平成 19 年度 機械情報工学科演習
コンピュータグラフィクス (1)
三次元コンピュータグラフィクスと OpenGL

担当：谷川 智洋 講師，西村邦裕 助教
TA：仲野 潤一，山口 真弘，藤澤 順也

2007 年 12 月 4 日

1 演習の目的

本演習では，3 回にわたり，C 言語と OpenGL ライブラリを用いたプログラムの作成を通して，三次元コンピュータグラフィクス (3DCG) の理解と応用について学ぶ．

まず，演習「シミュレーション」で使用するシミュレーション環境の基礎を理解する．3DCG のプログラミングには，様々なプラットフォームで使用されている OpenGL ライブラリを使用する．GUI の実装には，GLUT という OpenGL 支援用ライブラリと GLUT にボタンなどのコントロールを提供する GLUI というライブラリを使用する．また，演習「画像処理」を踏まえ，EyeToy でキャプチャした画像をテクスチャとして貼り付ける．最後に，インタフェースとして Wii リモコンを利用して，三次元コンピュータグラフィックスとのインタラクションについて学習する．

1.1 コンピュータグラフィクスの演習の予定

演習は以下のスケジュールで行う．第 1 回目では，主に 3DCG プログラムの基礎となる座標変換の学習を中心におこなう．第 2 回目では，3DCG プログラムに GUI を付け加え，インタラクティブなプログラムの作成を目指す．また，取り込んだカメラ画像をテクスチャに貼り付ける．2 回目，3 回目の演習には USB カメラを持参すること．第 3 回目では，3DCG と Wii リモコンを利用したインタラクションについて学習する，

12/4(**火**)　3DCG と OpenGL の基礎
12/11(**火**)　OpenGL+GUI(GLUI) による 3DCG の応用，テクスチャマッピング
12/14(**金**)　3DCG とインタラクション

1.2 出席・課題の確認について

出席の確認は，課題の確認によって行う．課題が終了したら，教員・TA を呼び，指示に従って実行して説明せよ．

1.3 資料など

演習の資料などは,

```
http://www.cyber.t.u-tokyo.ac.jp/~kuni/enshu2007/
```

においてある.

1.4 ネットワークの設定

ネットワークに接続できないときは,「システム」→「システム管理」→「ネットワーク」にて,「無線 LAN 接続」のクリックを外して,しばらくしてから再度クリックし,「インタフェースの設定を変更しています」が出るまで待つ.無線 LAN に接続できるようになることが多い.

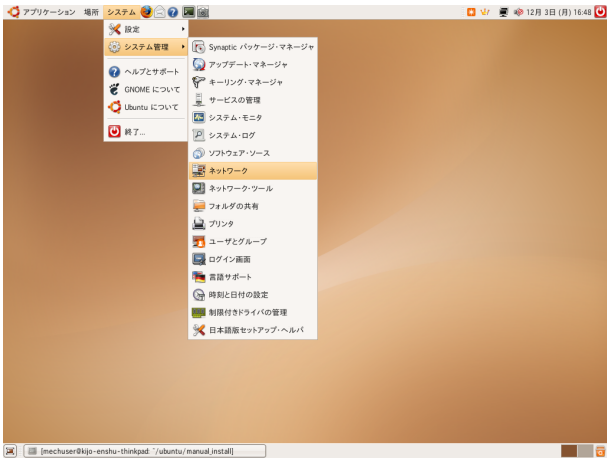

図 1 システム→システム管理→ネットワーク

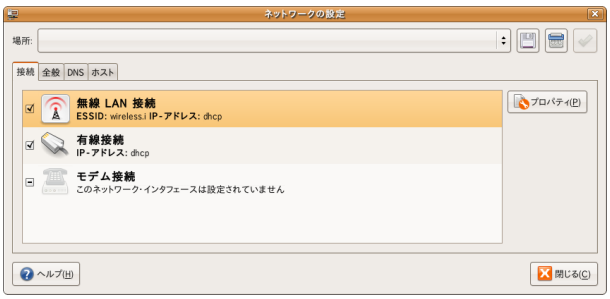

図 2 ネットワークの設定

2 OpenGL

OpenGL(Open Graphics Library) とは、Silicon Graphics 社 (SGI) が中心となって開発した 3D グラフィックスのためのプログラムインターフェイス。OpenGL は、SGI をはじめ、HP, SUN, IBM などの UNIX ワークステーションの他，Linux, FreeBSD などの PC UNIX に加え、Windows, Mac OS 等の様々な計算機で使用できるクロスプラットフォームの API である．また，携帯電話，PDA（携帯情報端末），家電など組み込み用途向け OpenGL のサブセット版である OpenGL ES も存在する。

無償で公開され、幅広い処理系に対応しているため、広く一般に普及している。非常に高速に動作し、高精度な 3D 画像を描画できる。有償・無償の豊富な補助ライブラリがあるのも特色として挙げられる。

2.1 OpenGL 関連ライブラリ

OpenGL には、強力かつ単純なレンダリング・コマンド群を装備しており、これらのコマンドを通じて高度な描画作業を行うことができる．ただし，OpenGL は各種のハードウェア・プラットフォームで実現するハードウェアに依存しないインタフェースとして設計されている．そのため，その画面表示の前提として必要なウィンドウ表示やユーザーインターフェイスに必要なコールバック関数等の機能は持ち合わせていない。

そこで、多くのプラットホームで共通に OpenGL ウィンドウを表示するためのライブラリである GLUT ライブラリが OpenGL と一緒に配布されている。主要な機能は、ウィンドウの表示、メニューの提供、イベントループ、キーボードとマウスからの入力、文字の表示である．

GL　OpenGL のコアライブラリ
　OpenGL コマンドは、”gl ” で始まり、コマンド名を構成する各語の最初の文字を大文字表記

GLU　OpenGL Utility Library
　OpenGL 開発環境の一部。射影のための行列設定や複雑なポリゴン分解などの機能を提供
　GLU ルーチンの接頭子は、”glu”

インターフェース・ライブラリ　各 Window システムが、OpenGL レンダリングをサポートするための機能を拡張するライブラリ (Window 管理・イベント処理など)
　X Window System では GLX, Windows NT/98/95 では WGL, OS/2 では PGL, Power Mac では AGL になる OS, Window システムに依存する

GLUT　OpenGL Utility Toolkit(Window システム非依存のツールキット)
　OpenGL 初期化、ウィンドウ操作、イベント処理を行うライブラリ
　演習では、ウィンドウ操作、イベント処理にこのライブラリを使用する
　GLUT ルーチンの接頭子は、”glut”
　本演習で使用する OpenGL のライブラリ

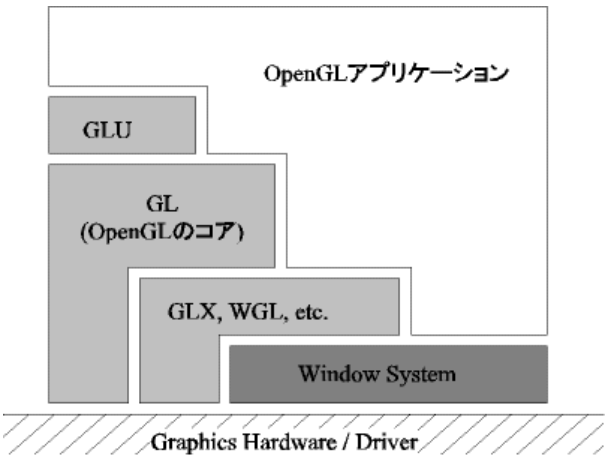

図 3 OpenGL と関連ライブラリ

3 OpenGL プログラミングの実際

3.1 OpenGL の構造

OpenGL は、以下のような OpenGL レンダリング・パイプラインと呼ばれる一連の処理に沿って行われる．

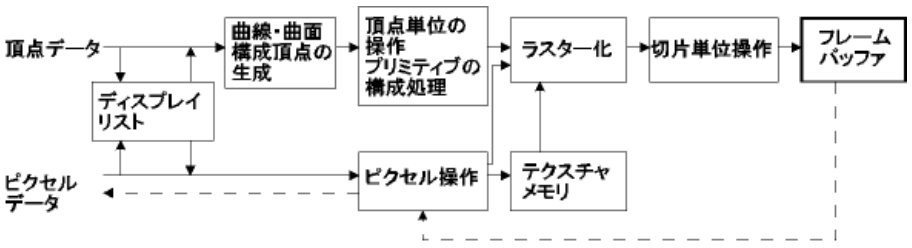

図 4 OpenGL レンダリング・パイプライン

幾何学データ (頂点、線分、ポリゴン) は、まず頂点データ (空間的座標) に 4x4 の浮動小数点行列により変換される．

空間座標は、三次元空間内の位置から画面上の位置に射影される．

一方、ピクセルデータ (ピクセル、画像、ビットマップ) は、拡大縮小・ピクセル演算など行われる．

このようにして、処理された幾何学データとピクセルデータの双方を、フラグメント (フレームバッファ内のピクセルに対応) に転換する．

最後にフレームバッファに実際に保存するフラグメンとの変更・破棄などを行い、適切はバッファに書き込むことで表示が行われる．

OpenGL は、仮想的なステート・マシンとして実装されている．OpenGL は、変更されるまで効果が持続する各種のステータス (モード) で使用される．たとえば以下の例では、色の状態を一度セットすると変更されるまで、その色でオブジェクトは描画される

3.2 OpenGL プログラムの例

```
main(){
/* Window オープン・初期化 */
   InitializeAWindowPlease();

/* 画面クリア */
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glClear (GL_COLOR_BUFFER_BIT);
/* オブジェクトの色指定 */
   glColor3f (1.0, 1.0, 1.0);
/* 投影変換の指定 */
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
/* オブジェクト指定 */
   glBegin(GL_POLYGON);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();
/* 描画コマンドの実行 */
   glFlush ();

/* Window 管理・イベント処理 */
   UpdateTheWindowAndCheckForEvents();
}
```

3.3 簡単な OpenGL プログラムの流れ図

GLUT ライブラリを使用した簡単な OpenGL プログラム (sample_clr.c) の流れ図を示す．プログラムは上記の Web に置いてある．

3.4 コマンドの説明

3.4.1 画面をクリアする色の指定

```
glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampfalpha);
```

GLglampf は、0.0 から 1.0 の間の単精度浮動小数点

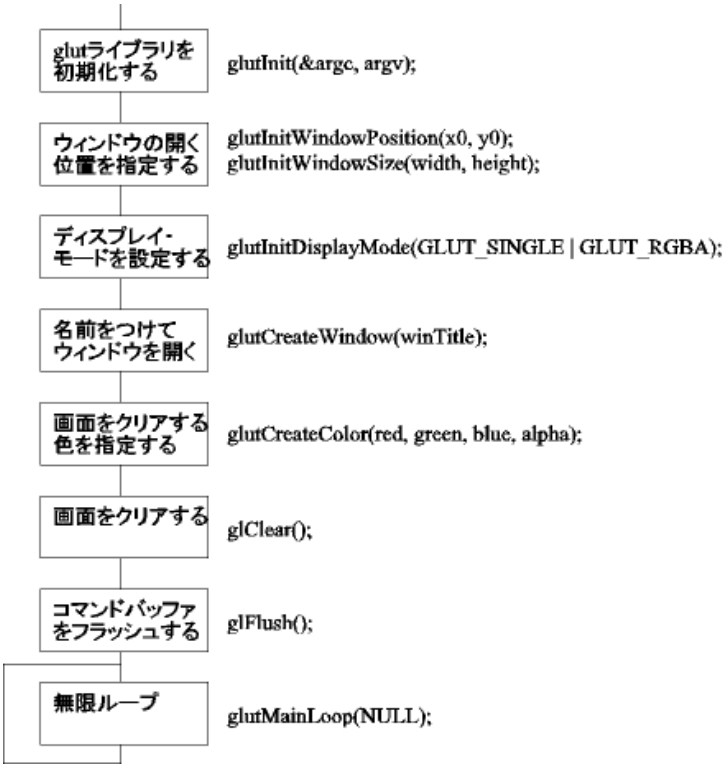

図 5 OpenGL プログラム (sample-clr.c) の流れ図

alpha は、透明度を示すパラメータ

例 画面をクリアする色を黒に設定．`glutClearColor(0.0, 0.0, 0.0, 0.0);`

3.4.2 画面を glClearColor でクリアする

```
glClear(GL_COLOR_BUFFER_BIT);
```

3.4.3 バッファリングされている描画コマンドを全てサーバに送る．

```
glFlush();
```

1 つのシーンに対する描画処理が終わった時点でコールする．ダブルバッファー・モードでは不要

3.5 glut コマンド

本演習ではウィンドウ操作、イベント処理に GLUT(OpenGL Utility Toolkit) ライブラリを使う．GLUT を用いれば、Windows、X window system の両方で動作するプログラムを非常に簡単に作成することが可能になる．GLUT の現時点での最新バージョンは、3.7 ベータ．また，GLU と同じようにいくつかのよく使用される立体を簡単に作成するために、球、円筒、多面体など基本的な 3 次元図形の描画をサポートする．

GLUT の初期化を glutInit でおこなう

```
void glutInit(int *argcp, char **argv);
```

argcp には、main 関数からコマンドライン引数を示す argc のアドレスを渡す．GLUT で解釈した引数については，その数を減じる．

argv は、main 関数の argv そのものである．GLUT で解釈した引数は、配列から消される．

X Window System 上への実装の場合は、GLUT の解釈するコマンドライン引数には、-display, -geometry などがある．詳細は、man X を参照のこと．

ウィンドウの開く位置・大きさを指定

```
void glutInitWindowPosition(int x, int y);
void glutInitWindowSize(int width, int height);
```

x, y は、ウィンドウ内側左上の点のスクリーン座標値.
width, height は、スクリーン上のピクセル数でウインドウの幅と高さを指定.

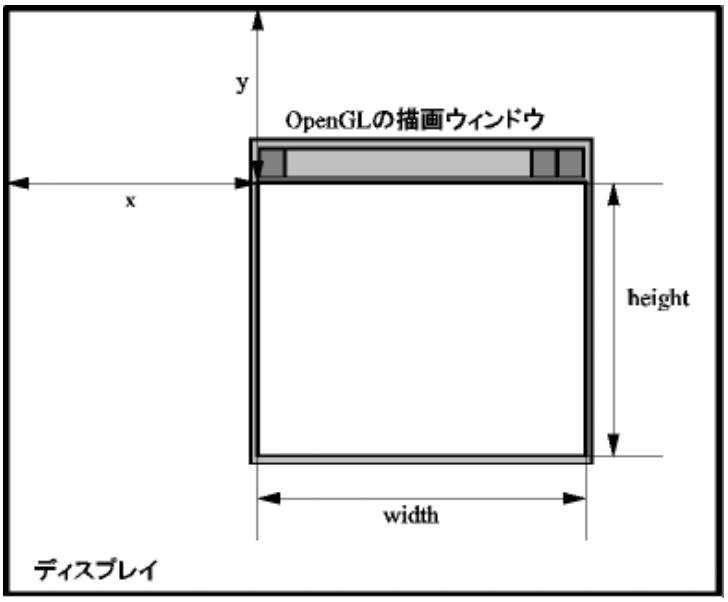

図 6 OpenGL の描画ウインドウを開く

glutInitWindow*関数がウィンドウシステムと連絡して OpenGL 初期化処理をする．

glutInitWindow*関数のパラメータで指定する座標系はウィンドウシステム固有のものを使い、画面左上を原点とする (OpenGL の座標形のとり方とは異なる).

ウィンドウ表示モードの初期化を glutInitDisplayMode **でおこなう**

```
void glutInitDisplayMode(unsigned int mode);
```

mode は以下に示すフラグ

- RGBA モードかカラーインデックス・モードか (GLUT_RGBA or GLUT_INDEX)
- 隠面消去するかどうか (GLUT_DEPTH)
- アニメーションするかどうか (GLUT_SINGLE or GLUT_DOUBLE)

必要なフラグを"—"でつないで指定する.

たとえば、RGBA モードで隠面消去してアニメーションするときは、

```
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
```

OpenGL ウィンドウを初期化して開く

```
int glutCreateWindow(char *titleString);
```

ウィンドウのタイトルを titleString で指定する.

OpenGL を使って描画するためのキャンバスとなるウィンドウを OpenGL ウィンドウを呼ぶ.

カレントウィンドウの再描画が必要であるということをマーキングする

```
void glutPostRedisplay(void);
```

次のイベント処理ループでノーマルプレーンの再描画のために display コールバックが呼ばれる
1 つのイベント処理ループの中で複数の再描画マークは 1 つにまとめられる

イベント処理の無限ループに入る

```
void glutMainLoop(void);
```

イベント処理の無限ループに入る.
glutDisplayFunc などの各種コールバック関数でイベントに対する処理を指定する

4 2 次元図形の描画

以上に基づいて，2 次元図形 (長方形) の描画をするプログラム (sample-hello.c) の流れ図を以下にしめす.

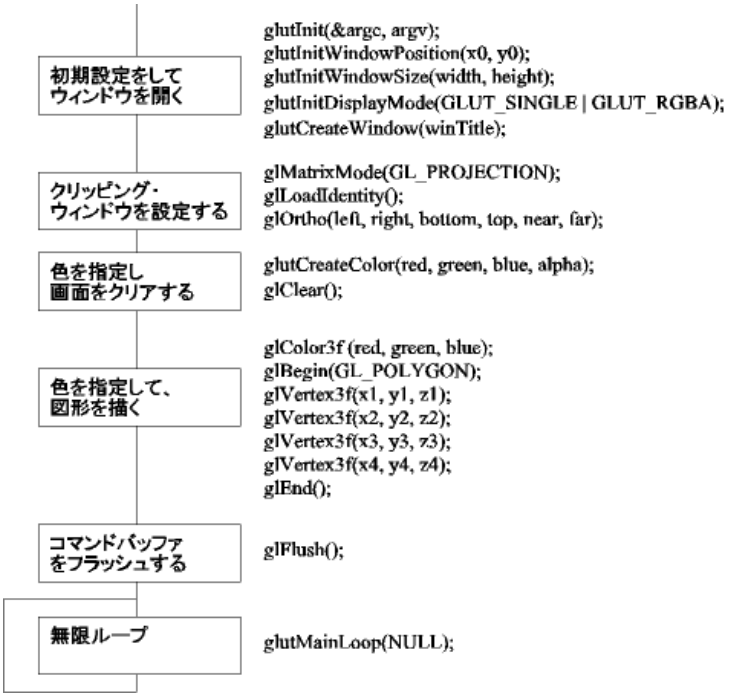

図 7　OpenGL プログラム (sample-hello.c) の流れ図

4.1 OpenGL 描画関数

色の指定

```
glColor3f(GLclampf red, GLclampf green, GLclampf blue);
```

0.0 から 1.0 の範囲の 3 つの浮動小数点で、赤・緑・青の強さを指定
例 黄色を指定. `glColor3f(1.0f, 1.0f, 0.0f);`

オブジェクトの描画

```
glBegin(GLenum mode);
```

幾何学的プリミティブを記述する頂点データのリストの先頭をマークする.
プリミティブの形式は mode が示し、以下のものがよく使われる.

- 単純な凸型ポリゴンの境界線 (GL_POLYGON)
- 三角形をつないで扇型にしたもの (GL_TRIANGLE_FAN)
- 三角形をつないでストリップ (帯状) にしたもの (GL_TRIANGLE_STRIP)
- 個々の線分として解釈される頂点の組 (GL_LINES)
- 個々の点 (GL_POINTS)

```
glEnd();
```

頂点データのリストの終端をマークする

```
glVertex3f(GLfloat x, GLfloat y, GLfloat z);
```

幾何学的オブジェクトの記述に使用する頂点を指定する

glVertex*関数は、いくつかのバージョンが存在する”3”は、3 つの引数 (x, y, z) をとることを示し、”f”は、32bit 浮動小数 (GLfloat) をとることを示す

4.2 描画の設定

ユーザーは、ワールド空間という仮想的な空間の任意の位置に図形を書くことができる

はじめにワールド空間中の図形を書こうと思う領域に、クリッピング・ウィンドウを設定する

クリッピング・ウィンドウ内部に書いた図形のみが、ディスプレイ上の OpenGL の描画ウィンドウにマッピングされる

クリッピングウィンドウの設定は gluOrtho2D 関数を使用する.

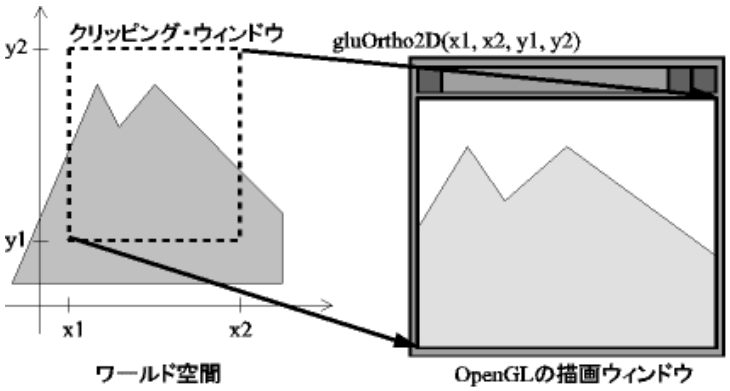

図 8 gluOrtho2D(x1,x2,y1,y2);

OpenGL には 3 つの変換マトリックスがあり、それぞれ投影変換、モデルビュー変換、テクスチャ座標変換に使う. マトリックスの設定は 3 つの中からカレントの処理対象マトリックスとして指定されているもの 1 つに対しておこなう. この設定をマトリックスモードの設定という

クリッピング・ウィンドウを設定するために、あらかじめマトリックスモードを投影変換マトリックスにして処理を行う

マトリックスモードの設定関数

```
void glMatrixMode(GLenum mode);
```

mode 引数で操作対象のマトリックスを指定する.

- 投影変換マトリックスを操作対象とする (GL_PROJECTION)
- モデルビュー変換マトリックスを操作対象とする (GL_MODELVIEW)
- テクスチャー座標変換マトリックスを操作対象とする (GL_TEXTURE)

マトリックスモードの変換マトリックスを単位マトリックスで初期化する.

```
void glLoadIdentity(void);
```

2 次元ワールド空間中にクリッピング・ウィンドウ (描画エリア) を設定する.

```
void gluOrtho2D(GLdouble left, GLdouble right,
                GLdouble bottom, GLdouble top);
```

クリッピング・ウィンドウ内に書いたものだけがディスプレイのウィンドウ内に表示される

gluOrtho2D 関数は、3 次元平行投影変換を設定する glOrtho 関数の引数 near, far をそれぞれ-1.0, 1.0 に設定した場合と同じである.

すなわち OpenGL の 2 次元描画は 3 次元の特殊な場合として処理されている

例　各方向-1.0 から 1.0 までの範囲に描画するとき、`gluOrtho2D(-1.0, 1.0, -1.0, 1.0);` または、`glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);`

次の手順でクリッピング・ウィンドウを設定する

1. マトリックスモードを投影変換のマトリックスにする
   `glMatrixMode(GL_PROJECTION);`
2. 投影変換のマトリックスを単位マトリックスで初期化
   `glLoadIdentity(void);`
3. クリッピングウィンドウの設定を行う
   `gluOrtho2D(left, right, bottom, top);`
4. 投影変換の処理が終わったら、処理対象マトリックスをモデルビューマトリックスにする
   `glMatrixMode(GL_MODELVIEW);`

4.3 ウィンドウサイズ変更とビューポートの設定

OpenGL の描画ウィンドウの中で、シーンのマッピングされる矩形領域をビューポートと呼ぶ. OpenGL の描画ウィンドウが最初に開いた時点ではビューポートは描画ウィンドウに一致している. 描画ウィンドウのサイズが変更になった場合は、それに応じてビューポートの再設定が必要になる (通常は描画ウィンドウに一致させる) .

ビューポート設定関数

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

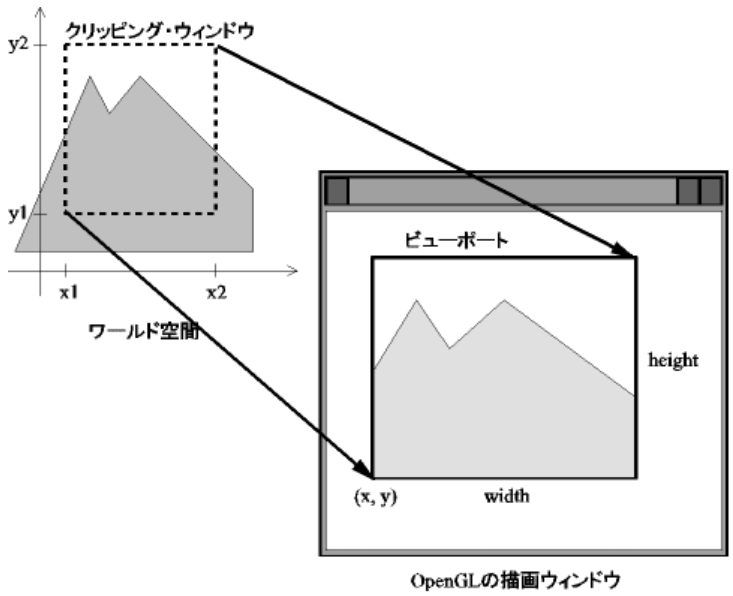

図 9 ビューポートの設定

(x,y) は左下端のディスプレイ座標値、
width と height はそれぞれピクセル単位でのビューポートの幅と高さ
通常、ウィンドウサイズに変更があったとき、glutReshapeFunc で指定するコールバック関数の中で指定する.

ウィンドウサイズに変更があった場合に呼び出す関数の指定

```
void glutReshapeFunc(void (*func)(int width, int height));
```

OpenGL ウィンドウが開いたとき、サイズが変更されたときに呼び出すコールバック関数を指定する.
コールバック関数の 2 つの仮引数にはウィンドウの幅と高さのピクセル数が渡される

5 OpenGL プログラムのコンパイル

5.1 ヘッダファイルとライブラリの指定

OpenGL プログラムをコンパイルするためには、適切なヘッダーファイルのインクルードと、ライブラリのリンクが必要になる

ヘッダーファイル

OpenGL のライブラリ関数を使う場合は次の 3 種類のヘッダーファイルを必要に応じてインクルードする．ただし，glut をインクルードする場合，省略可．

```
#include <GL/gl.h>      /* 必ず入れる */
#include <GL/glu.h>     /* glu ライブラリを使うとき */
#include <GL/glx.h>     /* X Window System の場合 */
```

GLUT ライブラリを使う場合は次のファイルだけインクルードする

```
#include <GL/glut.h>
```

gl.h や glu.h は、glut.h の中でインクルードされる．
glut.h が環境に応じて window.h などを適切な順番でインクルードするので、UNIX と WIndows でポータブルなソースコードになる．

本演習で使用するソースコードは GLUT を使用する

ライブラリのリンクには次のオプションが必要となる

```
-lglut -lGLU -lGL
```

例 simple.c. というプログラムから simple という実行モジュールを作成する

```
% gcc simple.c -o simple -lglut -lGLU -lGL
```

5.2 Makefile の例

```
#MAKEFILE

CC = gcc
RM = rm -f

CFLAGS  = -Wall
LDFLAGS = -lglut -lGLU -lGL -lm

TARGET  = sample-clr
OBJS    = sample-clr.o

.c.o:
${CC} -c ${CFLAGS} $<

TARGET: ${OBJS}
${CC} -o ${TARGET} ${OBJS} ${LDFLAGS}

clean:
${RM} ${TARGET} *.o *~
```

5.3 ウインドウの表示

まずは，サンプルのコンパイルを行って，実行ファイルを作成する．でき上がったファイルを実行し，スクリーン上にウィンドウが現れれば，コンパイルは成功である．

◇ **課題** 1 ◇

1. サンプルプログラム sample-clr.c をコンパイルして実行せよ．
2. ウインドウサイズやクリア色を変更せよ．
3. サンプルプログラム sample-hello.c をコンパイルして実行せよ．
4. 関数 `glOrtho()` のパラメータを適当に変更し，見え方の変化を確認せよ．
   また，関数 `glVertex3f()` のパラメータをを変更し，変化を確認せよ．

6　三次元画像の表示

6.1　三次元画像レンダリングの概要

希望するシーンを生成するための変換処理は、カメラを使用した写真撮影に類似したものとなっている。

1. 三脚を固定し、カメラをシーンに向ける。
2. 撮影するシーンが、希望するような構成になるように配置する。
3. カメラのレンズを選んで、ズームを調節する。
4. 最終的な写真の大きさを決定する。

OpenGL のプログラムにおいても、同様な順序で変換を指定する。

1. ワールド座標系において、カメラの位置姿勢を決定する。(視界変換)
2. オブジェクトを、ワールド内に配置する。(モデリング変換)
3. カメラモデル (画角・視体積) を決定する。(射影変換)
4. ウィンドウ座標にマッピングする。(ビューポート変換)

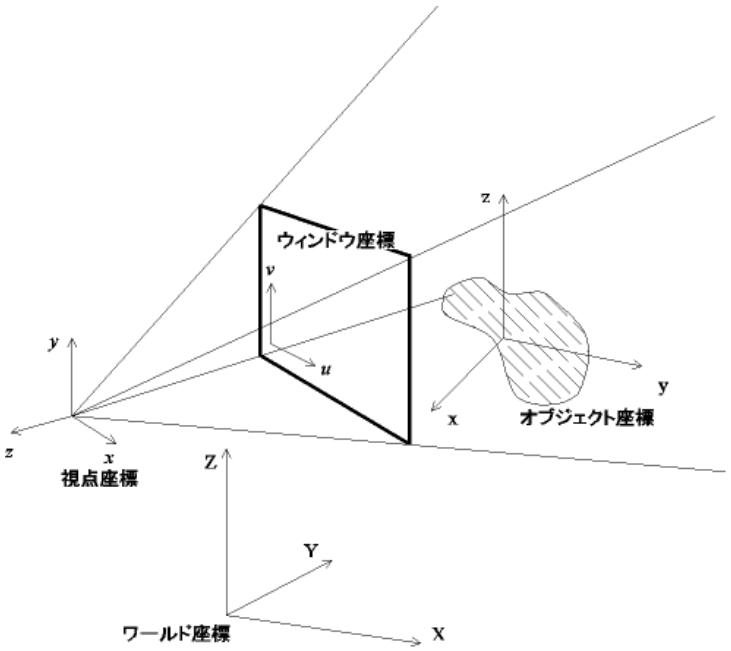

図 10　OpenGL における座標変換

OpenGL では、視界変換・モデリング変換・射影変換は、4x4 の行列 M を構築し、これをシーン内の各頂点 v の座標と乗算される。

$$v' = Mv$$

頂点の座標 v は、常に 4 つの座標 (x, y, z, w) であり、ほとんどの場合 w は 1 である。

指定する視界変換とモデリング変換は、組み合わさってモデルビュー行列を形成する。形成されたモデルビュー行列は、オブジェクト座標に適用され、視点座標を生成する。

指定されたカメラモデルから形成される射影行列を適用して、視点座標からクリップ座標を生成する。この変換は、視体積を定義し、この外側にあるオブジェクトは描画されないようにクリッピングされる。

この後、座標値 w で除算し、正規化座標を生成する。

最後に、ビューポート変換を適用して、変換された座標がウィンドウ座標に変換される。

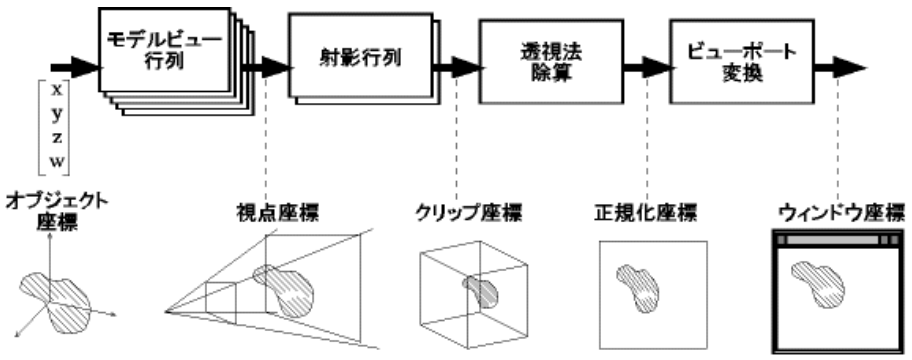

図 11 OpenGL におけるレンダリングパイプライン

6.2 簡単な例

次の例 (sample-cube.c) は、モデリング変換により拡大縮小した立方体を描画している。

```
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}

void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
   glLoadIdentity ();             /* clear the matrix */
           /* viewing transformation  */
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   glScalef (1.0, 2.0, 1.0);      /* modeling transformation */
   glutWireCube (1.0);
   glFlush ();
}
```

```
void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
   glMatrixMode (GL_MODELVIEW);
}

/* ARGSUSED1 */
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27:
         exit(0);
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&amp;argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

視界変換 gluLookAt() は、立方体が描画される方向にカメラを配置して向けている。射影変換とビューポート変換は、glFrustum() と glViewport() を用いて指定している。

6.3 座標変換の概要

視界変換

視界変換は、カメラの配置と方向設定と同等に機能する。

この例では、視界変換をする前に、glLoadIdentity() により現在の行列 (current matrix) を単位行列にセットしている。変換コマンドの大部分が、現在の行列に対し指定した行列を乗算し、その結果を現在の行列に設定するため、このステップが必要となる。

行列を初期化した後、gluLookAt() で視界変換を指定している。個々で使用している引数は、カメラを (0, 0, 5) に配置し、(0, 0, 0) の方向にレンズを向け、上方向ベクトルを (0, 1, 0) に指定している。

gluLookAt() を呼び出さない場合、カメラの位置と方向は初期設定の内容になる。初期設定では、カメラは原点に位置し、負の z 軸方向をさし、上方向ベクトルは (0, 1, 0) となっている。そのため、全体的な効果としては、gluLookAt() がカメラを z 軸に沿って 5 単位分移動することに相当する。

モデリング変換

モデリング変換は、モデルの配置と方向設定に使用する. たとえば、回転・平行移動・拡大縮小を実行したり、これらの作業を組み合わせたりできる。

この例では、glScalef() がモデリング変換である。このコマンドの引数は、3 つの軸に沿ってどのように拡大縮小するかを指定している。この場合、立方体は y 方向に 2 倍の大きさでかかれる。

カメラを移動して立方体を見る (視界変換) 代りに、カメラから立方体を移動させる (モデリング変換) ことも可能である。たとえば、gluLookAt() の呼び出しを、モデリング変換 glTranslatef(0.0, 0.0, -5.0) に変更した場合、その結果はまったく同一のものとなる。

OpenGL では、視界変換とモデリング変換を組み合わせてモデルビュー行列 (modelview matrix) を構成し、頂点座標に対して適用されている。

モデリング変換と視界変換は、立方体を描画する glutWireCube() の呼び出しと共に、ルーチン display() にふくまれる。glutDisplayFunc として指定される display() ルーチンは、ウインドウの再描画などの際に反復使用される。そのため、正しく毎回描画するためには、視界変換やモデリング変換を実行する前に、単位行列をロードすることが必要となる。

射影変換

射影変換は、カメラのレンズを選択することに相当する。この変換は、視野や視体積を特定し、その中に位置するオブジェクトと、それがどのように見えるかを決定する。

OpenGL では、2 種類の射影方式が用意されている。ひとつが透視 (perspective) 変換であり、われわれが実生活で世界を見ている方法に相当する。このプログラム例では、コマンド glFrustum() で指定している。もうひとつが、正射影 (orthographic) であり、CAD などのアプリケーションなどで使用される。

ルーチン reshape() で示すように、コマンド glMatrixMode(GL_PROJECTION) にして、現在の行列が射影変換を指定する。その上で、指定した射影変換のみが効果を持つようにするため glLoadIdentity() で初期化し、glFrustum() を呼び出して射影行列を生成している。

ビューポート変換

射影変換とビューポート変換は、シーンがどのように画面上に写像されるかを決定する。

ビューポートは、画像がコンピュータの画面上で占める領域を指定するために、写真のサイズと位置を定義することに相当する。glViewport() の引数は、ウィンドウ内で使用可能な画面空間の原点 (この例では (0, 0))、使用可能な画面領域の幅と高さ (ピクセル単位) を記述する。

射影変換とビューポート変換の双方が、ウィンドウが最初に作成されるときや、移動形状変更のたびに呼び出されるルーチン reshape() に含まれる。射影 (射影視体積の幅と高さの縦横比) とビューポートの適用は、画面上のウィンドウのサイズや縦横比に直接的に関係してくる。そのため、このように reshape() 内でビューポートの指定と射影変換の決定をする必要がある。

6.4 OpenGL 座標変換コマンド

一般的な変換コマンド

変換コマンドを供給する前に、モデルビュー行列・射影行列を変形するかの指示が必要である。行列の選択には、glMatrixMode() で実行する。

```
void glMatrixMode(GLenum mode);
```

mode に引数 GL_MODELVIEW、GL_PROJECTION、GL_TEXTURE のいずれかを使用して、モデルビュー、射影、またはテクスチャ行列を変形するか指定する。それ以降の変換コマンドは、指定した行列に影響する。初期設定では、変形可能な行列はモデルビュー行列になっている.

glLoadIdentity() は、次以降の変換コマンドに対して現在の行列を消去する際に使用する。通常、射影もしくは視界変換の指定の前に呼び出すが、モデリング変換の指定の前に呼び出す場合もある。

```
void glLoadIdentity(void);
```

現在の変形可能な行列を、4x4 の単位行列に設定する。

また、単純なオブジェクトを組み合わせて複雑なオブジェクトを構築する、階層モデルの構築などに行列スタックを使用する。

モデリング変換

モデリング変換用の 3 つの OpenGL ルーチンは、glTranslate*()、glRotate*()、glScale*() である。これらのルーチンは、オブジェクトを移動・回転・伸縮させて、オブジェクト (または座標系) を変換する。3 つのコマンドは、それぞれ平行移動、回転、拡大縮小の行列を生成し、現在の行列に対し乗算を行い、現在の行列として保存を行っているのと同等である.

```
void glTranslate{fd}(TYPE x, TYPE y, TYPE z);
```

指定した x, y, z の値分、オブジェクトを移動 (平行移動) させる。または、同じ量だけローカル座標系を移動させる。

```
void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);
```

原点から (x, y, z) の点を通過する線を中心に左回り方向にオブジェクト (またはローカル座標系) を回転させる。引数 angle は、回転の角度を度数で指定する。

glRotatef(45.0, 0.0, 0.0, 1.0) は、z 軸を中心に 45 度回転する。

```
void glScale{fd}(TYPE x, TYPE y, TYPE z);
```

オブジェクトを軸に沿って伸縮、または線対称変換する。オブジェクト内の各点の x, y, z 座標は、各々対応する引数 x, y, z と乗算される。または、ローカル座標の軸は、x, y, z 分だけ拡大縮小され、関連するオブジェクトはそれらを使用して変換される。

視界変換

視界変換の方法には、いくつか方法がある。

ひとつは、モデリング変換コマンド (glTranslate*() と glRotate*()) を使用する方法である。視点変換は、モデルに対してカメラを向ける作業に相当するが、これはカメラを固定したままワールド内の全オブジェクトを動かすことと同じためである。たとえば、オブジェクトを左回りに回転させるモデリング変換は、カメラを右回りに回転させる視界変換に等しい。

OpenGL Utility Library(GLU) ルーチンの gluLookAt() で視界の境界を定義する。このルーチンは、一連の回転コマンドと平行移動コマンドを要約したものである.

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
               GLdouble centerx, GLdouble centery, GLdouble centerz,
               GLdouble upx, GLdouble upy, GLdouble upz);
```

視界行列を定義し、それを現在の行列に乗算する。視点は、eyex, eyey, eyez で指定する。引数 centerx, centery, centerz は、希望する視線に添った点を指定する。通常はシーンの中心部の点となる。引数 upx, upy, upz は、上を向いている方向を示す。

初期設定では、カメラは原点にあり、負の z 軸を見下ろし、正の y 軸が真上の方向を指している。

射影変換

射影変換コマンドは、glFrustum()、glPerspective()、glOrtho()、glOrtho2D() である。各射影変換コマンドは、特定の射影変換に完全に対応しているため、射影変換を別の変換と組み合わせて使用することは通常ない.

これらの変換コマンドを使用する前には、そのコマンドがモデルビュー行列ではなく射影行列に影響を与えるようにするために、以下を呼び出す必要がある。

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

■透視投影変換

```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom,
               GLdouble top, GLdouble near, GLdouble far);
```

透視法による錐体の行列を作成し、それを現在の行列と乗算する。(left, bottom, -near) と (right, top, -near) は、近くのクリップ面の左下と右上の座標を指定する。near と far は、視点から近く・遠くのクリップ平面までの距離 (常に正) を指定する.

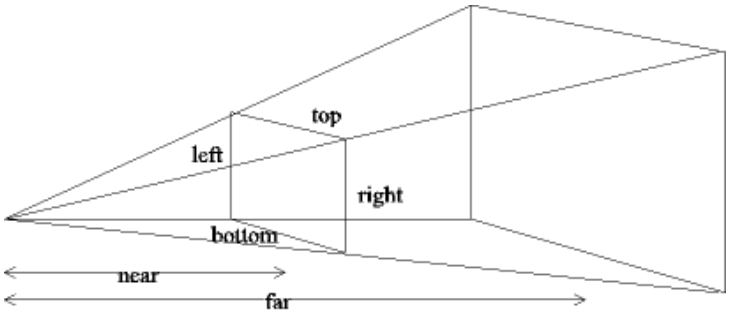

図 12　透視投影変換 glFrustum()

```
void glPerspective(GLdouble fovy, GLdouble aspect,
                   GLdouble near, GLdouble far);
```

左右対称な透視法錐体の行列を作成し、それを現在の行列と乗算する。fovy は、x-z 平面での視野の角度で、その値は [0.0, 180.0] の範囲であることが必要である。aspect は、錐体の縦横比で、その幅を高さで割ったもの。near と far の値は、負の z 軸に沿った、視点とクリップ平面間の距離 (常に正) である。

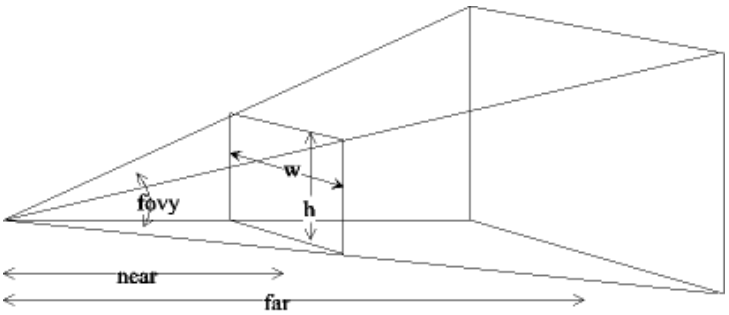

図 13　透視投影変換 glPerspective()

■正射影

```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,
             GLdouble top, GLdouble near, GLdouble far);
```

正射影による平行な視体積の行列を生成し、それを現在の行列と乗算する。(left, bottom, -near) と (right, top, -near) は、各々視体積の左下と右上の隅に写像される手前のクリップ平面上の点。
(left, bottom, -far) と (right, top, -far) は、奥のクリップ平面上の点。near と far は、どちらも正または負の値を取ることができる。

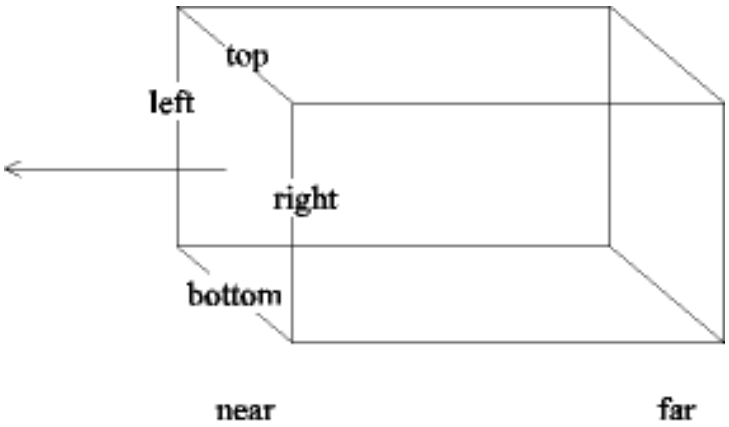

図 14 正射影変換 glOrtho()

```
void glOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);
```

2 次元座標を画面に射影する行列を生成し、それを現在の射影行列と乗算する。クリッピング領域は、左下隅を (left, bottom)、右上隅を (right, top) にした方形になる。

このルーチンは、シーン内のオブジェクトの z 座標がすべて-1.0〜1.0 の間に位置するという点を除いて、glOrtho() と同等である。2 次元の頂点コマンドを使用している場合、z 座標はすべて 0.0 として計算される。

ビューポートの定義

初期設定では、ビューポートは開かれるウインドウのピクセル方形全体にセットされている。より小さい描画領域を選択する場合、glViewport() を使用する。

ビューポートの縦横比は、通常は視体積の縦横比と等しくなる。2 つの比が異なる場合、ビュウポートに写像した時点で射影された画像がゆがんでしまう。アプリケーションは、ウインドウのサイズ変更イベントを検知し、ビューポートを適切に変換することが必要となる。

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

最終画像が写像されれるウインドウのピクセル方形を定義する。パラメータ (x, y) は、ビューポートの左下隅を指定し、width と height はビューポート方形のサイズである。初期設定では、ビューポート値は (0, 0, winWidth, winHeight) で、このとき winWidth, winHeight はウィンドウのサイズとなる。

7 図形の描画

7.1 図形の種類

OpenGL では、点・線分・ポリゴンという幾何学的プリミティブ (geometric primitive) 群から必要なモデルを構築する必要がある。

点

頂点 (Vertex) と呼ばれる浮動小数点値により表現される。
内部的な計算は、全ての頂点が 3 次元であるとして計算され、2 次元 (x, y) として指定された場合 z 座標は 0。

線分

始点と終点を、Vertex により指定したもの。
接続された一連の線分か、その一連の線分が閉じたものとなる。

ポリゴン

環状の線分によって閉じられた領域。
OpenGL によるポリゴンは、交差がなく常に凸型 (convex) なものに限られる。ポリゴンを形成する境界を形成する線分の数は制限されていない. 複雑なオブジェクトは GLU ライブラリにより単純な凸型ポリゴンの集合として扱われる。
頂点は常に三次元であるため、必ずしも空間内の同じ平面状に存在するとは限らない。この場合、正確にレンダリングされ得ない恐れがある。通常は、三角形ポリゴンを使用することで、これらの問題を回避している。

7.2 OpenGL コマンド

頂点の指定

OpenGL では、全ての幾何学プリミティブが適切な順序で配列された頂点群として最終的にかかれる。頂点の指定には、glVertex*() をしようする。

```
void glVertex{234}{sifd}[v](TYPE coords)
```

例:

```
glVertex2s(2, 3);
glVertex3f(2.3, 1.1, -2.2);
GLdouble dvect[3]={5.0, 9.0, 1992.0};
glVertex3dv(dvect);
```

描画プリミティブ

glVertex*() により指定した頂点から、点・線分・ポリゴンを生成する場合、glBegin() と glEnd() の呼び出し関数の間に各頂点群を指定する。glBegin() に渡される引数は、頂点からどのようなプリミティブが構築されるかを特定する。

例: 塗りつぶしたポリゴン

```
glBegin(GL_POLYGON);
   glVertex2f(0.0, 0.0);
   glVertex2f(0.0, 3.0);
   glVertex2f(4.0, 3.0);
   glVertex2f(6.0, 1.5);
   glVertex2f(4.0, 0.0);
glEnd();
```

よく使う図形

```
glBegin(GL_POLYGON);〜glEnd();：ポリゴン
glBegin(GL_TRIANGLE_FAN);〜glEnd();：三角形による扇形
glBegin(GL_TRIANGLE_STRIP);〜glEnd();：三角形による帯形
glBegin(GL_LINES);〜glEnd();：線分
```

その他の指定

```
GL_POINTS, GL_LINES,
GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES,
GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS,
GL_QUAD_STRIP, and GL_POLYGON.
```

glBegin() と glEnd() の間では、コマンド glVertex*() による頂点の座標の指定以外にも、カラー・法線ベクトル・テクスチャ座標といった頂点固有のデータを描く頂点に指定することが可能である。

例: 頂点の色指定

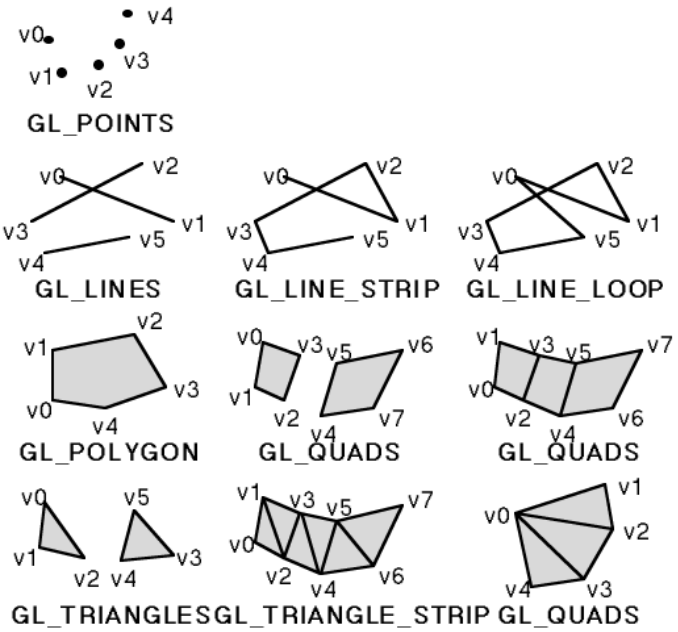

図 15　OpenGL プリミティブ

```
glBegin(GL_POINTS);
   glColor3f(0.0, 1.0, 0.0);
   glVertex(...);
   glColor3f(1.0, 1.0, 0.0);
   glVertex(...);
   glVertex(...);
glEnd();
```

また、反復実行などプログラムの言語構造を自由に入れることが可能である.

例: 円の輪郭線

```
#define PI 3.1415926535898
GLint circle_points=100;
glBegin(GL_LINE_LOOP);
for(i=0; i<circle_points; i++){
   angle = 2*PI*i/circle_points;
   glVirtex2f(cos(angle), sin(angle));
}
glEnd();
```

7.3 点・線・ポリゴンの表示

点について

レンダリングする点のサイズを変更する場合には、glPointSize() を使用し、引数に必要なサイズをピクセルで与える。

```
void glPointSize(GLfloat size);
```

レンダリングする点の幅をピクセルで設定。初期設定では 1.0 に設定されている。

線について

各種の幅の線や、点線・破線など各種の線を指定できる。

```
void glLineWidth(GLfloat width);
void glLineStipple(GLint factor, GLushort pattern);
```

レンダリングする線の幅 (width) をピクセルで設定。破線の場合、引数 pattern で、16 ビットの 0 と 1 の連続で必要に応じて反復されて描画される。1 はピクセル単位で描画が実行されることを示し、0 は描画が実行されないことを示す。factor は、pattern を何倍に引き伸ばすかを示す。

```
glLineStipple(1, 0x3F07);
glEnable(GL_LINE_STAPPLE);
```

この例では、パターン”0x3f07”(2 進数で”0011111100000111”) とすると、線はピクセル 3 個分を ON、5 個分を OFF、6 個分を ON、2 個分を OFF にして描画される。factor が 2 の場合、6 個分 ON、10 個分 OFF、12 個分 ON、4 個分 OFF となる。glLineStipple() で破線パターンを定義し、glEnable() で破線処理を有効にしている。

ポリゴンについて

ポリゴンは、前方面と後方面という二つの面があり、どちらが見ている側に面しているかにより、レンダリングが異なる。初期設定では、前方面・後方面共に同じように描画される。これを変更したい場合や、輪郭線・頂点だけによる描画を行いたい場合、glPolygonMode() を使用する。

```
void glPolygonMode(GLenum face, GLenum mode);
```

パラメタ face は、GL_FRONT_AND_BACK, GL_FRONT, GL_BACK、mode には、GL_POINT(点), GL_LINE(輪郭線), GL_FILL(塗りつぶし)。

```
void glFrontFace(GLenum mode);
void glCullFace(GLenum mode);
```

通常、画面上で頂点が左回り (反時計回り) の順序になっているポリゴンを”前方面”ポリゴンという。ある物体を構築するポリゴンの方向が一定であれば、適切な表面を提示できる。外側で右回りの方向になってしまった場合、glFrontFace() を使用することで通常では後方面と解釈される面を全方面のポリゴンに指定しなおせる。また、計算の省力化のため、あらかじめ後方面ポリゴンを破棄したい場合、glCullFace(GL_BACK) で破棄するポリゴンを指定し、glEnable(GL_CULL_FACE) で有効にする.

7.4 GLUT 関数

GLUT ライブラリには、複数の幾何学シェープを生成するためのルーチンがいくつか用意されている。これらのルーチンを使用することにより、簡潔なプログラムが可能になる。9 つの幾何学シェープが用意され、それぞれにつきワイヤーフレームとソリッドモデルの 2 つのルーチンが用意される。これらのルーチンは、純粋な OpenGL レンダリングルーチンとして実行することができる。

```
void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);
void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
```

glutSolidSphere と glutWireSphere は、指定された radius のモデリング座標原点を中心として、それぞれソリッドとワイヤーフレームの球をレンダリングする。

radius 球の半径
slices Z 軸を中心とする再分割の数 (経度のラインと同様)
stacks Z 軸に沿った再分割の数 (緯度のラインと同様)

```
void glutWireCube(GLdouble size);
void glutSolidCube(GLdouble size);
```

glutWireCube と glutSolidCube は、それぞれソリッドまたはワイヤーフレームの立方体をレンダリングする。立方体は、size によって大きさが設定され、モデリング座標原点にセンタリングされる。

size 立方体の一辺の長さ

```
void glutWireTorus(GLdouble innerRadius, GLdouble outerRadius,
                   GLint nsides, GLint rings);
void glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius,
                   GLint nsides, GLint rings);
```

glutSolidTorus と glutWireTorus は、軸が Z 軸と位置あわせされているモデリング座標原点を中心として、それぞれソリッドまたはワイヤーフレームのトーラスをレンダリングする。

innerRadius トーラスの内側の半径
outerRadius トーラスの外側の半径
nsides 個々の放射状のセクションのための分割の数
rings トーラスのための放射状の再分割の数

```
void glutWireIcosahedron(void);
void glutSolidIcosahedron(void);
void glutWireOctahedron(void);
void glutSolidOctahedron(void);
void glutWireTetrahedron(void);
void glutSolidTetrahedron(void);
void glutWireDodecahedron(void);
void glutSolidDodecahedron(void);
```

glutSolidIsosahedron と glutWireIsosahedron は、1.0 の半径と共にモデリング座標の原点を中心として、それぞれソリッドまたはワイヤーフレームの 20 面体をレンダリングする。

glutSolidOctahedron と glutWireOctahedron は、1.0 の半径と共にモデリング座標の原点を中心として、それぞれソリッドまたはワイヤーフレームの 8 面体をレンダリングする。

glutSolidTetrahedron と glutWireTetrahedron は、$\sqrt{3}$ の半径と共にモデリング座標の原点を中心として、それぞれソリッドまたはワイヤーフレームの 4 面体をレンダリングする。

glutSolidDodecahedron と glutWireDodecahedron は、$\sqrt{3}$ の半径と共にモデリング座標の原点を中心として、それぞれソリッドまたはワイヤーフレームの 12 面体をレンダリングする。

```
void glutWireCone(GLdouble radius, GLdouble height,
                  GLint slices, GLint stacks);
void glutSolidCone(GLdouble radius, GLdouble height,
                  GLint slices, GLint stacks);
```

glutSolidCone と glutWireCone は、Z 軸に沿ってそれぞれソリッドまたはワイヤーフレームの円錐をレンダリングする。円錐のベースを Z=0 に、頭頂を Z=height に配置される。また、Z 軸を中心として slices に

再分割され、Z 軸に沿って stacks に再分割される。

base 円錐のベースの半径
height 円錐の高さ
slices Z 軸を中心とする再分割の数
stacks Z 軸に沿った再分割の数

```
void glutWireTeapot(GLdouble size);
void glutSolidTeapot(GLdouble size);
```

glutSolidTeapot と glutWireTeapot は、それぞれソリッドまたはワイヤフレームのティーポットをレンダリングする。ティーポットのための平面法線とテクスチャ座標の両方が、OpenGL エバリュエータより生成される。

size ティーポットの相対的なサイズ

◇ **課題** 2 ◇

1. sample-cube.c を、make、実行する。
2. cube の色の変更や背景色の変更を行うこと。
3. 視点位置・視線方向の変更を行う。
4. glutLookAt() の呼び出しを、モデリング変換 glTranslatef(0.0, 0.0, -5.0) に変更すること。なぜ、同様の結果が得られるのか説明せよ。
5. glFrustum() の呼び出しを、より一般的に使用される gluPerspective(60.0. 1.0, 1.5, 20.0) に変更してみること。fovy と aspect の値を変えて試してみること。
6. cube のほかにも、いろいろなオブジェクトについても表示してみること。

8 座標変換によるオブジェクト操作

8.1 行列スタックの操作

これまでに作成・ロード・乗算したモデルビュー行列と射影行列は、実際にはそれらの行列の最上部に属している。行列のスタックは、単純なオブジェクトを組み合わせて複雑なオブジェクトを構築する、階層モデルの構築に効果的である。

前回までで説明した行列演算 (glLoadMatrix(), glMultMatrix(), glLoadIdentity() など) は、現在の行列、つまりスタックの最上部の行列を対象とする。スタックを操作するコマンドを使用することで、最上部に位置する行列を制御できる。glPushMatrix() は現在の行列をコピーし、そのコピーを最上部に追加する。glPopMatrix() は、スタックの最上部の行列 (現在の行列は常に最上部の行列) を破棄する。つまり、glPushMatrix() は、”現在位置の保存”、glPopMatrix() は、”前の位置に戻る”ことを意味する。

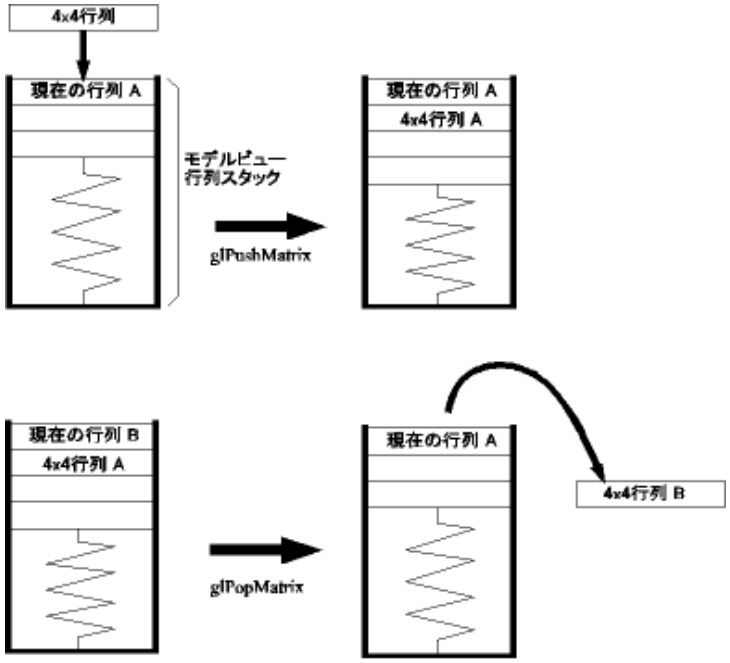

図 16　行列スタックの操作

```
void glPushMatrix(void);
```

現在のスタックの全行列を 1 段下にプッシュする。最上部の行列はコピーされ、その内容は最上部と上から 2 番目の行列の双方に複製される。現在のスタックは、glMatrixMode() が定義する。

```
void glPopMatrix(void);
```

スタックから行列をポップし、ポップされた行列の内容を破棄する。そして、上から 2 番目にあった行列が

最上部に押し上げられる。現在のスタックは、glMatrixMode() が定義する。スタックに行列が一つしか含まれない場合、glPopMatrix() を呼び出すとエラーが生ずる。

モデルビュー行列スタック

前回、説明したとおり、モデルビュー行列は視界変換とモデリング行列の乗算の累積である。各視界変換・モデリング変換は、現在のモデルビュー行列を乗算する新規の行列を作成する。新たに現在の行列なる行列は、現在の行列に乗算を行った合成変換を示す。

モデルビュー行列スタックは、最低 32 個の 4x4 行列を含む。初期状態では、最上部の行列が単位行列になっている。

射影行列スタック

射影行列は、視体積を記述する射影変換の行列を含む。通常は、射影行列を作成しないため、射影変換を実行する前に glLoadIdentity() を呼び出す。なお、射影行列スタックは、2 段だけにする必要がある。

スタックの 2 番目の行列は、3 次元のシーンを表示する通常のウィンドウのほかにテキストの入ったヘルプ・ウインドウの表示が必要なアプリケーションなどに使用できる。テキストの配置は、正射影がもっとも容易であるため、一時的に正射影に変更し、ヘルプを表示して、前の射影に戻す。

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();              /* save the current projection */
    glLoadIdentity();
    glOrtho(...);            /* set up for displaying help */
    display_the_help();
glPopMatrix();
```

8.2 行列スタック操作の例

例えば、4 つの車輪のおのおのを 5 本のボルトで取り付ける自動車を描画するケースを想定する。車輪を描画するルーチンをひとつ、ボルトを描画するルーチンを一つ使用する。これらのルーチンは、車輪やボルトを適切な位置と方向 (軸を一致させ、原点を中心に) で描画される。車輪を正確に配置するため、別々の変換を各々使用して車輪描画ルーチンを 4 回呼び出す。各車輪を描画する際には、ボルトを 5 回描画し、各車輪に対して適切に比例させて平行移動させる。

次の例は、車体・車輪・ボルトを描画するルーチンが前提として自動車を描画する。

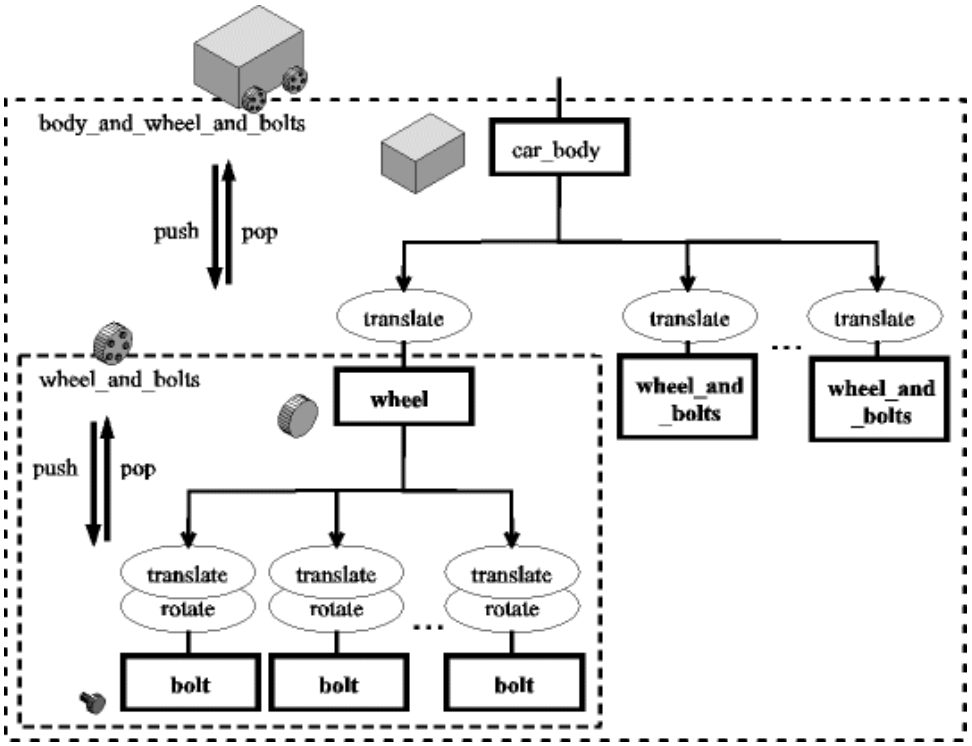

図 17 行列スタックの操作による自動車

```
draw_wheel_and_bolts()
{
   long i;

   draw_wheel();
   for(i=0;i<5;i++){
      glPushMatrix();
         glRotatef(72.0*i,0.0,0.0,1.0);
         glTranslatef(3.0,0.0,0.0);
         draw_bolt();
      glPopMatrix();
   }
}
```

```
draw_body_and_wheel_and_bolts()
{
   draw_car_body();

   glPushMatrix();
      glTranslatef(40,0,30);    /*move to first wheel position*/
      draw_wheel_and_bolts();
   glPopMatrix();
   glPushMatrix();
      glTranslatef(40,0,-30);   /*move to 2nd wheel position*/
      draw_wheel_and_bolts();
   glPopMatrix();
   ...                          /*draw last two wheel similarly*/

}
```

このプログラムでは、車輪とボルトの軸が一致しており、ボルトが車輪の中心から各々 72 度、3 単位分均等に配置され、前の車輪が車の原点の前方 40 単位、左右に 30 単位に配置している。

スタックがハードウェアで実現される場合、個別の行列よりも効率が改善される。行列をプッシュする場合、現在のデータをメイン・プロセスにコピーして戻す必要がなく、ハードウェアは同時に複数の行列要素をコピーできるためである。

8.3 太陽系の構築

同じ球体描画ルーチンを使用した太陽と惑星を含む単純な太陽系を描画する。このプログラムでは、太陽の周囲を回る惑星の公転と、自身の軸を中心とした惑星の自転に glRotate*() を使用する。惑星をその軌道から外し、太陽系の原点から遠ざける場合は、glTranslate*() を使用する。ルーチン glutWireSphere() に適切な引数を与えると、2 つの球体のサイズを指定できる。

```
    glPushMatrix();
    glutWireSphere(1.0, 20, 16);   /* draw sun */
    glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
    glTranslatef (2.0, 0.0, 0.0);
    glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
    glutWireSphere(0.2, 10, 8);    /* draw smaller planet */
    glPopMatrix();
```

最初の glRotate*() は、当初グローバル座標系と一致しているローカル座標系を回転する。次に、glTranslate*() が惑星の軌道上の位置にローカル座標系を移動する。移動した距離は軌道の半径と等しくなる。この

ようにして、glRotate*() が軌道上の惑星の位置 (公転) を決定する。

2 番目の glRotate*() は、ローカル軸の周囲でローカル座標系を回転させ、惑星の回転 (自転) を決定する。

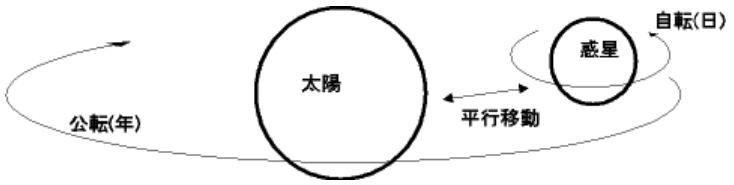

図 18　行列スタックの操作による太陽系

```
#include <GL/glut.h>
#include <stdlib.h>

static int year = 0, day = 0;

void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}

void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);

   glPushMatrix();
   glutWireSphere(1.0, 20, 16);   /* draw sun */
   glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
   glTranslatef (2.0, 0.0, 0.0);
   glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
   glutWireSphere(0.2, 10, 8);    /* draw smaller planet */
   glPopMatrix();
   glutSwapBuffers();
}
```

```
void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}

/* ARGSUSED1 */
void keyboard (unsigned char key, int x, int y)
{
   switch (key) {
      case 'd':
         day = (day + 10) % 360;
         glutPostRedisplay();
         break;
      case 'D':
         day = (day - 10) % 360;
         glutPostRedisplay();
         break;
      case 'y':
         year = (year + 5) % 360;
         glutPostRedisplay();
         break;
      case 'Y':
         year = (year - 5) % 360;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}
```

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

8.4 関節を持つロボットアームの構築

2 つの部分からなる、関節を持つロボット・アームを描画する。ロボット・アームには拡大縮小した立方体を使用している。ローカル座標系の原点は最初は立方体の中心であり、ローカル座標系を立方体の一方の端に移動させる必要がある。

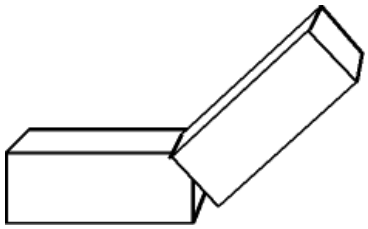

図 19 行列スタックの操作によるロボットアーム

glTranslate*() で回転軸を設定し、glRotate*() で立方体を回転させ、平行移動して立方体の中心に戻る。描画する前に、立方体の拡大縮小を行う。glPushMatrix() と glPopMatrix() により、glScale*() の効果をその場のみに制限している。以下は、腕の部分に使用するプログラムである。

```
        glTranslatef (-1.0, 0.0, 0.0);
        glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
        glTranslatef (1.0, 0.0, 0.0);
        glPushMatrix();
        glScalef (2.0, 0.4, 1.0);
        glutWireCube (1.0);
        glPopMatrix();
```

2 番目の部分を構築するときは、ローカル座標系を次の回転軸に移動する。この座標系は、すでに回転しているため x 軸は回転した腕に沿った方向になっている。そのため、x 軸に沿って平行移動するとローカル座標系が次の回転軸に移動することになる。回転軸に到達したら、最初の時と同じプログラムを使用して 2 番目の部分を描画する。

```
        glTranslatef (1.0, 0.0, 0.0);
        glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
        glTranslatef (1.0, 0.0, 0.0);
        glPushMatrix();
        glScalef (2.0, 0.4, 1.0);
        glutWireCube (1.0);
        glPopMatrix();
```

この作業は、それ以降の部分でも継続的に使用できる。(肩、肘、手首、指)

```
#include <GL/glut.h>
#include <stdlib.h>

static int shoulder = 0, elbow = 0;

void init(void)
{
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glShadeModel (GL_FLAT);
}
```

```
void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glTranslatef (-1.0, 0.0, 0.0);
   glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
   glTranslatef (1.0, 0.0, 0.0);
   glPushMatrix();
   glScalef (2.0, 0.4, 1.0);
   glutWireCube (1.0);
   glPopMatrix();

   glTranslatef (1.0, 0.0, 0.0);
   glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
   glTranslatef (1.0, 0.0, 0.0);
   glPushMatrix();
   glScalef (2.0, 0.4, 1.0);
   glutWireCube (1.0);
   glPopMatrix();

   glPopMatrix();
   glutSwapBuffers();
}

void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective(65.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef (0.0, 0.0, -5.0);
}
```

```
/* ARGSUSED1 */
void keyboard (unsigned char key, int x, int y)
{
   switch (key) {
      case 's':
         shoulder = (shoulder + 5) % 360;
         glutPostRedisplay();
         break;
      case 'S':
         shoulder = (shoulder - 5) % 360;
         glutPostRedisplay();
         break;
      case 'e':
         elbow = (elbow + 5) % 360;
         glutPostRedisplay();
         break;
      case 'E':
         elbow = (elbow - 5) % 360;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

9 イベント処理

9.1 GLUT ライブラリ

本演習では、キーボードの読みとりなどイベント処理に、GLUT ライブラリを使用している。以下に、イベント処理に必要な関数を示す。

入力イベントを処理するには、ウィンドウを作成した後、メインループにはいる前に、次のルーチンを使用してコールバック関数を登録することが必要である。

```
void glutDisplayFunc(void (*func)(void));
```

ウィンドウの内容を再描画する必要がある場合に呼び出す関数を指定する。ウィンドウの内容は、ウィンドウが最初に開かれたとき、ウィンドウがポップされウィンドウの損傷が露呈したとき、glutPostRedisplay() が明示的に呼び出されたときに再描画される。

```
void glutReshapeFunc(void (*func)(void));
```

ウィンドウをサイズ変更、または移動したとき呼び出す関数を指定する。引数 func は、ウィンドウの新規の幅と高さという 2 つの引数を要求する関数を示すポインタである。通常、ディスプレイが新規サイズにクリップされるように、func は glViewport() を呼び出し、射影された画像の縦横比がビューポートと適合し、縦横比のゆがみをさけるため、射影行列を再定義する。glutReshapeFunc() が呼び出されない場合、または NULL を渡して登録解除された場合は、初期の形状変更関数 glViewport(0, 0, width, height) が呼び出される。

```
void glutKeyboadFunc(void (*func)(unsigned int key, int x, int y));
```

ASCII 文字を生成するキーを押したときに呼び出す関数 func を指定する。コールバック・パラメタ key は、生成された ASCII 値である。コールバック・パラメタ x, y は、キーを押したときのマウスの位置 (ウインドウに比例した座標) を示す。

```
void glutMouseFunc(void (*func)(int button, int state, int x, int y));
```

マウスボタンを押したりはなしたりしたときに呼び出す関数 func を指定する。コールバック・パラメタ button は、GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON のいずれかになる。コールバック・パラメタ state は、マウスボタンの放し・押しによって、GLUT_UP または GLUT_DOWN になる。x と y は、イベント発生時のマウスの位置 (ウィンドウに比例した座標) をしめす。

```
void glutMotionFunc(void (*func)(int x, int y));
```

最低一つのマウスボタンを押している間に、ウィンドウ内でマウスポインタが移動したときに呼び出す関数 func を指定する。x と y は、イベント発生時のマウスの位置 (ウィンドウに比例した座標) をしめす。

```
void glutPostRedisplay(void);
```

現在のウインドウを、再描画が必要なものとしてマークする。次の機会に、glutDisplayFunc() が登録したコールバック関数が呼び出される。

イベントループがアイドルする場合など、ほかに未処理のイベントがない場合に実行する関数は、glutIdleFunc() で指定する。これは、連続するアニメーションや、その他のバックグラウンド処理の際に便利である。

```
void glutIdleFunc(void (*func)(void));
```

他に未処理にイベントがない場合に実行する関数 func を指定する。NULL が渡された場合、func の実行は無効化される。

すべての設定が終了したら、GLUT はイベント処理のループ、glutMainLoop() に入る。

```
void glutMainLoop(void);
```

GLUT 処理ループに入り、絶対に戻らない。登録されたコールバック関数は、それに対応するイベントに応答して呼び出される。

9.2 アニメーション

OpenGL で動画 (アニメーション) を作成したい場合、1 秒間に何回も視点やモデルの位置などを変えて描画したものを切り替えることによって実現する。

大部分の OpenGL の動作環境では、描画計算と表示を同時に行うために、2 つの完全なカラー・バッファを供給するハードウェア、もしくはソフトウェアによるダブル・バッファ処理 (double-buffer) の機能を備えている。一方が表示されている間、もう一方で描画がされている。フレームの描画が完了すると、2 つのバッファが交換され表示されていたバッファが描画に使用される。GLUT ライブラリを使用している場合は、次のルーチンを呼び出す。

```
void glutSwapBuffers(void);
```

大部分のアニメーションでは、例えば、ビューアの持つ視点 (viewpoint) の移動、車が道路を少し走る、オブジェクトがわずかに回転するなどと言った各種の変換を使用して、単純にシーン内のオブジェクトが再描画される。描画以外の作業に対して、重要な再計算が必要な場合、結果得られるフレーム速度は多くの場合低下する。

```
#include <GL/glut.h>
static GLfloat spin = 0.0;

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glRotatef(spin, 0.0, 0.0, 1.0);
   glColor3f(1.0, 1.0, 1.0);
   glRectf(-25.0, -25.0, 25.0, 25.0);
   glPopMatrix();
   glutSwapBuffers();
}

void spinDisplay(void)
{
   spin = spin + 2.0;
   if (spin > 360.0)
      spin = spin - 360.0;
   glutPostRedisplay();
}

void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}

void reshape(int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

```
/* ARGSUSED2 */
void mouse(int button, int state, int x, int y)
{
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN)
            glutIdleFunc(spinDisplay);
         break;
      case GLUT_MIDDLE_BUTTON:
         if (state == GLUT_DOWN)
            glutIdleFunc(NULL);
         break;
      default:
         break;
   }
}

/*
 *  Request double buffer display mode.
 *  Register mouse input callback functions
 */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```

この例は、glutSwapBuffers() を使用して、回転する正方形を描画している。また、GLUT を使用して入力デバイスを制御したり、アイドル機能をオン・オフする方法も示している。この場合、マウスボタンが回転のオン・オフを切り替えている。

◇ **課題** 3 ◇

1. sample-teapot.c をコンパイルし，ティーポットがマウスの操作にあわせて回転・拡大することを確認せよ．また，表示するオブジェクトを変更してみよ．
2. planet.tar.gz をダウンロードし，planet_ex.c に月を追加してみよ。また、複数の月や惑星を追加してみよ。座標系の位置と方向の保存・リストアには glPushMatrix() と glPopMatrix() を使用する。惑星の周囲に複数の月を描く場合、各々の月を配置する前に座標系を保存し、月を描き終えたら座標系をリストアすることが必要.
3. planet_ex.c において，太陽や惑星の自転・公転をアニメーションで実現せよ。
4. planet_ex.c において，マウスにより太陽系をいろいろな角度から観測できるようにせよ。
5. sample-robot.c を修正して、同じ位置に部分を追加してみる。例えば、手首に数本の指を追加してみる (下図)。手首における座標系の位置と方向の保存・リストアには glPushMatrix() と glPopMatrix() を使用する。手首に指を描画する場合は、各指を配置する前に現在の行列を保存し、指の描画を描き終えたら現在の行列をリストアすることが必要。 (オプション)

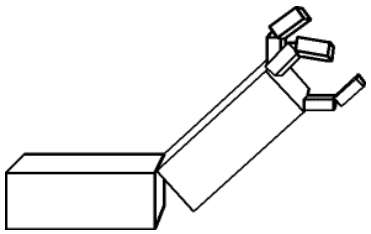

図 20　ロボットアーム