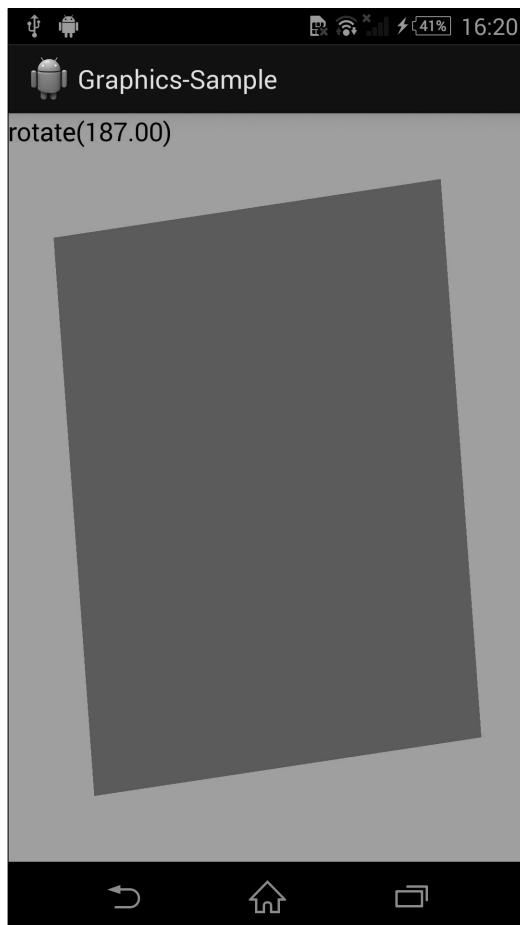


図20:回転処理の実行結果

回転行列の生成は「`android.opengl.Matrix`」クラスのstaticメソッド「`setRotateM`」で行えます。Androidでは別のパッケージにも同名の「`android.graphics.Matrix`」クラスが存在するため、importの指定には注意してください。

回転行列の引数は4つあり、指定方法が独特です。今は2Dの回転であるた

め、内容はシンプルに考えられます。しかし、このメソッドは本来3次元空間での回転を行うためにあります。3次元空間にはX/Y/Zの軸がそれぞれあります。X軸で回転するというのは、鉄棒の前回りや逆上がりを行うようなものです。Y軸での回転はコマの回転と同じです。Z方向の回転とは、壁時計の短針や長針のような動きになります。

さて、もう一度考えてください。3次元空間の回転軸の種類は、XYZの3種類だけでしょうか？ 答えはNOです。たとえば地球の地軸は公転面に対して傾いています。XYZだけでなく、そのような傾いた軸も3次元空間では扱わなければなりません。

そこで、「setRotateM」メソッドでは「回転軸の方向ベクトル」と「回転量」で回転行列を生成します。「方向ベクトル」とは、「ある地点からある地点への向き」を示したベクトル(XYZの値)です。あくまで「向き」であるため、物体がどれだけ離れていてもベクトルは同じです。たとえば、あなたが建物の1階にいるとして、2階にいる人も3階にいる人も、60階にいる人も、果ては屋上にいたとしても「上にいる」という意味で同じです。方向だけを指示す場合、「距離」は関係ないです。そのため、方向ベクトルは長さ(Xの2乗+Yの2乗+Zの2乗の平方根)は必ず1.0になります。

前置きが長くなりましたが、2Dの回転は前述の軸でいえば「時計の針」と同じZ軸の回転です。Z軸の方向ベクトルはXYZそれぞれ(0, 0, 1)となります(**リスト38**)。

#### リスト38: setRotateM(回転行列の生成)

```
Matrix.setRotateM(matrix, 0, rotate, 0, 0, 1.0f);
```

「Matrix.setRotateM」の第1引数には結果を格納する配列、第2引数は配列のオフセットを指定します。第3引数は回転量を指定します。これは「degree」、つまり360度系です。一般的なプログラミングの回転量は「ラジアン」で表されることが多いですが、このメソッドは360度系が採用されていることに注意してください。第4引数以降が回転軸の方向ベクトルです。前述の図のような計算はすべてメソッド内部に隠れていますため、シンプルに使用できます。

## TRY:X軸・Y軸の回転を試してみよう

前述のとおり、回転軸は無数にあります。X軸、Y軸、そしてその他の「斜め」の軸で回転させ、どのような見た目になるか確認してみましょう。



## 16-2-3 OpenGL ESにとっての「カメラ」

今まででは「ポリゴンを何らかの場所へ移動させる」ための行列作成を行ってきました。ポリゴン自体を明確に動かすだけでなく、「3D空間内にカメラを置いて、撮影する」という処理も行列によって記述できます。

### 演習: カメラを行列で指定する

言い方を変えれば、「ポリゴンをカメラから見た相対的な位置へ移動させる」という処理も行列で記述できます(リスト39)。

リスト39: ポリゴンをカメラから見た相対的な位置へ移動

```
public class Chapter02_05 extends Chapter01_01 {  
  
    中略...  
  
    /**  
     * 頂点に適用するワールド行列  
     * unif_matrix  
     */  
    protected int unif_world;  
  
    /**  
     * look at行列  
     */  
    protected int unif_look;  
  
    /**  
     * 射影行列  
     */  
    protected int unif_projection;  
  
    中略...  
  
    /**  
     * カメラ位置  
     * 初期位置は適当に決める  
     */  
    protected Vector3 cameraPos = new Vector3(10, 3, 10);  
  
    中略...  
  
    /**  
     * スクリーン幅  
     */  
    protected float screenWidth;
```

```
/***
 * スクリーン高
 */
protected float screenHeight;

/***
 * Surfaceが生成されたタイミングの処理
 */
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
{
    final String vertexShaderSource =
        "uniform mediump mat4 unif_world;" +
        "uniform mediump mat4 unif_look;" +
        "uniform mediump mat4 unif_projection;" +
        "attribute mediump vec4 attr_pos;" +
        "void main() {" +
        "    gl_Position = unif_projection * unif_look * unif_world * attr_pos;" +
        "}";
}

中略...
}

// attributeを取得する
{
    attr_pos = glGetAttribLocation(program, "attr_pos");
    assert attr_pos >= 0;

    unif_color = glGetUniformLocation(program, "unif_color");
    assert unif_color >= 0;

    unif_world = glGetUniformLocation(program, "unif_world");
    assert unif_world >= 0;

    unif_look = glGetUniformLocation(program, "unif_look");
    assert unif_look >= 0;

    unif_projection = glGetUniformLocation(program, "unif_projection");
    assert unif_projection >= 0;
}

glUseProgram(program);
}

@Override
public void onSurfaceChanged(GL10 gl, int width, int height) {
    glViewport(0, 0, width, height);
    this.screenWidth = width; ②
    this.screenHeight = height;
}

/***
 * ワールド行列設定
 */
private void setupWorld() {
```

```

float[] matrix = ES20Util.createMatrixIdentity();
Matrix.setRotateM(matrix, 0, rotate, 0, 0, 1.0f);
glUniformMatrix4fv(unif_world, 1, false, matrix, 0);

rotate += 1;
}

/**
 * カメラ情報のセットアップを行う
 */
private void setupCamera() {
    // カメラ注視
    float cameraLookX = 0; ③
    float cameraLookY = 0;
    float cameraLookZ = 0;

    // カメラ天地
    float cameraUpX = 0; ④
    float cameraUpY = 1;
    float cameraUpZ = 0;

    // 画角
    float fovY = 45.0f; ⑤

    // アスペクト比
    float aspect = screenWidth / screenHeight; ⑥

    // ニアクリップ/ファークリップ
    float nearClip = 1.0f;
    float farClip = 100.0f;

    {

        // look行列を生成する
        float[] matrix = ES20Util.createMatrixIdentity();
        Matrix.setLookAtM(matrix, 0, cameraPos.x, cameraPos.y, cameraPos.z, cameraLookX, cameraLookY, cameraLookZ, cameraUpX, cameraUpY, cameraUpZ);

        // アップロード
        glUniformMatrix4fv(unif_look, 1, false, matrix, 0); ⑦

        // カメラを移動する
        cameraPos.x -= 0.01f;
        cameraPos.z -= 0.005f;
    }

    {

        // projection行列を生成する
        float[] matrix = ES20Util.createMatrixIdentity();
        Matrix.perspectiveM(matrix, 0, fovY, aspect, nearClip, farClip); ⑧

        // アップロード
        glUniformMatrix4fv(unif_projection, 1, false, matrix, 0);
    }

    // デバッグ用メッセージを表示する
    SampleUtil.setDebugText(getActivity(), String.format("camera\n pos(%2f, %2f, %2f)\n loo

```

```
k(%.2f, %.2f, %.2f)\n    fovy(%.1f) aspect(%.2f)",  
        cameraPos.x, cameraPos.y, cameraPos.z,  
        cameraLookX, cameraLookY, cameraLookZ,  
        fovy, aspect));  
}  
  
@Override  
public void onDrawFrame(GL10 gl) {  
  
    中略...  
  
    // 各行列の設定を行う  
    setupWorld();  
    setupCamera();  
  
    中略...  
}  
}
```

このプログラムを実行すると図21のような画面が表示されます。

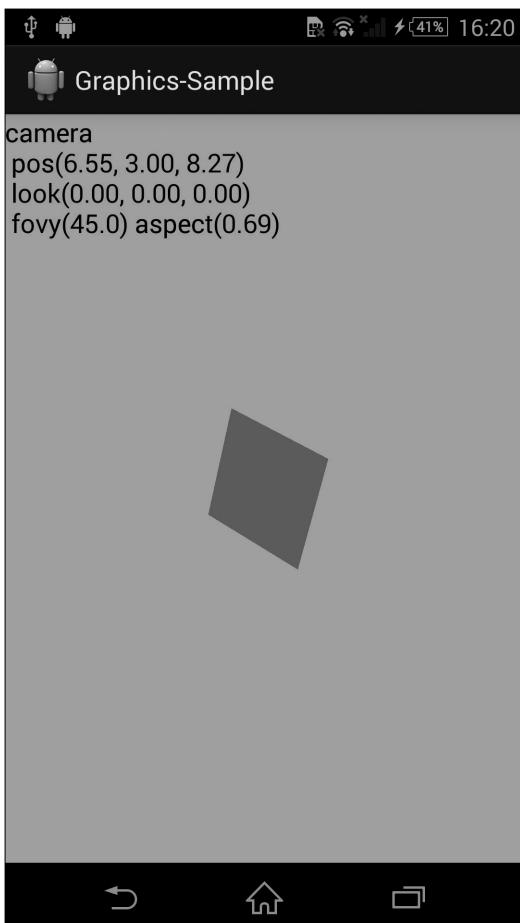


図21:カメラの移動処理の実行結果

皆さんは、デジタルカメラを持っていますか。スマートフォン搭載のカメラだけではなく、「一眼レフ」や「ミラーレス一眼」、「コンパクトデジタルカメラ」と呼ばれる種類のカメラです。2014年現在は数千円～と非常に安価に買えるため、複数台も持っている方が多いのではないでしょうか。現実世界でデジカメの「撮影」を行う際に使う

動作と、OpenGL ESにおけるカメラの処理は非常に似ています。そのため、まずは現実世界の「カメラ」について考えてみましょう。

3DCGというのは、仮想空間に浮かぶ「ジオラマ」のようなものです。限りある空間の中に3DCGの「物体」を浮かべ、それを画面に映し出すことが3DCGにおける「描画」です。ですので、博物館にあるジオラマや、プラモデルを並べたジオラマを撮影することを想像してください。

GPUは浮動小数点演算を行います。精度は4byte程度ですので、それ以上の広大な空間を扱うことができません。

あなたは撮影をする際、まずは配置すべき「建物」や「人形」といったミニチュアを作成します。時には数十種類ものミニチュア(Model)を用意しなければなりません。現実世界では「プロモデラー」という方々や造形師がこれを行います。ゲーム開発等では、これは「3dsMax」や「Maya」といった専用のツールを使った上で、デザイナーがモデルのデータを作成します。

次に、箱庭の中に「建物」や「人形」を「適切に配置する」必要があるでしょう。これは作られたデータを先ほどの「移動行列」、「回転行列」、「拡大縮小行列」を駆使して配置する作業です。これが今まで行っていた行列の作業です。これでまず被写体となるジオラマができ上がりました。OpenGL ESでいえば、行列によって「どこに配置するか」を確定できたということです。

ですが、まだ撮影を行えません。なぜなら、カメラの準備がまったく終わっていないからです。撮影を行うためには、まずカメラを三脚に固定します。つまり「位置を決め」ます。次に、被写体に対してカメラを向けます。つまり「注視場所を決め」ます。さらに、ズームイン・ズームアウトをして被写体をどれくらいの大きさで表示させるか考へるでしょう。一眼レフで言うならば、小さいレンズからバズーカ砲と呼ばれるサイズまで、自在にレンズを切り替えたりするでしょう。これらはすべて「画角を決定」「レンズ設定の決定」に相当します。

シャッターを押すと、その情景をイメージセンサーが捉えます。これが「glDrawArrays」コマンドに相当します。カメラとジオラマとの位置関係によって、当然ながら映り込む内容が異なります。これは「相対的な位置が違う」といえます。この相対的な座標を計算するのが「頂点シェーダー」であり、行列です。もしかしたら、あなたのデジカメには「セピア色で撮影」等のモードが付いているかもしれません。これは「フラグメントシェーダー」の計算処理です。

今回のシェーダーから、行列が3つに増えています。そのため、今までは単純に「unif\_matrix」と種別のない書き方をしていましたが、今回からは「unif\_world」「unif\_look」「unif\_projection」という3つの行列が登場しています。

「World行列」は、「ジオラマの中の配置」を担当します。これは今までのunif\_matrixに相当します。「Look行列」は、「カメラの配置」を担当します。「Projection行列」は、「カメラのレンズ」を担当します。

まずはリスト39の①のシェーダーに注目してください。「gl\_Position」の計算

方法は、次のようになっています。前述のように、シェーダー内の計算は「先に適用するものを右に書く」という法則があります。現実世界での処理順番は、例に倣うと次のとおりです。

1. モデルの作成(`attr_pos`)
2. モデルの配置(`World`行列)
3. カメラの位置決め(`Look`行列)
4. カメラのレンズ設定(`Projection`行列)

カッコ内はOpenGL ESの相当箇所です。それらがすべて「右から順番に」記述されているのがわかりますね(リスト40)。

リスト40:`unif_projection(gl_Positionの計算部分)`

```
gl_Position = unif_projection * unif_look * unif_world * attr_pos;
```

リスト39の②では、「Viewport」の高さと幅を保存しています。カメラでは端末の縦横サイズと「レンズ」の画角の両方を計算しなければならないため、ここではメンバ変数に保存を行っています。

リスト39の③～⑧の「setupCamera」メソッドが実際にカメラの行列を生成している部分です。

③の部分ではカメラが「どこを注視するか」を指定しています。今回はオブジェクトが原点(XYZ座標0, 0, 0)に配置されているため、常にカメラは原点を見つめるようになっています。

④の部分は、「カメラの天井方向」を指定しています。これは少しだけわかりにくいかもしれません。ですが、カメラで撮影をする際にカメラを横に持ったり、縦に持ったりしたことはあるかと思います。その「縦に持つ」「横に持つ」を表現するのがこの「天井方向」です。カメラを正しい向きで構えた状態は「カメラの天井が上に向いている」といえます。カメラを縦に構えた場合は、「カメラの天井が横方向に向いている」といえます。もちろん、斜め等もこの天井方向として表現可能です。

⑤では、レンズの画角を指定します。サンプルでは45.0度の画角です。一眼レフやカメラのExif情報では「○○mm(ミリ)」として表現されることが多いですが、計算の都合上ここでは「角度」になっています。もちろん、ここで指定するのは「degree」です。

⑥はOpenGL ES独特の事情によるパラメータです。まず、カメラは最終的に「画面(Viewport)」へと映像を投影しなければなりません。シンプルに計算してしまうと、「Viewport」の解説を行った時のように「縦に広がる」「横に広がる」等の「歪み」が出ててしまいます。それを抑えるため、OpenGL ESのカメラには「画面(Viewport)のアスペクト比」を与えて、歪みを抑えるのです。

次に、ニアクリップとファークリップです。「3DCGの空間」という限りある空間を効率的に使用するためのパラメーターです。たとえば、「これ以上遠くには何も映っていない」「これ以上近くには何も無い」という情報をカメラに伝えれば、不要な計算をしなくて済みます。ニアクリップよりも近くにあるピクセルは表示されず、ファークリップよりも遠くにあるピクセルもまた表示されません。

⑦の部分で実際に「Look行列」を生成しています。なお、カメラの位置はアニメーションさせる都合上メンバ変数として定義されています。その際の「Vector3型」は筆者が提供しているクラスで、xyzの3要素があるだけの簡素なものです(リスト41)。

リスト41:Vector3(Vector3クラスの実装)

```
public class Vector3 {  
    public float x = 0;  
    public float y = 0;  
    public float z = 0;  
  
    public Vector3(float x, float y, float z) {  
        this.x = x;  
        this.y = y;  
        this.z = z;  
    }  
}
```

リスト42に示すように、「Look行列」を作成するには、「Matrix.setLookAtM」メソッドを使用します。

リスト42:setLookAtM(Look行列を生成する)

```
Matrix.setLookAtM(matrix, 0, cameraPos.x, cameraPos.y, cameraPos.z, cameraLookX, cameraLookY, cameraLookZ, cameraUpX, cameraUpY, cameraUpZ);
```

第1、第2引数は結果を格納する配列を指定します。

第3～第5引数は、それぞれカメラ位置のXYZ座標を指定します。

第6～第8引数は、それぞれカメラの注視点のXYZ座標を指定します。

第9～第11引数は、それぞれカメラの天井方向の方向ベクトルを指定します。

リスト39の⑧でProjection行列を生成している部分を詳しく見てみましょう(リスト43)。

リスト43:perspectiveM(Projection行列を生成する)

```
Matrix.perspectiveM(matrix, 0, fovY, aspect, nearClip, farClip);
```

第1、第2引数は結果を格納する配列を指定します。

第3引数は画角を指定します。この画角は「Y」というキーワードが付いているように、「Y(縦)方向の画角」です。たとえば、映画のように「横長の画面」では縦方向と横方向の画角が一致しません。そこで、OpenGL ESでは「縦方向の画角」を基本として使用します。

横方向の画角は第4引数である画面のアスペクト比により自動で計算されます。

第5引数はニアクリップを指定し、カメラから見てこの数値よりも近いオブジェクトは画面に映りません。

第6引数はファークリップを指定し、カメラから見てこの数値よりも近いオブジェクトは画面に映りません。

アップロードについてはいつものように「glUniformMatrix4fv」で行います。この状態で描画を行うと、各種行列により「カメラから見た内容」のポリゴンが写し出されます。

## TRY: ポリゴンにテクスチャを貼り付けてみよう

このままではあまりにシンプルで面白みに欠ける画面です。シェーダー等を工夫し、ポリゴンに再度テクスチャを貼り付けてみてください。

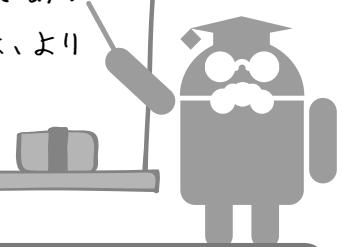
## CHALLENGE: シェーダーに記述する行列を1つだけにしてみよう

サンプルでは頂点シェーダーに3つの行列が記述されています。これでも十分な速度で処理が行えますが、さらなる高速化のために頂点シェーダー内に記述する行列を1つだけにしてみましょう。

# 16-3 OpenGL ES の 3D 描画の最適化

著 : 山下武志

前節までの知識さえあれば、すでに3DCGの複雑な描画さえ行うことができます。ただし、それは「可能」というだけであって、現実的な描画速度は得られていません。この節では、より高速な描画方法について解説します。



## この節で学ぶこと

- ・バッファオブジェクトを使用した描画
- ・深度バッファの有効化
- ・インデックスバッファを使用した描画

## この節で出てくるキーワード一覧

バッファオブジェクト  
深度バッファ  
インデックスバッファ  
`glDrawElements`  
`glGenBuffers`



## 16-3-1 立方体での例

### 課題:立方体を描画する

まずは「今までの知識」と少しだけの工夫を追加した状態で「テクスチャを貼り付けた立方体」を描画し、どのような問題点があるかを探ってみましょう(リスト44)。

リスト44:テクスチャを貼り付けた立方体の描画

```
public class Chapter03_01 extends Chapter01_01 {  
  
    中略...  
  
    /**  
     * テクスチャUniform  
     */  
    protected int unif_texture;  
  
    /**  
     * テクスチャオブジェクト  
     */  
    protected int texture;  
  
    中略...  
  
    /**  
     * Surfaceが生成されたタイミングの処理  
     */  
    @Override  
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {  
        {  
            final String vertexShaderSource =  
                "uniform mediump mat4 unif_world;" +  
                "uniform mediump mat4 unif_look;" +  
                "uniform mediump mat4 unif_projection;" +  
                "attribute mediump vec4 attr_pos;" +  
                "attribute mediump vec2 attr_uv;" +  
                "varying mediump vec2 vary_uv;" +  
                "void main() {" +  
                "    gl_Position = unif_projection * unif_look * unif_world * attr_pos;" +  
                "    vary_uv = attr_uv;" +  
                "}" ;  
  
            final String fragmentShaderSource =  
                "uniform sampler2D unif_texture;" +  
                "varying mediump vec2 vary_uv;" +  
                "void main() {" +  
                "    gl_FragColor = texture2D(unif_texture, vary_uv);" +  
                "}" ;  
  
            // コンパイルとリンクを行う
```

```

        this.program = ES20Util.compileAndLinkShader(vertexShaderSource, fragmentShaderSource);
    }

    // attributeを取得する
    {
        attr_pos = glGetUniformLocation(program, "attr_pos");
        assert attr_pos >= 0;

        unif_world = glGetUniformLocation(program, "unif_world");
        assert unif_world >= 0;

        unif_look = glGetUniformLocation(program, "unif_look");
        assert unif_look >= 0;

        unif_projection = glGetUniformLocation(program, "unif_projection");
        assert unif_projection >= 0;

        attr_uv = glGetUniformLocation(program, "attr_uv");
        assert attr_uv >= 0;

        unif_texture = glGetUniformLocation(program, "attr_uv");
        assert attr_uv >= 0;
    }

    texture = ES20Util.loadTextureFromAssets(getActivity(), "sample512x512.png");
    assert texture != 0;

    glUseProgram(program);
}

```

中略...

```

/**
 * サーフェイスの塗りつぶし処理
 */
protected void clearSurface() {
    glClearColor(0.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL10.GL_COLOR_BUFFER_BIT);
}

```

```

/**
 * 毎描画時の処理
 */
@Override
public void onDrawFrame(GL10 gl) {
    // サーフェイスを単色で塗りつぶす
    clearSurface();

    中略...

    glBindTexture(GL_TEXTURE_2D, texture);

    {
        // キューブを構築する
        final float LEFT = -1.0f;
        final float RIGHT = 1.0f;

```

```

final float FRONT = -1.0f;
final float BACK = 1.0f;
final float TOP = 1.0f;
final float BOTTOM = -1.0f;

final float[] cubeVertices = {
    // 上下面
    LEFT, BOTTOM, FRONT, 0, 0, RIGHT, BOTTOM, FRONT, 1, 0, LEFT, BOTTOM, BACK, 0, 1, //
    LEFT, BOTTOM, BACK, 1, 0, RIGHT, BOTTOM, FRONT, 0, 1, RIGHT, BOTTOM, BACK, 1, 1, //
    //
    LEFT, TOP, FRONT, 0, 0, LEFT, TOP, BACK, 1, 0, RIGHT, TOP, FRONT, 0, 1, //
    LEFT, TOP, BACK, 1, 0, RIGHT, TOP, BACK, 0, 1, RIGHT, TOP, FRONT, 1, 1, //

    // 左右面
    RIGHT, TOP, FRONT, 0, 0, RIGHT, TOP, BACK, 1, 0, RIGHT, BOTTOM, FRONT, 0, 1, //
    RIGHT, TOP, BACK, 1, 0, RIGHT, BOTTOM, BACK, 0, 1, RIGHT, BOTTOM, FRONT, 1, 1, //
    //
    LEFT, TOP, FRONT, 0, 0, LEFT, BOTTOM, FRONT, 1, 0, LEFT, TOP, BACK, 0, 1, //
    LEFT, TOP, BACK, 1, 0, LEFT, BOTTOM, FRONT, 0, 1, LEFT, BOTTOM, BACK, 1, 1, //

    // 前後面
    LEFT, TOP, BACK, 0, 0, LEFT, BOTTOM, BACK, 1, 0, RIGHT, TOP, BACK, 0, 1, //
    RIGHT, TOP, BACK, 1, 0, LEFT, BOTTOM, BACK, 0, 1, RIGHT, BOTTOM, BACK, 1, 1, //
    //
    LEFT, TOP, FRONT, 0, 0, RIGHT, TOP, FRONT, 1, 0, LEFT, BOTTOM, FRONT, 0, 1, //
    RIGHT, TOP, FRONT, 1, 0, RIGHT, BOTTOM, FRONT, 0, 1, LEFT, BOTTOM, FRONT, 1, 1, //
};

}

```

```

FloatBuffer buffer = ES20Util.wrap(cubeVertices);
glVertexAttribPointer(attr_pos, 3, GL_FLOAT, false, 4 * 3 + 4 * 2, buffer);
glVertexAttribPointer(attr_uv, 2, GL_FLOAT, false, 4 * 3 + 4 * 2, buffer.position(3)); // ③

```

注意:positionメソッドはbyte単位ではなく要素単位で指定する

```

    glDrawArrays(GL_TRIANGLES, 0, 36);
}
}
}

```

②

③

このプログラムを実行すると図22のような画面が表示されます。

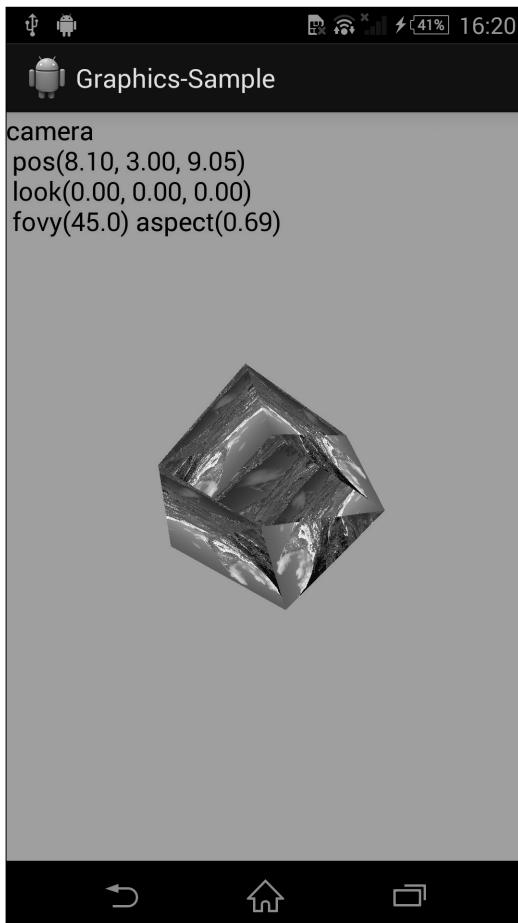


図22:テクスチャを貼り付けた立方体の描画例

このサンプルは後々少しだけ修正するため、リスト44の①の部分で「`glClearColor`」「`glClear`」をメソッド化しています。処理内容に変わりはありません。

リスト44の②は、頂点の「位置情報」と「UV情報」を一つの配列に詰め込んでいます。これは「頂点構造体」という技術で、2つの配列から個別に情報を読み出すより、1つの配列からデータを一定の法則で読み出すほうがGPUにとって効率よく処理できます。

今回からは本格的な3D情報を扱うため、位置情報はXYZ座標の3要素、「UV座標」は今までどおり2要素、合計で5要素/頂点となります。非常に効率よく処理できる反面、今回のサンプルでは36個もの頂点を構築するため、データが非常に見づらくなってしまっています。1頂点5要素×36頂点ですので、合計180個のデータができ上がります。情報量が多く、書くだけでも非常に大変ですね。実際にサンプルを作っていて(とてもとてもとても)苦労しましたがそれはまた別な話ですね。

頂点構造体を使用する場合、「`glVertexAttribPointer`」コマンドに少しだけ工夫が必要です。リスト44の③の「`glVertexAttribPointer`」コマンド部分に注目してください。まず、「`ttr_pos`」に設定する際の第2引数が、XYZの3要素を扱うため「3」に変更されています。それだけでなく、第5引数が「`4 * 3 + 4 * 2`」と記述されています。これは頂点構造体を使用する場合の「お約束」で、同じ配列から複数のデータを読み込む場合、「1頂点のバイト数」を伝えなければなりません。

ん。今回は4byte(float)×3要素(位置情報)+4byte(float)×2要素(UV情報)なので、合計20バイトが1頂点の大きさとなります。

以上の状態で描画を行うと、一見して立方体が正常に描画できているかのように見えます。しかしそう見ると、ポリゴンの前後関係が正しくありません。「後ろにある」はずのポリゴンが何故か前に見えてしまっているせいで、これが本当に立方体なのかどうかもわかりづらいでしょう。

## 課題：深度を正しく扱う

OpenGL ES 2.0では、「奥行き」を管理するための機能がデフォルトでOFFになっています。なぜなら、使わなければそれだけ処理速度の向上に役立つからです。たとえば2D描画だけしか使用しない場合は奥行きの機能は要りませんね。

次のサンプルコードで深度を正しく扱う方法を学びましょう(**リスト45**)。

リスト45:深度バッファ(Depth Buffer)を有効にする

```
public class Chapter03_02 extends Chapter03_01 {  
    @Override  
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {  
        super.onSurfaceCreated(gl, config);  
  
        // 深度バッファを有効化する ①  
        glEnable(GL_DEPTH_TEST);  
    }  
  
    @Override  
    protected void clearSurface() {  
        glClearColor(0.0f, 1.0f, 1.0f, 1.0f);  
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); ②  
    }  
}
```

このプログラムを実行すると、図23のような画面が表示されます。

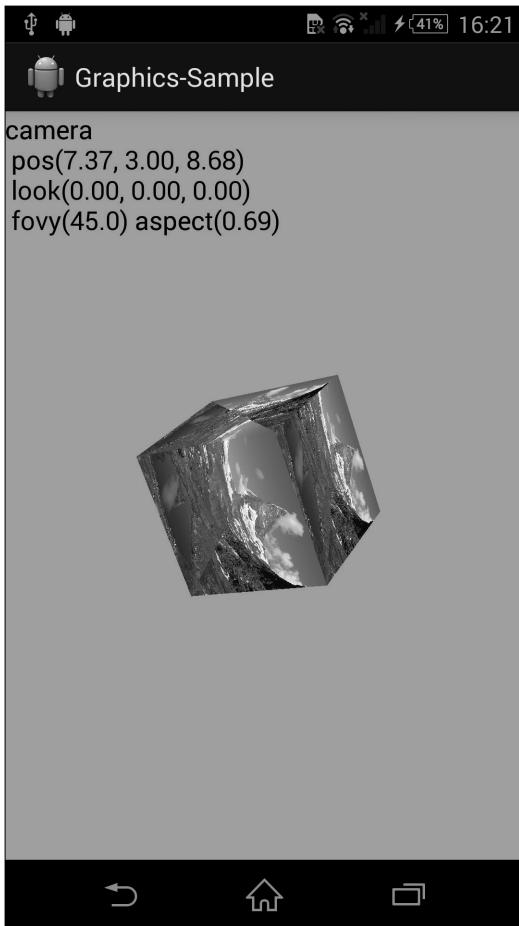


図23:深度を正しく扱った描画例

OpenGL ESの「奥行き機能」を「深度バッファ(Depth Buffer)」と呼びます。深度バッファを有効化するためには、2箇所に修正が必要ですが、難しくはありません。

**リスト45の①**では、「onSurfaceCreated」の初期化部分に、「glEnable」コマンドの呼び出しを追加します。このコマンドは、OpenGL ES 2.0の一部機能を「有効化」するためのものです。ここでは引数に「GL\_DEPTH\_TEST」を追加して、深度バッファを有効化しています。ちなみに、逆に機能を無効化する場合は「glDisable」というコマンドが用意されています。

**リスト45の②**では、「glClear」コマンドの引数に対して「GL\_DEPTH\_BUFFER\_BIT」を「追加して」います。「glClear」コマンドの引数は1つしかありませんが、このコマンドは引数を「|」でつないで、ビットOR演算を行うことで、「フラグを複数同時に伝える」ことができます。

Androidに限らず、公開されているAPIにはこのように「ON/OFFの状態を「|」演算子で組み合わせて送る」という引数が、たびたび登場します。1bitに付き1つの「ON/OFF」を管理できますので、int型(32bit)では最大32個のON/OFF組み合わせを保持できます。頭の片隅に覚えておくとよいでしょう。

「GL\_DEPTH\_BUFFER\_BIT」は「深度バッファを再度使用できる状態にする」ことを示します。この2つの操作を行うことで、図のように「正しく前後判定を行って」描画ができるのです。

## 課題：インデックスバッファで描画する

これだけでは速度の向上にはまだ不十分です。なぜなら、この頂点データには大量の「不要データ」が含まれているからです。たとえば、3DCGとは関係なく立方体の「頂点」を数えた場合、たった8個しかありません。前述の課題で作成していた大量のデータの中には、重複したデータが存在してしまっています。

OpenGL ESには、重複したデータを効率的に扱うための仕組み「インデックスバッファ」が存在します。これは頂点データ(頂点バッファ)の他に「インデックスバッファ」というもう1つのデータを作り出すことで、効率的なアクセスができるように取り計らってくれます。

「インデックスバッファが増えるのにデータが減るのか?」と疑問に思うかもしれません、見ればすぐに分かります。GPUは「頂点バッファ」と「インデックスバッファ」を与えられると、まずインデックスバッファに対してアクセスを行います。インデックスバッファには、その名のとおり、「index=頂点の番号」が記録されています。するとGPUは、インデックスバッファに記録された番号の頂点を読み込みにいきます。

重複した頂点を表現できるため、「インデックスバッファの個数 > 頂点の個数」でも問題ないという特徴があります。

また、インデックスが同じであれば、同じ頂点を読み込みに行きます。そのため、インデックスバッファは重複した頂点を表現すること得意としているのです(**リスト46**)。

一度処理した頂点はキャッシュが保持されるため、重複した頂点に何度も頂点シェーダー計算を行うことも避けられます。便利ですね。

リスト46:Chapter03\_03.java

```
public class Chapter03_03 extends Chapter03_02 {

    /**
     * 毎描画時の処理
     */
    @Override
    public void onDrawFrame(GL10 gl) {
        中略...

        {
            // キューブを構築する
            final float LEFT = -1.0f;
            final float RIGHT = 1.0f;
            final float FRONT = -1.0f;
            final float BACK = 1.0f;
            final float TOP = 1.0f;
            final float BOTTOM = -1.0f;
        }
    }
}
```

```

final float[] cubeVertices = {
    LEFT, TOP, FRONT, 0, 1, // 左上手前
    LEFT, TOP, BACK, 0, 0, // 左上奥
    RIGHT, TOP, FRONT, 1, 1, // 右上手前
    RIGHT, TOP, BACK, 1, 0, // 右上奥
    LEFT, BOTTOM, FRONT, 1, 1, // 左下手前
    LEFT, BOTTOM, BACK, 1, 0, // 左下奥
    RIGHT, BOTTOM, FRONT, 0, 1, // 右下手前
    RIGHT, BOTTOM, BACK, 0, 0, // 右下奥
};

final short cubeIndices[] = {
    //
    0, 1, 2, //
    2, 1, 3, //

    2, 3, 6, //
    6, 3, 7, //

    6, 7, 4, //
    4, 7, 5, //

    4, 5, 0, //
    0, 5, 1, //

    1, 5, 3, //
    3, 5, 7, //

    0, 2, 4, //
    4, 2, 6, //
};

FloatBuffer buffer = ES20Util.wrap(cubeVertices);
glVertexAttribPointer(attr_pos, 3, GL_FLOAT, false, 4 * 3 + 4 * 2, buffer);
glVertexAttribPointer(attr_uv, 2, GL_FLOAT, false, 4 * 3 + 4 * 2, buffer.position(3));
// 注意:positionメソッドはbyte単位ではなく要素単位で指定する

    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, ES20Util.wrap(cubeIndices)); ③
}
}
}

```

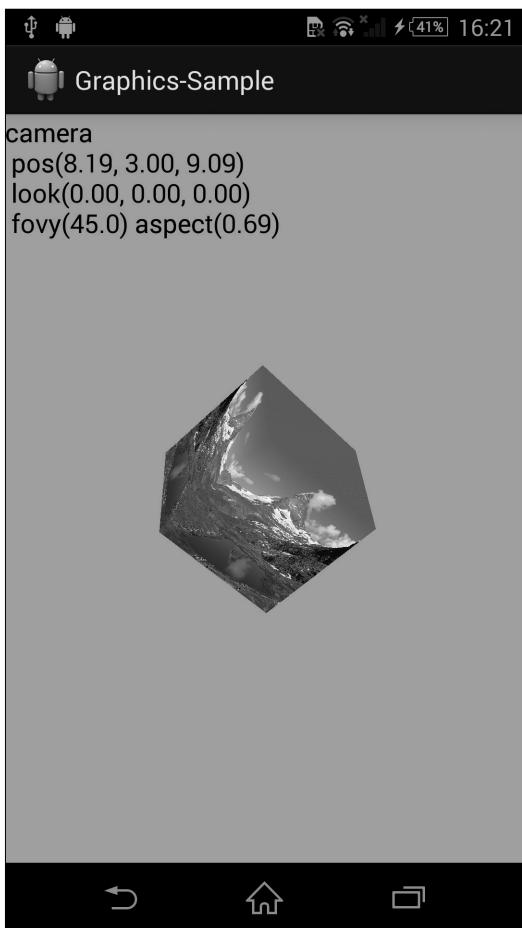


図24:インデックスバッファを使用した描画例

①の頂点バッファを見ると、データ量が一気に減って非常にシンプルになっていることがわかります。これは、重複している頂点情報を除いたためです。前述のとおり、立方体は頂点が8つしか存在しないため、実は非常にシンプルな頂点として表現できるのです。

その代わり、「インデックス何番の頂点を示すか」を表すためのインデックスバッファは、ちゃんと36個存在しなければなりません。②では12ポリゴン(2ポリゴン×正六面体)分のインデックスバッファを用意しています。

最後に、③では「glDrawElements」コマンドを使用して描画を行います。描画コマンド名が違うことに注目してください。インデックスバッファを使用する場合はこの「glDrawElements」コマンドで描画を行います。

第1引数は今までどおり「GL\_TRIANGLES」です。

第2引数は使用するインデックスバッファの個数を指定します。今回はインデックスバッファのデータが36個ありますので、36を指定します。

第3引数はインデックスバッファの型を指定します。このサンプルでのインデックスバッファは2byte整数で現していますので、「GL\_UNSIGNED\_SHORT」を指定します。

第4引数はインデックスバッファを渡します。頂点バッファの時と同じように、Bufferオブジェクトにラップする必要がありますが、流れは同じです。

これでインデックスバッファによる描画が行えるようになりました。頂点データが減つて、全体的にコードがスッキリしたのでは無いでしょうか。ですが、まだ足りません。まだ満足な描画速度を得るには足りないのです。

## 課題: バッファオブジェクトで描画する

さらなる高速化、効率化のためには、この頂点をVRAMに格納することが重要です。今まで使用していた「Buffer」オブジェクトはもともとCPUが効率よくデータを読み書きするためのクラスです。ですが、それはあくまでCPUにとって都合がよいというだけで、GPUにとって都合がいいとは限りません。

それだけでなく、CPU用のメモリ(RAM)はGPUから物理的に切り離されている場合があるため、効率のよいアクセスが行えない場合があります。そこで、頂点バッファやインデックスバッファを予めVRAMにアップロードし、GPUにとって都合がいいメモリ配置を行うことができます。OpenGL ESでは、そのように「事前にVRAMへアップロードするオブジェクト」のことを「バッファオブジェクト」と呼んでいます(図25)。

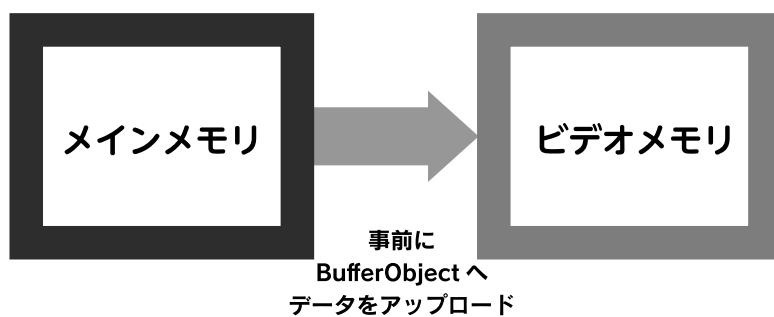
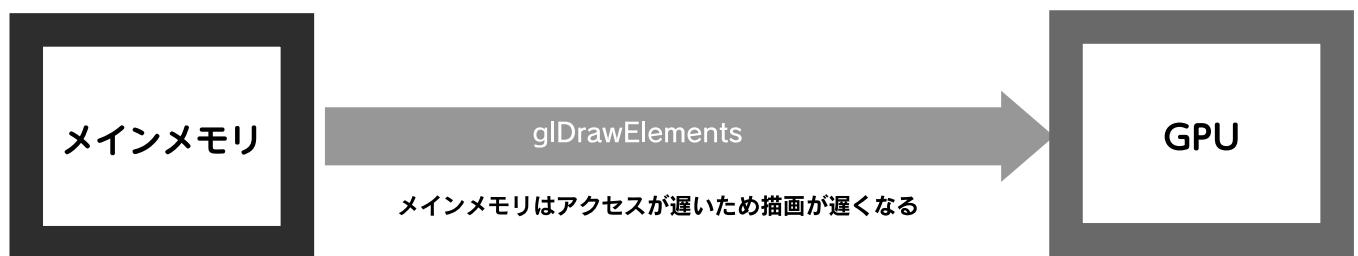


図25:25graphics-03-bufferobject.png/バッファオブジェクトの概念

このオブジェクトはテクスチャと同じく、int型のIDで管理されます(リスト46)。

リスト46:Chapter03\_04.java

```

public class Chapter03_04 extends Chapter03_02 {

    /**
     * 頂点バッファオブジェクト
     */
    int vertexBufferObject; ①

    /**
     * インデックスバッファオブジェクト
     */
    int indexBufferObject;

    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        super.onSurfaceCreated(gl, config);

        // オブジェクトを確保する
        {
            int[] objects = {0, 0};

            glGenBuffers(2, objects, 0);
            vertexBufferObject = objects[0]; ②
            indexBufferObject = objects[1];

            assert vertexBufferObject != 0;
            assert indexBufferObject != 0;
        }

        // 頂点バッファを生成する
        {
            // キューブを構築する
            final float LEFT = -1.0f;
            final float RIGHT = 1.0f;
            final float FRONT = -1.0f;
            final float BACK = 1.0f;
            final float TOP = 1.0f;
            final float BOTTOM = -1.0f;

            final float[] cubeVertices = {
                LEFT, TOP, FRONT, 0, 1, // 左上手前
                LEFT, TOP, BACK, 0, 0, // 左上奥
                RIGHT, TOP, FRONT, 1, 1, // 右上手前
                RIGHT, TOP, BACK, 1, 0, // 右上奥
                LEFT, BOTTOM, FRONT, 1, 1, // 左下手前
                LEFT, BOTTOM, BACK, 1, 0, // 左下奥
                RIGHT, BOTTOM, FRONT, 0, 1, // 右下手前
                RIGHT, BOTTOM, BACK, 0, 0, // 右下奥
            };

            glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
            glBufferData(GL_ARRAY_BUFFER, cubeVertices.length * 4, ES20Util.wrap(cubeVertices),
            GL_STATIC_DRAW);
            glBindBuffer(GL_ARRAY_BUFFER, 0);
        }
    }
}

```

```

// インデックスバッファを生成する
{
    final short cubeIndices[] = {
        //
        0, 1, 2, //
        2, 1, 3, //

        2, 3, 6, //
        6, 3, 7, //

        6, 7, 4, //
        4, 7, 5, //

        4, 5, 0, //
        0, 5, 1, //

        1, 5, 3, //
        3, 5, 7, //

        0, 2, 4, //
        4, 2, 6, //
    };

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferObject);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, cubeIndices.length * 2, ES20Util.wrap(cubeIndices), GL_STATIC_DRAW);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}
}

@Override
public void onDrawFrame(GL10 gl) {
    中略...

    {
        glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferObject);

        glVertexAttribPointer(attr_pos, 3, GL_FLOAT, false, 4 * 3 + 4 * 2, 0);
        glVertexAttribPointer(attr_uv, 2, GL_FLOAT, false, 4 * 3 + 4 * 2, 4 * 3); // 注意:positionメソッドはbyte単位ではなく要素単位で指定する
        glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, 0);
    }
}
}

```

このプログラムを実行すると、図26のような画面が表示されます。

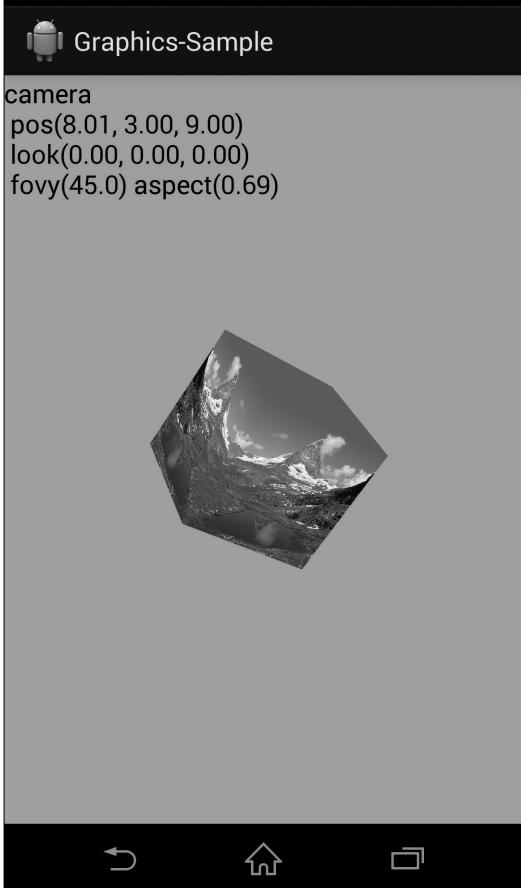


図26:26graphics-03-Chapter03\_04.pngバッファオブジェクトを利用して描画した結果

①ではバッファオブジェクトを保持するためのメンバ変数を追加しています。実際に確保を行っているのは②です。バッファオブジェクトの生成は「頂点バッファ」や「インデックスバッファ」の区別なく、「glGenBuffers」コマンドにより行えます。引数の構成はテクスチャの生成時と同じです。テクスチャの生成は1コマンドに付き1オブジェクトしか行っていませんでしたが、今回は2つ一括で確保しています。

③ではfloat[][]型である「cubeVertices」を頂点バッファへとアップロードしています。大まかな流れはテクスチャと同じです。

1. 頂点バッファをOpenGL ESにバインドする
2. 必要なデータをアップロードする
3. 頂点バッファをアンバインドする

まず、頂点バッファのバインドです。バッファオブジェクトをバインドするためには、glBindBufferコマンドを使用します。

第1引数には、バインド対象のオブジェクトの種類を指定します。頂点バッファオブジェクトであれば「GL\_ARRAY\_BUFFER」を指定し、インデックスバッファオブジェクトであれば「GL\_ELEMENT\_ARRAY\_BUFFER」を指定します。OpenGL ES 2.0では、このどちらかしかありません。

第2引数にはバインドするバッファオブジェクトを指定します。

次に行うのは頂点データのアップロードです。アップロードは「glBufferData」コマンドで行います。

第1引数には「GL\_ARRAY\_BUFFER」を指定します。

第2引数には頂点データのバイト数を指定します。今回はfloat型配列ですので、「cubeVertices」変数の「length×4」バイトを指定すればよいということになります。

第3引数はアップロードするデータ自体を指定します。

第4引数はメモリの使われ方のヒントを与えます。与えられる種類はいくつかあります。多くの場合、この引数には「GL\_STATIC\_DRAW」を指定してあげれば問題ありません。その他に「GL\_STREAM\_DRAW」「GL\_DYNAMIC\_DRAW」という種類があります。

最後に、データのアンバインドです。アンバインド処理は「glBindBuffer」コマンドの第2引数に「0」を指定するだけで済みます。

④はインデックスバッファのアップロードですが、各コマンドの第1引数が「GL\_ELEMENT\_ARRAY\_BUFFER」に変化している以外に違いはありませんので、問題なく行えるでしょう。

これでVRAMへのアップロードは完了しました。

⑤の部分で描画処理を行います。Java言語のOpenGL ESはいくつかのコマンドがオーバーロードされており、それを使用することで今までどおり「直接バッファを使用して描画」と「バッファオブジェクトを使用して描画」を切り替えられます。描画直前に「glBindBuffer」を2回、頂点バッファオブジェクトとインデックスバッファオブジェクトでそれぞれ呼び出しています。これは「今からこのバッファオブジェクトで描画を行うよ」という宣言をGPUに行うことになります。

次に、「glVertexAttribPointer」コマンドの呼び出しです。このコマンドでは、最後の引数が「Buffer」オブジェクトではなくint型に変化しています。このint型引数「offset」は、「バッファオブジェクトの何バイト目以降を使用するか」をGPUに教えるためのものです。たとえば、「cubeVertices」は頂点構造体により、「位置情報」「UV情報」の2つのデータが一括でアップロードされています。この時、「位置情報はバッファの何バイト目以降に入っているか」「UV情報はバッファの何バイト目以降に入っているか」を教えなければ、正しくアクセスすることができません。そこで、最後の引数により「バッファオブジェクトの何バイト目にデータがあるか」を教えるのです。今回は「位置情報」が0バイト目(先頭)に、「UV情報」が12byte目(位置情報の直後)に入っているため、それぞれ「0」と「12」を指定しています。

最終的な描画コマンドである「glDrawElements」コマンドも、同じく最後の引数は整数となっています。インデックスバッファは数値が押し込まれているだけの単純な情報ですので、「0」で問題ありません。



## まとめ

かなりの駆け足になりましたが、以上でグラフィックの章(OpenGL ESの章)の解説は終了です。教科書では紙面の関係でOpenGL ESのほんの入口しか触れていません。もしもっと高度な描画やさらなる詳細な情報を求める場合、市販の書籍を読むとよいでしょう。

「山下武志 OpenGL ES」をWebで検索!!

第16章

グラフィック

check!

### Android とiOSの方針の違い

Android 「L」では「OpenGL ES 3.1」+「Android Extension Pack」が発表されました。OpenGL ES 3.1では「Compute Shader」等のGPGPUが登場し、描画以外の場所でもOpenGL ES (GPU)が活躍できる箇所が増えようとしています。「Android Extension Pack」ではOpenGL ES 3.1が定める標準機能に加えて、「ジオメトリシェーダー」等の現代的なシェーダーにも対応しました。これはデスクトップPCの世界では一般的になりつつあり、移植性を高めることに役立つとの発表があります。

一方でApple社は2014年6月のイベントでOpenGL ESに変わるAPI「Metal」を発表しました。これはiOSデバイスに搭載されたGPUに最適化されたAPIで、OpenGL ESよりも少ないオーバーヘッドで実行できることを特徴としています。Appleの発表ではOpenGL ESよりもよりハードウェアに近い内容の操作が行え、かつオブジェクト指向のインターフェースを提供しています。ただし、このイベントではOpenGL ES 3.1への対応は発表されませんでした。

OpenGL ES 2.0は現在「現役」といえるAndroid端末はほぼすべて対応しています。それに対し、OpenGL ES 3.0は一部のGPUを搭載したAndroid 4.3以上が必要で、さらにAndroid Ext

ension Packを実装した端末も存在します。

Android 「L」ではジオメトリシェーダー等により、より高度なグラフィックを描画することができます。ですがその端末が普及するのは(少なくとも日本では)数年の時間が必要になります。多くのアプリや環境がES 2.0以下にしか対応していない端末を「切り捨て」するのではなく、Android 2.3ですら、日本では捨てるのを許さない人々がいるのですから。

2013年代まではOpenGL ES 2.0という共通インターフェースにより開発者は少ない労力で2つのプラットフォームに対応できました。しかし同時期(2014年6月)に発表された2社の方針の違いは、2014年以降のグラフィック処理において「OpenGL ES 3.0で共通処理を行う」のか、「それぞれのプラットフォームに最適化する」のかの判断を迫っています。

「Unity」や「Unreal Engine」等のエンジンに特化した製品は、間違いなく後者を選択するでしょう。しかし、独自のレンダリングエンジン、ゲームエンジンによってマルチプラットフォームに対応した開発を行ってきた開発者がどの方向へ舵取りを行うか、今後の展開が非常に楽しみです。

