JOGLによるOpenGL入門

この文書は,JOGLを用いたOpenGLのチュートリアルです。日本のOpenGL界隈では有名な、和歌山大学の床井先生の「OpenGL による「手抜き」OpenGL入門」を基にしています。末尾の条件に従い自由に再配布・改変してくださっても構いません。

なお、この文書の公開を快諾してくださった床井先生に感謝いたします。

初版 2015/11/12

更新 2015/11/15

目次

1.はじめに

1.1想定している読者

この文書では、Javaの基本は理解されている読者が、JOGLを使いたい場合に参考にしていただくことを目的として描かれています。ま た、 C, C++などの言語でOpenGLを使った経験はあるが、JOGLを使いたい方にも参考となることを目指しています。

筆者は主にOSX(Marvericks)を用いて動作確認しています。

1.2なぜJOGLか

床井先生の「OpenGL手抜き入門」には、以下のような一文があります。

> OpenGLとGLUTを組み合わせれば、
>
> - UNIX系OS(Linux、FreeBSD等を含む)とWindowsとMacのいずれでも動く、
> - リアルタイムに三次元表示を行うプログラムが、
> - とっても簡単に書けてしまう、
>
> という三拍子そろったメリットが得られます。

これについては、"Write once、run anywhere"がウリのJavaであり、JOGLももちろん複数のプラットフォームで使えるようになっています。
プラットフォームとしては、Windows/OSX/Unix系だけでなく、Android用のサンプルがAPKファイルの形で提供されてい ることから、Androidを採用したスマホやタブレットでも使えるようです。なお、iPhone/iPadについては、現時点では不明で す。

1.3 JOGL以外の選択肢

現時点では、Java系ではProcessing、LWJGLが OpenGLをサポートしています。libGDXはOpenGL ESをサポートしています。

Javaで、OpenGL系列ではない技術となると、１０年ほど前ならJava3Dが唯一の選択肢だったと思います。Java 8からは、JavaFX 3Dが登場しています。
JavaFXとJOGLの組み合わせが動かせないか、疑問に思われる方もいらっしゃると思いますが、「動 くはず」という情報だけは見つかりました。動作確認は行っていません。

1.4 UIフレームワーク

JavaのUIはAWT、Swing、SWTがあり、JOGLはいずれにも対応しています。JOGLはこれとは別にNEWTと呼ばれる独自 のライブ ラリが用意されています。但し、NEWTはSwingのJFrameやAWTのWindowとは違い、ボタンやテキストフィールドなどを貼り付けるよ うな使い方は想定されていないようです。
この文書で解説するのは、主にNEWTとします。

1.5 OpenGLのバージョン

この文書では、OpenGLの固定パイプライン機能を対象としています。残念ながら、最近のOpenGLで使われるGLSLについては対象 外です。

2.JOGLのインストール

(1)Mavenを使う方法

JOGLはMaven セントラル・リポジトリに登録されているので、以下の内容("*"は具体的な数字で置き換えます)をpom.xmlに書いておけ ば、"mvn install"によりダウンロードされます。
JavaDocをダウンロードしたい場合、"mvn dependency:resolve -Dclassifier=javadoc"、Javaソースをダウンロードしたい場合、"mvn dependency:resolve -Dclassifier=sources"とします。

```
org.jogamp.jogl
jogl-all-main
2.*.*
```

(2)コンパイル済みバイナリの入手

JOGLのコンパイル済みのバイナリはhttp://jogamp.org/か ら入手できます。Windows用、Linux用、OSX用がすべて一つの圧縮ファイルとして提供されています。

なお、検索エンジンで探すとhttps://kenai.com/projects/jogl/pages/Homeが 見つかることがありますが、これは古いです。

先のページの下の、Builds/Downloadsの"Current" の右にある"zip"クリックした先のページにある、jogamp-all-platforms.7zに、 必要となる全てのJARファイルが含まれています。
7z形式のアーカイブファイルを解凍できるソフトが必要となります。Windowsでは7-zip、 OSXではEz7z、 Unix系ではP7ZIPが使えるようです。

余談ですが、前述の圧縮ファイルには、JOCL(OpenCL関係)、JOAL(サウンド関係)などのJARファイルも同梱されています。
これらを適切なフォルダ、あるいはディレクトリに格納してください。なお、このときにディレクトリの構造は解凍した状態から変えないようにし ます。

(3)パッケージ管理ツールを使う方法
Ubuntuではlibjogl2-javaというパッケージにより提供されています。Debianでも同様と思わ れます。 RedHat系では未確認です。
OSXでは、HomeBrewには**無い**ことを確認しました。MacPortsでは未確認です。

(4)ソースからのコンパイル

この文書では対象外としますが、http://jogamp.org/jogl/doc/HowToBuild.htmlに 方法が書かれています。

(5)JavaDocおよびJavaソースコードの入手

mavenを使う場合(1)項を、それ以外の場合、リファレンスを見てくださ い。

3.コーディング

3.1 パッケージ名変更について

バージョン2.3より、JOGL関連クラスのパッケージ名が以下のとおり変わったようです。従って、既存のソースを、2.3以降でコンパイ ルする場 合には、以下のように修正する必要があります。

javax.media.opengl.* → com.jogamp.opengl.*
javax.media.nativewindow.* → com.jogamp.nativewindow.*

なお、Unix系のOS(OSXも含む)では以下のスクリプトにより、サブフォルダー内のソースを一括して変えられるはずです。 Windowsでの 方法は未調査です、すいません。
注意：このスクリプトのバグによりソースが失われる可能性もありますので、きちんとバックアップを取ってから実行してください。
このスクリプトの、'-i ".bak"'というパラメーターにより、"*.java.bak"というファイル名でバックアップを作成しますが、バックアップが不要なら削除してください。

```
grep -lr 'javax\.media\.opengl' --include="*.java" * | xargs sed -i ".bak" -e 's/javax\.media\.opengl/com\.jogamp\.opengl/g'
```

3.2JOGLの設定

jogl-all.jarとgluegen-rt.jarの２つのファイルを、クラスパスに設定します。

jogl-all.jarは、全てのクラスファイルを含むので、これさえ使っていれば必要なクラスが見つからないといったことにはならない のです が、各プラットフォームや、使いたいUI(AWT、Swing、NEWT)に応じて別のJARファイルも用意されているので、これらを使えば、プロ ジェクト全体を配布する場合に、ファイルサイズが小さくなる可能性があります。
また、実行時にはプラットフォーム依存のネイティブ・ライブラリも必要となるのですが、JOGLには実行環境を調査して、必 要なネイティブ・ライブラリを自動的にロードする仕組みが備わっているので、先ほどのJARファイルをクラスパスに設定しておけ ば、環境 変数PATHや、LD_LIBRARY_PATH、Eclipseでのネイティブ・ライブラリの場所などを設定する必要はありません。 この仕組みは、ディレクトリ構成がアーカイブファイルを解凍したときのままであることが前提条件のようですので、アーカイブファイルを解凍したらディレクトリ構成を変えな いようにします。

また、JDK/JREのextフォルダーにJOGL,GLUEGENのJARファイルを置くのは厳禁です。

3.3空のウィンドウを開く

いよいよプログラムの作成に入ります。ウィンドウを開くだけのプログラムは、JOGL+NEWTを使うとこんな風になります。このソースプ ログラム をFirstStepNewt.javaというファイル名で作成し、実行してみてください。

```
package demos.basic;

import com.jogamp.opengl.GL;
import com.jogamp.opengl.GLAutoDrawable;
```

```
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.Animator;

public class FirstStepNewt implements GLEventListener { //(1)

	public static void main(String[] args) {
		new FirstStepNewt();
	}

	public FirstStepNewt() {
		GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));//(2)
		GLWindow glWindow = GLWindow.create(caps); //(3)
		glWindow.setTitle("First demo (Newt)"); //(4)
		glWindow.setSize(300, 300); //(5)

		glWindow.addWindowListener(new WindowAdapter() { //(6)
			@Override
			public void windowDestroyed(WindowEvent evt) {
				System.exit(0);
			}
		});
		glWindow.addGLEventListener(this); //(7)

		Animator animator = new Animator(); //(8)
		animator.add(glWindow);
		animator.start();
		glWindow.setVisible(true); //(10)
	}

	@Override
	public void init(GLAutoDrawable drawable) {}

	@Override
	public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {}

	@Override
	public void display(GLAutoDrawable drawable) {}

	@Override
	public void dispose(GLAutoDrawable drawable) {
		if(animator != null) animator.stop();
	}
}
```

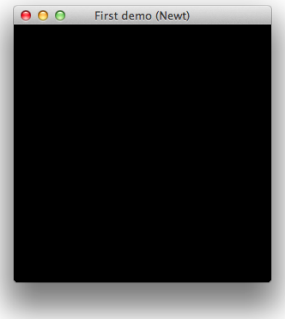
以下のような真っ黒なウィンドウが表示されます。

今度は、同じことをSwingを使ってやってみます。
このソースプログラムをFirstStepSwing.javaというファイル名で作成し、実行してみてください。

```
package demos.basic;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import com.jogamp.opengl.GL;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.opengl.awt.GLCanvas;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class FirstStepSwing implements GLEventListener { //(1)
	public static void main(String[] args) {
		SwingUtilities.invokeLater(new Runnable() {
			@Override
			public void run() {
				new FirstStepSwing();
			}
		});
	}

	public FirstStepSwing() {
		GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2)); //(2)
		JFrame frame = new JFrame(); //(3)
		frame.setTitle("First demo (Swing)"); //(4)

		frame.addWindowListener(new WindowAdapter() { //(6)
			@Override
			public void windowClosing(WindowEvent e) {
				System.exit(0);
			}
		});

		GLCanvas canvas = new GLCanvas(caps);
		canvas.setPreferredSize(new Dimension(300, 300)); //(5)

		canvas.addGLEventListener(this); //(7)

		frame.add(canvas, BorderLayout.CENTER);
```

```
                frame.setLocation(300, 300); //(9)
                frame.pack();
                frame.setVisible(true); //(10)
        }

        @Override
        public void init(GLAutoDrawable drawable) {}

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {}

        @Override
        public void display(GLAutoDrawable drawable) {}

        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(animator != null) animator.stop();
        }
}
```

これも同じように以下のような真っ黒なウィンドウが表示されます。



FirstStepSwing.javaについて、以下のように、GLCanvasの代わりにGLJPanelを使っても、同様に真っ暗なス クリーン になります。

```
import com.jogamp.opengl.awt.GLJPanel;
//GLCanvas canvas = new GLCanvas(caps); コメントアウト
GLJPanel canvas = new GLJPanel(caps);
```

GLCanvas、GLJPanelの使い分けは、jogamp フォーラムによると、特に問題ない限りGLCanvasを使い(こちらの方が高速)、以下のような状況で何かトラブルが起きた場 合は GLJPanelを使おうということのようです。

- JInternalFrameを使う。
- JOGLとJava2Dが同じJFrame上に混在している。

この文書では、GLCanvasを使うことにします。
ソースコードについて順を追って解説していきます。まず、FirstStepNewt.java、FirstStepSwing.javaの どちらに も、(1)の行にGLEventListenerというインターフェースが書かれています。これは以下のようなシグネチャとなっています。JOGLを 使うプログラマはこれを実装することにより、JOGLのパワーを引き出すことが出来るという訳です。
なお、C言語上のGLUTでは、

```
int main(int argc, char *argv[]) {
        glutInit(&argc, argv);
        glutCreateWindow(argv[0]);
        glutDisplayFunc(display);
        glutMainLoop();
        return 0;
}
```

の"glutDisplayFunc(display);"のように、関数名はプログラマが自分で決めて使いますが、JOGLの場合、メ ソッド名は 以下のように定義されているものを使うことになります。

```
public void init(GLAutoDrawable drawable);
public void reshape(GLAutoDrawable drawable, int x,int y, int width, int height);
public void display(GLAutoDrawable drawable);
public void dispose(GLAutoDrawable drawable);
```

いずれのメソッドでも、引数に、GLAutoDrawableクラスのインスタンスであるdrawableが渡されています。これについて は3.4 節で説明します。

init()は、OpenGLコンテキストの起動時に呼ばれますので、プログラマはここで一度しか行う必要のない初期化処理を記述します。 注意すべ きことは、下位のOpenGLコンテキストが破棄・再生成された場合にも呼ばれるため、２回以上呼ばれることがあるということです。(筆者も一度だけ 経験したことがあり、一度だけしか呼ばれないはずと不思議に思っていましたが、今回の執筆にあたりJavadocを再確認したところ、このよ うに書か れていましたので、納得しました)

dispose()は、OpenGLコンテキストが破棄された時に呼ばれますので、プログラマはここでリソースの解放などの処理を行いま す。こちら もinit()と同様、OpenGLコンテキストが破棄された場合に呼ばれますので、２回以上呼ばれることがあります。

reshape()は、アプリのウィンドウサイズが変更された場合に呼ばれます。プログラマは、必要に応じて、後述のビュー ポートや視錐台を更新します。

display()は、最も頻繁に呼ばれるメソッドです。プログラマは、ここで各種のプリミティブ(後述)に色を付けて描画したりします。 ゲームな どの動きのあるアプリケーションでは、スクリーン上に見えるキャラクターが滑らかに移動しているように見せかけるため、適切に処理する必要がありま す。

次に(2)の

```
GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2)); //(2)
```

については14章で解説します。

(3)から(6)についてまとめて説明します。NEWT版は

```
GLWindow glWindow = GLWindow.create(caps); //(3)
glWindow.setTitle("First demo (Newt)"); //(4)
glWindow.setSize(300, 300); //(5)
glWindow.addWindowListener(new WindowAdapter() { //(6)
        @Override
        public void windowDestroyed(WindowEvent evt) {
                System.exit(0);
        }
});
glWindow.addGLEventListener(this); //(7)
Animator animator = new Animator(); //(8)
```

```
animator.add(glWindow);
animator.start();
glWindow.setPosition(500, 500); //(9)
glWindow.setVisible(true); //(10)
```

のとおり、Swing版は、

```
JFrame frame = new JFrame(); //(3)
frame.setTitle("First demo (Swing)"); //(4)
frame.addWindowListener(new WindowAdapter() { //(6)
        @Override
        public void windowClosing(WindowEvent e) {
                System.exit(0);
        }
});
GLCanvas canvas = new GLCanvas(caps);
canvas.setPreferredSize(new Dimension(300, 300)); //(5)
canvas.addGLEventListener(this); //(7)
frame.add(canvas, BorderLayout.CENTER);
frame.setLocation(500, 500); //(9)
frame.pack();
frame.setVisible(true);//(10)
```

となっています。NEWT版では、Swing版とほぼ一対一に対応していることがわかると思います。NEWT版とSwing版を比較してみ ると、 NEWT版ではGLWindowがSwing版のJFrameとGLCanvasを兼ねたような役割を果たしていることがわかります。
Swingについては他にも良い解説がたくさんありますので、説明は割愛して、NEWTで使われるGLWindowクラスについて説明しま す。

(3)ではGLWindowのインスタンスを作成しています。(4)でタイトルバー上のタイトルを設定、(5)でウィンドウのサイズを設 定、(6) でタイトルバー上のクローズアイコンをクリックした時にアプリケーションを終了するように定義しています。(9)でウィンドウの位置を設定し、 (10)でウィンドウが見える状態に設定します。
(7)がJOGLアプリを作成する上で重要で、どのクラスがJOGLからのイベントを受け取るのかを定義しています。ここでは、 FirstStepNewtクラスとFirstStepSwingクラス自身が処理するように定義しています。今のところ、これらのメソッドは何もし ていないため、ウィンドウは真っ黒になるということです。
(8)はNEWT版だけにあります。(Animatorの解説は後で行います)

```
Animator animator = new Animator(); //(8)
animator.add(glWindow);
animator.start();
```

これがないとどうなるか、コメントアウトして試してみてください。ウィンドウが数秒間だけ表示され、すぐに終了することがわかります。
GLWindowについては、setAlwaysOnTop()で常に最前面に表示、setFullscreen(true)でフルスクリー ンを設定 できます。
この文書の冒頭で書いたとおり、NEWTはSwingのJFrameとは違い、ボタンやテキストフィールドなどを貼り付けるような使い方はで きません ので、これらの要素を使いたいならJFrameを、そうでなければNEWTを採用するのも一つの方法です。もちろんコントロールなしでもJFrame を使うという選択もありだと思います。

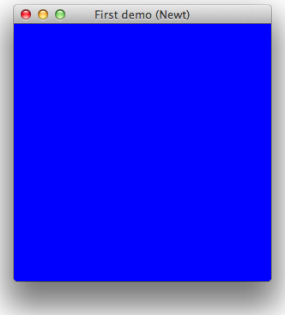
以降ではSwing版の解説は割愛し、NEWT版についてだけ書くことにします。

3.4ウィンドウを塗りつぶす

今まではdisplay()メソッドの中に何も記述していなかったので、真っ黒なウィンドウが表示されていました。そこで、今度は開いた ウィンドウ を塗りつぶしてみます。
FirstStepNewt.javaに以下の**太字**のところを追加し、もう一度プログラムを実行してみてください。

```
import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL2;
//(省略)
public class FirstStepNewt implements GLEventListener { //(1)

        public static void main(String[] args) {
                new FirstStepNewt();
        }

        public FirstStepNewt() {
                //変更なし
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();//追加
                //ウィンドウを青く塗りつぶす
                gl.glClearColor(0.0f, 0.0f, 1.0f, 1.0f);//追加
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {}

        @Override
        public void display(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();//追加
                gl.glClear(GL.GL_COLOR_BUFFER_BIT);//追加
        }

        @Override
        public void dispose(GLAutoDrawable drawable) {}
}
```

これを実行すると、以下のようなウィンドウが表示されます。



まず、init()メソッドでは以下を行っています。これは、display()メソッドでも同じです。

```
GL2 gl = drawable.getGL().getGL2();
```

これらのメソッドでは、引数として、GLAutoDrawableクラスのインスタンスであるdrawableが渡されています。JOGL では、 init()メソッド、display()メソッドだけでなく、reshape()、dispose()でも、上記のよう

にしてGLのインスタンスを 取得し、このインスタンスのメソッドを呼び出して必要な処理を行っていくことになります。

getGL().getGL2()のように呼び出しを連鎖しているのは、getGL()で得られるのはGLクラスのインスタンスであり、さ らに getGL2()を呼んで必要なGL2クラスのインスタンスを取得します。

OpenGLは、バージョンによって使える機能に違いがあり、JOGLではこの機能の相違を明確にするため、クラスを分けて定義しているよう です。詳 細は13章のProfileで説明します。
以下のクラスが定義されています。

- GL
- GL2
- GL3
- GL4

なお、これ以外にも、OpenGLESの機能を反映したGL2ES2などのクラスがありますが、この文書での説明は割愛します。

init()メソッド内では、続いて以下のメソッドを呼びます。

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

JOGLではこのメソッドの引数はfloat型ですので、"1.0"ではなく"1f"あるいは"1.0f" とする必要があります（"0.0"や"1.0"ではコンパイルエラーになります。"0"や"1"だけだと大丈夫です)

void glClearColor(float r, float g, float b, float a)
glClear(GL_COLOR_BUFFER_BIT)でウィンドウを塗りつぶす際の色を指定します。r、g、bはそれぞれ赤、 緑、青色 の成分の強さを示すfloat型の値で、0から1.0までの値を持ちます。1が最も明る く、この三つに(0f、0f、0f)を指定すれば黒色、 (1f、1f、1f)を指定すれば白色になります。上の例ではウィンドウは青色で 塗りつ ぶされます。最後のAはα値と呼ばれ、OpenGLでは不透明度として扱われます(0で透明、1.0fで不透明)。ここではとりあえず1.0fに しておいてください。

display()内ではGL2インスタンスを取得した後、以下を行っています。
glClearColor()は、プログラムの実行中に背景色を変更することがなければ、最初に一度だけ設定すれば十分です。そこでこのよう な初期化 処理は、init()内でまとめて行います。

```
gl.glClear(GL.GL_COLOR_BUFFER_BIT);
```

void glClear(int mask)
ウィンドウを塗りつぶします。maskには塗りつぶすバッファを指定します。OpenGLが管理する画面上のバッファ(メモリ)には、色を格納す るカラーバッファの他、隠面消去処理に使うデプスバッファ、凝ったことをするときに使うステンシルバッファ、カラーバッファの上に重ねて 表示され るオーバーレイバッファなど、いくつかのものがあり、これらが一つのウィンドウに重なって存在しています。

JOGLではGLクラスのスタティック変数としてこれらのマスク値が定義されているの で、"GL.GL_COLOR_BUFFER_BIT"のよう に指定する必要があります。(先頭の"GL"の後の"."に注意)
GLクラスをstatic importすれば以下のように"GL"というプレフィックス無しで、 GL_COLOR_BUFFER_BIT を指定することも可能です。C言語などから移植する場合はこちらの方が便利でしょう。

```
import static com.jogamp.opengl.GL.*; //"*"とすると、GLクラスで定義されている全てのstatic定数がインポートされる
 -
 -
gl.glClearBuffer(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT); //GLを省略可能
 -
```

サンプルコードのようにmaskにGL.GL_COLOR_BUFFER_BITを指定したときは、カラーバッファだけが塗りつぶされま す。デプス バッファをクリアするにはGL.GL_DEPTH_BUFFER_BITを、ステンシルバッファをクリアするには GL.GL_STENCIL_BUFFER_BITを指定します。以下のように、これらを'|'でまとめて指定することもできます。

```
gl.glClearBuffer(GL.GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

glFlush()
glFlush()はまだ実行されていないOpenGLの命令を全部実行します。OpenGLは関数呼び出しによって生成される OpenGL の命令をその都度実行するのではなく、いくつか溜め込んでおいてまとめて実行します。このため、ある程度命令が溜まらないと関数を呼び出しても実 行が開始されない場合があります。glFlush()はそういう状況でまだ実行されていない残りの命令の実行を開始します。ひんぱんにglFlush()を呼び出すと、かえって描画速度が低下します。

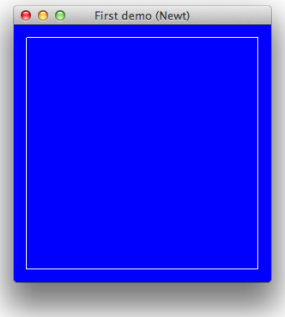
4.二次元図形を描く

4.1線を引く

ウィンドウ内に線を引いてみます。プログラムを以下のように変更し、実行してください。

```
import static com.jogamp.opengl.GL2.*;//追加
public class FirstStepNewt implements GLEventListener { //(1)

	public static void main(String[] args) {
		new FirstStepNewt();
	}

	public FirstStepNewt() {
		//変更なし
	}

	@Override
	public void init(GLAutoDrawable drawable) {
		//変更なし
	}

	@Override
	public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {}


	@Override
	public void display(GLAutoDrawable drawable) {
		GL2 gl = drawable.getGL().getGL2();
		gl.glClear(GL.GL_COLOR_BUFFER_BIT);
		//以下を追加
		gl.glBegin(GL_LINE_LOOP);
		gl.glVertex2f(-0.9f,-0.9f);
		gl.glVertex2f(0.9f, -0.9f);
		gl.glVertex2f(0.9f, 0.9f);
		gl.glVertex2f(-0.9f, 0.9f);
		gl.glEnd();
		//ここまでを追加
	}


	@Override
	public void dispose(GLAutoDrawable drawable) {}
}
```

以下のようウィンドウが表示されます。白い線が追加されていますね。



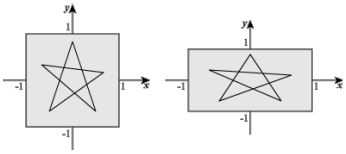
`void glBegin(int mode)`
: modeには4.2節で説明する図形のタイプを指定します。glBegin()と glEnd() の間で使えるメソッドは、glVertex, glColor, glNormal, glTexCoordなどに限定されています。詳しくはglBegin のリファレンスを参照してください。

`void glEnd()`
: 図形を描くには、glBegin()〜glEnd()の間にその図形の各頂点の座標値を設定するメソッドを置きます。

`void glVertex2f(float x,float y)`
: glVertex2f()は二次元の座標値を設定するのに使います。引数の型はfloatです。引数がdouble型のときは以下の glVertex2d()、int型のときはglVertex2i()を使います。通常は、float型で十分なケースが多いと思いま す。

`void glVertex2d(double x,double y)`
: glVertex2d()は引数の型がdoubleであることを除けば、glVertex2f()と同じです。

描かれる図形は、(-0.9、-0.9)と(0.9、0.9)の2点を対角線とする正方形です。これがウィンドウに対して「一回り小さく」 描かれま す.このウィンドウの大きさと図形の大きさの比率は、ウィンドウを拡大縮小しても変化しません。これはウィンドウのx軸とy軸の範囲が、ともに [-1、1]に固定されているからです。



4.2図形のタイプ

glBegin()の引数modeに指定できる図形のタイプには以下のようなものがあります。詳しくはglBegin のリファレンスを参照してください。

`GL_POINTS`
: 点を打ちます。

`GL_LINES`
: 2点を対にして、その間を直線で結びます。

`GL_LINE_STRIP`
: 折れ線を描きます。

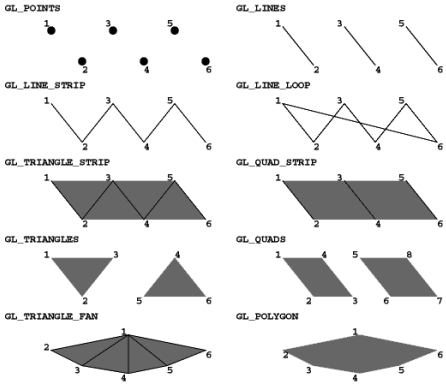
`GL_LINE_LOOP`
: 折れ線を描きます。始点と終点の間も結ばれます。

`GL_TRIANGLES/GL_QUADS`
: 3/4点を組にして、三角形/四角形を描きます。

`GL_TRIANGLE_STRIP/GL_QUAD_STRIP`
: 一辺を共有しながら帯状に三角形／四角形を描きます。

`GL_TRIANGLE_FAN`
: 一辺を共有しながら扇状に三角形を描きます。

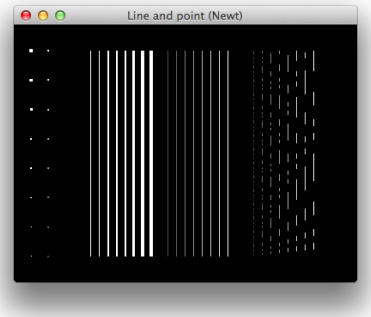
`GL_POLYGON`
: 凸多角形を描きます。



OpenGLを処理するハードウェアは、実際には三角形しか塗り潰すことができません(モノによっては四角形もできるものもあります)。こ のため GL_POLYGONの場合は、多角形を三角形に分割してから処理します。従って、もし描画速度が重要ならGL_TRIANGLE_STRIPや GL_TRIANGLE_FANを使うようプログラムを工夫してみてください。またGL_QUADSもGL_POLYGONより高速です。

4.3さまざまな点と線を描いてみる

これまでは単純な直線だけを描いてきましたが、さまざまな点や線を描画するにはどうしたらよいでしょうか。ここではまず実行結果を示して、 それを描 くにはどのようなプログラムにすればいいかを示すことにしま



す。
これを表示するためのプログラムは、次のとおりです。

```
package demos.basic;

import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.FPSAnimator;
import static com.jogamp.opengl.GL2.*;

public class LineAndPointNewt1 implements GLEventListener {
	public static void main(String[] args){
		new LineAndPointNewt1();
	}

	private float[] colors;
	private final short linePattern = 0b111100011001010; //破線のパターンを定義 (1)

	public LineAndPointNewt1() {
		initColors();
		GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
		GLWindow glWindow = GLWindow.create(caps);
		glWindow.setTitle("Line and point (Newt)");
		glWindow.setSize(400, 300);
		glWindow.addWindowListener(new WindowAdapter() {
			@Override
			public void windowDestroyed(WindowEvent evt) {
				System.exit(0);
			}
		});
		glWindow.addGLEventListener(this);
		FPSAnimator animator = new FPSAnimator(60);
		animator.add(glWindow);
		animator.start();
		glWindow.setVisible(true);
	}

	private void initColors() {
		colors = new float[8];
		for(int i = 0; i < 8; i++) {
			colors[i] = 0.3f + (0.1f * i);
		}
	}

	@Override
	public void display(GLAutoDrawable drawable) {
		final GL2 gl2 = drawable.getGL().getGL2();
		gl2.glClear(GL_COLOR_BUFFER_BIT);

		//大きさを変えて、点を描く。
		for(int i = 0; i < 8; i++) {
			gl2.glPointSize((i + 1) * 0.5f); //(2)
			gl2.glColor3f(1.0f, 1.0f, 1.0f);
			gl2.glBegin(GL_POINTS); //(3)
			gl2.glVertex2f(-0.9f, (i-7)*(1.6f/7f) + 0.8f);
			gl2.glEnd();
		}

		//灰色の濃度を変えて、点を描く。
		for(int i = 0; i < 8; i++) {
			gl2.glPointSize(2f);
			gl2.glBegin(GL_POINTS);
			gl2.glColor3f(colors[i], colors[i], colors[i]);
			//-0.8から+0.8の範囲になるよう計算
			gl2.glVertex2f(-0.8f, (i-7)*(1.6f/7f) + 0.8f);
			gl2.glEnd();
		}

		//太さを変えて、線を描く。
		for(int i = 1; i < 9; i++) {
			gl2.glLineWidth(i * 0.5f); //(4)
			gl2.glColor3f(1.0f, 1.0f, 1.0f);
			gl2.glBegin(GL_LINES);
			gl2.glVertex2f(-0.6f + i*0.05f, -0.8f);
			gl2.glVertex2f(-0.6f + i*0.05f, +0.8f);
			gl2.glEnd();
		}

		//灰色の濃度を変えて、線を描く。
		gl2.glLineWidth(1f);
		for(int i = 0; i < 8; i++) {
			gl2.glColor3f(colors[i], colors[i], colors[i]);
			gl2.glBegin(GL_LINES);
			gl2.glVertex2f(-0.1f + i*0.05f, -0.8f);
			gl2.glVertex2f(-0.1f + i*0.05f, +0.8f);
			gl2.glEnd();
		}

		//破線の色と、破線のスケールを変えて、線を描く。
		gl2.glEnable(GL_LINE_STIPPLE); //破線を描くことを設定 (5)
		for(int i = 0; i < 8; i++) {
			gl2.glLineStipple(i+1, linePattern); //(6)
			gl2.glColor3f(colors[i], colors[i], colors[i]);
```

```
                        gl2.glBegin(GL_LINES);
                        gl2.glVertex2f(+0.4f + i*0.05f, -0.8f);
                        gl2.glVertex2f(+0.4f + i*0.05f, +0.8f);
                        gl2.glEnd();
                }
                gl2.glDisable(GL_LINE_STIPPLE);//破線を描く設定を解除（7）
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                GL2 gl2 = drawable.getGL().getGL2();
                gl2.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) { }

        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(animator != null) animator.stop();
        }
}
```

プログラムについて解説します。(3)で、前項のGL_POINTSを使い、点を描画しますが、このときに(2)で指定した点の大きさが使 われま す。

void glPointSize(float size)
glPointSize()はこれから描画する点の大きさを指定します。引数の型はfloatです。何も指定しない場合は1.0fで す。0を 指定すると無視されます。次に別の値を設定するまで、同じ値が使われ続けます。
現在設定されている大きさを調べたい場合、例えば以下のようにして 調べることができます。

```
java.nio.DoubleBuffer glGetBuf = com.jogamp.common.nio.Buffers.newDirectfloatBuffer(bufSize);
gl2.glGetFloatv(paramType, glGetBuf);
glGetBuf.rewind();
System.out.print(paramName + ":");
for(int i = 0; i < bufSize; i++) {
        System.out.print(glGetBuf.get(i) + ", ");
}
System.out.println();
```

ここで、glGetFloatv()の引数のparamTypeはGLクラスで定義されているint型の定数です。glGetBufは java.nio.DoubleBuffer型のインスタンスで、これに結果が格納されます。bufSizeは結果を格納するために必要な要素数であ り、これが結果を格納するために必要な値より小さな場合、正常な結果が得られないようです。
点の大きさの場合、paramTypeはGL.GL_POINT_SIZEを、bufSizeは1を指定します。
paramTypeに指定できる値は、OpenGLの状態に応じた多数の定数があり、ここでは紹介しきれませんので、詳しくはglGet() 関数のリファレンスで確認してください。
上ではglGetFloatv()を使いましたが、この場合glGetDoublev()でも正常な結果が得られました。他にも、以下のよう に型に応 じたメソッドが用意されています。
なお、C/C++言語で用意されているglGetPointerv()は用意されていません。

```
void glGetBooleanv(int pname, java.nio.BooleanBuffer buf);
void glGetDoublev(int pname, java.nio.DoubleBuffer buf);
void glGetFloatv(int pname, java.nio.floatBuffer params);
void glGetIntegerv(int pname, java.nio.IntBuffer buf);
```

(4)について

void glLineWidth(float width)
glLineWidth()はこれから描画する線の幅を指定します。引数の型はfloatで、デフォルト(このメソッドを呼ばない) では 1.0fとなっています。0を指定すると無視されます。glPointSize()と同様に、次に別の値を設定するまで同じ値が使われ続けます。 現在設定されている大きさを調べたい場合、上記のglGetFloatv()にGL.GL_LINE_WIDTHと1を指定して呼び出し ます。

(5)について

void glEnable(int mode)
glEnable()はOpenGLの機能を有効化するために使われます。機能を無効にするglDisable(mode)とペアで 使われま す。modeには定義済みの定数を指定します。これも紹介しきれないほどたくさんの種類がありますので、興味があればglEnable() 関数のリファレンスをご覧ください。サンプルプログラムでは、gl.glBegin(GL_LINES)で直線を描くときに 破線を描 くよう、GL_LINE_STIPPLEを指定しています。

(7)について

void glDisable(int mode)
glEnable()で有効にした機能を無効にするために使われます。サンプルコードでは、GL_LINE_STIPPLEによる破 線の設定 を無効にしています。

(1)と(6)について

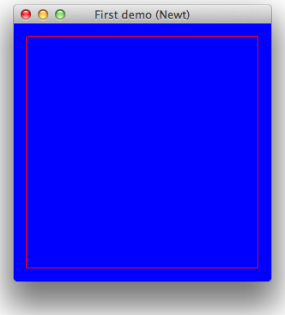
void glLineStipple(int factor, short pattern)
glLineStipple()のpatternはこれから描画する破線のパターンを２進数のパターンとして指定します。0は描かれ ず、1の ところだけが線が引かれます。short型ですので16bitのパターンを指定できます。
factorは描画する際の拡大率を指定します。どのよ うに描画されるかは、サンプルプログラムの実行結果で確認してください。

4.4線に色を付ける

線に色を付けてみます。4.1節のサンプルプログラムを以下のように変更し、実行してください。プログラムを実行したら線は何色で表示され たでしょ うか?

```
public class FirstStepNewt implements GLEventListener { //(1)
        public static void main(String[] args) {
                new FirstStepNewt();
        }

        public FirstStepNewt() {
                //変更なし
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                //変更なし
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {}

        @Override
        public void display(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                gl.glClear(GL.GL_COLOR_BUFFER_BIT);
                gl.glColor3f(1.0f, 0.0f, 0.0f); //この行を追加。1.0f, 0.0fのように'f'を付けていることに注意。
                gl.glBegin(GL_LINE_LOOP);
                gl.glVertex2f(-0.9f,-0.9f);
                gl.glVertex2f(0.9f, -0.9f);
                gl.glVertex2f(0.9f, 0.9f);
                gl.glVertex2f(-0.9f, 0.9f);
                gl.glEnd();
        }
        @Override
        public void dispose(GLAutoDrawable drawable) {
        }
}
```

以下のように線が赤くなりました。



void glColor3f(float r, float g, float b)

これから描画するものの色をRGBの色空間で指定します。glColor3fはfloat型を引数としますが、引数の型に応じて、以 下のメ ソッドが用意されています。

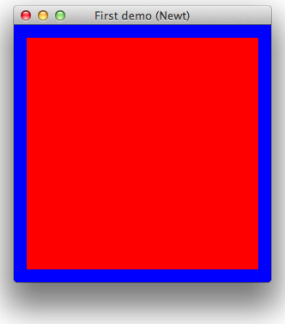
メソッド名と引数	値の範囲	備考
glColor3f(float r, float g, float b)	0から1.0まで	
glColor3d(double r, double g, double b)	0から1.0まで	
glColor3b(byte r, byte g, byte b)	0からByte.MAX_VALUE(127)まで	負の値は0と見なされる
glColor3s(short r, short g, short b)	0からShort.MAX_VALUE(32767)まで	負の値は0と見なされる
glColor3i(int r, int g, int b)	0からInteger.MAX_VALUE(2147483647)まで	負の値は0と見なされる
glColor3fv(FloatBuffer buf)	0から1.0まで	
glColor3dv(DoubleBuffer buf)	0から1.0まで	
glColor3bv(ByteBuffer buf)	0からByte.MAX_VALUE(127)まで	負の値は0と見なされる
glColor3sv(ShortBuffer buf)	0からShort.MAX_VALUE(32767)まで	負の値は0と見なされる
glColor3iv(IntBuffer buf)	0からInteger.MAX_VALUE(2147483647)まで	負の値は0と見なされる
glColor3fv(float[] array, int index)	0から1.0まで	
glColor3dv(double[] array, int index)	0から1.0まで	
glColor3bv(byte[] array, int index)	0からByte.MAX_VALUE(127)まで	indexは配列の添え字
glColor3sv(short[] array, int index)	0からShort.MAX_VALUE(32767)まで	indexは配列の添え字
glColor3iv(int[] array, int index)	0からInteger.MAX_VALUE(2147483647)まで	indexは配列の添え字

4.5図形を塗りつぶす

図形を塗りつぶしてみます。GL_LINE_LOOPをGL_POLYGONに変更しましょう。

```
import static com.jogamp.opengl.GL2.GL_POLYGON;//追加
//(省略)
public class FirstStepNewt implements GLEventListener { //(1)

	public static void main(String[] args) {
		new FirstStepNewt();
	}
	public FirstStepNewt() {
		//変更なし
	}
	@Override
	public void init(GLAutoDrawable drawable) {
		GL2 gl = drawable.getGL().getGL2();
		//ウィンドウを青く塗りつぶす。
		gl.glClearColor(0f, 0f, 1.0f, 1.0f);
	}
	@Override
	public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {}
	@Override
	public void display(GLAutoDrawable drawable) {
		GL2 gl = drawable.getGL().getGL2();
		gl.glClear(GL.GL_COLOR_BUFFER_BIT);
		gl.glColor3f(1.0f, 0.0f, 0.0f);
		gl.glBegin(GL_POLYGON);//変更
		gl.glVertex2f(-0.9f,-0.9f);
		gl.glVertex2f(0.9f, -0.9f);
		gl.glVertex2f(0.9f, 0.9f);
		gl.glVertex2f(-0.9f, 0.9f);
		gl.glEnd();
	}

	@Override
	public void dispose(GLAutoDrawable drawable) {}
}
```

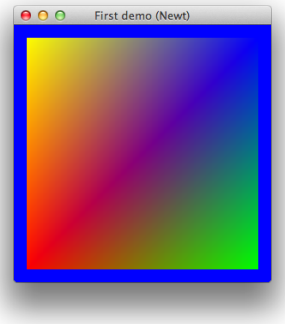
このようになりました。



色は頂点毎に指定することもできます。glBegin()の前のglColor3f()を消して、かわりに四つのglVertex2f() の前に glColor3f()を置きます。サンプルプログラムを以下のように変更してください。プログラムを実行すると、どういう色の付き方になったでしょ うか。

```
@Override
public void display(GLAutoDrawable drawable) {
        GL2 gl = drawable.getGL().getGL2();
        gl.glClear(GL.GL_COLOR_BUFFER_BIT);
        //gl.glColor3f(1.0f, 0.0f, 0.0f); //ここは削除

        gl.glBegin(GL_POLYGON);
        gl.glColor3f(1.0f, 0.0f, 0.0f); // 赤
        gl.glVertex2f(-0.9f,-0.9f);
        gl.glColor3f(0.0f, 1.0f, 0.0f); // 緑
        gl.glVertex2f(0.9f, -0.9f);
        gl.glColor3f(0.0f, 0.0f, 1.0f); // 青
        gl.glVertex2f(0.9f, 0.9f);
        gl.glColor3f(1.0f, 1.0f, 0.0f); // 黄
        gl.glVertex2f(-0.9f, 0.9f);
        gl.glEnd();
}
```

以下のように、多角形の内部は頂点の色から補間した色で塗りつぶされたと思います。



この段階でのサンプルプログラムは以下のとおりになっているはずです。

```
package demos.basic;

import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.FPSAnimator;
import static com.jogamp.opengl.GL2.*;

public class FirstStepNewt implements GLEventListener {
        public static void main(String[] args){
                new FirstStepNewt();
        }

        public FirstStepNewt() {
                GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
                GLWindow glWindow = GLWindow.create(caps);
                glWindow.setTitle("First demo (Newt)");
                glWindow.setSize(300, 300);
                glWindow.addWindowListener(new WindowAdapter() {
                        @Override
                        public void windowDestroyed(WindowEvent evt) {
                                System.exit(0);
                        }
                });
                glWindow.addGLEventListener(new FirstStepNewt());
                FPSAnimator animator = new FPSAnimator(10); //(2)
                animator.add(glWindow);
                animator.start();
                glWindow.setVisible(true);
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                //ウィンドウを青く塗りつぶす。
                gl.glClearColor(0f, 0f, 1f, 1.0f);
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) { }
```

```
        @Override
        public void display(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                gl.glClear(GL.GL_COLOR_BUFFER_BIT);
                //gl.glColor3f(1.0f, 0.0f, 0.0f); //ここは削除
                gl.glBegin(GL_POLYGON);
                gl.glColor3f(1.0f, 0.0f, 0.0f); // 赤
                gl.glVertex2f(-0.9f,-0.9f);
                gl.glColor3f(0.0f, 1.0f, 0.0f); // 緑
                gl.glVertex2f(0.9f, -0.9f);
                gl.glColor3f(0.0f, 0.0f, 1.0f); // 青
                gl.glVertex2f(0.9f, 0.9f);
                gl.glColor3f(1.0f, 1.0f, 0.0f); // 黄
                gl.glVertex2f(-0.9f, 0.9f);
                gl.glEnd();
        }

        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(animator != null) animator.stop();
        }
}
```

5.座標系とマウス・キーボードによる操作

5.1座標軸とビューポート

ウィンドウ内に表示する図形の座標軸は、そのウィンドウ自体の大きさと図形表示を行う"空間"との関係で決定します。開いたウィンドウの位 置や大き さはマウスを使って変更することができますが、その情報はウィンドウマネージャを通じて、イベントとしてプログラムに伝えられます。
これまでのプログラムでは、ウィンドウのサイズを変更すると、表示内容もそれにつれて拡大縮小していました。これを、表示内容の大きさを変え ずに、表 示領域のみを広げるようにしてみましょう。

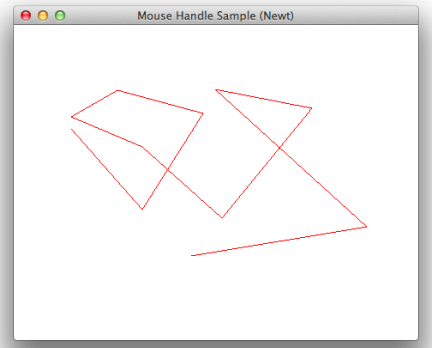
5.2マウスボタンのクリックと、マウスイベント

マウスのボタンが押されたことを知るには、GLWindowのインスタンスに対し、addMouseListener(リスナーインスタン ス)によ りイベントリスナーを設定します。
以下のソースプログラムをNewtMouseHandleSample.javaというファイル名で作成し、実行してみてください。

```
package demos.basic;

import java.awt.geom.Point2D;
import java.awt.geom.Point2D.Float;
import java.util.ArrayList;
import java.util.List;
import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.MouseEvent;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.FPSAnimator;
import static com.jogamp.opengl.GL2.*;

public class NewtMouseHandleSample implements GLEventListener, com.jogamp.newt.event.MouseListener { //(1)

        public static void main(String[] args) {
                new NewtMouseHandleSample();
        }

        private final List points;

        public NewtMouseHandleSample() {
                points = new ArrayList<>();

                GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
                final GLWindow glWindow = GLWindow.create(caps);
                glWindow.setTitle("Mouse Handle Sample (Newt)");
                glWindow.setSize(300, 300); //(2)

                glWindow.addWindowListener(new WindowAdapter() {
                        @Override
                        public void windowDestroyed(WindowEvent evt) {
                                System.exit(0);
                        }
                });

                glWindow.addGLEventListener(this);
                glWindow.addMouseListener(this); //(3)
                FPSAnimator animator = new FPSAnimator(30); //(4)
                animator.add(glWindow);
                animator.start();

                glWindow.setPosition(500, 500);
                glWindow.setVisible(true);
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                GL gl = drawable.getGL();
                //背景を白く塗りつぶす。
                gl.glClearColor(1f, 1f, 1f, 1.0f);
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x,int y, int width, int height) {
                GL2 gl = drawable.getGL().getGL2();
                //gl.glViewport(x, y, width, height); //(5)Jogl内部で実行済みなので不要。
                gl.glMatrixMode(GL_PROJECTION); //(6)透視変換行列を指定
                gl.glLoadIdentity();              //(7)透視変換行列を単位行列にする
                System.out.printf("x:%d, y:%d, w:%d, h:%d, %n", x, y, width, height);
                //これによりウィンドウをリサイズしても中の図形は大きさが維持される。
                //また、第3、第4引数を入れ替えることによりGLWindowの座標系(左上隅が原点)とデバイス座標系(左下隅が原点)の違いを吸収している。
                gl.glOrthof(x, x + width, y + height, y, -1.0f, 1.0f); //(8)

                gl.glMatrixMode(GL_MODELVIEW); //(9)モデルビュー変換行列を指定
                gl.glLoadIdentity(); //(10)モデルビュー変換行列を単位行列にする
        }

        @Override
        public void display(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                gl.glClear(GL_COLOR_BUFFER_BIT);
                gl.glColor3f(1.0f, 0.0f,0.0f); // 赤
```

```
                gl.glBegin(GL_LINES);

                //p1のところで+1しているので、iが範囲を超えないようループ回数を一つ減らしている。
                for(int i = 0; i < points.size() - 1; i++) { //(11)
                        Point2D.Float p0 = (Float) points.get(i);
                        Point2D.Float p1 = (Float) points.get(i + 1);
                        gl.glVertex2d(p0.getX(), p0.getY()); // 今の位置
                        gl.glVertex2d(p1.getX(), p1.getY()); // 次の位置
                }
                gl.glEnd();
        }

        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(animator != null) animator.stop();
        }

        //ここから下で、com.jogamp.newt.event.MouseListenerインターフェースのメソッドを実装。 (12)

        @Override
        public void mouseClicked(com.jogamp.newt.event.MouseEvent e) {
                System.out.printf("%d, %d%n", e.getX(), e.getY());
                points.add(new Point2D.Float(e.getX(), e.getY())); //(13)
        }

        @Override
        public void mouseEntered(com.jogamp.newt.event.MouseEvent e) { }

        @Override
        public void mouseExited(com.jogamp.newt.event.MouseEvent e) { }

        @Override
        public void mousePressed(com.jogamp.newt.event.MouseEvent e) { }

        @Override
        public void mouseReleased(com.jogamp.newt.event.MouseEvent e) { }

        @Override
        public void mouseMoved(com.jogamp.newt.event.MouseEvent e) { }

        @Override
        public void mouseDragged(com.jogamp.newt.event.MouseEvent e) { }

        @Override
        public void mouseWheelMoved(com.jogamp.newt.event.MouseEvent e) { }
}
```

これを実行し、ウィンドウ内で適当にクリックしてみてください。次のように、クリックした点が結ばれて表示されます。また、ウィンドウのサ イズを変 えても中の図形の大きさは維持されていることがわかると思います。



(3)でGLWindowのインスタンスに対しキーボードのイベントリスナーを登録していますが、ここでは NewtMouseHandleSampleクラスのインスタンス自身をthisとして登録しています。そこで(1)のようにリスナーを実装している ことを宣言し、(12)以降で必要なメソッドを記述しています。

(13)でマウスがクリックされた位置をList pointsに 保存し、 display()メソッドの中で使っています。

(1)のcom.jogamp.newt.event.MouseListenerは以下のメソッドを持つインターフェースです。 Swingの MouseListener,MouseMotionListener,MouseWheelListenerのメソッドを全て併せ持つインター フェースになっています。マウスがクリックされたり、ドラッグされたり、マウスをクリックせずに動かしたとき、マウスホイールが回転されたと きなどに 呼ばれます。

```
@Override
public void mouseClicked(com.jogamp.newt.event.MouseEvent e);
@Override
public void mouseEntered(com.jogamp.newt.event.MouseEvent e);
@Override
public void mouseExited(com.jogamp.newt.event.MouseEvent e);
@Override
public void mousePressed(com.jogamp.newt.event.MouseEvent e);
@Override
public void mouseReleased(com.jogamp.newt.event.MouseEvent e);
@Override
public void mouseMoved(com.jogamp.newt.event.MouseEvent e);
@Override
public void mouseDragged(com.jogamp.newt.event.MouseEvent e);
@Override
public void mouseWheelMoved(com.jogamp.newt.event.MouseEvent e);
```

mouseClicked()、mousePressed()、mouseReleased()は、 java.awt.event.MouseEventクラスと同様、ボタンを押すとmousePressed()が呼ばれ、離すと mouseReleased()の後、mouseClicked()の順で呼ばれます。

ところで、サンプルプログラムの後ろのほうに何もしないメソッドが並んでいますが、Swingではこのような場合 java.awt.event.MouseAdapterクラスを使い必要なメソッドだけを実装します。
NEWTでも同様のcom.jogamp.newt.event.MouseAdapterクラスが用意されていますので、以下のように短く すること ができます。

```
public class NewtMouseHandleSample implements GLEventListener {// com.jogamp.newt.event.MouseListener を削除
        //glWindow.addMouseListener(this);//(3)削除
        glWindow.addMouseListener(new com.jogamp.newt.event.MouseAdapter() {
                @Override
                public void mouseClicked(com.jogamp.newt.event.MouseEvent e) {
                        System.out.printf("%d, %d%n", e.getX(), e.getY());
                        points.add(new Point2D.Float(e.getX(), e.getY()));
```

```
                }
        });
        //(省略)
        //ここから下で、com.jogamp.newt.event.MouseListenerインターフェースのメソッドを実装。 (12)
        //これ以降のMouseListenerインターフェースを実装していた部分を削除
        // @Override
        // public void mouseClicked(com.jogamp.newt.event.MouseEvent e) { //(13)
                // System.out.printf("%d, %d%n", e.getX(), e.getY());
                // points.add(new Point2D.Float(e.getX(), e.getY()));
        // }
        //
        // @Override
        // public void mouseReleased(com.jogamp.newt.event.MouseEvent e) {
                // System.out.println("mouse released");
        // }
        // @Override
        // public void mouseEntered(com.jogamp.newt.event.MouseEvent e) {}
        //
        // @Override
        // public void mouseExited(com.jogamp.newt.event.MouseEvent e) {}
        //
        // @Override
        // public void mousePressed(com.jogamp.newt.event.MouseEvent e) {}
        //
        // @Override
        // public void mouseMoved(com.jogamp.newt.event.MouseEvent e) { }
        //
        // @Override
        // public void mouseDragged(com.jogamp.newt.event.MouseEvent e) { }
        //
        // @Override
        // public void mouseWheelMoved(com.jogamp.newt.event.MouseEvent e) {}
```

com.jogamp.newt.event.MouseEventはマウスが操作されたときの情報が格納されています。以下の主要なメ ソッドがあ ります。 メソッドは他にもありますので、詳しくはAPI docをご覧 ください。プレッシャーや３次元 デバイスの操作にも対応しているようです。

```
int getX();
int getY();
short getClickCount();

short getButton();
int getModifiers();
float[] getRotation();

float getRotationScale();

float[] getRotationXYZ(float rotationXorY, int mods);

float getMaxPressure();

float[] getAllPressures();

float getPressure(boolean normalized);

float getPressure(int index, boolean normalized);
```

getX()、getY()で得られる座標は、GLWindowの**左上隅(タイトルバーを除く)を原点(0,0)**と した画面上の画素の位置になります。**デバイス座標系とは上下が反転している**ので気をつけてください。な お、 Swingの場合、GLCanvas,GLJPanelの各コンポーネントの左上隅が原点になります。

ダブルクリックはgetClickCount()で調べられます。
どのボタンが押されたのかを調べるには、Swingと同様、getButton()を使います。MouseEventクラスで定義されている 以下の定 数が返されます。なお、これはBUTTON_9まで定義されているので、３ボタンマウスだけでなく、ゲームパッドのようなものも想定しているのだと思 われます。

- BUTTON1:左クリック
- BUTTON2:中クリック
- BUTTON3:右クリック

なお、SwingUtilities.isRightMouseButton()などに相当するメソッドは用意されていません。

キーボードの特殊キー(シフトキー、コントロールキー、WindowsのWindowsキー、OS XのコマンドキーとOptionキー)を押しながらマウスをクリックした場合、以下の通り検出できるようになっています。

getModifiers()はcom.jogamp.newt.event.InputEventで定義されている定数 SHIFT_MASK,CTRL_MASK,ALT_MASK,META_MASK,ALT_GRAPH_MASKなどを返します。他にも BUTTON1_MASKからBUTTON9_MASKが定義されているので、これもゲームパッドへの対応が考慮されているのかもしれませ ん。

getModifiers()の他に、特殊キーの状態を直接得るための、以下のメソッドも用意されています。

```
public final boolean isAltDown();

public final boolean isAltGraphDown();

public final boolean isControlDown();

public final boolean isMetaDown();

public final boolean isShiftDown();
```

以下のようにサンプルソースを変えて動かしてみると、動きが理解出来ると思います。

```
glWindow.addMouseListener(new com.jogamp.newt.event.MouseAdapter() {
        @Override
        public void mouseClicked(com.jogamp.newt.event.MouseEvent e) { //(4)
                System.out.printf("%d, %d%n", e.getX(), e.getY());
                points.add(new Point2D.Float(e.getX(), e.getY()));
                System.out.println("mouse clicked count:" + e.getClickCount());
                System.out.println("mouse source :" + e.getButton());
                System.out.println("mouse button 1:" + MouseEvent.BUTTON1);
                System.out.println("mouse button 2 :" + MouseEvent.BUTTON2);
                System.out.println("mouse button 3 :" + MouseEvent.BUTTON3);
        }

        @Override
        public void mouseReleased(com.jogamp.newt.event.MouseEvent e) {
                System.out.println("mouse released");
        }
});
```

本節の冒頭のサンプルコードの(4)から(9)について説明します。以下に再掲します。

```
@Override
public void reshape(GLAutoDrawable drawable, int x,int y, int width, int height) {
        GL2 gl = drawable.getGL().getGL2();
        //gl.glViewport(x, y, width, height); //(5)Jogl内部で実行済みなので不要
        gl.glMatrixMode(GL_PROJECTION); //(6)透視変換行列を指定
        gl.glLoadIdentity();              //(7)透視変換行列を単位行列にする
        System.out.printf("x:%d, y:%d, w:%d, h:%d, %n", x, y, width, height);
```

```
        //これによりウィンドウをリサイズしても中の図形は大きさが維持される。
        //また、第3、第4引数を入れ替えることによりGLWindowの座標系(左上隅が原点)とデバイス座標系(左下隅が原点)の違いを吸収している。
        gl.glOrthof(x, x + width, y + height, y, -1.0f, 1.0f); //(8)

        gl.glMatrixMode(GL_MODELVIEW); //(9)モデルビュー変換行列を指定
        gl.glLoadIdentity(); //(10)モデルビュー変換行列を単位行列にする
}
```

void glViewport(int x,int y,int width,int height)
ビューポートを設定します。ビューポートとは、開いたウィンドウの中で、実際に描画が行われる領域のことをいいます。正規化デバイス 座標系の 2点(-1,-1)、(1,1)を結ぶ線分を対角線とする矩形領域がここに表示されます。最初の二つの引数x、yにはその領域の左下隅の位置、 widthには幅、heightには高さをデバイス座標系での値、すなわちディスプレイ上の画素数で指定します。関数reshape() の引数 width、heightにはそれぞれウィンドウの幅と高さが入っていますから、glViewport(0,0,width,height)はリ サイズ後のウィンドウの全面を表示領域に使うことになります。

なお、C言語などではglViewport()が書かれている場合が多いようですが、JOGLの場合、reshape()が呼ばれる前に内部で実 行されて いますので、別のパラメータを設定する必要がない限り、プログラマが明示的に行う必要はありません。(reshape()のJavaDocに以下のように 記載されています。)

```
For efficiency the GL viewport has already been updated via glViewport(x, y, width, height) when this method is called.
```

void glMatrixMode(int mode)
操作の対象とする変換行列を指定します。modeにはGL_MODELVIEW、GL_PROJECTIONを指定します。他にもあ ります が、割愛します。

(4)の以下のコードは、後のアニメーションのところで説明します。

```
FPSAnimator animator = new FPSAnimator(30); //(4)
```

5.3 モデリング座標とビューイング変換について

座標変換のプロセスは、

1. 図形の空間中での位置を決める「モデリング変換」
2. その空間を視点から見た空間に直す「ビューイング(視野)変換」
3. その空間をコンピュータ内の空間にあるスクリーンに投影する「透視変換」
4. スクリーン上の図形をディスプレイ上の表示領域に切り出す「ビューポート変換」

という四つのステップで行われます。これまではこれらを区別せずに取り扱ってきました。すなわち、これらの投影を行う行列式を掛け合わせる ことで、 単一の行列式として取り扱ってきたのです。

しかし、図形だけを動かす場合は、モデリング変換の行列だけを変更すればいいことになります。また、後で述べる陰 影 付けは、透視変換を行う前の座標系で計算する必要があります。

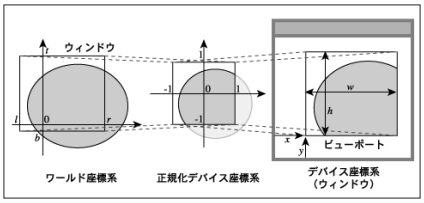
そこでOpenGLでは、「モデリング変換－ビューイング変換」の変換行列(モデルビュー変換行列)と、「透視変換」の変換行列を独立して 取り扱う 手段が提供されています。モデルビュー変換行列を設定する場合はglMatrixMode(GL_MODELVIEW)、透視変換行列を設定する場合 はglMatrixMode(GL_PROJECTION)を実行します。

カメラの画角などのパラメータを変更しない場合、ウィンドウを開いたときに一回だけ透視変換行列を設定すればよいので、これは reshape()の 中で設定すればよいでしょう。あとは全てモデリグ－ビューイング変換行列に対する操作なので、透視変換行列を設定した直後に、 gl.glMatrixMode(GL_MODELVIEW)を実行します。

void glLoadIdemtity()
これは対象としている変換行列を初期化します(単位行列にする)。座標変換の合成は行列の積で表されますから、変換行列には初期値と して単位 行列を設定しておきます。

void glOrthof(float left,float right,float bottom,float top,float near,float far)
glOrtho()はワールド座標系を正規化デバイス座標系に平行投影(orthographicprojection:正射影)す る行列を 変換行列に乗じます。引数には左から、leftに表示領域の左端の位置、rightに右端の位置、bottomに下端の位置、topに上端の位 置、nearに前方面の位置、farに後方面の位置を指定します。これは、ビューポートに表示される空間の座標軸を設定します。なお、引 数の型が 全てdoubleとなる、glOrthod()という関数もあります。以下の説明では総称してglOrtho*()と記します。

reshape()の処理によって、プログラムはglViewport()で指定した領域にglOrtho*()で指定した領域内の図形を 表示する ようになります。ここでglOrtho*()で指定する領域の大きさをビューポートの大きさに比例するように設定すれば、表示内容の大きさをビュー ポートの大きさにかかわらず一定に保つことができます。ここでビューポートの大きさは開いたウィンドウの大きさと一致させていますから、ウィ ンドウの リサイズしても表示内容の大きさを一定に保つことができます。



図形はワールド座標系と呼ばれる空間にあり、その2点(l,b),(r,t)を結ぶ線分を対角線とする矩形領域を、2点(-1,-1), (1,1) を対角線とする矩形領域に投影します。この投影された座標系を正規化デバイス座標系(あるいはクリッピング座標系)と呼びます。

この正規化デバイス座標系の正方形領域内の図形がデバイス座標系(ディスプレイ上の表示領域の座標系)のビューポートに表示されますから、 結果的に ワールド座標系からglOrtho*()で指定した矩形領域を切り取ってビューポートに表示することになります。

ワールド座標系から切り取る領域は、"ＣＧ用語"的には「ウィンドウ」と呼ばれ、ワールド座標系から正規化デバイス座標系への変換は「ウィ ンドウイ ング変換」と呼ばれます。しかしウィンドウシステム(MS Windowsや、Unix系OSで使われるX Window System等)においては、「ウィンドウ」はアプリケーションプログラムがディスプレイ上に作成する表示領域のことを指すので、ここの説明ではこれを「座標軸」と呼んで います。なお、正規化デバイス座標系からデバイス座標系への変換はビューポート変換と呼ばれます。

glOrtho*()では引数としてleft、right、top、bottomの他にnearとfarも指定する必要があります。実は OpenGLは二次元図形の表示においても内部的に三次元の処理を行っており、ワールド座標系は奥行き(Z)方向にも軸を持つ三次元空間に なっていま す。nearとfarには、それぞれこの空間の前方面(可視範囲の手前側の限界)と後方面(可視範囲の遠方の限界)を指定します。nearより手前に ある面やfarより遠方にある面は表示されません。

二次元図形は奥行き(Z)方向が0の三次元図形として取り扱われるので、ここではnear(前方面、可視範囲の手前の位置)を-1.0、 far(後 方面、遠方の位置)を1.0にしています。

glOrtho*()を使用しなければ変換行列は単位行列のままなので、ワールド座標系と正規化デバイス座標系は一致し、ワールド座標系の 2点 (-1,-1)、(1,1)を対角線とする矩形領域がビューポートに表示されます。ビューポート内に表示する空間の座標軸が変化しないため、この状態 でウィンドウのサイズを変化させると、それに応じて表示される図形のサイズも変わります。

表示図形のサイズをビューポートの大きさにかかわらず一定にするには、glOrtho*()で指定するの領域の大きさをビューポートの大き さに比例 するように設定します。例えばワールド座標系の座標軸が上記と同様にleft、right、top、bottom、near、farで与えられてお り、もともとのウィンドウの大きさがWxH、リサイズ後のウィンドウの大きさがwxhなら、glOrtho* (left*w/W,right*w /W,bottom*h/H,top*h/H,near,far)とします。これまでのプログラムでは、ワールド座標系の2点(-1,-1)、 (1,1)を対角線とする矩形領域を300x300の大きさのウィンドウに表示した時の表示内容の大きさが常に保たれるよう設定しています。

マウスの位置をもとに図形を描く場合は、マウスの位置からウィンドウ上の座標値を求めなければなりません。このサンプルプログラムでは ちょっと工夫 して、ワールド座標系がこのマウスの座標系に一致するよう、また同時にウィンドウの上下も反転するよう、glOrthof()を設定しています。

5.4 座標変換について

コンピューターグラフィックスの世界では、ある物体が世界のどこにあって、どちらを向いているかが非常に重要になります。どこにあるかを決めるためには、基準となる原点を 定めて、
そこからどれだけ離れているかを定める必要があります。原点は任意に定めても構いませんが、一度決めたらこれを統一して使わないと混乱するこ とになり ます。

例えば、地球上では、グリニッジ天文台を通る子午線を経度０とし、赤道を緯度０とするようなものです。

物体の姿勢については、物体の特徴に合ったベクトルがどちらを向いているかにより表すことになります。このときに、基準となる、世界に対し不 同の 基準方向が必要になります。
先の例えでは、地上にいる人の姿勢を表すために、へそが向いている方向を人体の基準軸とし、北方向を方位角０度、地平線を向いている場合を仰 角０ 度とするようなものです。

緯度・経度は、地上にいる人の位置を表現するには十分ですが、例えばロケットに乗って地球を離れている人の位置を表すことはできませんので、 この ような状況では、適切な座標系を選び直す必要があることに注意しないといけません。

OpenGLの内部では、変換行列の拡大縮小(scale)、回転(rotate)、平行移動(translate)を、4行×4列の行列を 使って表 現して います。

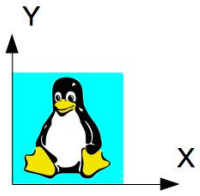
$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

変換行列は、概念としては4行x4列の行列ですが、OpenGLで扱う実際のデータは、16個の要素からなる一次元配列となっています。行列 と一次元 配列との対応は、この図のアルファベット順のように、まず一列目の1行目から4行目までを使い、次に２列目以降を使うようになっています。これを column majorと呼びます。これとは対照的に、Java3DやDirect3Dは、row majorといって、横方向を優先してたどっていくようになっています。

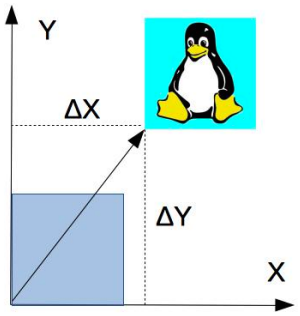
ここで左辺の（x', y', z', w')は変換後の座標系、右辺の(x, y, z, w)が変換前の座標系です。3次元の変換なのに一つ次元を増やしているのは、これにより拡大縮小・回転による座標変換と、平行移動による座標変換を統一した形で表すことが 出来て、都合がよいからです。d, h, l, pから成る追加した行を、同次座標と呼びます。

平行移動

以下のような図形が、変換前の座標系にあるとします。



これをΔX、ΔYだけ平行移動することを考えます。



これを表現するプログラムコードはこのようになります。最後の引数ΔZは、図では省略しています。

```
gl.glTranslatef(ΔX, ΔY, ΔZ);
```

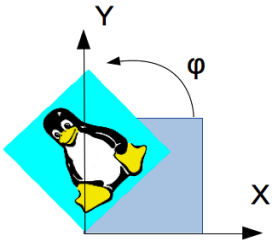
これを表現する行列式はこのようになります。

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \Delta X \\ 0 & 1 & 0 & \Delta Y \\ 0 & 0 & 1 & \Delta Z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

変換行列の赤い文字の部分だけが使われていることがわかります。この赤字の部分を transform成分と呼ぶことがあります。

回転

元の図形をZ軸を中心として、反時計回りにφだけ回転してみます。



これを表現するプログラムコードはこのようになります。

```
gl.glRotatef(φ, 0, 0, 1);    //z軸による回転
```

ここで、最初の引数φの単位は度(degree)になります。Java.lang.Mathパッケージのsin(), cos(), atan2()などの角度の単位はラジアンなので、注意が必要です。回転の方向は、OpenGLでは上図 のとおり反時計回りが正と定義されています。２番から４番目までの引数で、回転の軸となるベクトルを表現しています。上記の場合Ｚ軸になります。
これを表現する行列式はこのようになります。

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

X軸、Y軸が軸となる回転は、それぞれ以下のようになります。

```
gl.glRotatef(θ, 1, 0, 0);    //X軸による回転
```

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

```
gl.glRotatef(ψ, 0, 1, 0);    //Y軸による回転。
```

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \cos\psi & 0 & \sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

これを見ると、青色の3x3の部分が変化していることがわかります。この部分をscaleRotate成分と呼ぶことがあるようです。

上記の回転は、X軸、Y軸、あるいはZ軸だけを軸としていましたが、一般には、任意の角度の姿勢を表したいですよね。このような任意の姿勢を表す には、以下の方法があります。いずれも回転の中心が原点と一致しているという前提があります。

(1)オイラー角による回転

Wikipediaに わかりやすい解説があります。３つのパラメーターで表されます。回転軸そのものが動くことがポイントです。

オイラー角による回転を表す行列は以下のようになります。まずβとγを掛けてから、αを掛けています。
出典

- http://www6.ocn.ne.jp/~simuphys/daen1-1.html
- http://irobutsu.a.la9.jp/mybook/ykwkrAM/sim/EulerAngle.html

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & \sin\beta & 0 \\ 0 & -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} \cos\gamma & \sin\gamma\cos\beta & \sin\gamma\sin\beta & 1 \\ -\sin\gamma & \cos\gamma\cos\beta & \cos\gamma\sin\beta & 0 \\ 0 & -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} \cos\gamma\cos\beta-\cos\beta\sin\alpha\sin\gamma & \cos\gamma\sin\alpha+\cos\beta\cos\alpha\sin\gamma & \sin\gamma\sin\beta & 0 \\ -\sin\gamma\cos\alpha-\cos\beta\sin\alpha\cos\gamma & -\sin\gamma\sin\alpha+\cos\beta\cos\alpha\cos\gamma & \cos\gamma\sin\beta & 0 \\ \sin\beta\sin\alpha & -\sin\beta\cos\alpha & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

(2)ピッチ、ロール、ヨーによる回転

ピッチ角(φ)、ロール角(θ)、ヨー角(ψ)の３つのパラメーターで表されます。
飛行機を例にとると、ピッチが左右の翼を結ぶ軸を中心とした回転、ロールが機首と尾翼を結ぶ線を中心軸とした回転、ヨーが機首の東西南北の向きに なります。
これを表す行列は以下のようになります。まずφとθを掛けてから、ψを掛けています。

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ \sin\varphi\sin\theta & \cos\varphi & -\sin\varphi\cos\theta & 0 \\ -\cos\varphi\sin\theta & \sin\varphi & \cos\varphi\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} \cos\theta\cos\psi & -\cos\theta\sin\psi & \sin\theta & 0 \\ \sin\varphi\sin\theta\cos\psi+\cos\varphi\sin\psi & -\sin\varphi\sin\theta\sin\psi+\cos\varphi\cos\psi & -\sin\varphi\cos\theta & 0 \\ -\cos\varphi\sin\theta\cos\psi+\sin\varphi\sin\psi & \cos\varphi\sin\theta\sin\psi+-\sin\varphi\cos\psi & \cos\varphi\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

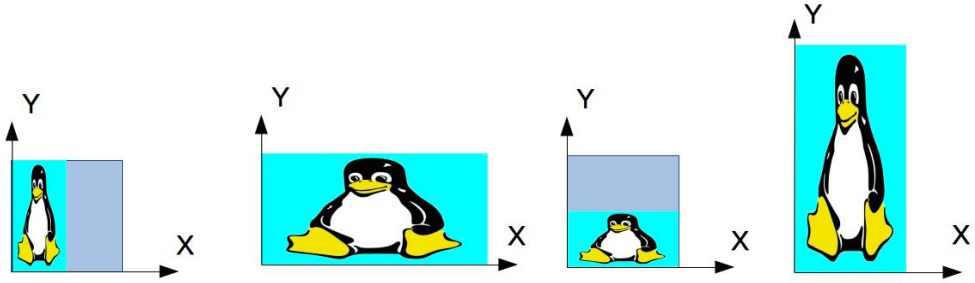
出典：http://www.songho.ca/opengl/gl_anglestoaxes.htmlのRxRyRzの項

(3)クォータニオンによる回転

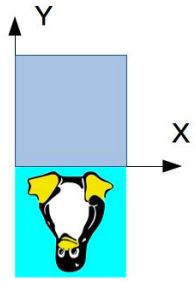
クォータニオン(四元数)と呼ばれる表現形式があります。この文書での解説は対象外としますが、Java3D のVecmathについての解説の、「回転とQuaternion」の項がわかりやすいと思います。３次元ベクトルで表した任意 軸を中心 とした回転を容易に行うために使われます。
(1)、(2)による回転は、ある物体の姿勢を表すには十分ですが、ゲームなどで使われる、２つの姿勢の間を補間したい場合には向いていない ので、 クォータニオンが使われるようです。例えば、シューティングゲームで、コンピュータで制御された敵のキャラクターが、最初は別の方向を向いていた銃 を、プレイヤーに向ける場合の銃の動きなどが該当します。
なお、JOGLにはcom.jogamp.opengl.math.Quaternionという、クォータニオンをサポートするためのクラス が用意さ れています。

拡大・縮小

これは、X、Y、Zの各方向について独立してスケールを指定できま す。
以下はそれぞれX方向に1/2、X方向に２倍、Y方向に1/2、Y方向に２倍しています。



また、以下のように軸に対して反転するよう指定することが出来ます。



これを行うためのプログラムコードは以下のようになります。

```
        gl.glScalef(0.5f, 1.0f, 1.0f);   //X方向を1/2
        gl.glScalef(2.0f, 1.0f, 1.0f);   //X方向を2倍
        gl.glScalef(1.0f, 0.5f, 1.0f);   //Y方向を1/2
        gl.glScalef(1.0f, 2.0f, 1.0f);   //Y方向を2倍
        gl.glScalef(1.0f, -1.0f, 1.0f); //Y方向を反転
```

また、変換行列は以下のようになります。

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} scaleX & 0 & 0 & 0 \\ 0 & scaleY & 0 & 0 \\ 0 & 0 & scaleZ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

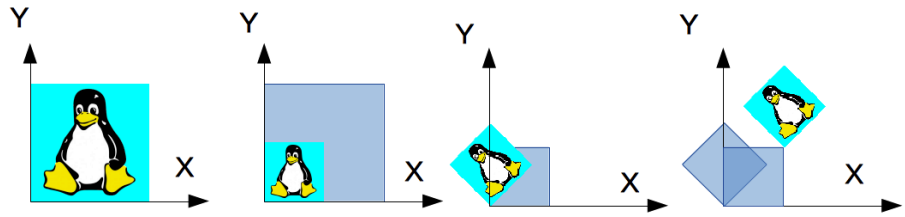
これを見ると、拡大・縮小では対角成分だけが使われています。一番下の行は平行移動、回転、拡大／縮小のいずれの操作でも{0, 0, 0, 1}となっていることが分かります。このような変換をアフィン変換と呼びます。

平行移動、回転、拡大／縮小の組み合わせ

前の方で、「回転の中心が原点と一致しているという前提があります」と述べましたが、一般的には任意の位置での拡大・縮小、回転、平行移動が必要となります。これを行うには、対象の物体のローカル座標で、拡大・縮小、回転、平行移動の順に行います。
行列で表現すると、スケール行列 * 回転行列 * 平行移動行列となります。(平行移動行列に対し回転移動行列を掛けてから、さらにスケール行列を掛けています)。
これをプログラムで表すと、以下のとおりの順番になります。行列を掛ける順番と逆になっていることに留意してください。

```
        gl.glTranslatef(x, y, z);   //平行移動
        gl.glRotatef(rot, ax, ay, zz);   //回転
        gl.glScalef(sx, sy, sz);   //拡大、縮小
```

これを図で表すと以下のようになります。



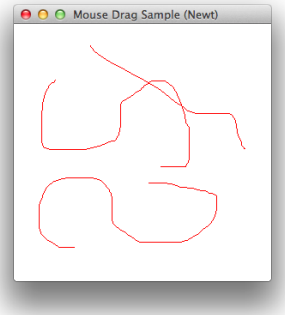
5.5 マウスのドラッグ

既出のcom.jogamp.newt.event.MouseListenerクラスにはドラッグ中に呼ばれるmouseDragged()メソッドと、ボタンを離してマウスを動かしたときに呼ばれるmouseMoved()メソッドがありますので、これを使って、簡単なペイントソフトもどきを作ってみます。以下のソースプログラムをNewtMouseDraggingSample.javaというファイル名で作成し、実行してみてください。

```
package demos.basic;

import java.awt.geom.Point2D;
import java.awt.geom.Point2D.Float;
import java.util.ArrayList;
import java.util.List;
import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.FPSAnimator;
import static com.jogamp.opengl.GL2.*;

public class NewtMouseDraggingSample implements GLEventListener, com.jogamp.newt.event.MouseListener { //(1)

        public static void main(String[] args) {
                new NewtMouseDraggingSample();
        }

        private final List> pointsList;
        private final List points;
        public NewtMouseDraggingSample() {
                pointsList = new ArrayList<>();
                points = new ArrayList<>();

                GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
                final GLWindow glWindow = GLWindow.create(caps);
                glWindow.setTitle("Mouse Drag Sample (Newt)");
```

```
                glWindow.setSize(300, 300);
                glWindow.addWindowListener(new WindowAdapter() {
                        @Override
                        public void windowDestroyed(WindowEvent evt) {
                                System.exit(0);
                        }
                });

                glWindow.addGLEventListener(this);
                glWindow.addMouseListener(this); //(2)
                FPSAnimator animator = new FPSAnimator(60);
                animator.add(glWindow);
                animator.start();
                glWindow.setPosition(500, 500);
                glWindow.setVisible(true);
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                //showGLInfo(drawable);
                GL gl = drawable.getGL();
                //背景を白く塗りつぶす。
                gl.glClearColor(1f, 1f, 1f, 1.0f);
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
                GL2 gl = drawable.getGL().getGL2();
                gl.glMatrixMode(GL_PROJECTION);//透視変換行列を指定
                gl.glLoadIdentity();//透視変換行列を単位行列にする
                System.out.printf("x:%d, y:%d, w:%d, h:%d, %n", x, y, width, height);
                gl.glOrthof(x, x + width, y + height, y, -1.0f, 1.0f);

                gl.glMatrixMode(GL_MODELVIEW);//モデルビュー変換行列を指定
                gl.glLoadIdentity();//モデルビュー変換行列を単位行列にする
        }

        @Override
        public void display(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                gl.glClear(GL_COLOR_BUFFER_BIT);
                gl.glColor3f(1.0f, 0.0f,0.0f); // 赤
                //現在ドラッグ中の図形を描画 (3)
                render(gl, points);
                //前の図形を描画 (4)
                for(List p : pointsList) {
                        render(gl, p);
                }
        }

        private void render(GL2 gl, List p) {
                gl.glBegin(GL_LINES);
                //p1のところで+1しているので、iが範囲を超えないようループ回数を一つ減らしている。
                for(int i = 0; i < p.size() - 1; i++) {
                        Point2D.Float p0 = (Float) p.get(i);
                        Point2D.Float p1 = (Float) p.get(i + 1);
                        gl.glVertex2d(p0.getX(), p0.getY()); // 今の位置
                        gl.glVertex2d(p1.getX(), p1.getY()); // 次の位置
                }
                gl.glEnd();
        }
        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(animator != null) animator.stop();
        }

        //ここから下で、com.jogamp.newt.event.MouseListenerインターフェースのメソッドを実装。 (5)
        @Override
        public void mouseDragged(com.jogamp.newt.event.MouseEvent e) {
                points.add(new Point2D.Float(e.getX(), e.getY())); //(6)
        }

        @Override
        public void mouseReleased(com.jogamp.newt.event.MouseEvent e) {
                pointsList.add(new ArrayList(points)); //(7)
                points.clear();
        }

        @Override
        public void mouseClicked(com.jogamp.newt.event.MouseEvent e) {}

        @Override
        public void mouseEntered(com.jogamp.newt.event.MouseEvent e) {}

        @Override
        public void mouseExited(com.jogamp.newt.event.MouseEvent e) {}

        @Override
        public void mousePressed(com.jogamp.newt.event.MouseEvent e) {}
        @Override
        public void mouseMoved(com.jogamp.newt.event.MouseEvent e) {}
        @Override
        public void mouseWheelMoved(com.jogamp.newt.event.MouseEvent e) {}
}
```

これを実行し、ウィンドウ内で適当にドラッグしてみてください。次のように、ドラッグしたところには点が連続して描画され、ボタンを離した状態でマウスを動かすと描画されなくなります。また、ウィンドウのサイズを変えても中の図形の大きさは維持されていることがわかると思います。



プログラムは、以下のようになっています。
(2)でGLWindowのインスタンスに対しキーボードのイベントリスナーを登録しています。ここではNewtMouseDraggingSampleクラスのインスタンス自身を登録しています。そこで(1)のようにリスナーを実装していることを宣言し、(5)で必要なメソッドを記述しています。
(6)マウスのドラッグ中に呼ばれるmouseDragged()内で、点をList型の変数pointsに追加しています。
(7)マウスが離されたときに呼ばれるmouseReleased()で、points変数を別のList>型の変数pointsListに追加し、pointsについては次回ドラッグされた時に備えてクリアしています。
また、display()メソッド内での描画は、(3)でマウスドラッグ中の図形を表示し、(4)で、上記(7)で記録された、過去に描画された図形をまとめて描画するようにしています。

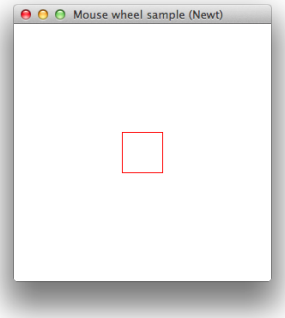
5.6 マウスホイールの操作

マウスホイールの操作により、図形の大きさを変えられるようにしてみます。以下のサンプルプログラムを、NewtMouseWheelSample.javaというファイル名で作成し、実行してみてください。

```
package demos.basic;

import static com.jogamp.opengl.GL.*;
import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.KeyEvent;
import com.jogamp.newt.event.KeyListener;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.FPSAnimator;

public class NewtMouseWheelSample implements GLEventListener, KeyListener {
        private static final char KEY_ESC = 0x1b;
        private static final float INIT_SCALE = 20f;
        private static final int HEIGHT = 300;
        private static final int WIDTH = 300;

        public static void main(String[] args) {
                new NewtMouseWheelSample();
        }

        private float scale = INIT_SCALE;

        public NewtMouseWheelSample() {

                GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
                final GLWindow glWindow = GLWindow.create(caps);
                glWindow.setTitle("Mouse wheel sample (Newt)");
                glWindow.setSize(WIDTH, HEIGHT);

                glWindow.addWindowListener(new WindowAdapter() {
                        @Override
                        public void windowDestroyed(WindowEvent arg0) {
                                System.exit(0);
                        }
                });

                glWindow.addGLEventListener(this);
                glWindow.addMouseListener(new com.jogamp.newt.event.MouseAdapter() {
                        @Override
                        public void mouseWheelMoved(com.jogamp.newt.event.MouseEvent e) {       //(1)
                                float[] rot = e.getRotation(); //(2)
                                scale *= (rot[1] > 0 ? 1.005f : 0.995f); //(3)
                                System.out.println("scale:" + scale);
                        }
                });

                glWindow.addKeyListener(this);

                FPSAnimator animator = new FPSAnimator(10);
                animator.add(glWindow);
                animator.start();
                glWindow.setPosition(500, 500);
                glWindow.setVisible(true);
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                //showGLInfo(drawable);
                GL gl = drawable.getGL();
                //背景を白く塗りつぶす。
                gl.glClearColor(1f, 1f, 1f, 1.0f);
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
                GL2 gl = drawable.getGL().getGL2();
                gl.glOrthof(x, x + width, y, y + height, -1.0f, 1.0f); //(4)
        }

        @Override
        public void display(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                gl.glClear(GL_COLOR_BUFFER_BIT);
                gl.glColor3f(1.0f, 0.0f,0.0f); // 赤
                gl.glBegin(GL_LINE_LOOP);
                gl.glVertex2f(WIDTH/2 - scale, HEIGHT/2 - scale); //(5)
                gl.glVertex2f(WIDTH/2 + scale, HEIGHT/2 - scale);
                gl.glVertex2f(WIDTH/2 + scale, HEIGHT/2 + scale);
                gl.glVertex2f(WIDTH/2 - scale, HEIGHT/2 + scale);
                gl.glEnd();
        }

        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(animator != null) animator.stop();
        }

        @Override
        public void keyPressed(KeyEvent e) {
        }

        @Override
        public void keyReleased(KeyEvent e) {
                char keyChar = e.getKeyChar();

                if( keyChar == ' ' ) {
                        scale = INIT_SCALE;
                        System.out.println("scale:" + scale);
                }
                if(keyChar == 'q' || keyChar == KEY_ESC) System.exit(0);
        }
```

```
}
```

以下のウィンドウが表示されました。マウスホイールを操作してみてください。中の図形が拡大したり縮小したりすると思います。スペースキーを押すと最初の大きさに戻ります。



(1)で、マウスホイールが操作されたときのイベントを受け取ります。(2)でマウスホイールの回転を取り出します。配列になっているのは、３Dマウスによる３軸の動きに対応するためだと思われます。ホイールの回転は添え字1の配列に格納されていました。回転方向により正負どちらかになるので、これに応じて(3)でscaleを変えています。
(4)では、glOrthof()によりビューポートを設定していますが、NewtMouseDraggingSample.javaで行っていたy軸方向の入れ替えはここでは行っていません。
(5)で、ウィンドウの中心から上下左右にscaleだけ離れた頂点をもつ正方形を描いています。

5.7 キーボードの操作

OpenGLのアプリケーションプログラムが開いたウィンドウには、Windowsのコマンドプロンプトのように、入力されたキーをウィンドウ上に表示するようにはなっていません。そのかわりマウスのボタン同様、キーをタイプするごとに実行する関数を指定できます。それにはGLWindowクラスのインスタンスに対し、addKeyListener(リスナーインスタンス名)を行い、キーリスナーを登録します。

また、これまで作ったプログラムを終了するには、タイトルバーのクローズボタンをクリックするか、WindowsではALTキーとファンクションキーのF4を、OSXではCommandキーとqを同時に押す方法がありましたが、'q'のキーやESCキーをタイプしたときにも終了するするようにします。以下のようなプログラムを、NewtKeyboardInputSample.javaとして保存して実行してください。

```
package demos.basic;

import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.KeyEvent;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.FPSAnimator;

public class NewtKeyboardInputSample implements GLEventListener, com.jogamp.newt.event.KeyListener { //(1)
        private static final char KEY_ESC = 0x1b;

        public static void main(String[] args) {
                new NewtKeyboardInputSample();
        }

        private final GLWindow glWindow;

        public NewtKeyboardInputSample() {
                GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
                glWindow = GLWindow.create(caps);
                glWindow.setTitle("Keyboard input demo (Newt)");
                glWindow.setSize(300, 300);

                glWindow.addWindowListener(new WindowAdapter() {
                        @Override
                        public void windowDestroyed(WindowEvent evt) {
                                System.exit(0);
                        }
                });

                glWindow.addGLEventListener(this);
                glWindow.addKeyListener(this); //(2)
                FPSAnimator animator = new FPSAnimator(10);
                animator.add(glWindow);
                animator.start();
                glWindow.setVisible(true);
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                //背景を白く塗りつぶす。
                gl.glClearColor(1f, 1f, 1f, 1.0f);
                showGLInfo(drawable);
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {}

        @Override
        public void display(GLAutoDrawable drawable) {}

        @Override
        public void dispose(GLAutoDrawable drawable) {
                System.out.println("dispose()");
                if(animator != null) animator.stop();
        }

        //ここから下で、com.jogamp.newt.event.MouseListenerインターフェースのメソッドを実装。 (3)
        @Override
        public void keyPressed(KeyEvent e) {
                char keyChar = e.getKeyChar();
                printKeyChar(keyChar, "pressed");

                if(e.isAltDown()) {
                        System.out.println("ALT key pressed");
                }
                if(e.isShiftDown()) {
                        System.out.println("Shift key pressed");
                }
                if(e.isControlDown()) {
```

```
                        System.out.println("Ctrl key pressed");
                }
                if(e.isAltGraphDown()) {
                        System.out.println("AltGraph key pressed");
                }
                if(e.isMetaDown()) {
                        System.out.println("Meta key pressed");
                }
        }

        @Override
        public void keyReleased(KeyEvent e) {
                char keyChar = e.getKeyChar(); //(4)
                printKeyChar(keyChar, "released"); //(5)

                if(Character.isISOControl(keyChar)) {
                        System.out.println(Integer.valueOf(keyChar) + " released");
                } else {
                        System.out.println(keyChar + " released");
                }


                if(e.isAltDown()) {
                        System.out.println("ALT key released");
                }
                if(e.isShiftDown()) {
                        System.out.println("Shift key released");
                }
                if(e.isControlDown()) {
                        System.out.println("Ctrl key released");
                }
                if(e.isAltGraphDown()) {
                        System.out.println("AltGraph key released");
                }
                if(e.isMetaDown()) {
                        System.out.println("Meta key released");
                }

                if(keyChar == KEY_ESC || keyChar == 'q' || keyChar == 'Q' ) { //(6)
                        glWindow.destroy();
                }
        }

        private void printKeyChar(char keyChar, String type) {
                if(Character.isISOControl(keyChar)) {
                        System.out.println(Integer.valueOf(keyChar) + type);
                } else {
                        System.out.println(keyChar + type);
                }
        }
}
```

(2)でGLWindowのインスタンスに対しキーボードのイベントリスナーを登録していますが、ここではNewtKeyboardInputSampleクラスのインスタンス自身を登録しています。
そこで(1)のようにリスナーを実装していることを宣言し、(3)で必要なメソッドを記述しています。

Swingのjava.awt.event.KeyAdapterクラスと同様に、NEWTでもcom.jogamp.newt.event.KeyAdapterクラスが用意されていますので、以下のように必要なメソッドだけを実装できます。

```
glWindow.addKeyListener(new com.jogamp.newt.event.KeyAdapter() {
        @Override
        public void keyPressed(KeyEvent e) {}
        @Override
        public void keyReleased(KeyEvent e) {}
}
```

そして(4)で文字を取得し、(5)で'q'あるいはESCキーだった場合にGLWindowインスタンスのdestroy()メソッドを呼び出しています。
ここでSystem.exit(0)を呼び出すと、内部でリソースの解放が行われない可能性があるので、望ましくありません。

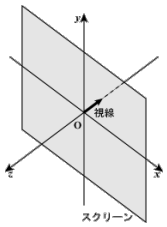
ALT,Control,Shift,Command,ファンクションキーのような文字キー以外のタイプを検出するときは、このプログラムを実行して調べたいキーを押すと、コードが10進数で表示されますので、参考にしてください。

なお、上記のとおりキー単位での入力には対応していますが、日本語を入力したいといったニーズには向いていません。どうしても入力したい場合、NEWTではなくSwingのJTextFieldなどを使うことになると思われます。

6.三次元図形を描く

6.1二次元と三次元

これまでは二次元の図形の表示を行ってきましたが、OpenGLの内部では実際には三次元の処理を行っています。すなわち画面表示に対して垂直にZ軸が伸びており、これまではその三次元空間のxy平面への平行投影像を表示していました。



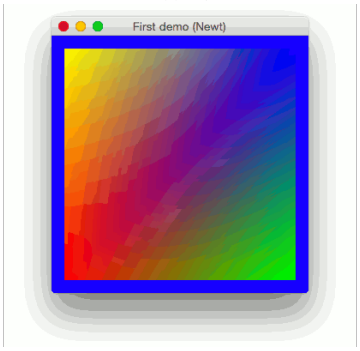
試しに4.5節で作成したプログラム(FirstStepNewt.java)において、図形をy軸中心に25度回転してみましょう。

```
@Override
public void display(GLAutoDrawable drawable) {
        GL2 gl = drawable.getGL().getGL2();
        gl.glClear(GL.GL_COLOR_BUFFER_BIT);
        //gl.glColor3f(1.0f, 0.0f, 0.0f); //ここは削除
        gl.glRotatef(25.0f, 0f, 1f, 0f); //追加
        gl.glBegin(GL_POLYGON);
        gl.glColor3f(1.0f, 0.0f, 0.0f); // 赤
        gl.glVertex2f(-0.9f,-0.9f);
        gl.glColor3f(0.0f, 1.0f, 0.0f); // 緑
        gl.glVertex2f(0.9f, -0.9f);
        gl.glColor3f(0.0f, 0.0f, 1.0f); // 青
        gl.glVertex2f(0.9f, 0.9f);
        gl.glColor3f(1.0f, 1.0f, 0.0f); // 黄
        gl.glVertex2f(-0.9f, 0.9f);
        gl.glEnd();
}
```

glRotatef(float angle,float x,float y,float z)

変換行列に回転の行列を乗じます。引数はいずれもfloat型で、一つ目の引数angleは回転角(単位は度)、残りの三つの引数x、y、zは回転軸の方向ベクトルです。引数がdouble型ならglRotated()を使います。原点を通らない軸で回転させたい場合は、glTranslatef()を使って一旦軸が原点を通るように図形を移動し、回転後に元の位置に戻します。7.1節でこのような操作を行っています。

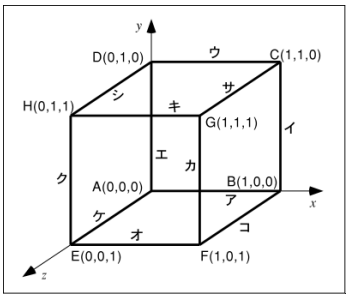
変更したプログラムを実行して、描かれる図形を見てください。



アニメーションの機能により、Y軸中心に回転しているため、横方向に延びたり縮んだりしていると思います。これは変換行列にglRotatef()による回転の行列が積算されるからです。これを防ぐには描画の度に変換マトリクスをglLoadIdentity()で初期化するか、後で述べるglPushMatrix()/glPopMatrix()を使って変換行列を保存します。

6.2線画を表示する

それでは、こんどは以下のような三次元の立方体を線画で描いてみましょう。JOGLにはGLUTから移植したglutWireCube()など、いくつか基本的な立体を描く関数があるのですが、ここでは自分で形状を定義してみたいと思います。



この図形は8個の点を12本の線分で結びます。点の位置(幾何情報)と線分(位相情報)を別々にデータにします。

```
float[][] vertex = {
        { 0.0f, 0.0f, 0.0f}, /* A */
        { 1.0f, 0.0f, 0.0f}, /* B */
        { 1.0f, 1.0f, 0.0f}, /* C */
        { 0.0f, 1.0f, 0.0f}, /* D */
        { 0.0f, 0.0f, 1.0f}, /* E */
        { 1.0f, 0.0f, 1.0f}, /* F */
        { 1.0f, 1.0f, 1.0f}, /* G */
        { 0.0f, 1.0f, 1.0f} /* H */
};
int[][] edge = {
        { 0, 1}, /* ア (A-B) */
        { 1, 2}, /* イ (B-C) */
        { 2, 3}, /* ウ (C-D) */
        { 3, 0}, /* エ (D-A) */
        { 4, 5}, /* オ (E-) */
        { 5, 6}, /* カ (-G) */
        { 6, 7}, /* キ (G-H) */
        { 7, 4}, /* ク (H-E) */
        { 0, 4}, /* ケ (A-E) */
        { 1, 5}, /* コ (B-) */
        { 2, 6}, /* サ (C-G) */
        { 3, 7} /* シ (D-H) */
};
```

この場合、例えば"点C"(1,1,0)と"点D"(0,1,0)を結ぶ線分"ウ"は、以下のようにして描画できます。glVertex3dv()は、引数に三つの要素を持つfloat型の配列を与えて、頂点を指定します。

```
gl.glBegin(GL_LINES);
gl.glVertex3dv(vertex[edge[2][0]], 0); // 線分 "ウ" の一つ目の端点 "C"
gl.glVertex3dv(vertex[edge[2][1]], 0); // 線分 "ウ" の二つ目の端点 "D"
gl.glEnd();
```

従って立方体全部を描くプログラムは以下のようになります。なお、立方体がウィンドウからはみ出ないように、glOrtho()で表示する座標系を(-2,-2)～(2,2)にしています。ウィンドウ中央より右上の領域に表示されます。

以下の内容のCubeSample.javaというファイル名で作成してください。

```
package demos.basic;

import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.FPSAnimator;
import static com.jogamp.opengl.GL2.*;

public class CubeSample implements GLEventListener {
        public static void main(String[] args){
                new NewtWireCube();
        }

        float[][] vertex = {
                { 0.0f, 0.0f, 0.0f}, /* A */
                { 1.0f, 0.0f, 0.0f}, /* B */
```

```
                { 1.0f, 1.0f, 0.0f}, /* C */
                { 0.0f, 1.0f, 0.0f}, /* D */
                { 0.0f, 0.0f, 1.0f}, /* E */
                { 1.0f, 0.0f, 1.0f}, /* F */
                { 1.0f, 1.0f, 1.0f}, /* G */
                { 0.0f, 1.0f, 1.0f} /* H */
        };

        int[][] edge = {
                { 0, 1}, /* ア (A-B) */
                { 1, 2}, /* イ (B-C) */
                { 2, 3}, /* ウ (C-D) */
                { 3, 0}, /* エ (D-A) */
                { 4, 5}, /* オ (E-) */
                { 5, 6}, /* カ (-G) */
                { 6, 7}, /* キ (G-H) */
                { 7, 4}, /* ク (H-E) */
                { 0, 4}, /* ケ (A-E) */
                { 1, 5}, /* コ (B-) */
                { 2, 6}, /* サ (C-G) */
                { 3, 7} /* シ (D-H) */
        };

        public CubeSample() {
                GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
                GLWindow glWindow = GLWindow.create(caps);
                glWindow.setTitle("First demo (Newt)");
                glWindow.setSize(300, 300);
                glWindow.addWindowListener(new WindowAdapter() {
                        @Override
                        public void windowDestroyed(WindowEvent evt) {
                                System.exit(0);
                        }
                });
                glWindow.addGLEventListener(this);
                FPSAnimator animator = new FPSAnimator(10); //(2)
                animator.add(glWindow);
                animator.start();
                glWindow.setVisible(true);
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                //背景を白く塗りつぶす.
                gl.glClearColor(1f, 1f, 1f, 1.0f);
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
                GL2 gl = drawable.getGL().getGL2();
                //gl.glViewport(0f, 0f, width, height); // Jogl内部で実行済みなので不要。
                gl.glOrtho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
        }

        @Override
        public void display(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                gl.glClear(GL.GL_COLOR_BUFFER_BIT);
                // 黒色を指定
                gl.glColor3f(0.0f, 0.0f, 0.0f);
                //図形の描画
                gl.glBegin(GL_LINES);
                for (int i = 0; i < 12; i++) {
                        gl.glVertex3fv(vertex[edge[i][0]], 0); //(1)
                        gl.glVertex3fv(vertex[edge[i][1]], 0); //(1)
                }
                gl.glEnd();
        }

        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(animator != null) animator.stop();
        }
}
```

glVertex3fv(float[] v,int offset)

glVertex3fv()は三次元の座標値を指定するのに使います。引数vは3個の要素を持つfloat型配列を指定します。v[0]にはx座標値、v[1]にはy座標値、v[2]にはz座標値を格納します。offsetには配列のオフセットを指定します。これは通常0です。C++言語ではこの引数は不要ですので、移植時には注意が必要です。double型の配列にしたい場合、glVertex3dv()を使います。なお、JOGLの場合、配列ではなくjava.nio.FloatBufferを引数とする同名の以下のメソッドが用意されています。

glVertex3fv(java.nio.FloatBuffer v)

引数vはx座標値、y座標値、z座標値の3個の要素を順番に格納したjava.nio.FloatBufferを指定します。引数をdouble型にしたい場合、java.nio.DoubleBufferを引数とするglVertex3dv()を使います。なお、ここでいうDoubleBufferは、グラフィック関係のアプリケーションで頻出するダブルバッファリングとは関係ありません。

6.3透視投影する

前のプログラムでは、立方体が画面に平行投影されるため、正方形しか描かないと思います。そこで現実のカメラのように透視投影をしてみます。これにはglOrtho()の代わりにgluPerspective()を使います。

gluPerspective()は座標軸の代わりに、カメラの画角やスクリーンのアスペクト比(縦横比)を用いて表示領域を指定します。またglOrtho()同様、前方面や後方面の位置の指定も行います。

視点の位置の初期値は原点なので、このままでは立方体が視点に重なってしまいます。そこでglTranslatef()を使って立方体の位置を少し奥にずらしておきます。

```
import com.jogamp.opengl.glu.GLU;//追加
private final GLU glu;//追加
public NewtWireCube() {
        GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
        GLWindow glWindow = GLWindow.create(caps);
glu = new GLU();//追加
        glWindow.setTitle("First demo (Newt)");
        //略
}
@Override
public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
        GL2 gl = drawable.getGL().getGL2();
        //gl.glViewport(0f, 0f, width, height); // Jogl内部で実行済みなので不要。
        //gl.glOrtho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0); コメントアウト
         glu.gluPerspective(30.0, (double)w / (double)h, 1.0, 100.0);gl.glTranslatef(0.0f, 0.0f, -5.0f);
}
```

GLU

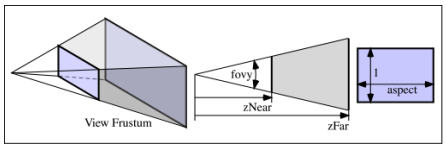
C/C++言語のGLUをJOGLに移植したもので、com.jogamp.opengl.glu.GLUパッケージとして提供されています。詳細は12章で説明します。

GLUT

ついでですが、GLUTについても移植されていて、com.jogamp.opengl.util.gl2.GLUTパッケージになっています。詳細は12章で説明します。

gluPerspective(float fovy, float aspect, float zNear, float zFar)

変換行列に透視変換の行列を乗じます。最初の引数fovyはカメラの画角であり、度で表します。これが大きいほどワイドレンズ(透視が強くなり、絵が小さくなります)になり、小さいほど望遠レンズになります。二つ目の引数aspectは画面のアスペクト比(縦横比)であり、1であればビューポートに表示される図形のx方向とy方向のスケールが等しくなります。三つ目の引数zNearと四つ目の引数zFarは表示を行う奥行き方向の範囲で、zNearは手前(前方面)、zFarは後方(後方面)の位置を示します。この間にある図形が描画されます。この立体は視錐台と呼ばれます。



glTranslatef(float x, float y, float z)

変換行列に平行移動の行列を乗じます。引数はいずれもfloat型で、三つの引数x、y、zには現在の位置からの相対的な移動量を指定します。引数がdouble型ならglTranslated()を使います。

ウィンドウをリサイズしたときに表示図形がゆがまないようにするためには、gluPerspective()で設定するアスペクト比aspectを、glViewport()で指定したビューポートの縦横比(w/h)と一致させます。

上のプログラムのように、リサイズ後のウィンドウのサイズをそのままビューポートに設定している場合、仮にaspectが定数であれば、ウィンドウのリサイズに伴って表示図形が伸縮するようになります。したがって、ウィンドウをリサイズしても表示図形の縦横比が変わらないようにするために、ここではaspectをビューポートの縦横比に設定しています。

6.4視点の位置を変更する

前のプログラムのように、視点の位置を移動するには、図形の方をglTranslated()やglRotated()を用いて逆方向に移動することで実現できます。しかし、視点を任意の位置に指定したいときにはgluLookAt()を使うと便利です。

```
@Override
public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
        GL2 gl = drawable.getGL().getGL2();
        //gl.glViewport(0f, 0f, width, height); // Jogl内部で実行済みなので不要。
        //gl.glOrtho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0); コメントアウト
        glu.gluPerspective(30.0, (double)w / (double)h, 1.0, 100.0);
        gl.glTranslatef(0.0f, 0.0f, -5.0f);
        gluLookAt(3.0f, 4.0f, 5.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
}
```

void gluLookAt(float ex, float ey, float ez, float cx, float cy, float cz, float ux, float uy, float uz)

この最初の三つの引数ex、ey、ezは視点の位置、次の三つの引数cx、cy、czは目標の位置、最後の三つの引数ux、uy、uzは、ウィンドウに表示される画像の「上」の方向を示すベクトルです。

この例では(3,4,5)の位置から原点(0,0,0)を眺めますから、立方体の中心の頂点がウィンドウの中心に来ると思います。

7.アニメーション

7.1図形を動かす

ここまでできたら、今度はこの立方体を回してみましょう。
一つ注意しなければいけないことがあります。繰り返し描画を行うには、描画の度に座標変換の行列を設定する必要があります。
カメラ(視点)の位置を動かすアニメーションを行う場合は、描画のたびにgluLookAt()によるカメラの位置や方向の設定(ビューイング変換行列の設定)を行う必要があります。同様に物体が移動したり回転したりするアニメーションを行う場合も、描画のたびに物体の位置や回転角の設定(モデリング変換行列の設定)を行う必要があります。したがって、これらはdisplay()の中で設定します。

ここで、マウスの左ボタンをクリックしている間、立方体が回転するようなプログラムを作ります。ついでに右ボタンをクリックすると立方体が1ステップだけ回転し、'q'、'Q'、またはESCキーでプログラムが終了するようにします。
以下の内容のCubeSample2.javaというファイルを作ってください。

```
package demos.basic;

import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.opengl.glu.GLU;
import com.jogamp.newt.event.KeyEvent;
import com.jogamp.newt.event.KeyListener;
import com.jogamp.newt.event.MouseEvent;
import com.jogamp.newt.event.MouseListener;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.FPSAnimator;
import static com.jogamp.opengl.GL2.*;

public class CubeSample2 implements GLEventListener, MouseListener, KeyListener {

        public static void main(String[] args){
                new CubeSample2();
        }

        float[][] vertex = {
        { 0.0f, 0.0f, 0.0f}, /* A */
        { 1.0f, 0.0f, 0.0f}, /* B */
        { 1.0f, 1.0f, 0.0f}, /* C */
        { 0.0f, 1.0f, 0.0f}, /* D */
        { 0.0f, 0.0f, 1.0f}, /* E */
        { 1.0f, 0.0f, 1.0f}, /* F */
        { 1.0f, 1.0f, 1.0f}, /* G */
        { 0.0f, 1.0f, 1.0f} /* H */
        };

        int[][] edge = {
        { 0, 1}, /* ア (A-B) */
        { 1, 2}, /* イ (B-C) */
        { 2, 3}, /* ウ (C-D) */
        { 3, 0}, /* エ (D-A) */
        { 4, 5}, /* オ (E-) */
        { 5, 6}, /* カ (-G) */
        { 6, 7}, /* キ (G-H) */
        { 7, 4}, /* ク (H-E) */
        { 0, 4}, /* ケ (A-E) */
        { 1, 5}, /* コ (B-) */
        { 2, 6}, /* サ (C-G) */
        { 3, 7} /* シ (D-H) */
        };

        private final GLU glu;
        private final Animator animator; //(1)
```

```
	private final GLWindow glWindow;
	private boolean willAnimatorPause = false;
	private static final char KEY_ESC = 0x1b;

	//回転角
	float r = 0;

	public CubeSample2() {
		GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
		glu = new GLU();

		glWindow = GLWindow.create(caps);
		glWindow.setTitle("Cube demo (Newt)");
		glWindow.setSize(300, 300);
		glWindow.addGLEventListener(this);

		glWindow.addWindowListener(new WindowAdapter() {
			@Override
			public void windowDestroyed(WindowEvent evt) {
				System.exit(0);
			}
		});

		glWindow.addMouseListener(this);
		glWindow.addKeyListener(this);
		animator = new Animator(); //(2)
		animator.add(glWindow); //(3)
		animator.start(); //(4)
		animator.pause(); //(5)
		glWindow.setVisible(true);
	}

	@Override
	public void init(GLAutoDrawable drawable) {
		GL2 gl = drawable.getGL().getGL2();
		//背景を白く塗りつぶす.
		gl.glClearColor(1f, 1f, 1f, 1.0f);
	}

	@Override
	public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
		GL2 gl = drawable.getGL().getGL2();

		gl.glMatrixMode(GL_PROJECTION);
		gl.glLoadIdentity();
		glu.gluPerspective(30.0, (double)width / (double)height, 1.0, 300.0);

		// 視点位置と視線方向
		glu.gluLookAt(3.0f, 4.0f, 5.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
		gl.glMatrixMode(GL_MODELVIEW);
	}

	@Override
	public void display(GLAutoDrawable drawable) {
		GL2 gl = drawable.getGL().getGL2();
		gl.glClear(GL.GL_COLOR_BUFFER_BIT);

		gl.glLoadIdentity();

		// 図形の回転
		gl.glRotatef(r, 0.0f, 1.0f, 0.0f); //(6)
		// 図形の描画
		gl.glColor3f(0.0f, 0.0f, 0.0f);
		gl.glBegin(GL_LINES);
		for (int i = 0; i < 12; i++) {
			gl.glVertex3fv(vertex[edge[i][0]], 0);
			gl.glVertex3fv(vertex[edge[i][1]], 0);
		}
		gl.glEnd();

		//一周回ったら回転角を 0 に戻す
		if (r++ >= 360.0f) r = 0;
		System.out.println("anim:" + animator.isAnimating() + ", r:" + r);
		if(willAnimatorPause) {
			animator.pause(); //(8)
			System.out.println("animoator paused:");
			willAnimatorPause = false;
		}
	}

	@Override
	public void dispose(GLAutoDrawable drawable) {
		if(animator != null) animator.stop();
	}

	@Override
	public void keyPressed(KeyEvent e) {}

	@Override
	public void keyReleased(KeyEvent e) {
		char keyChar = e.getKeyChar();
		if(keyChar == KEY_ESC || keyChar == 'q' || keyChar == 'Q') {
			glWindow.destroy();
		}
	}

	@Override
	public void mouseClicked(MouseEvent e) { }

	@Override
	public void mouseEntered(MouseEvent e) { }

	@Override
	public void mouseExited(MouseEvent e) {}

	@Override
	public void mousePressed(MouseEvent e) {
		switch(e.getButton()) {
			case MouseEvent.BUTTON1:
				animator.resume(); //(9)
				System.out.println("button 1, left click");
				break;
			case MouseEvent.BUTTON2:
				System.out.println("button 2");
				break;
			case MouseEvent.BUTTON3:
				System.out.println("button 3, right click");
				willAnimatorPause = true; //(10)
				animator.resume(); //(11)
				break;
			default:
				//empty
```

```
                }
        }

        @Override
        public void mouseReleased(MouseEvent e) {
                animator.pause();
        }

        @Override
        public void mouseMoved(MouseEvent e) { }

        @Override
        public void mouseDragged(MouseEvent e) {}

        @Override
        public void mouseWheelMoved(MouseEvent e) {}
}
```

JOGLの場合、定期的に画面を更新するために、AnimoatorクラスまたはFPSAnimatorクラスを使います。実は、これは既に6.1節で図形が横方向に延びたり縮んだりするところで使っています。(NEWTを使う場合、FPSAnimatorクラスを使わないとアプリケーションが起動してから数秒で終了してしまうため、使わざるを得なかったという事情がありました)。

FPSAnimoatorクラスまたはAnimatorクラスの使い分けについては、前者のFPSは"Frame Per Second"の略で、ゲームなどのように画面の更新頻度を一定に保ちたい場合にこちらを使い、後者は特にFPSを指定する必要がない場合に使えば良いと思います。但し、FPSAnimatorを使っていても、パソコンの負荷の状況によっては指定されたレートが保証される訳ではないようです。
サンプルでは、特にFPSを指定する必要がないため、Animatorクラスを使っています。

(1)で変数animatorを定義し、CubeSample2クラスのコンストラクタ内で(2)animatorを初期化、(3)glWindowとの関係を定義、(4)のanimator.start();でアニメーション開始を指定しています。通常は、(4)のanimator.start()だけで良いのですが、ここではマウスクリック時だけ回転させるため、直後に(5)のanimator.pause()で停止しています。
そして、(9)でマウス左ボタンをクリックしている間は、animator.resume()を呼ぶので、display()メソッドが呼ばれて、図形が回転し続けるようにしています。
(10)でマウス右ボタンがクリックされたことを検出したら、変数willAnimatorPauseをtrueにすると共に、(11)でanimator.resume()を呼ぶので、display()メソッドが呼ばれます。display()内では、willAnimatorPauseがtrueなら(8)でanimator.pause()を呼んでいるので、アニメーションは停止し、この後display()が呼ばれなくなるので、図形の回転も止まります。従って、図形は１ステップだけ回転しますが、右ボタンを押し続けても回転しません。回転させるには、いったんマウスを放して、もう一度右クリックします。

ここで、(1)のanimator変数のクラスをFPSAnimatorに変え、(2)をanimator = FPSAnimator(30);のように変えても同じように動作します。FPSAnimatorのコンストラクタは以下のようなバリエーションがあります。２番目以降は、内部で、最初のコンストラクタを、デフォルトの引数(nullあるいはfalse)を指定して呼び出しています。

FPSAnimator(final GLAutoDrawable drawable, final int fps, final boolean scheduleAtFixedRate)
drawableにはGLAutoDrawableのインスタンスを指定します。(以下の捕捉参照)fpsには設定したいフレームレートを、scheduleAtFixedRateには、指定したフレームレートに従い、アニメーションを行うか否かを指定します。(原文は"a flag indicating whether to use fixed-rate scheduling."となっており、true/falseを指定した場合の動作の違いが不明確ですが、trueを指定した場合には、指定されたフレームレートを守ろうと出来るだけ頑張るけど、falseだったらそこまでは頑張らないよ、という意味ですかね？)

FPSAnimator(final int fps)
fpsには設定したいフレームレートを指定します。

FPSAnimator(final int fps, final boolean scheduleAtFixedRate)
fpsには設定したいフレームレートを、scheduleAtFixedRateには、指定したフレームレートに従い、アニメーションを行うか否かを指定します。

FPSAnimator(final GLAutoDrawable drawable, final int fps)
drawableにはGLAutoDrawableのインスタンスを、fpsには設定したいフレームレートを指定します。

FPSAnimatorの全ての引数を持つコンストラクタを使う場合、以下のようにします。GLWindowクラスはGLAutoDrawableインターフェースを実装しているので、GLWindowのインスタンスが指定できます。

```
        private final FPSAnimator animator;
        //（省略）
        public CubeSample2() {
                //（省略）
                glWindow.addMouseListener(this);
                glWindow.addKeyListener(this);
                animator = new FPSAnimator(glWindow, 30, true);
                //次の行をコメントアウトしないと、"IllegalArgumentException: Drawable already added to animator"が投げられる。
                //animator.add(glWindow);
                animator.start();
                animator.pause();
                glWindow.setVisible(true);
        }
```

ところで、このプログラムでは、以下のように、図形の中心を、x,y,z共に0.5fだけずらし、回転させてから、元の位置に戻しています。 このようにしないと、回転の中心が物体の中心と一致しません。

```
// 図形の回転
gl.glTranslatef(0.5f, 0.5f, 0.5f);//追加
gl.glRotatef(r, 0.0f, 1.0f, 0.0f);
gl.glTranslatef(-0.5f, -0.5f, -0.5f);//追加
```

7.2ダブルバッファリング

グラフィック関係のソフトウェアでは、更新途中の画面を表示することによるちらつきを防ぐため、ダブルバッファリングという技法が使われることがあります。これは画面描画用に二つのバッファを用意し、一方を表示している間に(見えないところで)もう一方に図形を描き、それが完了したらこの二つの画面を入れ換える方法です。JOGLでは、デフォルトでこれを行うように設定されています。

それでは、CubeSample2.javaでダブルバッファリングを行わないようにしてみましょう。

```
@Override
public void init(GLAutoDrawable drawable) {
        GL2 gl = drawable.getGL().getGL2();//背景を白く塗りつぶす
        gl.glClearColor(1f, 1f, 1f, 1.0f);
        System.out.println("auto swap:" + drawable.getAutoSwapBufferMode());//追加
        drawable.setAutoSwapBufferMode(false);//追加
}
```

これを実行しても、ウィンドウの中に何も表示されません。これを表示するようにするには、display()メソッドの最後に以下のように`drawable.swapBuffers()`を追加する必要があります。

```
@Override
public void display(GLAutoDrawable drawable) {
        //省略
        drawable.swapBuffers(); //追加
}
```

これはC/C++版のGLUTの`glutSwapBuffers()`に相当する機能です。前述のとおり、デフォルトでダブルバッファリングが行われていますので、特別な事情がない限り、setAutoSwapBufferMode(false)でダブルバッファーを使わないように設定する必要はないでしょう。

7.3 Animoatorとスレッドについて

サンプルコード CubeSample2.javaでは、描画はFPSAnimatorというアニメーション用のクラスから呼ばれたスレッドが担当し、マウス操作については、SwingのEventDispatchThreadに相当する別のスレッドが動いています。
CubeSample2.javaに以下を追加して動かしてみます。

```
@Override
public void display(GLAutoDrawable drawable) {
	dumpThread("display");
	//他は同じ
}

@Override
public void mouseDragged(com.jogamp.newt.event.MouseEvent e) {
	dumpThread("dragging");
	//他は同じ
}

private void dumpThread(String name) {
	Thread th = Thread.currentThread();
	System.out.println(name + ":" + th.getName() + "," + th.getState());
}
```

以下の２つのスレッドが動いているのが分かります。OS X(Marvericks)で動かした場合ですので、別のOSでは異なる結果になるはずです。
前者は何もしなくても常時表示されますが、後者はマウスをドラッグしたときだけ表示されています。

```
display:main-FPSAWTAnimator#00-Timer0, RUNNABLE（実際にはこれが何度も表示される）
dragging:main-Display-.macosx_nil-1-EDT-1, RUNNABLE
```

現時点では、このように別々のスレッドが動いていることを提示するだけにとどめますが、プログラムが複雑になると、自分でスレッドを起動する必要にせまられ、複数のスレッドの相互作用を考慮しないとならない機会もあるかと思います。
なお、JOGLではSwingWorkerに相当するユーティリティー・クラスが提供されているかどうか、SwingWorkerとJOGLを組み合わせることが出来るかどうかは現時点では未確認です。申し訳ありません。

8.隠面消去処理

8.1多面体を塗りつぶす

それでは、次に立方体の面を塗りつぶしてみましょう。面のデータは、稜線とは別に以下のように用意します。

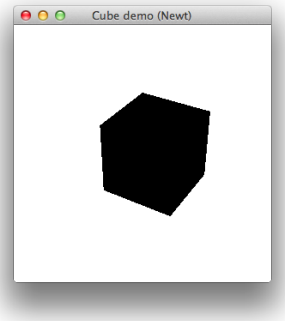
```
private final int face[][] = {
	{ 0, 1, 2, 3 }, // A-B-C-D を結ぶ面
	{ 1, 5, 6, 2 }, // B-F-G-C を結ぶ面
	{ 5, 4, 7, 6 }, // F-E-H-G を結ぶ面
	{ 4, 0, 3, 7 }, // E-A-D-H を結ぶ面
	{ 4, 5, 1, 0 }, // E-F-B-A を結ぶ面
	{ 3, 2, 6, 7 } // D-C-G-H を結ぶ面
};
```

このデータを使って、線を引く代わりに6枚の四角形を描きます。以下の内容の、CubeSample3.javaというファイルを作成してください。

```
package demos.basic;

import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.opengl.glu.GLU;
import com.jogamp.newt.event.MouseEvent;
import com.jogamp.newt.event.MouseListener;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.FPSAnimator;
import static com.jogamp.opengl.GL2.*;

public class CubeSample3 implements GLEventListener, MouseListener {


	public static void main(String[] args) {
		new CubeSample3();
	}

	private final float[][] vertex = {
	{ 0.0f, 0.0f, 0.0f}, /* A */
	{ 1.0f, 0.0f, 0.0f}, /* B */
	{ 1.0f, 1.0f, 0.0f}, /* C */
	{ 0.0f, 1.0f, 0.0f}, /* D */
	{ 0.0f, 0.0f, 1.0f}, /* E */
	{ 1.0f, 0.0f, 1.0f}, /* F */
	{ 1.0f, 1.0f, 1.0f}, /* G */
	{ 0.0f, 1.0f, 1.0f} /* H */
	};

	private final int face[][] = {
	{ 0, 1, 2, 3 }, // A-B-C-D を結ぶ面
	{ 1, 5, 6, 2 }, // B-F-G-C を結ぶ面
	{ 5, 4, 7, 6 }, // F-E-H-G を結ぶ面
	{ 4, 0, 3, 7 }, // E-A-D-H を結ぶ面
	{ 4, 5, 1, 0 }, // E-F-B-A を結ぶ面
	{ 3, 2, 6, 7 } // D-C-G-H を結ぶ面
	};


	private final GLU glu;

	private final FPSAnimator animator;

	private boolean willAnimatorPause = false;

	//回転角
	float r = 0;

	public CubeSample3() {
		GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
		glu = new GLU();
		GLWindow glWindow = GLWindow.create(caps);
		glWindow.setTitle("Cube demo (Newt)");
		glWindow.setSize(300, 300);
		glWindow.addWindowListener(new WindowAdapter() {

			@Override
			public void windowDestroyed(WindowEvent evt) {
				System.exit(0);
			}
		});
```

```
                glWindow.addGLEventListener(this);
                glWindow.addMouseListener(this);
                animator = new FPSAnimator(30);
                animator.add(glWindow);
                animator.start();
                animator.pause();
                glWindow.setVisible(true);
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                //背景を白く塗りつぶす.
                gl.glClearColor(1f, 1f, 1f, 1.0f);
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
                GL2 gl = drawable.getGL().getGL2();
                gl.glMatrixMode(GL_PROJECTION);
                gl.glLoadIdentity();
                glu.gluPerspective(30.0, (double)width / (double)height, 1.0, 300.0);
                glu.gluLookAt(3.0f, 4.0f, 5.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
                gl.glMatrixMode(GL_MODELVIEW);
        }

        @Override
        public void display(GLAutoDrawable drawable) {
                GL2 gl = drawable.getGL().getGL2();
                gl.glClear(GL_COLOR_BUFFER_BIT);

                gl.glLoadIdentity();

                // 図形の回転
                gl.glTranslatef(0.5f, 0.5f, 0.5f);
                gl.glRotatef(r, 0.0f, 1.0f, 0.0f);
                gl.glTranslatef(-0.5f, -0.5f, -0.5f);
                // 図形の描画
                gl.glColor3f(0.0f, 0.0f, 0.0f);
                gl.glBegin(GL_QUADS);
                for (int j = 0; j < 6; ++j) {
                        for (int i = 0; i < 4; ++i) {
                                gl.glVertex3fv(vertex[face[j][i]], 0);
                        }
                }
                gl.glEnd();

                //一周回ったら回転角を 0 に戻す
                if (r++ >= 360.0f) r = 0;
                System.out.println("anim:" + animator.isAnimating() + ", r:" + r);
                if(willAnimatorPause) {
                        animator.pause();
                        System.out.println("animoator paused:");
                        willAnimatorPause = false;
                }
        }

        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(animator != null) animator.stop();
        }

        @Override
        public void mouseClicked(MouseEvent e) { }

        @Override
        public void mouseEntered(MouseEvent e) { }

        @Override
        public void mouseExited(MouseEvent e) {}

        @Override
        public void mousePressed(MouseEvent e) {
                switch(e.getButton()) {
                        case MouseEvent.BUTTON1:
                        animator.resume();
                        System.out.println("button 1, left click");
                        break;
                        case MouseEvent.BUTTON2:
                        System.out.println("button 2");
                        break;
                        case MouseEvent.BUTTON3:
                        System.out.println("button 3, right click");
                        willAnimatorPause = true;
                        animator.resume();
                        break;
                }
        }

        @Override
        public void mouseReleased(MouseEvent e) {
                animator.pause();
        }

        @Override
        public void mouseMoved(MouseEvent e) { }

        @Override
        public void mouseDragged(MouseEvent e) {}

        @Override
        public void mouseWheelMoved(MouseEvent e) {}
}
```

これを実行すると次のようになります。真っ黒で何もわかりません。



面ごとに色を変えてみましょう。色のデータは以下のように作ってみます。

```
private final float color[][] = {
{ 1.0f, 0.0f, 0.0f }, // 赤
{ 0.0f, 1.0f, 0.0f }, // 緑
{ 0.0f, 0.0f, 1.0f }, // 青
{ 1.0f, 1.0f, 0.0f }, // 黄
{ 1.0f, 0.0f, 1.0f }, // マゼンタ
{ 0.0f, 1.0f, 1.0f } // シアン
};
```

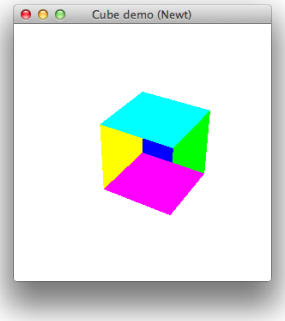
一つの面を描く度に、この色を設定してやります。display()メソッドを以下のように修正します。

```
@Override
public void display(GLAutoDrawable drawable) {
        // 図形の描画
         //gl.glColor3f(0.0f, 0.0f, 0.0f);// コメントアウト
        gl.glBegin(GL_QUADS);
        for (int j = 0; j < 6; ++j) {
                gl.glColor3fv(color[j], 0); //追加
                for (int i = 0; i < 4; ++i) {
                        gl.glVertex3fv(vertex[face[j][i]], 0);
                }
        }
        gl.glEnd();
}
```

void glColor3fv(float[]v,imtoffset)

glColor3fv()はglColor3f()と同様にこれから描画するものの色を指定します。引数vは三つの要素を持ったfloat型の配列で、v[0]には赤(R)、v[1]には緑(G)、v[2]には青(B)の強さを、0以上1以下の範囲で指定します。offsetには0を指定します。

でもこれだとなんか変な表示になるかもしれません。



前のプログラムではデータの順番で面を描いていますから、先に描いたものが後に描いたもので塗りつぶされてしまいます。ちゃんとした立体を描くには**隠面消去処理**を行う必要があります。

8.2デプスバッファを使う

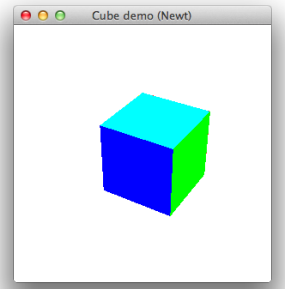
隠面消去処理を行なうには、glEnable(GL_DEPTH_TEST)を実行します。こうすると、描画のときにデプスバッファを使うようになります。このため、画面(フレームバッファ、カラーバッファ)を消去するときにデプスバッファも一緒に消去しておきます。それにはglClear()の引数にGL_DEPTH_BUFFER_BITを追加します。

デプスバッファを使うと、使わないときより処理速度が低下します。そこで、必要なときだけデプスバッファを使うようにします。デプスバッファを使う処理の前でglEnable(GL_DEPTH_TEST)を実行し、使い終わったらglDisable(GL_DEPTH_TEST)を実行します。**このプログラムでは常にデプスバッファを使うので、init()の中でglEnable(GL_DEPTH_TEST)を一度だけ実行し、glDisable(GL_DEPTH_TEST)の実行を省略しています。**

CubeSample3.javaを以下のように修正します。

```
@Override
public void init(GLAutoDrawable drawable) {
GL2 gl = drawable.getGL().getGL2();
        //背景を白く塗りつぶす
        gl.glClearColor(1f, 1f, 1f, 1.0f);
        gl.glEnable(GL_DEPTH_TEST);//追加
}
@Override
public void display(GLAutoDrawable drawable) {
        GL2 gl = drawable.getGL().getGL2();
         gl.glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //GL_DEPTH_BUFFER_BITを追加
        //(省略)
}
```

以下のように正常に表示されるようになりました。



8.3カリング

立方体のように閉じた立体の場合、裏側にある面、すなわち視点に対して背を向けている面は見ることはできません。そういう面をあらかじめ取り除いておくことで、隠面消去処理の効率を上げることができます。

視点に対して背を向けている面を表示しないようにするにはglCullFace(GL_BACK)、表を向いている面を表示しないようにするにはglCullFace(GL_FRONT)、両方とも表示しないようにするにはglCullFace(GL_FRONT_AND_BACK)を実行します。ただし、この状態でも点や線などは描画されます。

また、glCullFace()を有効にするにはglEnable(GL_CULL_FACE)、無効にするにはglDisable(GL_CULL_FACE)を実行します。サンプルプログラムを以下のように変えてみてください。

```
@Override
public void init(GLAutoDrawable drawable) {
        GL2 gl = drawable.getGL().getGL2();
        //背景を白く塗りつぶす.
        gl.glClearColor(1f, 1f, 1f, 1.0f);
        gl.glEnable(GL_DEPTH_TEST);
        gl.glEnable(GL_CULL_FACE);//追加
        gl.glCullFace(GL_BACK);//追加
}
```

このプログラムも、多分妙な表示になります。裏側の面を表示しないはずなのに、実際は表側の面が削除されています。実は、面の表裏は頂点をたどる順番で決定しています。配列face[]ではこれを右回り(時計回り)で結んでいます。ところがOpenGLでは、標準では視点から見て頂点が左回りになっているとき、その面を表として扱います。試しにglCullFace(GL_FRONT)としてみてください。あるいは、face[]において頂点を右回りにたどるようにしてみてください。

なお、頂点が右回りになっているときを表として扱いたいときは、glFrontFace(GL_CW)を実行します。左回りに戻すにはglFrontFace(GL_CCW)を実行します。

一般にカリングはクリッピングや隠面消去処理の効率を上げるために、視野外にある図形など見えないことが分かっているものを事前に取り除いておいて、隠面消去処理(可視判定)の対象から外しておくことを言います。これには様々な方法が存在しますが、glCullFace()による方法(背面ポリゴンの除去)は、そのもっとも基本的なものです。

9.陰影付け

9.1光を当ててみる

次は面ごとに色を付けるかわりに、光を当ててみましょう。この場合、glColor()で指定した物体の表面の色は使われず、代わりに光が照らす方向と、表面での光の反射方向に従い、表面の陰影が計算されます。（実際は表面の色も使われますが、後で説明します）

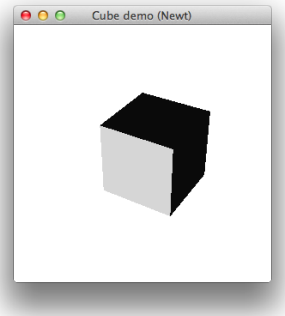
この計算を行うためには、面ごとの色の代わりに法線ベクトルを与えます。

```
private final float normal[][] = {
{ 0.0f, 0.0f, -1.0f},
{ 1.0f, 0.0f, 0.0f},
{ 0.0f, 0.0f, 1.0f},
{-1.0f, 0.0f, 0.0f},
{ 0.0f, -1.0f, 0.0f},
{ 0.0f, 1.0f, 0.0f}
};
```

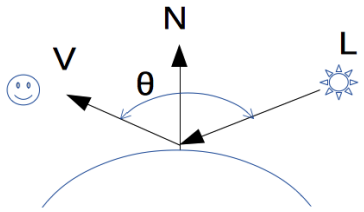
また、glColor3fv()のかわりにglNormal3fv()を使います。

```
@Override
public void init(GLAutoDrawable drawable) {
        GL2 gl = drawable.getGL().getGL2();
        //背景を白く塗りつぶす.
        gl.glClearColor(1f, 1f, 1f, 1.0f);
        gl.glEnable(GL_DEPTH_TEST);
        gl.glEnable(GL_CULL_FACE);
        gl.glCullFace(GL_FRONT);//GL_BACKから変更
        gl.glEnable(GL_LIGHTING);//追加
        gl.glEnable(GL_LIGHT0);//追加
        gl.glEnable(GL_LIGHT1);//追加
}
@Override
public void display(GLAutoDrawable drawable) {
        //省略
        gl.glBegin(GL_QUADS);
        for (int j = 0; j < 6; ++j) {
                //gl.glColor3fv(color[j], 0);//コメントアウト
                gl.glNormal3fv(normal[j], 0);//追加
                for (int i = 0; i < 4; ++i) {
                        gl.glVertex3fv(vertex[face[j][i]], 0);
                }
        }
        gl.glEnd();
        //省略
}
```

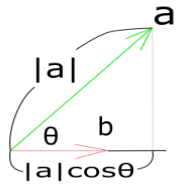
これを実行すると次のようになります。



ところで、上記で、法線ベクトルを与えたのはどうしてなのでしょうか？物体の表面の明るさは、光の方向と、法線の向きの間の角度によって決まると言われています。具体的には、この角度をθとすると、cosθに比例する明るさとなります。



光の方向と、法線をベクトルで表しているので、以下のように、ベクトルの内積を計算すればよいことになります。



ベクトルの内積(ドット積とも呼ばれる)とは、２つのベクトルa(a_x,a_y,a_z)とb(b_x,b_y,b_z)があったとき、以下の式で表されるスカラー量です。

$$a \cdot b = a_x b_x + a_y b_y + a_z b_z$$

この値は、aとbの間の角度をθとすると、

$$a \cdot b = |a||b|\cos\theta$$

とも等しくなります。従って、以下の式で明るさのスケールが求められます。

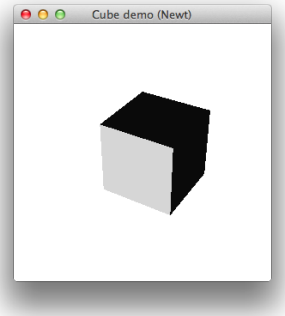
$$\cos\theta = \frac{a \cdot b}{|a||b|} = \frac{a_x b_x + a_y b_y + a_z b_z}{\sqrt{(a_x^2 + a_y^2 + a_z^2)(b_x^2 + b_y^2 + b_z^2)}}$$

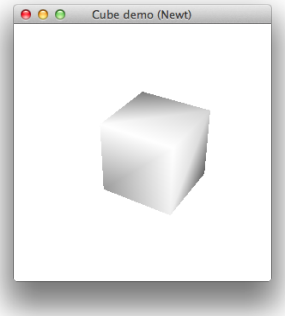
aとbをあらかじめ正規化（絶対値を１とする）しておけば、内積を計算するだけでcosθが求められます。

ここで、前述のコードの一部を、以下のように変えて実行してみてください。

```
 // 図形の描画
gl.glBegin(GL_QUADS);
for (int j = 0; j < 6; ++j) {
        //gl.glNormal3fv(normal[j], 0);//コメントアウト
        for (int i = 0; i < 4; ++i) {
                gl.glNormal3fv(vertex[face[j][i]], 0);//追加
                gl.glVertex3fv(vertex[face[j][i]], 0);
        }
}
gl.glEnd();
```

これにより各頂点での法線ベクトルは、立方体の中心から各頂点へ延びる線分になります。
左が変更前、右が変更後です。頂点が不明瞭になっていることが分かると思いますが、これはさいころのような形だと、さいころの各面に垂直な法線ベクトルを使うべきであるためです。





プログラムを元に戻しておいてください。

9.2光源を設定する

ここまでで表示した結果は、上面と側面がほぼ同じような黒さで、まだ立体感があるとはとうてい言えないような状態となっています。そこで、光源を設定して見栄えをよくしてみましょう。OpenGLには、最初からいくつかの光源が用意されています。いくつの光源が用意されているかはシステムによって異なりますが、仕様上8個までは必ず用意することになっています。この数は以下のようにして問い合わせることができます。

```
int[] maxLights = new int[1];
gl.glGetIntegerv(GL2.GL_MAX_LIGHTS, maxLights, 0);
System.out.println(maxLights[0]);
```

0番目の光源(GL_LIGHT0)を有効にする(点灯する)にはglEnable(GL_LIGHT0)、無効にする(消灯する)にはglDisable(GL_LIGHT0)を実行します。
陰影付けを行うと、陰影付けを行わないより処理速度は低下します。陰影付けを有効にするにはglEnable(GL_LIGHTING)、無効にするにはglDisable(GL_LIGHTING)を実行します。

なお、陰影付けが有効になっているときは、glColor3f()などによる色指定は無視されます。glColor3f()などで色を付けたいときは、一旦glDisable(GL_LIGHTING)を実行して陰影付けを行わないようにする必要があります。一方、上のプログラムのように常に陰影付けを行う場合や、光源を点灯したままにしておく場合は、glEnable(GL_DEPTH_TEST)同様、glEnable(GL_LIGHTING)やglEnable(GL_LIGHTm)をinit()の中で一度実行するだけで十分です。また、このときはglDisable(GL_LIGHTING)やglDisable(GL_LIGHTm)を実行する必要はありません。

それでは光源を二つにして、それぞれの位置と色を変えてみましょう。最初の光源(GL_LIGHT0)の位置をZ軸方向の斜め上(0,3,5)に、二つ目の光源(GL_LIGHT1)をx軸方向の斜め上(5,3,0)に置き、二つ目の光源の色を緑(0,1,0)にします。これらのデータはいずれも四つの要素を持つfloat型の配列に格納します。四つ目の要素は1にしておいてください。

```
private final float[] light0pos = {0.0f, 3.0f, 5.0f, 1.0f};
private final float[] light1pos = {5.0f, 3.0f, 0.0f, 1.0f};
private final float[] green = {0.0f, 1.0f, 0.0f, 1.0f};
```

これらをglLightfv()を使ってそれぞれの光源に設定します。サンプルプログラムを以下のように変更してください。

```
private final float[] light0pos = {0.0f, 3.0f, 5.0f, 1.0f};//追加
private final float[] light1pos = {5.0f, 3.0f, 0.0f, 1.0f};//追加
private final float[] green = {0.0f, 1.0f, 0.0f, 1.0f};//追加
@Override
public void init(GLAutoDrawable drawable) {
        GL2 gl = drawable.getGL().getGL2();
        //背景を白く塗りつぶす.
        gl.glClearColor(1f, 1f, 1f, 1.0f);
        gl.glEnable(GL_DEPTH_TEST);
        gl.glEnable(GL_CULL_FACE);
        gl.glCullFace(GL_FRONT);
        gl.glEnable(GL_LIGHTING);
        gl.glEnable(GL_LIGHT0);
        gl.glEnable(GL_LIGHT1);
        gl.glLightfv(GL_LIGHT1, GL_DIFFUSE, green, 0);//追加
        gl.glLightfv(GL_LIGHT1, GL_SPECULAR, green, 0);//追加
}
@Override
public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
        GL2 gl = drawable.getGL().getGL2();
        gl.glMatrixMode(GL_PROJECTION);
        gl.glLoadIdentity();
        glu.gluPerspective(30.0, (double)width / (double)height, 1.0, 300.0);

        //視点位置と視線方向
        //glu.gluLookAt(3.0f, 4.0f, 5.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);//コメントアウト

        gl.glMatrixMode(GL_MODELVIEW);
}
@Override
public void display(GLAutoDrawable drawable) {
        GL2 gl = drawable.getGL().getGL2();
        gl.glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //GL_DEPTH_BUFFER_BITを追加
        gl.glLoadIdentity();
        glu.gluLookAt(3.0f, 4.0f, 5.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);//追加

        // 光源の位置設定
        gl.glLightfv(GL_LIGHT0, GL_POSITION, light0pos, 0);//追加
        gl.glLightfv(GL_LIGHT1, GL_POSITION, light1pos, 0);//追加
        //以降省略
}
```

void glLightfv(int light, int pname, float[] params)
光源のパラメータを設定します。最初の引数lightには設定する光源の番号(GL_LIGHT0～GL_LIGHT**n**、**n**はシステムによって異なります)です。二つ目の引数pnameは設定するパラメータの種類です。最後の引数paramsは、pnameに指定したパラメータの種類に設定する値です。
pnameとparamsに指定するパラメータの組み合わせの一部を以下の表に示します。

pname	params
GL_POSITION	**光源の位置を設定。x,y,z,wの4要素。wが0なら光源からの指向性のある平行光線(例：太陽光)で、0以外なら点光源(例：スポットライト、街灯) 点光源の場合、光源からの距離に応じてどのように減衰するかを設定できる。**
GL_DIFFUSE	**拡散光(特定の方向からの面光源。ワールド内の物体の、光源に面している側だけが照らされる)**の色を設定する。r,g,bの3要素+4番目の固定値1。
GL_SPECULAR	鏡面光(特定の方向からの光束。物体の一部だけを照らす)の色を設定する。r,g,bの3要素+4番目の固定値1。
GL_AMBIENT	環境光(ワールド内を均一に照らす、どこから照らされているかわからないような光)の色を設定する。r,g,bの3要素+4番目の固定値1。

なお、上記のr,g,bの3要素にはint型、float型、double型を指定できます。

上の表で拡散光、鏡面光、環境光の３つを挙げたのは、CGでは、この３つの光源を使うことにより、現実世界での光を近似できると言われているからのようです。

陰影付けの計算はワールド座標系で行われるので、glLightfv()による光源の位置(GL_POSITION)の設定は、**視点の位置を設定した後に行う**必要があります。また、上のプログラムのglRotate3d()より後でこれを設定すると、光源もいっしょに回転してしまいます。

- 座標変換のプロセスは"モデリング変換→ビューイング変換→透視変換→・・・"という順に行われると書きましたが、プログラムのコーディング上は、これらの設定が**逆順になる**ことに注意してください。
  1. glLoadIdentity()でモデルビュー変換行列を初期化
  2. gluLookAt()等でビューイング変換を設定
  3. glTranslated()やglRotated()等でモデリング変換を設定
  4. glBegin()～glEnd()等による描画
- 1-2の間で光源の位置を設定した場合は、光源は視点と一緒に移動します。このとき、光源の方向を(0、0、-1、0)、すなわちZ軸の負の方向に設定すれば、自動車のヘッドライトのような効果を得ることができます。
- 2-3の間で光源の位置を設定した場合は、光源の位置は視点や図形の位置によらず固定になります。通常はここで光源の位置を設定します。
- 3-4の間で光源の位置を設定した場合は、光源の位置は図形と一緒に移動します。

glLightfv()による光源の色の設定(GL_DIFFUSE等)は、必ずしもdisplay()内に置く必要はありません。プログラムの実行中に光源の色を変更しないなら、glEnable(GL_DEPTH_TEST)やglEnable(GL_LIGHTING)同様init()の中で一度実行すれば十分です。

glLightf*()で設定可能なパラメータは、GL_POSITIONやGL_DIFFUSE以外にもたくさんあります。光源を方向を持ったスポットライトとし、その方向や広がり、減衰率なども設定することもできます。詳しくは http://wisdom.sakura.ne.jp/system/opengl/gl16.html に詳しく解説されていますので、参考としてください。

9.3材質を設定する

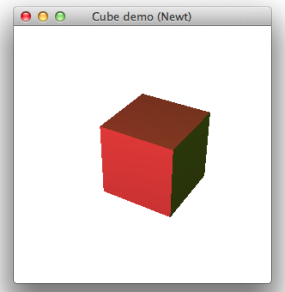
前の例では図形に色を付けていませんでしたから、立方体はデフォルトの色(白)で表示されたと思います。今度はこの色を変えてみましょう。この場合も光源の時と同様に四つの要素を持つGLfloat型の配列を用意し、個々の要素に色をR、G、BそれにAの順に格納します。四つ目の要素(A)は、ここではとりあえず1にしておいてください。

```
private final float[] red = {0.8f, 0.2f, 0.2f, 1.0f};
```

glColor*()で色を付けるときと同様、図形を描く前にglMaterialfv()を使ってこの色を図形の色に指定します。サンプルプログラムを以下のように変更してください。

```
 private final float[] red = {0.8f, 0.2f, 0.2f, 1.0f};
//追加

@Override
 public void display(GLAutoDrawable drawable) {
        //ここまで省略
        // 図形の回転
        gl.glTranslatef(0.5f, 0.5f, 0.5f);
        gl.glRotatef(r, 0.0f, 1.0f, 0.0f);
        gl.glTranslatef(-0.5f, -0.5f, -0.5f);
        // 図形の色（赤）
        gl.glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, red, 0);//追加
        // 図形の描画
        gl.glBegin(GL_QUADS);
        //以降省略
}
```

これを実行すると、次のように赤くなりました。



void glMaterialfv(imt face、imt pname、float[] params,imt offset)

glMaterialfv()は図形の材質パラメータを設定します。引数faceにはGL_FRONT、GL_BACKあるいはGL_FRONT_AND_BACKが指定でき、それぞれ面の表、裏、あるいは両面に材質パラメータを設定します。設定できる材質pnameにはGL_AMBIENT(環境光に対する反射係数)、GL_DIFFUSE(拡散反射係数)、GL_SPECULAR(鏡面反射係数)、GL_EMISSION(発光係数)、GL_SHININESS(ハイライトの輝き)、あるいはGL_AMBIENT_AND_DIFFUSE(拡散反射係数と鏡面反射係数の両方)があります。他にインデックスカラーモード(GLUT_INDEX)であればGL_COLOR_INDEXESも使用できますが、この資料では使用していません。引数paramsは一つまたは四つの要素を持つfloat型の配列で、四つの要素を持つ場合(GL_SHININESS、GL_COLOR_INDEXES以外)は、色の成分RGBおよび透明度Aに対する係数を指定します。この初期値は(0.8f,0.8f,0.8f,1.0f)ですが、1を越える値も設定できます。offset(JOGLで追加された引数)は0を指定します。

void glMaterialfv(int face,int pname, FloatBuffer params)

3つめの引数がFloatBufferである点以外は、float配列を引数とするメソッドと同じです。なお、こちらの場合４つめのoffsetは不要です。

図形に色を付けるということは、図形の物理的な材質パラメータを設定することに他なりません。**GL_DIFFUSEで設定する拡散反射係数が図形の色**に相当します。GL_AMBIENTは環境光(光源以外からの光)に対する反射係数で、光の当たらない部分の明るさに影響を与えます。GL_DIFFUSEとGL_AMBIENTには同じ値を設定することが多いので、これらを同時に設定するGL_AMBIENT_AND_DIFFUSEが用意されています。

GL_SPECULARは光源に対する鏡面反射係数で、図形表面の光源の映り込み(ハイライト)の強さです。GL_SHININESSはこの鏡面反射の細さを示し、大きいほどハイライトの部分が小さくなります。この材質パラメータの要素は一つだけなので、glMaterialf()を使って設定することもできます。

GL_DIFFUSE以外のパラメータを設定することによって、図形の質感を制御できます。たとえばGL_SPECULAR(鏡面反射係数)を白(1111)に設定してGL_SHININESSを大きく(10～40とか／最大128)すればつややかなプラスチックのようになりますし、GL_SPECULAR(鏡面反射係数)をGL_DIFFUSEと同じにしてGL_AMBIENTを0に近づければ金属的な質感になります。ただしGL_SPECULARやGL_AMBIENTを操作するときは、glLightfv()で光源のこれらのパラメータも設定してやる必要があります。

10.階層構造

次に図形の階層構造を表現してみます。ここでの階層構造とは、例えばロボットの体のように、胴体や手足などが、肩や肘、手首などの複数の関節で繋がれている構造のことです。このような構造では、手は手首の動きに追従して動き、肘から先は肘の動きに追従し、さらに二の腕は肩の動きに追従するというように、連鎖的な構造になっています。

ここでは、手のひらと指を例として階層化してみます。

FingersSampleNewt.java、Palm.java、Finger.javaいうファイルを、それぞれ以下の内容で作成してください。

```
package demos.fingers;

import static com.jogamp.opengl.GL.GL_COLOR_BUFFER_BIT;
import java.nio.IntBuffer;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.KeyAdapter;
import com.jogamp.newt.event.KeyEvent;
import com.jogamp.newt.event.MouseAdapter;
import com.jogamp.newt.event.MouseEvent;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.Animator;

public class FingersSampleNewt implements GLEventListener {
        protected GLCapabilities caps;
        private final Animator animator;

        private float viewScale = 0.04f;
        private int prevMouseX = -1;

        // フィンガー
        private static final int FINGERS_COUNT = 3*5;
        private static Finger[] fingers;
        private Palm palm;
        private int fingerNumber = 1;
```

```
	private final float[] red = {1f, 0f, 0f};
	private final float[] green = {0f, 1f, 0f};
	private final float[] blue = {0f, 0f, 1f};
	private final float[] orange = {1f, 0f, 1f};

	public static void main(String[] args) {
		System.out.println("指をドラッグして角度を変えられます。第一関節(指先)を操作するにはコントロールキーを、第二関節を操作するにはシフトキーを押しながらドラッグしてください。");
		System.out.println("操作する指を変えるには、１(小指)から5(親指)を押してください。");
		new FingersSampleNewt();
	}

	public FingersSampleNewt() {
		GLProfile prof = GLProfile.get(GLProfile.GL2);
		caps = new GLCapabilities(prof);

		setupFingers();

		GLWindow glWindow = GLWindow.create(caps);
		glWindow.setTitle("Finger demo (Newt)");
		glWindow.setSize(500, 500);
		glWindow.addGLEventListener(this);

		glWindow.addWindowListener(new WindowAdapter() {
			@Override
			public void windowDestroyed(WindowEvent evt) {
				quit();
			}
		});

		glWindow.addMouseListener(new MouseAdapter() {
			@Override
			public void mouseReleased(MouseEvent e) {
				prevMouseX = -1;
			}

			@Override
			public void mouseDragged(MouseEvent evt) {
				int x = evt.getX() ;
				if(prevMouseX != -1) {
					float rotDelta = (prevMouseX - x);
					int fingerIndex = (fingerNumber -1) * 3;
					if(evt.isControlDown()) {
						fingers[2 + fingerIndex].updateRotation(rotDelta);
					} else if(evt.isShiftDown()) {
						fingers[1 + fingerIndex].updateRotation(rotDelta);
					} else {
						fingers[0 + fingerIndex].updateRotation(rotDelta);
					}
				}
				// 現在のマウスの位置を保存
				prevMouseX = x;
			}

			@Override
			public void mouseWheelMoved(MouseEvent e) {
				float[] rot = e.getRotation();
				viewScale *= (rot[1] > 0 ? 1.005f : 0.995f);
			}
		});

		glWindow.addKeyListener(new KeyAdapter() {
			@Override
			public void keyPressed(KeyEvent key) {
				switch (key.getKeyChar()) {
					case KeyEvent.VK_ESCAPE:
					quit();
					break;

					case 'q':
					quit();
					break;
					case '1':
					fingerNumber = 1;
					System.out.println("1:小指");
					break;
					case '2':
					fingerNumber = 2;
					System.out.println("2:薬指");
					break;
					case '3':
					fingerNumber = 3;
					System.out.println("3:中指");
					break;
					case '4':
					fingerNumber = 4;
					System.out.println("4:人差し指");
					break;
					case '5':
					fingerNumber = 5;
					System.out.println("5:親指");
					break;
					default:
					break;
				}
			}
		});

		glWindow.setVisible(true);
		animator = new Animator(glWindow);
		animator.start();
	}

	private void setupFingers() {
		fingers = new Finger[FINGERS_COUNT];

		palm = new Palm(10, 0, -20, orange);

		//positionは指の根元の場合に使う。二番目以降については
		//親のアンカーの位置に合わせる。

		// 小指の設定
		fingers[0] = new Finger(-9.0f, 10f, 7f, 0f, red);
		fingers[1] = new Finger(fingers[0], 7f, 0f, blue);
		fingers[2] = new Finger(fingers[1], 5f, 0f, green);

		// 薬指の設定
		fingers[3] = new Finger(-4.5f, 10f, 8f, 0f, red);
		fingers[4] = new Finger(fingers[3], 8f, 0f, blue);
		fingers[5] = new Finger(fingers[4], 7f, 0f, green);

		//中指の設定
		fingers[6] = new Finger(0.0f, 10f, 9f, 0f, red);
```

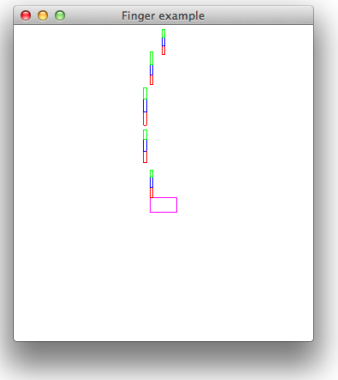
```
                fingers[7] = new Finger(fingers[6], 9f, 0f, blue);
                fingers[8] = new Finger(fingers[7], 8f, 0f, green);

                // 人差し指の設定
                fingers[9] = new Finger(4.5f, 10f, 7f, 0f, red);
                fingers[10] = new Finger(fingers[9], 7f, 0f, blue);
                fingers[11] = new Finger(fingers[10], 9f, 0f, green);

                // 親指の設定
                fingers[12] = new Finger(9.0f, 7f, 6f, 0f, red);
                fingers[13] = new Finger(fingers[12], 6f, 0f, blue);
                fingers[14] = new Finger(fingers[13], 5f, 0f, green);
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                GL2 gl2 = drawable.getGL().getGL2();
                gl2.glClearColor(1f, 1f, 1f, 1f);
                gl2.glClearDepth(1.0f);
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {}

        @Override
        public void display(GLAutoDrawable drawable) {
                GL2 gl2 = drawable.getGL().getGL2();
                gl2.glClear(GL_COLOR_BUFFER_BIT);
                gl2.glLoadIdentity();
                gl2.glScalef(viewScale, viewScale, viewScale); //(1)

                //手のひらを描く
                palm.render(gl2);

                //指を描く
                for (int i = 0; i < FINGERS_COUNT; i++) {
                        fingers[i].render(gl2);
                }
        }

        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(animator != null) animator.stop();
        }

        private void quit() {
                animator.stop();
                System.exit(0);
        }
}
```

```
package demos.fingers;

import com.jogamp.opengl.GL2;

public class Palm {
        private static final float[][] palmVertex = {{-1f, 0f}, {+1f, 0f}, {+1f, 1f}, {-1f, 1f}};

        private final float positionX;
        private final float positionY;
        private final float size;
        private final float[] color;

        public Palm(float size, int x, int y, float[] color) {
                this.size = size;
                this.positionX = x;
                this.positionY = y;
                this.color = color;
        }

        // 手のひら？を描画する
        protected void render(GL2 gl2) {
                gl2.glTranslatef(positionX, positionY, 0f);

                gl2.glPushMatrix();
                gl2.glScalef(size, size, 1f);
                gl2.glColor3fv(color, 0);
                gl2.glBegin(GL2.GL_LINE_LOOP);
                for (int i = 0; i < palmVertex.length; i++) {
                        gl2.glVertex3fv(palmVertex[i], 0);
                }
                gl2.glEnd();

                gl2.glPopMatrix();
        }
}
```

```
package demos.fingers;

import com.jogamp.opengl.GL2;

public class Finger {
        private static final float[][] fingerVertex = {{-1f, 0f}, {+1f, 0f}, {+1f, 1f}, {-1f, 1f}};
        private final float positionX;
        private final float positionY;
        private final float length;
        private float rotationAngle;
        private final float[] color;

        public Finger(Finger parent, float length, float rotationAngle, float[] color) {
                this.positionX = parent.getJointX();
                this.positionY = parent.getJointY();
                this.length = length;
                this.rotationAngle = rotationAngle;
                this.color = color;
        }

        public Finger(float x, float y, float length, float rotationAngle, float[] color) {
                this.positionX = x;
                this.positionY = y;
                this.length = length;
                this.rotationAngle = rotationAngle;
                this.color = color;
        }

        float getJointY() {
                return length;
        }
```

```
        float getJointX() {
                return 0;
        }

        public void updateRotation(float angle) {
                this.rotationAngle += angle;
        }

        // 指を描画する
        protected void render(GL2 gl2) {
                gl2.glTranslatef(positionX, positionY, 0f);
                gl2.glRotatef(rotationAngle, 0f, 0f, 1f);

                gl2.glPushMatrix();
                gl2.glScalef(1f, length, 1f);
                gl2.glColor3fv(color, 0);
                gl2.glBegin(GL2.GL_LINE_LOOP);
                for (int i = 0; i < fingerVertex.length; i++) {
                        gl2.glVertex3fv(fingerVertex[i], 0);
                }
                gl2.glEnd();

                gl2.glPopMatrix();
        }
}
```

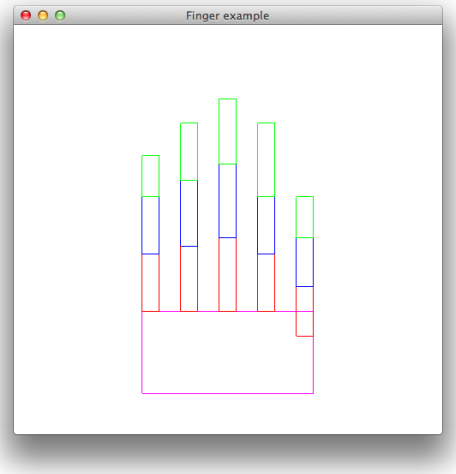
FingersSampleNewt.javaを実行すると、以下のようなウィンドウになります。指が離れてしまっていますね。何が起きたのでしょう？



FingersSampleNewt.javaのdisplay()メソッドを以下のように直しましょう。

```
@Override
public void display(GLAutoDrawable drawable) {
        //(省略)
        //指を描く
        //for (int i = 0; i < FINGERS_COUNT; i++) { //削除
                // fingers[i].render(gl2); //削除
        //} //削除

        //ここから追加
        gl2.glPushMatrix(); //(1)
        fingers[0].render(gl2);
        fingers[1].render(gl2);
        fingers[2].render(gl2);
        gl2.glPopMatrix(); //(2)

        gl2.glPushMatrix(); //(3)
        fingers[3].render(gl2);
        fingers[4].render(gl2);
        fingers[5].render(gl2);
        gl2.glPopMatrix(); //(4)

        gl2.glPushMatrix(); //(5)
        fingers[6].render(gl2);
        fingers[7].render(gl2);
        fingers[8].render(gl2);
        gl2.glPopMatrix(); //(6)

        gl2.glPushMatrix(); //(7)
        fingers[9].render(gl2);
        fingers[10].render(gl2);
        fingers[11].render(gl2);
        gl2.glPopMatrix(); //(8)

        gl2.glPushMatrix(); //(9)
        fingers[12].render(gl2);
        fingers[13].render(gl2);
        fingers[14].render(gl2);
        gl2.glPopMatrix(); //(10)
        //ここまでを追加
}
```

これを実行すると次のようなウィンドウになります。これでも手のひらのつもりです。



glScalef(float x、float y、float z)
変換行列にスケーリング係数を乗じます。引数はいずれもfloat型で、三つの引数x、y、zには現在のスケールに対する相対的なスケール係数を指定します。x、y、z軸それぞれ独立して係数を指定します。
引数がdouble型ならglScaled()を使います。

ウィンドウ上でマウスを左右にドラッグしてみてください。いちばん左の指（小指のつもりです）の角度が根元で変わります。
シフトキーを押しながらドラッグすると、小指の第二関節(指先から２番目）の角度が変わり、コントロールキーを押しながらだと第一関節（指先に一番近い）の角度が変わります。また、１から５の数字を押すと、ドラッグ操作の対象が、１から順に小指、薬指、中指、人差し指、親指に切り替わります。
上の修正を行う前は、小指から順に図形を描いていたので、小指は正常に描かれていましたが、薬指から後はglTranslatef()による変換行列への変更が蓄積され、おかしな位置に描画されていたのでした。
これを防ぐために、(1)(3)(5)(7)(9)のglPushMatrix()で、それまでのglTranslatef()、glRotatef()、glScalef()（これらを総称してtranformと呼ぶことがあります）が適用された変換行列を待避(push)しておき、(2)(4)(6)(8)(10)のglPopMatrix()で復元(pop)するようにしたため、薬指から後も正常に描かれるようになったのです。

上の例では、まず手のひらを描き、小指の根元から指先まで順に変換行列をtransformしながら描いていきますが、次に薬指を描くときに使う変換行列は小指の根元を描くときに使ったのと同じで構いません。そこで、小指を描く前に待避(push)しておいて、小指の描画が終わったら復元(pop)して再利用するようにします。中指以降も同様です。

なお、glPushMatrix()とglPopMatrix()の操作は、これを実行する前にglMatrixMode(GL_MODELVIEW)を実行していた場合、ModelView行列が対象となり、glMatrixMode(GL_PROJECTION)を実行していた場合、Projection行列が対象となります。
glMatrixMode()を一度も実行しないと、GL_MODELVIEWが指定されていることになります。
glMatrixMode()に指定できるパラメータには､GL_MODELVIEW、GL_PROJECTION以外に、GL_TEXTUREがありますが、これらについては割愛します。

push、popの操作で使われるデータ構造はスタック、あるいはLIFO(先入れ後出し)と呼ばれています。こちらの解説がわかりやすいと思います。
スタックのサイズ、つまり、glPushMatrix()できる回数には、制限があります。OpenGLの仕様では、GL_MODELVIEWは32、GL_PROJECTIONは２が必須となっていますが、OpenGL実装によってはこれ以上の操作が許されており、以下のコードで問い合わせることができます。

```
int[] maxStackSize = new int[1];
gl.glGetIntegerv(GL2.GL_MODELVIEW_STACK_DEPTH, maxStackSize, 0);//GL_PROJECTION_STACK_DEPTHあるいはGL_TEXTURE_STACK_DEPTH
System.out.println(maxStackSize[0]);
```

Pushの操作を、許容された回数以上に行うと、その操作は無視され、エラーとなります。Popの操作についても、Pushの操作ときちんと対応している必要があり、Pushした回数を超えるとエラーとなります。OpenGLのエラーについては13章で説明します。
既出のglLoadIdentity()の場合、変換行列を初期化してしまいます。使い分ける基準としては、それまでに行った変換行列への変更を無効化しても構わない場合はこれを使い、そうではない場合pushしておいたのをpopして使うことになります。
同様の操作としては、glPushAttributes()とglPopAttributes()の組み合わせがありますが、ここでは詳細な説明は割愛します。

11.テクスチャ

以下の内容のファイルをTextureSample.javaという名前で作ります。

```
package demos.texture;

import java.io.IOException;
import java.io.InputStream;
import com.jogamp.opengl.DebugGL2;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLException;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.KeyAdapter;
import com.jogamp.newt.event.KeyEvent;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.Animator;
import com.jogamp.opengl.util.texture.Texture;
import com.jogamp.opengl.util.texture.TextureIO;

public class TextureSample implements GLEventListener {
        private static final String IMAGE_FILE = "nehe.png"; //(1)
        private final Animator animator;
        private Texture texture; //(2)

        public static void main(String[] args) {
                new TextureSample();
        }

        public TextureSample() {
                GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
                GLWindow glWindow = GLWindow.create(caps);

                glWindow.setTitle("Texture sample (Newt)");
                glWindow.setSize(500, 500);
                glWindow.addGLEventListener(this);

                glWindow.addWindowListener(new WindowAdapter() {
                        @Override
```

```
                        public void windowDestroyed(WindowEvent evt) {
                                quit();
                        }
                });

                glWindow.addKeyListener(new KeyAdapter() {
                        @Override
                        public void keyPressed(KeyEvent key) {
                                switch (key.getKeyChar()) {
                                        case KeyEvent.VK_ESCAPE:
                                        quit();
                                        break;

                                        case 'q':
                                        quit();
                                        break;
                                        default:
                                        break;
                                }
                        }
                });

                glWindow.setVisible(true);
                animator = new Animator(glWindow);
                animator.start();
        }

        @Override
        public void init(GLAutoDrawable drawable) {
                final GL2 gl2 = drawable.getGL().getGL2();
                drawable.setGL(new DebugGL2(gl2));
                gl2.glClearColor(1f, 1f, 1f, 1f);

                try {
                        InputStream resourceStream = this.getClass().getResourceAsStream(IMAGE_FILE);     //(3)
                        texture = TextureIO.newTexture(resourceStream, false, TextureIO.PNG); //(4)
                } catch (GLException | IOException e) {
                        e.printStackTrace();
                }
        }

        @Override
        public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) { }

        @Override
        public void display(GLAutoDrawable drawable) {
                final GL2 gl2 = drawable.getGL().getGL2();
                gl2.glClear(GL2.GL_COLOR_BUFFER_BIT);
                gl2.glLoadIdentity();
                gl2.glScalef(0.9f, 0.9f, 0.9f);

                texture.enable(gl2); //(5)

                gl2.glBegin(GL2.GL_QUADS);
                gl2.glTexCoord2f(0.0f, 0.0f); //(6)
                gl2.glVertex2f(-1.0f, -1.0f);
                gl2.glTexCoord2f(1.0f, 0.0f);
                gl2.glVertex2f(1.0f, -1.0f);
                gl2.glTexCoord2f(1.0f, 1.0f);
                gl2.glVertex2f(1.0f, 1.0f);
                gl2.glTexCoord2f(0.0f, 1.0f);
                gl2.glVertex2f(-1.0f, 1.0f);
                gl2.glEnd();
        }

        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(texture != null) {
                        final GL2 gl2 = drawable.getGL().getGL2();
                        texture.destroy(gl2); //(7)
                        if(animator != null) animator.stop();
                }
        }

        private void quit() {
                animator.stop();
                System.exit(0);
        }
}
```

そして以下の画像をTextureSample.javaと同じフォルダーに"nehe.png"というファイル名で格納しておきます。

これを忘れたり、ファイル名が違っていたり、誤って別のフォルダーに置くと以下のような例外が表示されます。サンプルプログラムの(4)のところで、画像ファイルが見つからないためです。

```
java.io.IOException: Stream was null
at com.jogamp.opengl.util.texture.TextureIO.newTextureDataImpl(TextureIO.java:834)
at com.jogamp.opengl.util.texture.TextureIO.newTextureData(TextureIO.java:246)
at com.jogamp.opengl.util.texture.TextureIO.newTexture(TextureIO.java:506)
at demos.texture.TextureSample.init(TextureSample.java:77)
(以下省略)
```

これを実行すると、以下のようなウィンドウが表示されます。

(1)で、上記の画像ファイル名を指定し、(2)でcom.jogamp.opengl.util.texture.Textureクラスのインスタンス変数textureを宣言しています。
(3)で、(4)の引数とするInputStreamクラスの変数resourceStreamを、(1)のファイルを引数として作成しています。
(4)で、(2)で宣言したtexture変数に画像ファイルから読み込んだ内容をセットしています。これで、OpenGLで使うテクスチャの準備ができました。

com.jogamp.opengl.util.texture.TextureIO.newTexture(java.io.InputStream stream, boolean mipmap, String imageType)
指定されたstreamからOpenGLのテクスチャを用意します。mipmap引数はミップマップと呼ばれる、スケールの異なる複数のテクスチャを用意するか否かを指示します。画像ファイルフォーマットによってはミップマップをサポートしていて、ファイル内に格納されている場合もあり、このような場合にはそれが使われ、なければJOGL内部で自動生成します。ミップマップは、LOD(Level Of Detals)と呼ばれる、対象の物体がスクリーン内でどの程度の大きさに見えるかに応じてテクスチャを切り替える際に使います。(スクリーン内で小さくしか見えない物体に対し、詳細なテクスチャを適用するのはメモリの無駄遣いなので、小さめのテクスチャが使われます)
最後のimageTypeは、画像フォーマットを指定する名前を与えます。nullを指定すると自動検出を試みます。サポートされている画像フォーマットは、JPEG,PNG,GIF,TIFFなどで、それぞれ上記のimageTypeに`TextureIO.JPG`,`TextureIO.PNG`,`TextureIO.GIF`,`TextureIO.TIFF`を指定します。

なお、(3)(4)はinit()内で実行していますが、GLインスタンスを必要としないので、コンストラクターなど、他で実行しても構いません。

(3)は以下のようにすると、Javaのクラスファイルとは別のフォルダーに格納しておくことができます。状況に応じて使い分けるとよいでしょう。

```
String someFolder = "どこかのフォルダー";
File file = new File(someFolder, IMAGE_FILE);
InputStream resourceStream = new FileInputStream(file);
```

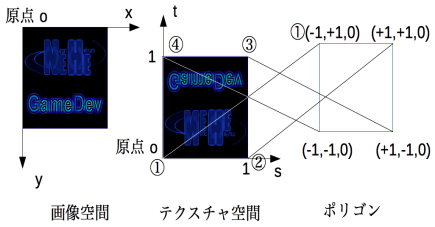
(5)から(7)で、(2)のtextureオブジェクトを使っています。(5)でテクスチャーを使うことを指定します。(6)以降でテクスチャ座標とポリゴン座標の対応を指定しています。
ウィンドウを閉じたときにdispose()が呼ばれるので、(7)でテクスチャーオブジェクトを廃棄しています。

画像ファイルの原点は、通常画像の左上となっていますが、テクスチャ座標は画像の左下を原点としています。C言語などでは、テクスチャ座標の上下を反転して扱う必要がありましたが、TextureIOを使えば内部で自動的に反転されるため、意識する必要はありません。

テクスチャ座標は、元の画像ファイルの縦・横のサイズとは無関係に、左下の(0,0)から右上の(1,1)の範囲にスケールされます。

サンプルプログラムでは、テクスチャ座標とポリゴン座標を以下のように対応させています。8.3節で、ポリゴンの頂点をたどる順が左回り(反時計回り)を表面とすると書きましたが、テクスチャ座標も同様に反時計回りとなっています。

位置	テクスチャ座標	ポリゴン座標
①	(0,0)	(-1,+1,0)
②	(1,0)	(+1,+1,0)
③	(1,1)	(+1,-1,0)
④	(0,1)	(-1,-1,0)



12.GLUTに定義済みの図形

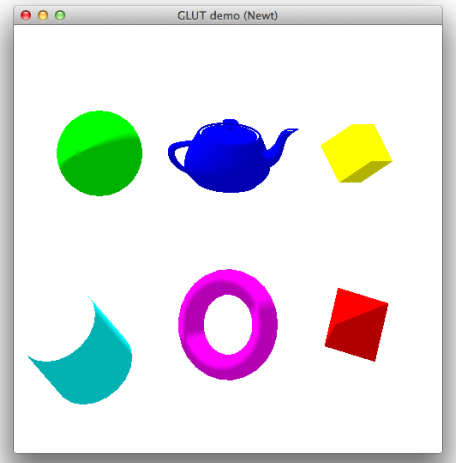
JOGLにはGLUTの機能が移植されています。GLUTで定義されている図形を描画するサンプルプログラムを以下に示します。

```
package demos.glut;

import static com.jogamp.opengl.GL.*;
import static com.jogamp.opengl.fixedfunc.GLLightingFunc.*;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.newt.event.KeyAdapter;
import com.jogamp.newt.event.KeyEvent;
import com.jogamp.newt.event.MouseAdapter;
import com.jogamp.newt.event.MouseEvent;
import com.jogamp.newt.event.WindowAdapter;
import com.jogamp.newt.event.WindowEvent;
import com.jogamp.newt.opengl.GLWindow;
import com.jogamp.opengl.util.Animator;
import com.jogamp.opengl.util.gl2.GLUT;

public class GlutSampleNewt implements GLEventListener {
        private final Animator animator;
        private final GLUT glut;

        private float viewScale = 0.02f;

        private final float[] white0 = {0.5f, 0.5f, 0.5f};
```

```
	private final float[] white1 = {1f, 1f, 1f};
	private final float[] red = {1f, 0f, 0f};
	private final float[] green = {0f, 1f, 0f};
	private final float[] blue = {0f, 0f, 1f};
	private final float[] yellow = {1f, 1f, 0f};
	private final float[] magenta = {1f, 0f, 1f};
	private final float[] cyan = {0f, 1f, 1f};;

	private final float[] light0pos = {-10f, 30.0f, 5.0f, 1.0f};
	private final float[] light1pos = {-10f, 30.0f, 5.0f, 1.0f};

	public static void main(String[] args) {
		new GlutSampleNewt();
	}

	public GlutSampleNewt() {
		GLProfile prof = GLProfile.get(GLProfile.GL2);
		glut = new GLUT();
		GLCapabilities caps = new GLCapabilities(prof);
		GLWindow glWindow = GLWindow.create(caps);
		glWindow.setTitle("GLUT demo (Newt)");
		glWindow.setSize(500, 500);
		glWindow.addGLEventListener(this);

		glWindow.addWindowListener(new WindowAdapter() {
			@Override
			public void windowDestroyed(WindowEvent evt) {
				quit();
			}
		});

		glWindow.addMouseListener(new MouseAdapter() {
			@Override
			public void mouseWheelMoved(MouseEvent e) {
				float[] rot = e.getRotation();
				viewScale *= (rot[1] > 0 ? 1.005f : 0.995f);
				System.out.println(viewScale);
			}
		});

		glWindow.addKeyListener(new KeyAdapter() {
			@Override
			public void keyPressed(KeyEvent key) {
				switch (key.getKeyChar()) {
					case KeyEvent.VK_ESCAPE:
					quit();
					break;

					case 'q':
					quit();
					break;
					default:
					break;
				}
			}
		});

		glWindow.setVisible(true);
		animator = new Animator(glWindow);
		animator.start();
	}

	@Override
	public void init(GLAutoDrawable drawable) {
		GL2 gl2 = drawable.getGL().getGL2();
		gl2.glClearColor(1, 1, 1, 1);
		gl2.glClearDepth(1.0f);
		gl2.glEnable(GL_DEPTH_TEST);
		gl2.glEnable(GL_CULL_FACE);
		gl2.glCullFace(GL_FRONT);
		gl2.glEnable(GL_LIGHTING);
		gl2.glEnable(GL_LIGHT0);
		gl2.glLightfv(GL_LIGHT0, GL_DIFFUSE, white0, 0);
		gl2.glLightfv(GL_LIGHT0, GL_POSITION, light0pos, 0);
		gl2.glEnable(GL_LIGHT1);
		gl2.glLightfv(GL_LIGHT1, GL_SPECULAR, white0, 0);
		gl2.glLightfv(GL_LIGHT1, GL_POSITION, light1pos, 0);
		gl2.glEnable(GL_LIGHT2);
		gl2.glLightfv(GL_LIGHT2, GL_AMBIENT, white0, 0);
	}

	@Override
	public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {}

	@Override
	public void display(GLAutoDrawable drawable) {
		GL2 gl2 = drawable.getGL().getGL2();
		gl2.glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
		gl2.glLoadIdentity();
		gl2.glScalef(viewScale, viewScale, viewScale);

		gl2.glPushMatrix();
		gl2.glTranslatef(-30f, -20, 0);
		gl2.glRotatef(30f, 0f, 1f, 0f);
		gl2.glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, green, 0);
		glut.glutSolidSphere(10, 30, 30); //(1)球
		gl2.glPopMatrix();

		gl2.glPushMatrix();
		gl2.glTranslatef(0f, -20, 0);
		gl2.glRotatef(30f, 1f, 1f, 0f);
		gl2.glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, blue, 0);
		glut.glutSolidTeapot(10); //(2)ティーポット
		gl2.glPopMatrix();

		gl2.glPushMatrix();
		gl2.glTranslatef(30f, -20, 0);
		gl2.glRotatef(30f, 0f, 1f, 0f);
		gl2.glRotatef(30f, 0f, 0f, 1f);
		gl2.glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, yellow, 0);
		glut.glutSolidCube(10); //(3)立方体
		gl2.glPopMatrix();

		gl2.glPushMatrix();
		gl2.glTranslatef(-40f, 20, 0);
		gl2.glRotatef(30f, 0f, 1f, 0f);
		gl2.glRotatef(30f, 1f, 0f, 0f);
		gl2.glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, cyan, 0);
		glut.glutSolidCylinder(10, 20, 20, 10); //(4)円筒
		gl2.glPopMatrix();

		gl2.glPushMatrix();
```

```
                gl2.glTranslatef(0f, 20, 0);
                gl2.glRotatef(30f, 0f, 1f, 0f);
                gl2.glRotatef(30f, 0f, 0f, 1f);
                gl2.glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, magenta, 0);
                glut.glutSolidTorus(3, 10, 15, 25); //(5)トーラス
                gl2.glPopMatrix();

                gl2.glPushMatrix();
                gl2.glTranslatef(30f, 20, 0);
                gl2.glRotatef(30f, 0f, 1f, 0f);
                gl2.glRotatef(30f, 0f, 0f, 1f);
                gl2.glScalef(10f, 10f, 10f);
                gl2.glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, red, 0);
                glut.glutSolidOctahedron(); //(6)正八面体
                gl2.glPopMatrix();
        }


        @Override
        public void dispose(GLAutoDrawable drawable) {
                if(animator != null) animator.stop();
        }

        private void quit() {
                animator.stop();
                System.exit(0);
        }
}
```

これを実行すると次のようなウィンドウが表示されます。左上から順にサンプルプログラムの(1)から(6)により描かれています。
青いのはUtahTeapotと呼ばれていて、コンピューターグラフィックスの黎明期から今までずっと使われている、由緒正しいものなのだそうです。



13.OpenGLのプロファイル

どれでもいいので、サンプルプログラムに以下のように追記して実行してみてください。

```
@Override
public void init(GLAutoDrawable drawable) {
        //省略
        showGLInfo(drawable);
}
private static void showGLInfo(GLAutoDrawable drawable) {
        System.err.println("利用可能なプロファイルのリスト");
        for(String prof : GLProfile.GL_PROFILE_LIST_ALL) {
                System.err.println(prof);
        }
        System.err.println();
        System.err.println("選択されたGLCapabilities: " + drawable.getChosenGLCapabilities());
        GL gl = drawable.getGL();
        System.err.println("INIT GL: " + gl.getClass().getName());
        System.err.println("GL_VENDOR: " + gl.glGetString(GL.GL_VENDOR));
        System.err.println("GL_RENDERER: " + gl.glGetString(GL.GL_RENDERER));
        System.err.println("GL_VERSION: " + gl.glGetString(GL.GL_VERSION));
}
```

すると、以下のようなメッセージがコンソールに表示されると思います。
以下は、Mac Book Air(2012 mid model、OSはMarvericks)での実行結果です。

```
利用可能なプロファイルのリスト
GL4bc
GL3bc
GL2
GL4
GL3
GLES3
GL4ES3
GL2GL3
GLES2
GL2ES2
GLES1
GL2ES1
選択されたGLCapabilities: GLCaps[rgba 8/8/8/0, opaque, accum-rgba 0/0/0/0, dp/st/ms 16/0/0, dbl, mono , hw, GLProfile[GL4/GL4.hw], offscr[fbo]]
INIT GL: jogamp.opengl.gl4.GL4bcImpl
GL_VENDOR: Intel Inc.
GL_RENDERER: Intel HD Graphics 4000 OpenGL Engine
GL_VERSION: 4.1 INTEL-8.28.33
```

OpenGLは、歴史的経緯により、次々と機能が拡張されてきました。OpenGLの規格を制定しているKhronos Groupでは、最近のバージョン3.1を策定したときに、過去との互換性を切り捨てて、固定機能パイプラインへのサポートを廃止し、GLSLなどの新しい機能だけを使えるようにしました。この方針は、開発者の猛反対にあったため、過去のバージョンとの互換性を維持するために、プロファイルという概念を導入し、これによって古い機能を指定して使えるようにしました。JOGLにもこの方針が反映されており、GLProfileクラスとして定義されています。

この文書では、一貫して、

```
GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
```

としていましたが、これはOpenGL2.0を使うように設定していたのです。ここには、以下のように

```
GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL3));
```

```
GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL4));
```

```
GLCapabilities caps = new GLCapabilities(GLProfile.getDefault());
```

などを指定できます。上記のリストは、プロファイルとして指定できる値を列挙していたのです。
ここで注意すべき点が一つあります。最後のサンプルプログラム(FingersSampleNewt.java)の

```
GLCapabilities caps = new GLCapabilities(GLProfile.get(GLProfile.GL2));
```

の"GL2"を"GL3"に変えてみてください。コンパイルは出来ますが、実行すると以下のようなメッセージが表示され正常に動きません。

```
Exception in thread "main" java.lang.RuntimeException: com.jogamp.opengl.GLException: Caught GLException: Not a GL2 implementation on thread
main-Display-.macosx_nil-1-EDT-1
```

これを修正するためには、init(),reshape(),display(),dispose()の各メソッド内で

```
GL2 gl2 = drawable.getGL().getGL2();
```

としているところを、

```
GL3 gl3 = drawable.getGL().getGL3();
```

のように変える必要があります。しかし、これだと以下のメソッドがコンパイル・エラーとなります。GL3では、これらのメソッドが前述の事情により廃止され、サポートされていないためです。対策として、固定機能パイプラインを使わず、GLSLというシェーダー言語と組み合わせてプログラムすることになります。GLSLについては、この文書では説明しません。

```
gl3.glLoadIdentity();
gl3.glScalef(viewScale, viewScale, viewScale);
gl3.glTranslatef(positionX, positionY, 0f);
gl3.glRotatef(rotationAngle, 0f, 0f, 1f);
gl3.glPushMatrix();
gl3.glColor3fv(color, 0);
gl3.glBegin(gl3.GL_LINE_LOOP);
gl3.glVertex3fv(fingerVertex[i], 0);
gl3.glEnd();
gl3.glPopMatrix();
```

なお、これ以外にも以下を指定できます。

```
GLCapabilities caps = new GLCapabilities(GLProfile.getMaxFixedFunc(true));
```

GLProfile.getMaxFixedFunction()、GLProfile.getDefault()を指定した場合にどのバージョンが使われるかはハードウェアやOSに依存し、上記のようにエラーとなる可能性があるため、バージョンを指定する方が望ましいでしょう。

14.デバッグ

(1)エラーの原因を表示する。以下により、OpenGLへの呼び出しの後に毎回エラーが起きていないかをチェックし、何かあったらエラーを標準出力に出すようになります。

```
@Override
public void init(GLAutoDrawable drawable) {
        GL2 gl2 = drawable.getGL().getGL2();
        drawable.setGL(new DebugGL2(gl2));
        //以降省略
}
```

(2)OpenGLへの呼び出しをトレースする。以下により、OpenGL呼び出しの度に、呼び出されたOpenGLのメソッドが、パラメータと共に、指定したファイルに出力されます。

```
private final GLPrintStream out;
//コンストラクタ
public SomeClass {
        try {
                out = new GLPrintStream("gltrace.txt");
        } catch (IOException e) {
                e.printStackTrace();
        }
}
@Override
public void init(GLAutoDrawable drawable) {
        GL2 gl2 = drawable.getGL().getGL2();
        drawable.setGL(newTraceGL2(gl2, out));
        //以降省略
}
```

(3)OpenGLに慣れていないうちは、プログラムを実行してもウィンドウ内に何も見えないことがあったりします。以下に主な原因と対策を示します。

原因	対策
頂点の位置が異常なため、意図したポリゴンになっていない。ポリゴンの各点が重なっていたり、三角形のつもりが 細い線になっている。	地道に各頂点の座標を調べる。
頂点の順番がおかしいため、物体の面の表裏が逆になり、カリングによって見えなくなっている。	glEnable(CULL_FACE)を止めてみる。
カメラの向きと物体のある位置が一致していない。 意外とありがちなのが、カメラが物体の中に潜り込んでいるケース。 平面上に描いた図形を真横から見ているため、線しか見えない場合もある。	カメラの位置を変えてみる。 視錐台を大きくしてみる。 物体の全ての頂点がカメラの視野の中に納まるか、全ての頂点の座標と視錐台の包含関係をチェックする。
glProjectionMatrix()によるMODELVIEWとPROJECTIONを忘れている。あるいは、変換行列に異常な値を設定している。	これらを正しく設定する。
glScalef()に渡す変数の初期化を忘れていて、０になっている。	初期化を追加。

スケールが大きすぎる、あるいは小さすぎる。	スケールを変えてみる。あるいは、スケールがマウスドラッグによって変わるように実装し、ドラッグしてみる。(5.6 節を参照)
物体の色や、照明、材質が正しく設定されていない。 あるいは、背景色と同じ色になっている。	これらを変えてみる。

15.サンプルコード

この文書内で使われているJavaのソースコードは、こちら(GitHub)からダウンロードできます。

JOGLのサンプルは、以下が参考になります。

- Jogamp.orgに用意されているサンプルコード (バイナリファイルへのリンク)
- Jogamp.orgの"Jogl Tutorial"
- Yet Another Tutorial on JOGL 2.1 Including Nehe JOGL Port これは冒頭で説明したパッケージ名の修正が必要です。
- JOGLのソースコードのアーカイブ(jogl-v2.*.*.tar.xz)の、src/testの下に、JUnitによるテストコードがあります。

16. リファレンス

- Jogamp.orgここの右上にフォーラムへのリンクがあり、英語での質疑応答ができます。
- Jogamp.orgにあるJOGLのJavaDoc
- JOGLのJavaDocのアーカイブ(jogl-javadoc.7z)
- JOGLのソースコードのアーカイブ(jogl-v2.*.*.tar.xz)　同じものがここからダウンロードしたアーカイブにもjocl-java-src.zipというファイル名で含まれています。
- stackoverflow.comにはJOGLというタグでの質疑応答があります。JOGLの開発者の一人のgouessejという方がアクティブに回答されています。
- 書籍「Javaによる3DCG入門」著作:山口 泰、出版：朝倉書店、出版：2015年5月

17.ライセンス

license.txtを見てください。