

## ゲーム開発には欠かせない「SurfaceView」とは

### ●定期的な再描画に向いているウィジェット

SurfaceView というのは、名前からも分かる通り、View のサブクラスです。今まで紹介した画面部品群「ウィジェット」も、同様に View のサブクラスですが、それらのウィジェットと SurfaceView には決定的な違いがあります。

それは、描画の方式が違うのです。パッケージ「android.widget」に属するウィジェットは、アプリケーションのスレッド内で描画が行われるため、定期的に再描画を繰り返すゲームなどには向いていません。

一方、SurfaceView は、アプリケーションのスレッドと描画処理のスレッドが独立している（同時並行処理）ため、定期的な再描画に向いています。見た目が滑らかな動きになります。

ちなみに、android.widget.VideoView クラスは、SurfaceView のサブクラスです。「SurfaceView が動画再生などの負荷の描画処理に向いている」という一例です。

### ゲーム開発には欠かせない「SurfaceView」とは

SurfaceView というのは、名前からも分かる通り、View のサブクラスです。以前連載第4回の「簡単でワクワクする Android ウィジェット 10 連発！」で紹介した「ウィジェット」も、同様に View のサブクラスですが、それらのウィジェットと SurfaceView には決定的な違いがあります。

それは、描画の方式が違うのです。パッケージ「android.widget」に属するウィジェットは、アプリケーションのスレッド内で描画が行われるため、定期的に再描画を繰り返すゲームなどには向いていません。一方、SurfaceView は、アプリケーションのスレッドと描画処理のスレッドが独立しているため、定期的な再描画に向いています。

ちなみに、android.widget.VideoView クラスは、SurfaceView のサブクラスです。「SurfaceView が動画再生などの負荷の描画処理に向いている」という一例です。

## ■コラム 「View でもゲームは作れる」

今回は、SurfaceView がテーマなのですが、通常の View クラスでも問題なくゲームは作れます。

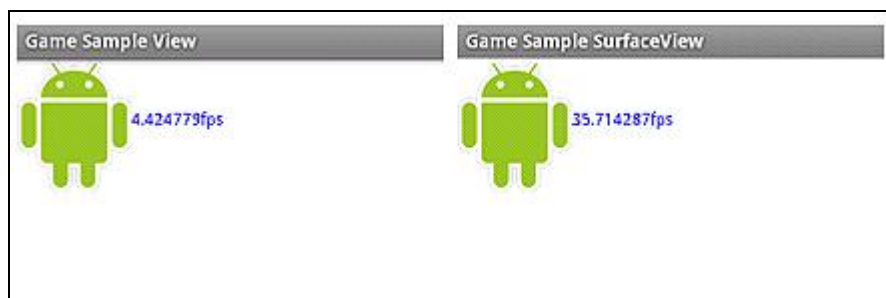
通常の View は invalidate() を呼び出して、間接的に onDraw(Canvas) を呼び出します。これまでの Java (Java SE や Java ME) は「repaint()」というメソッド名でしたが、Android では Windows API っぽい「invalidate()」というメソッド名が使用されています。

さて、「通常の View でもゲームは作れる」といいましたが、それでも以下のようなゲームには向きません。

- \* 3D を使用したゲーム

- \* FPS (1 秒間当たりの描画回数) が高いゲーム

今回のデモアプリに、View と SurfaceView をそれぞれ使用した、十字キーで Android マスコットが動くサンプルを含めています。このサンプルは、1 秒間当たりの描画回数を表示します。



## ■SurfaceView の使い方の一例

```
//オブジェクトの取得
```

```
SurfaceView surfaceView = (SurfaceView)v;
```

```
//
```

```
Canvas canvas = surfaceView.getHolder().lockCanvas();
```

```
Paint paint = new Paint();
```

```
canvas.drawColor(Color.WHITE);
```

```
paint.setColor(Color.BLUE);
```

```
paint.setAntiAlias(true);
```

```
paint.setTextSize(24);
```

```
canvas.drawText("Hello, SurfaceView!", 0, paint.getTextSize(), paint);
```

```
surfaceView.getHolder().unlockCanvasAndPost(canvas);
```

SurfaceView には、「getHolder()」という SurfaceHolder を取得するメソッドがあります。さらに SurfaceHolder には、「lockCanvas()」という Canvas を取得するメソッドがあります。ここで取得したキャンバスに自由に描画し、最後に SurfaceHolder の unlockCanvasAndPost(Canvas)を呼び出して描画を完了します。

SurfaceHolder の lockCanvas()メソッドはスレッドセーフ（synchronized による排他制御）に作られています。要するに、ほかのスレッドで lockCanvas()が呼ばれると、もう一方のスレッドは、unlockCanvasAndPost(Canvas)が呼び出されるまで描画を行えません。その待っている間にオフスクリーン（メモリ上）に描画を済ませてしまう、というのはテクニックの1つです。

ただし、このサンプルの使い方は、実は一般的ではありません。一般的には、SurfaceView は継承して使用します。サンプルの Hello.java を見てください。Hello.java では、SurfaceView のサブクラスを定義して、SurfaceHolder にコールバックを設定して、コールバック内で描画を行っています。

SurfaceView の基本が分かったところで、関連する重要なクラスとメソッドをまとめておきます。

表 1 SurfaceView に関連する重要なクラスとメソッド

SurfaceView	
SurfaceHolder getHolder()	SurfaceHolder を取得
SurfaceHolder	
void addCallback(SurfaceHolder.Callback)	コールバックを設定
removeCallback(SurfaceHolder.Callback)	コールバックを解除
Canvas lockCanvas()	描画を開始
void unlockCanvasAndPost(Canvas)	描画を終了
Canvas	
int save()	状態を保存
void restore()	状態を復元
Paint	

<code>setAntiAlias(boolean)</code>	アンチエイリアスを設定

## ■ サンプルコード（基本形-静止した画像の描画）

### ● プロジェクトの新規作成。

- \* プロジェクト名 : SampleSurView
- \* ビルド・ターゲット : Android 1.6
- \* アプリケーション名 : Sample Surface View
- \* パッケージ名 : `jp.sample.SampleSurView`
- \* Create Activity : SampleSurView

### ● リソースレイアウト(`main.xml`)の修正はなし

今回はリソースレイアウトを使用しないため、`main.xml` ファイルはデフォルトのままで OK。

### ● ソースコードの修正

`SampleSurView.java` ファイルを修正します。メインクラスである `SampleSurView` クラスの内部に、`SurfaceView` クラスのサブクラスとして `DrawSurfaceView` クラスを作ることになります（コードの全体は以下）。このサブクラスが `SurfaceView` を作って描画します。

```
package jp.sample.SampleSurView;
```

```
import android.app.Activity;
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.os.Bundle;
import android.util.Log;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
```

```

public class SampleSurView extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // *注意：ここはレイアウトリソースである R.layout.main を使用せず、
        // SurfaceView を継承したサブクラス DrawSurfaceView のインスタンスを渡す。
        setContentView( new DrawSurfaceView(this) );
    }

// SurfaceView を継承したサブクラス（内部クラスとして） -----
    class DrawSurfaceView extends SurfaceView implements SurfaceHolder.Callback {

        public DrawSurfaceView(Context context) {
            // コンストラクタ
            super(context);
            // SurfaceHolder を取得するために、getHolder メソッドを呼び出す
            // そして、コールバックを登録するために、addCallback メソッドも呼び出す。
            getHolder().addCallback(this);
        }

        @Override
        public void surfaceChanged(SurfaceHolder holder, int format, int width,
            int height) {
            // SurfaceView のサイズなどが変更されたときに呼び出されるメソッド。
            // 今回は何もしない（ログ表示のみ）。
            Log.d("SampleSurView", "surfaceChanged called.");
        }

        @Override
        public void surfaceCreated(SurfaceHolder holder) {
            // SurfaceView が最初に生成されたときに呼び出されるメソッド
            Log.d("SampleSurView", "surfaceCreated called.");
        }
    }
}

```

```

// SurfaceHolder から Canvas のインスタンスを取得する
Canvas canvas = holder.lockCanvas();
// Paint クラスのインスタンスを作る。これは、描画するときに使用する
// 色は青、アンチエイリアス ON、テキストサイズ 24
Paint paint = new Paint();
paint.setColor(Color.BLUE);
paint.setAntiAlias(true);
paint.setTextSize(24);
// Canvas の背景色を白で塗る
canvas.drawColor(Color.WHITE);

// Canvas に文字を書く。ここで Paint クラスのインスタンスを用いる。
// つまり、青色でテキストサイズが 24 でアンチエイリアスのかかった
「Sample Text」を描画する
canvas.drawText("Sample Test", 0, paint.getTextSize(), paint);

// 描画が終わったら呼び出すメソッド。
holder.unlockCanvasAndPost(canvas);
}

@Override
public void surfaceDestroyed(SurfaceHolder holder) {
    // SurfaceView が破棄されるときに呼び出されるメソッド
    // 今回は実装しない。
    Log.d("SampleSurView", "surfaceDestroyed");
}

} //ここまで-内部クラス-----
}

```

---

ポイント

このサブクラスは、SurfaceView を使用するために、SurfaceView クラスを継承 (extends) します。また、SurfaceView を使用するために、SurfaceHolder.Callback クラスのインタ

ーフェースを実装しないといけないため、implements します。

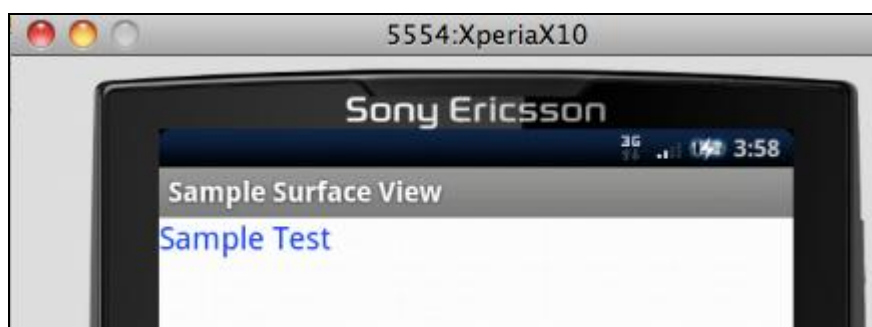
また、SurfaceHolder.Callback クラスを implements すると、3つのメソッド surfaceChanged、surfaceCreated、surfaceDestroyed を実装しなければなりません。今回のサンプルでは、surfaceCreated メソッド、つまり SurfaceView が生成されたときに呼び出されるメソッド、のみ実装しています。ここでは、SurfaceView を白で塗りつぶして、「Sample Text」というテキストを青色で表示します。

最後に重要なポイントとしては、SampleSurView クラスの onCreate メソッドで setContentView メソッドを呼んでいます。通常は R.layout.main というようにリソースレイアウトを指定します。

しかし今回は、DrawSurfaceView クラスのインスタンスを渡しています。

## ■実行結果

エミュレータで起動すると、アプリが起動して以下のような実行結果となります。



一応アンチエイリアスは ON にしていますが、見た目汚いように思います。テキストを表示するときは BMP ファイルなどにしたほうが良いかも知れません。

このサンプルは起動時に一度だけ描画するのみなので、SurfaceView のありがたみがまったく分かりませんでした。次は定期的・連続的に描画するプログラムを作成してみたいと思います。

## ■SurfaceViewによるユーザからのイベントに対する応答

ゲームでは、アクティビティ上に GUI のウェジットを配置していくのではなく、背景やキャラクターなどを独自に描画しなければなりません。

Android の場合は SurfaceView という専用の View を使います。この SurfaceView のサブクラスとして独自の View 系クラスを定義し、キーイベントや描画ロジックを実装していきます。

ユーザからのイベントの処理も、基本的には View での処理とまったく同じです。イベント処理のプログラムの例を、SurfaceView を使って実現してみます。

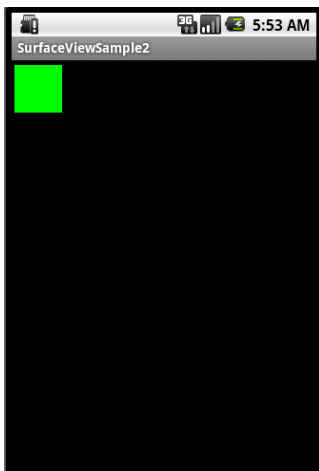
View クラスの処理と異なる点は、View クラスでは画面の更新時に onDraw メソッドが自動的に呼ばれ、グラフィックの描画がおこなわれていましたが、SurfaceView では呼び出されません。

また、画像を再描画したい場合には、invalidate メソッドを実行すれば、onDraw メソッドが呼び出されていましたが、SurfaceView では、invalidate メソッドを実行しても、onDraw メソッドは呼び出されません。

このプログラムでは、doDraw メソッドを定義して画像の更新処理をおこないたい場合は、このメソッドを呼び出すようにしています。

また、View クラスの描画処理をイメージしやすいように onDraw メソッドを定義して、doDraw メソッドから onDraw メソッドを呼び出していますが、onDraw メソッドが自動的に呼び出されるわけではないので、onDraw メソッドを定義せずに、doDraw メソッド内で直接グラフィックの描画をおこなってもかまいません。

## ■サンプルコード（ユーザからの入力イベントに対応した画像の描画）



画面上をタッチすると四角形が移動、十字キーでも移動



●プロジェクトの新規作成。

- \* プロジェクト名 : SurfaceViewSample2
- \* ビルド・ターゲット : Android 1.6
- \* アプリケーション名 : SurfaceViewSample2
- \* パッケージ名 : jp.sampleSurfaceView
- \* Create Activity : SurfaceViewSample2

●リソースレイアウト(main.xml)の修正はなし

今回はリソースレイアウトを使用しないため、main.xml ファイルはデフォルトのままでOK。

●ソースコード (SurfaceViewSample2.java)

```
package jp.sampleSurfaceView;

import android.app.Activity;
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.os.Bundle;
import android.view.KeyEvent;
import android.view.MotionEvent;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class SurfaceViewSample2 extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //下記 SurfaceView を継承したサブクラスを生成し、コンテンツに設定
        setContentView(new CustomSurfaceView(this));
    }
}
```

```
//-----
```

//今回は別クラスとして切り分けて記述してみました。必ずしも内部クラスにする必要はありません。

```
class CustomSurfaceView extends SurfaceView
    implements SurfaceHolder.Callback {

    private static final float rectWidth = 50;
    private static final float rectHeight = 50;

    private float x0 = 0, y0 = 0;
    private float dx = 0, dy = 0;
    private float xOffset = 0, yOffset = 0;

    public CustomSurfaceView(Context context) {
        super(context);

        setFocusable(true);
        getHolder().addCallback(this);
    }

    public void surfaceChanged(SurfaceHolder holder, int format, int width,
                               int height) {
        // SurfaceView が変化（画面の大きさ、ピクセルフォーマット）した時の
        イベントの処理を記述
    }

    public void surfaceCreated(SurfaceHolder holder) {
        // SurfaceView が作成された時の処理（初期画面の描画等）を記述
        doDraw(holder);
    }

    public void surfaceDestroyed(SurfaceHolder holder) {
        // SurfaceView が廃棄されたる時の処理を記述
    }
}
```

```

//独自に作成した再描画メソッド
private void doDraw(SurfaceHolder holder) {
    Canvas canvas = holder.lockCanvas();
    onDraw(canvas);//再描画
    holder.unlockCanvasAndPost(canvas);
}

//描画処理記述部分
@Override
protected void onDraw(Canvas canvas) {
    canvas.drawColor(Color.BLACK);

    Paint paint = new Paint();
    paint.setColor(Color.GREEN);
    //四角形を病害位置を更新して、描く
    canvas.drawRect(xOffset + dx, yOffset + dy, xOffset + dx + rectWidth,
        yOffset + dy + rectHeight, paint);
}

//キーイベント処理
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    switch (keyCode) {
        case KeyEvent.KEYCODE_DPAD_LEFT:
            xOffset--;
            break;
        case KeyEvent.KEYCODE_DPAD_RIGHT:
            xOffset++;
            break;
        case KeyEvent.KEYCODE_DPAD_UP:
            yOffset--;
            break;
        case KeyEvent.KEYCODE_DPAD_DOWN:
            yOffset++;
            break;
        default:
    }
}

```

```

        return true;
    }
    doDraw(getHolder());//上記の再描画メソッドを呼ぶ
    return true;
}

//タッチイベント処理
@Override
public boolean onTouchEvent(MotionEvent event) {
    float x = event.getX();
    float y = event.getY();
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            x0 = x;
            y0 = y;
            break;
        case MotionEvent.ACTION_MOVE:
            dx = x - x0;
            dy = y - y0;
            break;
        case MotionEvent.ACTION_UP:
            xOffset += (x - x0);
            yOffset += (y - y0);
            dx = 0;
            dy = 0;
            break;
        default:
            return true;
    }
    doDraw(getHolder());
    return true;
}
}

```

---

以上