R01–Ec5 プログラミング演習 IV テキスト

（R01/09/09–09/13，高橋　章）

【第 1 日】
2D グラフィックス

1　はじめに

本演習では，C/C++ 言語で 2D/3D グラフィックスを扱うための手段の一つである OpenGL[1] および OpenGL Utility Toolkit（GLUT）の使用法を，ごく簡単かつ大雑把に紹介する．紹介する方法は，あくまで「手段の一つ」であり「定番」or「最善の方法」とは限らない．また，記載事項にはタイプミスをはじめ様々な誤りが含まれる可能性がある．

細かい説明や正しい定義などは，入門書 [4, 5] や Web サイト [6] を参照して，各自で調べながら学習を進めてほしい．もう少し本格的に 3D グラフィックスを学びたいのであれば，入門書 [5] を購入することを推奨する（下敷きになったサイト [6] からも十分な情報が得られる）．

1.1　準備

OpenGL プログラミングを進めていく上で，準備すべきこと（モノ）は次の通り：

- **コンピュータ・OS**：OpenGL はプラットフォームに依存しない．Windows でも Mac でも Linux でもよい（はず）．本テキストでは Windows を利用するものとして話を進める．
- **C/C++ コンパイラ**：Windows アプリケーションが作成可能であること．以下では Embarcadero Free C++ compiler（BCC）を利用するものとして話を進める．
- **テキストエディタ**：ソースファイルを編集するために必要である．

1.2　GLUT のインストール

OpenGL ライブラリと補助ライブラリは，BCC に含まれているが，GLUT は含まれていない．そこで GLUT をインストールする必要がある．

1. GLUT の配布元 [2] から `glut-3.7.6-bin.zip` をダウンロードする．ここからドキュメント（英文）や，ソース `glut-3.7.6-src.zip` もダウンロードできる[*1]．ソースには多くのサンプルプログラムが収録されている．
2. `glut-3.7.6-bin.zip` を解凍（展開）する．
3. 解凍されたファイルがあるディレクトリで以下のコマンドを実行する[*2]：

```
implib glut32.lib glut32.dll
```

4. 以下の 3 ファイルを移動またはコピーする：
   - `glut32.dll`：32bit OS では `c:\Widnows\System32` へ，64bit OS では `c:\Windows\SysWOW64` へ[*3]
   - `glut.h`：BCC インストール先の `include\windows\sdk\gl` へ
   - `glut32.lib`：BCC インストール先の `lib\win32c\release\psdk` へ

1.3　OpenGL プログラムのコンパイル法

OpenGL プログラム `prog1.c` を BCC でコンパイルするには，次のコマンドを実行する：

```
bcc32c prog1.c
```

効率的あるいは大規模なプログラム開発を行うために，`MAKE` ユーティリティの利用も検討するとよい．`MAKE` を利用するには，予めプログラム開発を行う（ソースファイルを置く）ディレクトリにリスト 1 のような `MAKEFILE` を作成しておく．

ソースファイル `prog1.c` をコンパイルして `prog1.exe` を作成するには，次をコマンド入力する：

```
make prog1.exe
```

すると，`MAKE` はリスト 1 の 3，4 行目で指定されたルールに従って，コンパイル処理を行う．

また次のようにすると，コンパイル時に生成された中間ファイルを削除することができる[*4]：

[*1] 開発は長らく停止しているので，代わりに freeglut を検討してもよいだろう（p. 14 のコラム参照）．

[*2] 解凍ファイルにも `glut32.lib` が含まれるが，Visual C++ 用であるため BCC でリンクすることができない．

[*3] あるいは BCC インストール先の `bin` など，パスの通ったフォルダでもよいはず．

[*4] `clean` は擬似ターゲットと呼ばれる．リスト 1 には中間

リスト 1　MAKEFILE

```
1  CC = bcc32c
2
3  .c.exe:
4      $(CC) $(CFLAGS) $&.c
5
6  clean:
7      -@if exist *.tds del *.tds >nul
8
9  purge:
10     -@if exist *.tds del *.tds>nul
11     -@if exist *.exe del *.exe>nul
```

```
make clean
```

2　2D グラフィックスプログラム

Ec2 情報処理から学習してきた C 言語プログラムは，記述してある命令を順に実行するような方式（**逐次実行型**）であった．これに対し，今回学習するウィンドウを開く GUI アプリケーションは，**イベント駆動型（event-driven）**と呼ばれる方式で動作する．すなわち，ユーザが行うキー入力，マウスの移動やボタンクリック，ウィンドウの移動やサイズ変更などの操作をイベントとして扱い，イベント発生時に，それぞれどのような処理を行うかを**コールバック（callback）関数**として記述しておく．そして，プログラム起動後にコールバック関数を**イベントハンドラ（event handler）**としてシステムに登録し，イベント待機の処理に入る．

以下では簡単な 2 次元グラフィックスプログラムにおけるコールバック関数の記述法を紹介する．

2.1　最小限のプログラム

ウィンドウを開くだけの最小限の OpenGL プログラムをリスト 2 に示す（**付録 A** 参照）．

リスト 2 はイベント駆動型プログラムである．`main` 関数は，ウィンドウの作成やイベントハンドラ登録などの前処理部分（12〜15 行目）と，イベント待機の開始（16 行目）で構成される．

2.2　ウィンドウ内を塗りつぶす

画面再描画時に，画面内を指定の色で塗りつぶす機能をリスト 2 に付加してみよう．まず，色の指定を**光の三原色**（Red，Green，Blue）の**加法混色**で指定するために，`main` 関数の `glutCreateWindow` の

ファイルと実行ファイルを全て削除する擬似ターゲット `purge` も指定してある．

リスト 2　最小限の OpenGL プログラム

```
1  #include <GL/glut.h>
2
3  void display(void) {
4      /* 描画命令 */
5  }
6
7  void init(void) {
8      /* 初期化命令 */
9  }
10
11 int main(int argc, char** argv) {
12     glutInit(&argc, argv);
13     glutCreateWindow(argv[0]);
14     glutDisplayFunc(display);
15     init();
16     glutMainLoop();
17     return 0;
18 }
```

リスト 3　ウィンドウ内塗りつぶし

```
1  void display(void) {
2      glClear(GL_COLOR_BUFFER_BIT);
3      glFlush();
4  }
5
6  void init(void) {
7      glClearColor(0.0, 0.0, 0.0, 1.0);
8  }
```

前に次行を挿入する．

```
glutInitDisplayMode(GLUT_RGBA);
```

次に，`display` 関数と `init` 関数をリスト 3 のように定義する．`init` 関数内で呼び出している `glClearColor` 関数の 4 つの引数は，順に R，G，B，α 値を表す．R，G，B の各成分は 0〜1 の実数値で指定するが，すべて 0 であれば黒となる[*5]．

◇演習 1．加法混色： 光の三原色による加法混色について調べよ．次に `glClearColor` の引数を変更して Red，Green，Blue，Cyan，Magenta，Yellow，White などでウィンドウを塗りつぶすプログラムを作成せよ．

2.3　線図を描く

`display` 関数をリスト 4 のように定義すると，画面内に赤い十字が描かれる．`glColor3d` は，描画色を RGB 値で指定する関数である．`glBegin` と

[*5] α 値は不透明度を表し，0 で透明，1 で不透明となる．

リスト 4　赤い十字描画

```
1  void display(void) {
2      glClear(GL_COLOR_BUFFER_BIT);
3      glColor3d(1.0, 0.0, 0.0);
4      glBegin(GL_LINES);
5      glVertex2d(-0.8, 0.0);
6      glVertex2d( 0.8, 0.0);
7      glVertex2d( 0.0, -0.8);
8      glVertex2d( 0.0, 0.8);
9      glEnd();
10     glFlush();
11 }
```

glEnd は，それぞれ図形定義の開始と終了を意味する．glVertex2d は 2 次元の座標値を設定する関数である．リスト 4 では glBegin の引数として GL_LINES が指定されているので，2 点毎に線分が描画される．すなわち $(-0.8, 0)$ と $(0.8, 0)$ を両端点とする線分と，$(0, -0.8)$ と $(0, 0.8)$ を両端点とする線分が描画される．

◇**演習 2. 線図の種類：** リスト 4 の glBegin 関数の引数を GL_POINTS，GL_LINE_STRIP，GL_LINE_LOOP に変えたプログラムを作成し，それぞれの違いを確認せよ．

2.4　正三角形を描く

display 関数をリスト 5 のように定義すると，画面内に赤い正三角形が描かれる[*6]．

◇**演習 3. 正多角形の描画：** リスト 5 を参考に，正方形，正五角形，正六角形，正七角形，正八角形を描画するプログラムを作成せよ．

◇**参考：** glBegin 関数の引数として，他にも GL_POLYGON，GL_QUADS，GL_QUAD_STRIP，GL_TRIANGLE_STRIP，GL_TRIANGLE_FAN などが指定できる．

2.5　文字を出力する

display 関数をリスト 6 のように定義すると，x 軸，y 軸に相当する L 字型の赤い線が描かれ，原点（直角部分付近）に黄色い文字「0」が出力される．

文字を出力するには，基準座標（文字を囲う矩形領域の左下）を glRasterPos2d で定め，glutBitmapCharacter 関数でフォントと文字を引数に与えればよい．リスト 6 では，8×13 画素の固定幅フォントを選択している．1 文字出力すると，基準座標は自動的に 1 文字分右へずれるので，連続して文字を出力する場合に，1 文字ごとに glRasterPos2d を呼ぶ必要はない．この方法で描画可能な文字は ASCII 文字に限られ，π のような記号や日本語などの多バイト文字を出力することはできない．

◇**参考：** glutBitmapCharacter 関数の第 1 引数として，以下が指定できる：

- GLUT_BITMAP_9_BY_15
- GLUT_BITMAP_TIMES_ROMAN_10
- GLUT_BITMAP_TIMES_ROMAN_24
- GLUT_BITMAP_HELVETICA_10
- GLUT_BITMAP_HELVETICA_12
- GLUT_BITMAP_HELVETICA_18

リスト 5　赤い正三角形描画

```
1  void display(void) {
2      int i;
3      double theta,dt,x,y;
4      glClear(GL_COLOR_BUFFER_BIT);
5      glColor3d(1.0, 0.0, 0.0);
6      dt = 2.0*M_PI/3.0;
7      theta = 0.0;
8      glBegin(GL_TRIANGLES);
9      for (i = 0; i < 3; i++) {
10         x = cos(theta);
11         y = sin(theta);
12         glVertex2d(x, y);
13         theta += dt;
14     }
15     glEnd();
16     glFlush();
17 }
```

リスト 6　文字の出力

```
1  void display(void) {
2      glClear(GL_COLOR_BUFFER_BIT);
3      glColor3d(1.0, 0.0, 0.0);
4      glBegin(GL_LINE_STRIP);
5      glVertex2d(0, 1);
6      glVertex2d(0, 0);
7      glVertex2d(1, 0);
8      glEnd();
9      glColor3d(1.0, 1.0, 0.0);
10     glRasterPos2d(0, 0);
11     glutBitmapCharacter(
12       GLUT_BITMAP_8_BY_13, '0'
13     );
14     glFlush();
15 }
```

[*6] math.h をインクルードする必要がある．また，6 行目で使用している M_PI は円周率 π の近似値を定義したマクロで，BCC 以外では未定義でエラーとなる可能性がある．

2.6 ウィンドウサイズを変更する

ここまで作成したプログラムでは，最初に開かれるウィンドウサイズ[*7]，その中に描かれる図形の描画領域[*8]はデフォルトのままである．これでは，ユーザがウィンドウサイズを変更（resize）したとき，図形の縦横比（aspect ratio）が変わってしまう．

リサイズ時にも縦横比を一定に保つためには，リサイズイベントに対するコールバック関数を指定する必要がある．このためには OpenGL の座標系を理解し，図形を指定している**ワールド座標系**（**WCS**; world coordinate system）と，画面に表示されたウィンドウで用いられる**デバイス座標系**（**DCS**; device ···）の関係を正しく設定しなければならない．

2.6.1 OpenGL の座標系

OpenGL の WCS は 3 次元の右手座標系として定義される．2D で描画するときは xy 平面（$z = 0$）のみを扱うことになる．DCS は 2 次元の座標系で，ウィンドウの左下を原点とする整数値で指定する．

WCS の座標値の範囲は自由に定めてよいが，これを有限サイズの画面上で表現するために，描画されるべき領域を指定して切り出す**クリッピング処理**を行う必要がある．OpenGL では，この処理をプログラマが実装する必要はなく，適切な設定を行うだけでよい．そのためには，左下が $(-1,-1)$，右上が $(1,1)$ の正方形領域からなる**正規化デバイス座標系**を考え，WCS から正規化デバイス座標系への変換（**ウィンドウイング変換**）と，正規化デバイス座標系から DCS への変換（**ビューポート変換**）の 2 段階の変換を指定する．

2.6.2 ウィンドウサイズの初期設定

プログラム起動時に開くウィンドウサイズを 640×480 画素に指定するには，`glutCreateWindow` の前に次のように指定する：

```
glutInitWindowSize(640, 480);
```

2.6.3 ウィンドウリサイズのコールバック関数

ウィンドウがリサイズされたときに発生するイベントに対応するコールバック関数は，ウィンドウの幅 `w` と高さ `h` を画素単位で受け取ることができる．リサイズコールバック関数 `resize` をイベントハンドラに登録するには，次のようにする[*9]．

```
glutReshapeFunc(resize);
```

[*7] たぶん 300 × 300 画素

[*8] たぶん 1.0 × 1.0

[*9] `main` の `glutDisplayFunc` の後にでも指定すればよい．

リサイズ後のウィンドウ全体をビューポートとし，ビューポートの縦横比にあわせてクリッピングウィンドウを設定するためのコールバック関数例をリスト 7 に示す．

リスト 7　リサイズのコールバック関数例

```
1 void resize(int w, int h) {
2     glViewport(0, 0, w, h);
3     glMatrixMode(GL_PROJECTION);
4     glLoadIdentity();
5     gluOrtho2D(
6       -w/200.0, w/200.0,
7       -h/200.0, h/200.0
8     );
9 }
```

`glViewport` はビューポート変換を指定する関数で，ウィンドウ内で描画を行う長方形（矩形）領域を指定する．ウィンドウ左下から描画領域の左下までの余白を表す `(x,y)`，描画領域の幅 `w` と高さ `h` を画素単位で引数に与える：

```
glViewport(x, y, w, h);
```

関数 `glMatrixMode` は OpenGL が内部で扱う二つの変換行列の切り替えを行うもので，リスト 7 では**投影行列**と呼ばれる，WCS から正規化デバイス座標系への変換行列を選択している（**4.2** 参照）．

関数 `glLoadIdentity` は選択した変換行列を単位行列に設定する．関数 `gluOrtho2D` は WCS 中で描画対象の矩形領域を指定する．矩形領域の左下座標 `(l,b)`，右上座標 `(r,t)` から引数を与える：

```
gluOrtho2D(l, r, b, t);
```

リスト 7 では矩形領域を $(-\frac{w}{200},-\frac{h}{200})$〜$(\frac{w}{200},\frac{h}{200})$ としている．これにより，ウィンドウサイズが 300×200 画素の時は，WCS の $(-1.5,-1.0)$〜$(1.5,1.0)$ の範囲が描画される．また，ウィンドウサイズが 200×400 画素の時は，WCS の $(-1.0,-2.0)$〜$(1.0,2.0)$ の範囲が描画されることになる．すなわち，ウィンドウサイズを変更すると，その分だけ WCS 中の描画範囲が広がり，ウィンドウ中の図形の見かけの大きさや縦横比は変わらない．

◇演習 4．拡大縮小するウィンドウ： ウィンドウサイズを 2 倍にしたら，ウィンドウ中の図形が 2 倍に拡大されて見えるようなリサイズコールバック関数を作成せよ．ただし，ウィンドウの縦横比が変化しても，描画される図形の縦横比が変わらないよ

うにせよ．例えばウィンドウサイズが 300×200 画素の時は，WCS の $(-1.5,-1.0)$〜$(1.5,1.0)$ の範囲が描画され，ウィンドウサイズが 200×400 画素の時は，WCS の $(-1.0,-2.0)$〜$(1.0,2.0)$ の範囲が描画されるような変換を考えればよい．

3 2D アニメーションプログラム

図形が動いているような効果を**アニメーション**（**animation**）と呼ぶ．一般にアニメーションは，静止画を少しずつ変えながら連続表示させて実現する．これは，短い間隔で絵が切り替わるとき，人間の眼は絵の不連続性を認識できずに，滑らかに動いていると感じる**残像効果**を利用している．

3.1 アイドルコールバック関数の登録

システムが他に処理するイベントが無い空き状態を**アイドル**（**idle**）状態と呼ぶ．アイドル状態になった場合にアイドルコールバック関数を呼び出させることで，何がしかの処理をさせることができる．アイドルコールバック関数 idle のイベントハンドラの登録は，次のようにする[*10]．

```
glutIdleFunc(idle);
```

3.2 アイドルコールバック関数例

再描画要求のイベントを登録するだけのアイドルコールバック関数例をリスト 8 に示す．

リスト 8　アイドルコールバック関数例

```
1 void idle(void) {
2     glutPostRedisplay();
3 }
```

アイドルコールバック関数から再描画が要求されると，描画を行うコールバック関数（display）が呼び出される．リスト 4 やリスト 5 では，何度描き直されても見え方が変化しない．そこで，図形を原点まわりに少しずつ回転させてみよう．これを実現するには，次の二つの手段がある：

- 図形の頂点を回転させる．
- 座標系やビューポートを回転させる．

ここでは直感的に理解しやすい前者で実装を行う．

図形の回転を実現するには，回転角を記憶しておく必要があるが，描画用コールバック関数について，次の二つの制約がある：

- 関数を呼び出すのはシステムである．
- 関数は引数を受け取ることができない．

これらの制約のもとで回転角を記憶しておき，次回の描画で利用するための二つの方法を紹介する．

■**グローバル変数を使う：** 回転角 rotAng を**グローバル変数**として関数定義の外で定義し，利用する例をリスト 9 に示す．

■**スタティック変数を使う：** 回転角 rotAng を関数 display 内の**スタティック変数**として定義し，利用する例をリスト 10 に示す．

リスト 9，リスト 10 の何れも，1 回描画するたびに図形を 3 度ずつ回転させている[*11]．

リスト 9　グローバル変数の利用

```
 1 double rotAng=0.0;
 2
 3 void display(void) {
 4     int i;
 5     double theta,dt,x,y;
 6     glClear(GL_COLOR_BUFFER_BIT);
 7     glColor3d(1.0, 0.0, 0.0);
 8     dt = 2.0*M_PI/3.0;
 9     theta = rotAng;
10     glBegin(GL_TRIANGLES);
11     for (i = 0; i < 3; i++) {
12         x = cos(theta);
13         y = sin(theta);
14         glVertex2d(x, y);
15         theta += dt;
16     }
17     glEnd();
18     glFlush();
19     rotAng += 3.0*M_PI/180.0;
20 }
```

リスト 10　スタティック変数の利用

```
1 void display(void) {
2     static double rotAng=0.0;
3     int i;
4     double theta,dt,x,y;
5     /* グローバル変数利用の場合と同じ処理 */
6     glFlush();
7     rotAng += 3.0*M_PI/180.0;
8 }
```

[*10] これも glutDisplayFunc の後にでも指定すればよい．

[*11] C 言語の数学関数（cos, sin など）の引数は，弧度（radian）単位で与えなければならない．第 2 日に扱う glRotate 関数（**6.1**）とは単位が異なるので注意が必要である．

3.3 描画速度の制御

アイドルコールバックで再描画イベントを発生させるだけのアニメーションでは，コンピュータの処理速度や同時に起動されているアプリケーションの負荷，描画モデルの複雑さなどによって描画の更新速度が異なるという問題がある．これはアクションゲームなどを作成する場合に，実行環境によって難易度が変わるという問題に直結する．

1 回の描画が十分高速に終わる場合で，かつ次の描画開始のタイミングをモデルや実行するハードウェアに依存せずにほぼ一定に保つ手段として，GLUT にはタイマコールバックが用意されている．ユーザが作成する関数 `timer` を `msec` ミリ秒後に呼び出すためには次のように記述する：

```
glutTimerFunc(msec, timer, dummy);
```

`dummy` は `int` 型変数で，`timer` の引数にセットされる．`timer` 関数の実装例をリスト 11 に示す．この関数では 100 ミリ秒後に再び自分自身(関数 `timer`)を呼び出すようにタイマコールバックをセットし，再描画を要求する[*12]．1 回の描画（`display` の処理）が 100[ms] 未満で終了するならば，このプログラムは 10fps（frames per second）の描画速度で動作し，1 回の描画毎に物体が指定された角度だけ回転する[*13]．

リスト 11　タイマコールバック関数例

```
1 static void timer(int dummy) {
2     glutTimerFunc(100, timer, 0);
3     glutPostRedisplay();
4 }
```

◇演習 5. 正多角形の回転： 演習 3 で作成した正多角形を，タイマコールバックを利用して回転させるプログラムを作成せよ．

3.4 ダブルバッファリング

ここまで作成してきたアニメーションプログラムでは，表示画面にチラつきを感じるかもしれない．これは，OpenGL が描画している画面を，そのままユーザに見せるというシングルバッファ（single buffer）処理をしているためである．より滑らかなアニメーションを実現するためには，描画処理を行う画面と，表示画面を切り替えるダブルバッファリング（**double buffering**）を利用するとよい．ダブルバッファリングを利用するには，次のようにプログラムを書き換える．

- `glutInitDisplayMode` の引数を変更する：

  ```
  GLUT_RGBA | GLUT_DOUBLE
  ```

- `glFlush()` を `glutSwapBuffers()` で置き換える．

課題 I

1. **プログラミングのツール**：BCC には MAKE ユーティリティの他，grep や touch などプログラム開発を行う際に役に立つコマンドラインツールが含まれている．それぞれの機能や利用法について学習せよ．
2. **C 言語**：変数の「有効範囲（スコープ)」，「記憶域期間（記憶寿命)」について学習し，一般的な（ローカル）変数とグローバル変数，スタティック変数の違いを整理せよ．
3. 図 1，図 2 のような図形（星型正多角形）を描き，回転させるプログラムを作成せよ．
4. 図 3，図 4 のような図形（完全グラフ）を描き，回転させるプログラムを作成せよ．
5. リスト 12 を解析し，数学関数のグラフを描くプログラムを作成せよ．例えば θ を 0 から 2π まで変化させながら，次の関数をプロットするとどんな図形が描かれるだろうか（コラム参照)．余力がある学生は，座標軸に目盛（文字）を加えてみよ．

   (a) カージオイド（cardioid）

   $$x = \cos\theta\,(1 + \cos\theta) \tag{1}$$
   $$y = \sin\theta\,(1 + \cos\theta) \tag{2}$$

   (b) サイクロイド（cycloid）

   $$x = \theta - \sin\theta \tag{3}$$
   $$y = 1 - \cos\theta \tag{4}$$

   (c) 4 尖点の内サイクロイド（hypocycloid）

   $$x = \cos^3\theta \tag{5}$$
   $$y = \sin^3\theta \tag{6}$$

○ヒント：

- 課題 I–3 や I–4 では，正五角形や正七角形の頂点座標を配列に格納して利用するとよい．線分を描く頂点をどのようなルールで指定すると効率がよくなるか，考えながら実装せよ．

[*12] 自分自身を呼び出すような関数を**再帰関数**と呼ぶ．再帰関数は関数宣言の先頭に `static` を付けることを推奨する．リスト 11 も再帰関数の一種である．

[*13] 一般に 10fps 以上の描画速度を**リアルタイム**（**realtime**）の描画と呼ぶ．

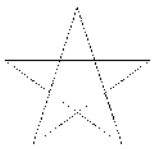

図 1　星型五角形

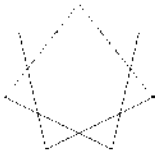

図 2　星型七角形

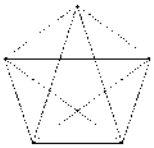

図 3　K_5

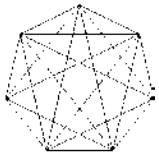

図 4　K_7

- 課題 I–5 では，それぞれの図形の x，y 座標値の範囲を事前に手計算で求め，適切なビューポート設定を行うとよい．余力がある学生は，配列に座標値を格納して，最小値・最大値を求め，自動的に描画範囲を調整する機能の実装を検討するとよい．

コラム：媒介変数形式

OpenGL で半径 r の円を描くことを考えてみよう．円の式は，$x^2 + y^2 = r^2$ であるが，この式をそのまま使って，円を描くことはできないであろう．そこで，これを y について解き，$-1 \leqq x \leqq 1$ について

$$y_1 = \sqrt{r^2 - x^2}$$
$$y_2 = -\sqrt{r^2 - x^2}$$

を順次求めて描画することも一案であるが，プロットする点の間隔が均等にならないため，あまり滑らかに描けない．そこで，一般的には**媒介変数** θ を導入し，$0 \leqq \theta < 2\pi$ について $(r\cos\theta, r\sin\theta)$ をプロットして円を描く．

リスト 12　グラフプロット例

```
1  #include <GL/glut.h>
2  #include <math.h>
3
4  void display(void) {
5      double x,y;
6      glClear(GL_COLOR_BUFFER_BIT);
7      glColor3d(0.0, 0.0, 0.0);
8      glBegin(GL_LINES);
9      glVertex2d(0.0, 0.0);
10     glVertex2d(2.0*M_PI, 0.0);
11     glVertex2d(0.0, -1.0);
12     glVertex2d(0.0, 1.0);
13     for (x = 1; x <= 2.0; x += 1.0) {
14         glVertex2d(x*M_PI, -0.05);
15         glVertex2d(x*M_PI, 0.05);
16     }
17     for (y = -1; y <= 1; y += 2) {
18         glVertex2d(-0.05, y);
19         glVertex2d( 0.05, y);
20     }
21     glEnd();
22     glColor3d(1.0, 0.0, 0.0);
23     glBegin(GL_LINE_STRIP);
24     for (x = 0; x < 2*M_PI; x += 0.2) {
25         y = sin(x);
26         glVertex2d(x, y);
27     }
28     glEnd();
29     glFlush();
30 }
31
32 void resize(int w, int h) {
33     glViewport(0, 0, w, h);
34     glMatrixMode(GL_PROJECTION);
35     glLoadIdentity();
36     gluOrtho2D(-0.5, 6.5, -1.5, 1.5);
37 }
38
39 void init(void) {
40     glClearColor(1.0, 1.0, 1.0, 1.0);
41 }
42
43 int main(int argc, char** argv) {
44     glutInit(&argc, argv);
45     glutInitWindowSize(700, 300);
46     glutInitDisplayMode(GLUT_RGBA);
47     glutCreateWindow(argv[0]);
48     glutDisplayFunc(display);
49     glutReshapeFunc(resize);
50     init();
51     glutMainLoop();
52     return 0;
53 }
```

【第 2 日】

3D グラフィックス

4　3D グラフィックスプログラム

OpenGL では 3D グラフィックス描画を**合成カメラモデル（synthetic-camera model）**と呼ばれる概念を使って実現する．これは仮想の 3 次元世界に物体，照明，カメラ等を配置して撮影する（シャッタを切る）と，2 次元の画像（写真）が生成され，画面に表示されると考えることができる．OpenGL のカメラは，人間の眼や現実のカメラを模した**透視投影（perspective projection）**モデルや，製図や数学で扱われる**正射影（orthogonal projection）**モデルから選択できる．ただし，物体形状の定義や仮想空間への配置，カメラモデルの選択など，ありとあらゆる設定を，しかるべき OpenGL 関数に適切な数値を引数として渡すことで行わなければならないため，数学（特に行列，ベクトルなどの線形代数の知識）が不可欠である．本演習では簡便な正射影モデルを扱う．

4.1　正射影の基礎

正射影は，物体の各辺の長さが変わらずに画像上に投影するため，**isometric projection** とも呼ばれる．正射影によって描かれた図をアイソメ図と呼ぶこともある．ちなみに orthogonal は，空間中の点と，その点が写しこまれた画像上の点（**投影像**と呼ぶ）を結ぶ線分が，画像平面と常に直交することを意味する．また，正射影は**平行投影（parallel projection）**と呼ばれることも多い．これは空間中の点と，その投影像を結ぶ線分が，画像平面の法線と常に平行であることを意味する．

4.2　正射影カメラの投影行列

OpenGL でカメラを設定するには，**投影行列**と呼ばれる行列を設定する関数を呼び出す[*14]：

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(l, r, b, t, n, f);
```

投影中心（視点）を原点として，`l`（left），`r`（right），`b`（bottom），`t`（top），`n`（near），`f`（far）の 6 変数で定まる直方体内部が描画処理の対象となる[*15]．

[*14] ウィンドウリサイズのコールバック関数あたりに書けばよい（**8** 参照）．なお，リスト 7 で用いた `gluOrtho2D` は，`glOrtho` で `n=-1`，`f=1` とするのと等価である．

[*15] 当然 `l` < `r`，`b` < `t`，`n` < `f` でなければならない．

■座標軸の描画：　x, y, z 軸をそれぞれ Red, Green, Blue で描画する例をリスト 13 に示す[*16]．

4.3　カメラの配置

リスト 13 は，2D グラフィックスと同じような描画結果となり，z 軸（青い線分）が描かれない．これは，デフォルトではカメラの投影中心（視点）が WCS 原点にあり，画像が xy 平面と一致し，z 軸が画面に垂直方向となっているためである．

典型的な 3 次元グラフィックスの出力を得るた

リスト 13　座標軸の描画例

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

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_LINES);
    glColor3d(1, 0, 0);
    glVertex3d(0, 0, 0);
    glVertex3d(1, 0, 0);
    glColor3d(0, 1, 0);
    glVertex3d(0, 0, 0);
    glVertex3d(0, 1, 0);
    glColor3d(0, 0, 1);
    glVertex3d(0, 0, 0);
    glVertex3d(0, 0, 1);
    glEnd();
    glFlush();
}

void resize(int w, int h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-2, 2, -2, 2, -2, 2);
}

void init(void) {
    glClearColor(0.0, 0.0, 0.0, 1.0);
}

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

[*16] `glOrtho` の引数は本来はすべて実数である．

めには，カメラを仮想三次元空間中の適切な位置・姿勢に配置しなければならない．カメラの視点を $E(\mathtt{ex},\mathtt{ey},\mathtt{ez})$，注視点を $A(\mathtt{ax},\mathtt{ay},\mathtt{az})$，画像の上方向を定めるベクトル（up ベクトル）を $\boldsymbol{u}=(\mathtt{ux},\mathtt{uy},\mathtt{uz})$ に配置するには，次のようにする：

```
gluLookAt(ex,ey,ez,ax,ay,az,ux,uy,uz);
```

このとき，画像は視点 E を通り，線分 EA に垂直な平面として配置される．線分 EA 周りの回転角を定めるためにベクトル $\boldsymbol{u}$ が利用される．ただし，線分 EA とベクトル $\boldsymbol{u}$ が直交する必要はない．

◇演習 6．視点の設定： リスト 13 の `init` 関数中で，視点を $(1,1,1)$，注視点を $(0,0,0)$，up ベクトルを $(0,1,0)$ に設定してみよ．

5 物体形状の取り扱い

5.1 物体形状モデル

OpenGL では物体の形状を**ワイヤフレーム（wireframe）**モデル，または**サーフェイス（surface）**モデルで表現する．前者は頂点と辺だけで物体を表現し，後者は面の集合として物体を表現する．なお，現実世界の実物体は**ソリッド（solid）**モデルと呼ばれる．サーフェイスモデルでは物体内部が空（**中空**）なのに対し，ソリッドモデルでは物体内部の物質（**中実**）を考慮しなければならない．

以下では，ワイヤフレームモデルを説明する．

5.2 ユーザ定義の形状

物体形状を定義して描画するには，2D グラフィックスの場合と同様に，物体の頂点座標を定義し，頂点間を結ぶ辺（線分）を OpenGL に渡せばよい．例えば，原点を中心とする半径 1 の球に内接する正四面体のワイヤフレームモデルを定義するには，リスト 14 のようにすればよい．

◇演習 7．正四面体の描画： リスト 14 を参考に正四面体を描画するプログラムを完成させよ．

○ヒント： リスト 13 の `display` 関数にリスト 14 を融合させればよい．

◇演習 8．正六面体・正八面体の描画： 原点を中心とする半径 1 の球に内接する正六面体を描画するプログラムを作成せよ．同様に半径 1 の球に内接する正八面体を描画するプログラムを作成せよ．

5.3 GLUT の基本立体

前節のように物体の頂点や辺の接続を，プログラマ自身が全て定義するのは煩雑である[*17]．数学的な知識が十分でないユーザでも手軽に CG 制作ができるように，GLUT では既製の物体として，表 1 に示す基本立体を用意している．球，円錐，トーラスの引数のうち `GLdouble` 型（実数）のものは半径や高さといった寸法を設定するパラメータであり，`GLint` 型（整数）のものはワイヤフレームの線の数（分割数）を設定するパラメータである．

リスト 14　ワイヤフレーム正四面体の定義

```
 1  int i,j;
 2  GLdouble p[4][3];
 3  p[0][0] = 1.0;
 4  p[0][1] = p[0][2] = p[1][2] = 0.0;
 5  p[1][0] = p[2][0] = p[3][0] = -1.0/3;
 6  p[2][1] = p[3][1] = -sqrt(2.0)/3;
 7  p[1][1] = -2.0*p[2][1];
 8  p[2][2] = sqrt(6.0)/3.0;
 9  p[3][2] = -p[2][2];
10  glBegin(GL_LINES);
11  for (i = 0; i < 3; i++) {
12      for (j = i+1; j < 4; j++) {
13          glVertex3dv(p[i]);
14          glVertex3dv(p[j]);
15      }
16  }
17  glEnd();
```

正四面体〜正二十面体は**プラトン（Plato）の立体**とも呼ばれる正多面体である．Teapot は「ユタ大学のティーポット」と呼ばれる立体モデルで，CG の例としてよく登場する．この立体は 192 個の頂点を持つらしい [4]．

◇演習 9．基本立体の描画： 表 1 の GLUT の基本立体の引数の意味を自分で調べ，これらを描くプログラムを作成せよ．

5.4 GLU の 2 次曲面

GLUT の基本立体にくらべて手続きが煩雑となるが，GLU[*18]の 2 次曲面として球，円柱（円錐台），円盤，部分円盤が用意されている．これらを描画するには，まずグローバル変数として次のような変数（厳密には構造体へのポインタ）を宣言する：

```
GLUquadric *myQuad;
```

次に，`init`（または `main`）関数に以下を加える：

```
myQuad = gluNewQuadric();
gluQuadricDrawStyle(myQuad, GLU_LINE);
```

そして `display` 関数で表 2 の関数を呼び出す．

[*17] 接触判定や照光処理などの高次な処理を行う場合には，自分で定義しなければならない場合が多くなる．

[*18] GLU は OpenGL Utility Library の略．

表 1　GLUT の基本立体

種類	関数プロトタイプ
球	`glutWireSphere(GLdouble radius, GLint slices, GLint stacks);`
円錐	`glutWireCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);`
トーラス	`glutWireTorus(GLdouble inner, GLdouble outer, GLint sides, GLint slices);`
正四面体	`glutWireTetrahedron();`
正八面体	`glutWireOctahedron();`
正十二面体	`glutWireDodecahedron();`
正二十面体	`glutWireIcosahedron();`
Teapot	`glutWireTeapot(GLdouble size);`

表 2　GLU の 2 次曲面

種類	関数プロトタイプ
球	`gluSphere(GLUquadric* obj, GLdouble radius, GLint slices, GLint stacks);`
円柱	`gluCylinder(GLUquadric* obj, GLdouble base, GLdouble top, GLdouble height, GLint slices, GLint stacks);`
円盤	`gluDisk(GLUquadric* obj, GLdouble inner, GLdouble outer, GLint slices, GLint rings);`
部分円盤	`gluPartialDisk(GLUquadric* obj, GLdouble inner, GLdouble outer, GLint slices, GLint rings, GLdouble start, GLdouble angle);`

◇演習 10. 基本立体の描画： 表 2 の GLU の 2 次曲面の引数の意味を自分で調べ，これらを描くプログラムを作成せよ．

6　3D アニメーション

3D グラフィックスでアニメーションを行うための手続きは，2D アニメーションで解説した方法と同じである．具体的には，まず繰り返し描画を行うためにアイドルコールバック関数やタイマコールバック関数を登録して，再描画イベントを発生させるようにする．そして，`display` 関数で適切な描画を行えばよい．その際，滑らかに描画を行うためにダブルバッファリングを使うとよい．

以下では，立体図形（物体）を回転させるアニメーションの実現を考える．

6.1　3D の回転変換

回転操作の対象物体が，プログラム中で頂点座標を指定するような図形（リスト 13 参照）の場合には，頂点の回転変換をプログラマ自身が記述することができる．

◇演習 11. 座標軸の回転： リスト 13 をもとにタイマコールバック関数を使って，z 軸まわりの回転変換を行うプログラムを作成してみよ（**付録 B** 参照）．

6.2　OpenGL の回転変換

GLUT 基本図形（**5** 参照）を回転させたい場合，プログラム中では頂点座標を直接扱わないので，プログラマ自身が頂点の回転変換を記述する方法は使えない．このような問題をスマートに解決するために，OpenGL 自身が座標変換機能を備えている．OpenGL の座標変換機能を利用すると，プログラマは座標変換行列の詳細を知らなくとも，平行移動や回転などの基本的な座標変換を施すことができる．

タイマコールバック関数 `timer` が呼び出されるたびに，物体を z 軸まわりに 3 度回転させるような座標変換を行う例をリスト 15 に示す．`glRotated` 関数は，第 1 引数が回転角（単位は degree），第 2 ～第 4 引数が回転軸に平行なベクトル（**方向ベクトル**）の x，y，z 成分である．実際の回転軸は，原点を通り方向ベクトルに平行な直線となる．

◇演習 12. 基本立体の回転： GLUT/GLU の基本立体をリスト 15 のタイマコールバック関数を使って，z 軸まわりに回転させるアニメーションを作成してみよ．

リスト 15　z 軸まわりの回転変換

```
1 static void timer(int dummy) {
2     glutTimerFunc(100, timer, 0);
3     glMatrixMode(GL_MODELVIEW);
4     glRotated(3.0, 0.0, 0.0, 1.0);
5     glutPostRedisplay();
6 }
```

7 User Interface

OpenGL 関数は本質的にはグラフィックス機能だけを提供するが，GLUT にはユーザからの入力を受け付ける UI（user interface）機能が備えられている．GLUT ではキー入力，マウス操作に対応している．より優れた UI として GLUI[7] などがあるが，今回は扱わない．

7.1 キー入力

キー入力を扱うには，`glutMainLoop` を呼び出す前にキーボードコールバック関数の登録を行う：

```
glutKeyboardFunc(keyin);
```

ESC キー[*19]または q（Q）キーを入力した場合にプログラムを終了させるキーボードコールバック関数の例をリスト 16 に示す．引数の `key` が押下されたキーの値，`x`，`y` は，その時点でのマウスの**ウィンドウ座標値**を意味する．ウィンドウ座標はウィンドウ左上を原点とし，右が x 軸正方向，下が y 軸正方向の値をとり，単位は画素である．DCS とは y 軸の正方向の取り方が逆になるので注意しなければならない（**2.6.1** 参照）．典型的なキーボードコールバック関数はリスト 16 のように，`switch` 文または `if` 文を用いて，想定するキー入力の数だけ場合分け（条件分岐）を行う構成となる．なお，プログラム終了関数 `exit` を呼び出すために，`stdlib.h` をインクルードしておく必要がある[*20]．

◇演習 13．3D 回転の制御： 演習 12 の 3D 回転アニメーションを機能拡張し，x/y/z のキー入力を受け付けて，入力されたキーに対応する座標軸まわりの回転が行われるようにしてみよ．

リスト 16　キーボードコールバック関数例

```
1  void keyin(
2      unsigned char key, int x, int y
3  ) {
4      switch(key) {
5        case '\033': /* ESC */
6        case 'q':
7        case 'Q': exit(0);
8        default: break;
9      }
10 }
```

[*19] ESC キーの値は`\033` に対応する．

[*20] `glut.h` より先にインクルードする方がよいらしい．

○ヒント： どの回転軸が選択されているかをグローバル変数で記憶するとよい．

◇演習 14．ズームイン/アウト： キー入力によってズームイン/アウトしているように見える処理を行う方法を考え，実装してみよ．

7.2 マウス入力

マウスボタンのクリックによる入力を扱うには，`glutMainLoop` を呼び出す前にマウスコールバック関数の登録を行う：

```
glutMouseFunc(mouse);
```

左ボタンをクリックした場合にプログラムを終了させるマウスコールバック関数の例をリスト 17 に示す．引数の `btn`，`state` はそれぞれ表 3，表 4 の値をとる．`x`，`y` はマウスのデバイス座標値である（**7.1** 参照）．

GLUT には，このほかにドラッグ（ボタンを押したままマウスを移動）に関するイベントを登録する関数 `glutMotionFunc` や，単なるマウスカーソルの移動に関するイベントを登録する関数 `glutPassiveMotionFunc` もある．興味があれば，自分で調べてみよ．

◇演習 15．アニメーション制御： 演習 13 の 3D 回転アニメーションを機能拡張し，マウスの左ボタ

リスト 17　マウスコールバック関数例

```
1  void mouse(
2      int btn, int state, int x, int y
3  ) {
4      if (
5        (btn == GLUT_LEFT_BUTTON)
6        &&
7        (state == GLUT_DOWN)) {
8          exit(0);
9      }
10 }
```

表 3　マウスのボタン（`btn` の値）

ボタン	GLUT の記述
左	GLUT_LEFT_BUTTON
中央	GLUT_MIDDLE_BUTTON
右	GLUT_RIGHT_BUTTON

表 4　マウスのボタン状態（`state` の値）

状態	GLUT の記述
押し下げ	GLUT_DOWN
リリース	GLUT_UP

ンをクリックすると回転アニメーションを停止し，右ボタンをクリックすると回転アニメーションを再開する機能を実装してみよ．

○ヒント： タイマコールバック関数による回転アニメーションを停止するには，次回の再起呼び出しをしないように，グローバル変数で制御すればよい[*21]．例えばグローバル変数として整数変数 isAnime を宣言し，アニメーションを行う場合は 1，停止する場合は 0 を代入することにすればよい．このように 2 値情報によって On/Off 制御を行う場合，その 2 値情報をフラグ（flag）と呼ぶ．ハードウェア寄りの組込みプログラムでは，記憶容量を効率化するためにビット単位でフラグを設定し，8 ビットの整数変数で 8 つのフラグを扱うことが多い．

8 モデル–ビュー変換

OpenGL は内部で座標変換を行う機能を持つ．OpenGL 内部では，**モデル–ビュー変換**行列と**投影変換**行列と呼ばれる二つの座標変換行列を記憶し，利用している．投影変換とは，仮想空間のカメラのシャッタを切ったときに，画像が生成されるまでの 3 次元から 2 次元への写像を扱う変換である．**4.1** で紹介した glOrtho 関数は，投影変換行列を設定する関数である．また，モデル–ビュー変換は，仮想空間の WCS 中でカメラや物体を移動・回転する変換を扱う．gluLookAt 関数や glRotatef 関数はモデル–ビュー変換に関係する．

二つの座標変換行列は，何れも**同次座標表現**（**付録 C** 参照）として 4×4 の行列として記憶され，座標変換を追加すると，それまでに加えられた全ての変換を合成する行列として更新される．

8.1 行列の選択と初期化

モデル–ビュー変換行列を選択するには，次の関数呼び出しを行う：

```
glMatrixMode(GL_MODELVIEW);
```

投影変換行列を選択するには，次の関数呼び出しを行う：

```
glMatrixMode(GL_PROJECTION);
```

選択した行列を単位行列に初期化するには，次の関数呼び出しを行う：

```
glLoadIdentity();
```

[*21] アイドルコールバック関数による回転アニメーションを停止するには，アイドルコールバック関数に NULL を再登録すればよい．

プログラム起動時には二つの行列とも単位行列に初期化されているが，慣例として，行列を選択して変換を加えていく前に初期化を呼び出すとよい．

8.2 基本的な座標変換

物体に加える基本的な座標変換としては，**平行移動**，**回転**，**スケーリング**があり，これらをまとめて**アフィン（Affine）変換**と呼ぶ．OpenGL で利用できる座標変換関数を表 5 に示す．

平行移動ではベクトル $\boldsymbol{T} = (t_x, t_y, t_z)$ に沿って移動される．回転では原点と点 (x, y, z) を結ぶ直線を回転軸として，angle 度だけ回転される．回転の向きは右ねじの向きである．スケーリングは x，y，z 軸方向にそれぞれ s_x，s_y，s_z 倍の拡大・縮小が行われる．$s_i = 1$ は等倍を意味し，$s_i < 0$ は鏡映を意味する．

8.3 座標変換行列のスタック

次の命令により，現在選択されている座標変換行列を一時的に**スタック**に退避（push）することができる：

```
glPushMatrix();
```

スタックからもとの行列を取り出す（pop）には，次の命令を使う：

```
glPopMatrix();
```

スタックを使うことにより，基本図形を変形したパーツを組み合わせて，より複雑な物体を表現したり，関節を持つようなロボットの動きを表現することが可能となる．

■**注意**： スタックの push/pop の回数は必ず同じにすること．

9 階層モデルによる物体表現

9.1 インスタンス変換

OpenGL の基本図形（**5** 参照）から，スケーリングや回転などのアフィン変換を使って，より複雑な物体のパーツを構成することができる．ベースとなる基本図形を**プリミティブ（primitive）**，あるいは**シンボル（symbol）**と呼ぶ．実際に出力されるパーツを，そのシンボルの**インスタンス（instance）**と呼ぶ．シンボルからインスタンスを生成することを**インスタンス化**と呼び，そのための変換を**インスタンス変換**と呼ぶ．

一般に，複数のパーツで物体を構成する場合には，各パーツごとに異なるインスタンス変換を適用したいはずである．このため，一つのパーツを描画

表 5　OpenGL の座標変換（GLdouble 型）

変換	関数プロトタイプ
平行移動	`glTranslated(GLdouble tx, GLdouble ty, GLdouble tz);`
回転	`glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z);`
スケーリング	`glScaled(GLdouble sx, GLdouble sy, GLdouble sz);`

するコードは次のようになる：

```
glPushMatrix();
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
/* インスタンス変換の定義 */
/* 基本物体の呼び出し */
glPopMatrix();
```

9.2　アームロボット

例として図 5 に示すようなアームロボットを，キー入力で操作する OpenGL プログラムの実装について考えてみよう．ロボットは平面 $z = 0$ に設置され，台座と下腕および上腕の三つのパーツからなる．台座の回転角 α_0，下腕の傾斜角 α_1，上腕の傾斜角 α_2 をキー入力で制御することとする．制御するパラメータが 3 個であることから，**3 自由度（3DOF; degree of freedom）**と呼ばれる．

まず，各パーツのインスタンス変換を考える．

- **台座**：GLU の円柱（2 次曲面）をインスタンス変換する．GLU の円柱は平面 $z = 0$ を底面とするので，x 軸（または y 軸）まわりに 90 度回転させてから適当な半径と高さを指定して `gluCylinder` を呼び出す．
- **下腕・上腕**：GLUT の立方体をインスタンス変換する．それぞれ，座標軸方向に適当な長さにスケール変換させてから `glutWireCube` を呼び出す．

次に階層表現を考える．すなわち，上腕の傾斜角 α_2 が変更された場合には上腕のみが傾くが，台座の回転角 α_0 が変更された場合にはロボットアーム全体が回転するような親子関係を実現する．**8** で述べたように，OpenGL のモデル–ビュー変換は累積されていくので，リスト 18 のような描画処理を行えばよい[*22]．

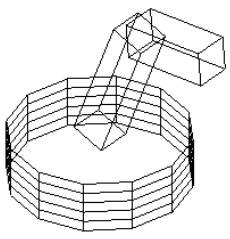

図 5　アームロボットのモデル

リスト 18 で各パーツのインスタンス化を全て `display` 関数に含めると，コードが長くなり見通しが悪化する．その対策として，各パーツのインスタンス化をユーザ定義関数とする方法が考えられる．また，OpenGL ではディスプレイリストを利用することもできる．次節では後者の方法を紹介する．

9.3　ディスプレイリスト

座標変換，描画色指定，図形描画などの一連の処理を繰り返すような場合，毎回該当する OpenGL 関数を呼び出すのでは，関数呼び出しのオーバーヘッドが発生するなどの理由で効率が悪い．定型処理を事前に登録しておき，グラフィックスハードウェアに効率のよい形で実行する方法として**ディスプレイリスト（display list）**がある．ディスプレイリストへの登録は `glutMainLoop` を呼ぶ前（`init` や `main` 関数中）で，次のように行う：

リスト 18　アームロボットの描画関数例

```
 1 GLfloat rotAng[3];
 2
 3 void display(void) {
 4     glClear(GL_COLOR_BUFFER_BIT);
 5     glMatrixMode(GL_MODELVIEW);
 6     glLoadIdentity();
 7     gluLookAt(1,1,1, 0,0,0, 0,1,0);
 8     glRotated(rotAng[0], 0, 1, 0);
 9     /* 台座のインスタンス変換 */
10     glTranslated(0, HEIGHT_B, 0);
11     glRotated(rotAng[1], 0, 0, 1);
12     /* 下腕のインスタンス変換 */
13     glTranslated(0, HEIGHT_L, 0);
14     glRotated(rotAng[2], 0, 0, 1);
15     /* 上腕のインスタンス変換 */
16     glutSwapBuffers();
17 }
```

[*22] 回転角 α_i はグローバル変数 `rotAng[i]` に格納する．また，`HEIGHT_B`，`HEIGHT_L` はそれぞれ台座，下腕の高さ（長さ）で，マクロ等で定義しておく（リスト 27 参照）．

```
glNewList(登録番号, GL_COMPILE);
/* 登録したい処理 */
glEndList();
```

登録番号（n）は 1 以上の正の整数である．**9.2** のアームロボットでは台座，下腕，上腕のそれぞれについて異なる番号で三つのリストを登録すればよい．例えば，下腕を登録するコードはリスト 19 のように書けるであろう[*23]．

リスト 19　アームロボット下腕の登録例

```
1  glNewList(ID_L, GL_COMPILE);
2  glColor3f(0.0, 1.0, 0.0);
3  glPushMatrix();
4  glTranslatef(0.0, 0.5*HEIGHT_L, 0.0);
5  glScalef(WIDTH_L, HEIGHT_L, WIDTH_L);
6  glutWireCube(1.0);
7  glPopMatrix();
8  glEndList();
```

登録したリストを実行するには，次の命令を使う：

```
glCallList(登録番号);
```

台座を ID_B，下腕を ID_L，上腕を ID_U の登録番号でディスプレイリストに登録した場合 [*23]，アームロボットの描画はリスト 20 のようになる．

リスト 20　ディスプレイリストによる描画例

```
1  GLfloat rotAng[3];
2
3  void display(void) {
4      glClear(GL_COLOR_BUFFER_BIT);
5      glMatrixMode(GL_MODELVIEW);
6      glLoadIdentity();
7      gluLookAt(1,1,1, 0,0,0, 0,1,0);
8      glRotated(rotAng[0], 0, 1, 0);
9      glCallList(ID_B);
10     glTranslated(0, HEIGHT_B, 0);
11     glRotated(rotAng[1], 0, 0, 1);
12     glCallList(ID_L);
13     glTranslated(0, HEIGHT_L, 0);
14     glRotated(rotAng[2], 0, 0, 1);
15     glCallList(ID_U);
16     glutSwapBuffers();
17 }
```

[*23] ID_L は下腕の登録番号で，マクロ等で定義しておく．HEIGHT_L，WIDTH_L も同様（リスト 27 参照）．

課題 II

1. 図 6 に展開図を示す正四面体 $ABCD$ について，$\triangle BCD$ の重心 G を原点 O と一致させ，GA を x 軸，GB を y 軸とする座標系を考える．正四面体の一辺の長さを w とするとき，各頂点の 3 次元座標を導出せよ．
2. 前問で求めた結果をもとに，原点 O を中心とする半径 1 の球に内接する正四面体の頂点がリスト 14 のように定まることを導出せよ．
3. **9.2** のアームロボットについて，キー入力で台座の回転と，各関節の傾斜角を制御できるような対話プログラムを作成してみよう．
4. タイマコールバックを使って，アームロボットを制御するようなプログラムを作成してみよう．アームロボットが踊っているかのように見せるには，どんな工夫ができるだろうか．

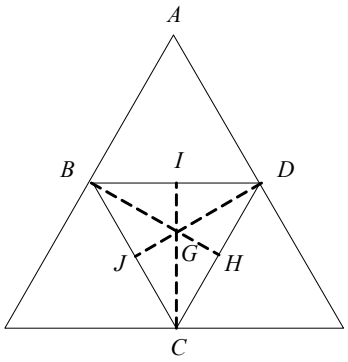

図 6　正四面体の展開図

コラム：freeglut について

freeglut[3] は，GLUT の上位互換を目指して開発されており，tea spoon や菱形 12 面体，シェルピンスキー四面体などの基本図形が追加されている．また，文字列を出力する関数も追加されている．残念ながら BCC では，文字出力に関する機能が利用できないため，freeglut の全機能を利用するには VisualStudio または gcc を開発環境に選ぶ必要がある．

【第 3 日】

3D モデルと照光処理

10　サーフェイスモデル

サーフェイスモデル（surface model）では面の集合（多面体）として物体を表現する．OpenGL では，面として平面や 2 次曲面などを扱うことができるが，今回は**ポリゴン（polygon）**と呼ばれる多角形だけを扱う．

一般に 3 頂点を定めると一つの平面（三角形）が一意に定まるが，4 頂点以上の場合には全ての点が平面上に存在することが保証されないうえ，多角形が凸となることも保証されない．OpenGL では，ねじれた面や凹多角形の描画を保証していないので注意が必要である[*24]．

10.1　ポリゴンの描画

演習 3 では，`glBegin` 関数の引数を `GL_POLYGON` とすれば，内部が塗りつぶされた赤い正三角形が描かれることを確認した．

◇演習 16．頂点間の色の補間： ポリゴンの頂点ごとに異なる色を指定すると，内部が補間されて塗りつぶされる．演習 3 で `GL_POLYGON` で描画するプログラムについて，`glVertex2d` で頂点座標を定める前に，`glColor3d` による色指定を 1 点ごとに行い，このことを確かめてみよ．

10.2　基本立体の描画

GLUT の基本立体をサーフェイスモデルで描画したい場合には，表 1 の関数名に含まれる `Wire` を `Solid` に置き換えればよい．GLU の 2 次曲面をサーフェイスモデルで描画したい場合には，描画スタイル指定を変更する：

```
gluQuadricDrawStyle(mySpr, GLU_FILL);
```

GLUT の基本立体や GLU の 2 次曲面は，単色で描画しても立体感は得られない．とりあえず立体的に見える立体を描くために，プログラマ自身が物体形状を記述し，頂点や面ごとに描画色を指定する方法がある．より実世界に近い立体感を得るには，照光処理を加える必要がある（**13** 参照）．

10.3　正四面体の描画

最も単純な立体として，半径 1 の球に内接する正四面体を描画することを考える．この正四面体を描くには，4 頂点と 4 面を定義すればよい．4 頂点は

[*24] GLU の機能を使えば三角形に分割できるらしい．

リスト 21　正四面体 4 頂点の近似値（ジオメトリ情報）

```
1 GLdouble vP[4][3]={
2   { 1.000, 0.000, 0.000 },
3   { -0.333, 0.943, 0.000 },
4   { -0.333, -0.471, 0.816 },
5   { -0.333, -0.471, -0.816 }
6 };
```

リスト 22　正四面体の 4 面（トポロジ情報）

```
1 int tP[4][3]={
2   { 0, 1, 2 }, { 0, 3, 1 },
3   { 0, 2, 3 }, { 1, 3, 2 }
4 };
```

リスト 23　正四面体の色情報

```
1 GLdouble cP[4][3]={
2   { 1.0, 0.0, 0.0 }, { 0.0, 1.0, 0.0 },
3   { 0.0, 0.0, 1.0 }, { 1.0, 1.0, 0.0 }
4 };
```

例えばリスト 21 のようにグローバル変数として定義すればよい[*25]．

面はリスト 21 の頂点をたどる順として，リスト 22 のようにグローバル変数として定義すればよい．また，各面を異なる色で塗るために，リスト 23 のように色をグローバル変数として定義しておく．

描画コールバック関数はリスト 24 のようにシンプルに書ける．このように頂点座標と，面を構成する頂点の接続情報を分けて定義する方法は，複雑な図形を描く場合に有効である．頂点座標を**ジオメトリ（geometry）**情報，頂点の接続情報を**トポロジ（topology）**情報と呼ぶ．

◇演習 17．正四面体の回転： リスト 21〜リスト 24 を実装し，正四面体の回転アニメーションを行うプログラムを完成させ，様々な方向から正四面体を観察せよ．4 面それぞれが正しく塗り分けられて表示されているだろうか．

11　カリング

凸多面体を描く際に OpenGL にポリゴンの表裏を判断させ，視点側の面（表）だけを描かせることができる．これを**カリング（culling）**と呼ぶ．

[*25] リスト 14 の 3〜9 行目で計算される頂点座標の近似値を用いた．

リスト 24　正四面体の描画コールバック関数

```
1  void display(void) {
2      int i,j;
3      glClear(GL_COLOR_BUFFER_BIT);
4      glBegin(GL_TRIANGLES);
5      for (i = 0; i < 4; i++) {
6          glColor3dv(cP[i]);
7          for (j = 0; j < 3; j++) {
8              glVertex3dv(vP[tP[i][j]]);
9          }
10     }
11     glEnd();
12     glutSwapBuffers();
13 }
```

11.1　面の向き

OpenGL はデフォルトでは，ポリゴン上の頂点を反時計回り（CCW; counterclockwise）にたどる向きを表，時計回り（CW; clockwise）にたどる向きを裏と定める．何かの都合で逆（CW を表，CCW を裏）にしたい場合は，次の関数を呼ぶ：

```
glFrontFace(GL_CW);
```

11.2　カリングの制御

カリングを有効にして，表と裏を描き分けるには次の関数を呼ぶ：

```
glEnable(GL_CULL_FACE);
```

カリングを無効にするには，次の関数を呼ぶ：

```
glDisable(GL_CULL_FACE);
```

カリングによって描画しない面はデフォルトでは裏（GL_BACK）であるが，表を描画しないようにするには，次の関数を呼ぶ[*26]：

```
glCullFace(GL_FRONT);
```

11.3　描画モードによる描き分け

カリングでは裏と判定された面が，完全に表示処理から省かれる．このため描画処理の高速化が期待できるが，ビューポートに図形が全く表示されない場合に，カメラ位置の設定の問題か，物体定義の問題かを見分けにくい．このような場合に有効なのが，表面と裏面とで描画モードを変更する方法である．描画モードの設定は，次の関数で行う：

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

[*26] glFrontFace と glCullFace を組み合わせると，混乱するのでむやみに変更しないこと．

表 6　face の指定

面	face の指定
表	GL_FRONT
裏	GL_BACK
両面	GL_FRONT_AND_BACK

表 7　mode の指定

描画法	mode の指定
頂点のみ	GL_POINT
ワイヤフレーム	GL_LINE
サーフェイス	GL_FILL

face は表 6，mode は表 7 から指定する[*27]．

◇演習 18．正四面体へのカリング： 演習 17 のプログラムでカリングを有効にしてみよ[*28]．

◇演習 19．正六面体・正八面体の描画： 半径 1 の球に内接する正六面体や正八面体のジオメトリ情報，トポロジ情報，色情報を定義して，サーフェイスモデルとして描画するプログラムを作成せよ．

12　奥行き判定

凹多面体や，複数の凸多面体を描く場合にはカリングだけでは適切な出力が得られない．物体の前後関係を正しく OpenGL に判定させて，視点に近い面を確実に可視面とするには，奥行き判定（**陰面消去**）を有効にすればよい．

そのためには，glutInitDisplayMode の引数に GLUT_DEPTH を加え，**デプスバッファ**（Z バッファ）を準備する．ダブルバッファリングも行う場合，関数呼び出しの引数は次のようになる：

```
GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH
```

次に陰面消去処理が必要となる描画の前で，奥行き判定を有効にする[*29]：

```
glEnable(GL_DEPTH_TEST);
```

陰面消去処理が不要なときは，奥行き判定を無効にすると処理速度が向上する：

```
glDisable(GL_DEPTH_TEST);
```

画面を消去するときには，デプスバッファもクリアする必要がある．従って glClear の関数呼び出しの引数は次のようになる：

[*27] glCullFace の引数も表 6 から選択できる．両面を選ぶと，表と裏を描き分ける意味がない．

[*28] 普通は途中で変えないはずなので，関数 init あたりで有効にすればよい．

[*29] init 関数に書けばよいだろう．

```
GL_COLOR_BUFFER_BIT |
    GL_DEPTH_BUFFER_BIT
```

◇演習 20．複数の正四面体描画： 前後関係のある複数の物体を描くようなアニメーションを作成し，奥行き判定の効果を確認してみよう．

13 照光処理

現実世界の物体は，それ自身が色を発しているのではなく，光源から放射されて届く光に対する反射特性によって見え方が決まる．これを忠実にシミュレートするには，光源ばかりでなく，周囲の物体の形状・反射特性・配置などの全てを知らなければならない．そのような描画法として**レイトレーシング**（ray tracing）法が知られているが，リアルタイム描画は実質的に不可能である．

OpenGL では，フォン（Phong）が提案したモデルにより，近似的な照光処理（シェーディング；shading）を実現している．フォンのモデルでは，物体表面上の点がどのように見えるかを拡散反射（diffuse reflection），鏡面反射（specular reflection），環境光反射（ambient reflection），放射光（emissive light）の合成として考える．

13.1 シェーディングの On/Off

OpenGL でシェーディングを有効にするには，次の関数を呼ぶ：

```
glEnable(GL_LIGHTING);
```

シェーディングを無効にする必要があるときは，次の関数を呼ぶ：

```
glDisable(GL_LIGHTING);
```

■注意 1： シェーディングを有効にすると，`glColor` による色設定は無視される．色指定の代わりに，各面の法線ベクトル[*30]を指定しなければならない．例えば，正四面体の描画を行う場合は，次のように法線ベクトルを格納するための配列をグローバルに宣言する：

```
GLdouble nV[4][3];
```

この配列に，各面の法線ベクトルを計算しておき（**付録 D** 参照），`i` 番目の面については，次の関数で法線ベクトルを指定する：

```
glNormal3dv(nV[i]);
```

[*30] スムーズシェーディングを行う場合には，各頂点の法線ベクトルを指定する必要がある（**13.3** 参照）．

■注意 2： 法線ベクトルは，単位ベクトルにする必要がある．単位ベクトル化のための計算（正規化）を自分で行うのが面倒な場合には，次の関数呼び出しで OpenGL に任せることもできる：

```
glEnable(GL_NORMALIZE);
```

しかし，この設定を行うと，OpenGL は表示処理のたびに全ての法線ベクトルの正規化処理を行うことになるので，表示速度が低下する．

なお，GLUT や GLU の基本立体（表 1，表 2 参照）を描画するときは，法線の指定を省略できる（描画関数側で正しく指定される）．

13.2 光源をオンにする

シェーディングを有効にしただけでは，物体に光が当たらないので，何も表示されない．そこで光源（照明）を設定して，点灯する必要がある．Windows 環境の OpenGL では 0 番〜7 番までの 8 個の光源を設定することができる（らしい）．

0 番の光源をオンにする（点灯させる）には，次の関数を呼ぶ：

```
glEnable(GL_LIGHT0);
```

デフォルトでは 0 番の光源は，z 軸方向の無限遠から白色光を照射する．光源位置を変更するには，

```
GLfloat lP[4];
```

のように宣言した配列に，同次座標で光源位置を設定し，次の関数を呼ぶ[*31]：

```
glLightfv(GL_LIGHT0, GL_POSITION, lP);
```

光源 1 について，光源位置を座標 $(0,1,2)$ に定め，拡散反射光（`GL_DIFFUSE`），鏡面反射光（`GL_SPECULAR`）を黄色 $(1,1,0)$ に定めるコードをリスト 25 に示す[*32]．

◇演習 21．基本立体の照光処理： GLUT の基本立体のサーフェイスモデルをシェーディング表示するプログラムを作ってみよ．

◇演習 22．正多面体の照光処理： 演習 18 や演習 19 で扱った正四面体・正六面体・正八面体の各面について，法線ベクトルを求め，リスト 26 の表示コールバック関数を参考に，それぞれを描画するプログラムを作成せよ．

[*31] `glLight` 関数には `GLdouble` を引数とする version が存在しない（**付録 A** 参照）．

[*32] このほか，環境反射光（`GL_AMBIENT`）を指定できる．

リスト 25　光源 1 の設定例

```
GLfloat lP1[]={ 0.0,1.0,2.0,1.0 };
GLfloat lC1[]={ 1.0,1.0,0.0,1.0 };

/* 以下はinitあたりに書く */
glLightfv(GL_LIGHT1,GL_POSITION,lP1);
glLightfv(GL_LIGHT1,GL_DIFFUSE,lC1);
glLightfv(GL_LIGHT1,GL_SPECULAR,lC1);
glEnable(GL_LIGHT1);
```

リスト 26　四面体の描画コールバック関数

```
GLdouble nV[4][3]; /* 単位法線ベクトル */

void display(void) {
    int i,j;
    glClear(
      GL_COLOR_BUFFER_BIT
      | GL_DEPTH_BUFFER_BIT
    );
    glBegin(GL_TRIANGLES);
    for (i = 0; i < 4; i++) {
        glNormal3dv(nV[i]);
        for (j = 0; j < 3; j++) {
            glVertex3dv(vP[tP[i][j]]);
        }
    }
    glEnd();
    glutSwapBuffers();
}
```

13.3　法線ベクトルとスムーズシェーディング

OpenGL では球面のような曲面を，平面（ポリゴン）の集合で近似表示している．このような場合，面ごとに法線ベクトルを設定するのではなく，頂点ごとに法線ベクトルを設定し，滑らかなシェーディング表示を実現すると高品質な出力が得られる．

一つの平面を一つの法線ベクトルを使ってシェーディングする（その結果，一つの面は単一色で塗られる）方法を**フラットシェーディング（flat shading）**と呼ぶ．頂点ごとに設定された法線ベクトルを使ってシェーディングする（その結果，一つの面内の色は補間によって滑らかに変化する）方法を**スムーズシェーディング（smooth shading）**と呼ぶ．

フラットシェーディングで表示したい場合は，

```
glShadeModel(GL_FLAT);
```

スムーズシェーディングで表示したい場合は，

```
glShadeModel(GL_SMOOTH);
```

を呼び出せばよい．

14　おわりに（この先には）

様々な色のついた物体や，様々な光沢の物体を OpenGL で出力するには，光源の種類の設定や，物体の反射特性のパラメータをきめ細かく設定する必要がある [4].

光源の設定は `glLight*`や `glLightModel*`，反射特性の設定は `glMaterial*`を利用することになる．また，OpenGL では半透明の物体を表示する**ブレンディング（blending）**処理を行ったり，表面に模様があるような物体を表示する**テクスチャマッピング（texture mapping）**処理を行うことも可能である．これらについては，各自で学習してほしい．

課題 III

1. 空間中の任意の 3 点 $P_i = (x_i, y_i, z_i), i = 1,2,3$ が与えられたとき，$\triangle P_1P_2P_3$ の単位法線ベクトルを全て求める式を導出せよ．
2. 空間中の任意の 3 点 $P_i = (x_i, y_i, z_i), i = 1,2,3$ が与えられたとき，$\triangle P_1P_2P_3$ のどちらが表（CCW）で，どちらが裏（CW）と判定できるか，判定方法を導出せよ．
3. Phong の照光モデルについて，詳しく調べよ．
4. 正四面体, 正六面体, 正八面体をスムーズシェーディングで表示するために，各頂点の単位法線ベクトルをどのように定めればよいか考え，実装せよ．
5. リスト 27 を参考に，アームロボットをシェーディング表示するものに変更してみよ．

レポート

第 1 日〜第 3 日までの課題を整理して 10 月 18 日（金）17:00 までに提出せよ．プログラムコードをただ挿入するのではなく，要点を簡潔に整理して作成すること．また，レポートの最後に感想と，今後の改善案の二つを**必ず**書くこと．

総合課題

2D または 3D グラフィックスを用いた専門科目の学習を支援するようなプログラム，あるいは小中学生が簡単にルールを理解できるゲームプログラムを作成して，ソースコードと実行ファイルを提出せよ．期限は 11 月 15 日（金）17:00 までとする．

リスト 27　リアルなロボットアーム（ロケット?）

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

#define ID_B 1
#define ID_L 2
#define ID_U 3
#define RADIUS_B 0.5
#define HEIGHT_B 0.3
#define WIDTH_L 0.2
#define HEIGHT_L 0.8
#define WIDTH_U 0.2
#define HEIGHT_U 0.5

typedef struct materialStruct {
  GLfloat ambient[4], diffuse[4], specular[4];
  GLfloat shininess;
} materialStruct;

materialStruct brassMaterials = {
  { 0.33, 0.22, 0.03, 1 }, { 0.78, 0.57, 0.11, 1 }, { 0.99, 0.91, 0.81, 1 }, 27.8 };
materialStruct redPlasticMaterials = {
  { 0.3, 0.0, 0.0, 1 }, { 0.6, 0.0, 0.0, 1 }, { 0.8, 0.6, 0.6, 1 }, 32.0 };
materialStruct greenPlasticMaterials = {
  { 0.0, 0.3, 0.0, 1 }, { 0.0, 0.6, 0.0, 1 }, { 0.6, 0.8, 0.6, 1 }, 32.0 };
GLUquadric *myQuad;

void materials(materialStruct *m) {
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, m->ambient);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, m->diffuse);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, m->specular);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, m->shininess);
}

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(1.0, 1.0, 1.0, 0.0, 0.5, 0.0, 0.0, 1.0, 0.0);
    glCallList(ID_B);
    glTranslatef(0.0, HEIGHT_B, 0.0);
    glCallList(ID_L);
    glTranslatef(0.0, HEIGHT_L, 0.0);
    glCallList(ID_U);
    glutSwapBuffers();
}

void resize(int w, int h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-2.0, 2.0, -2.0, 2.0, -4.0, 4.0);
}

void init(void) {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    myQuad = gluNewQuadric();
```

```
56      gluQuadricDrawStyle(myQuad, GLU_FILL);
57      glEnable(GL_DEPTH_TEST);
58      glEnable(GL_LIGHTING);
59      glEnable(GL_LIGHT0);
60      glNewList(ID_B, GL_COMPILE);
61      glPushMatrix();
62      glRotatef(-90.0, 1.0, 0.0, 0.0);
63      materials(&brassMaterials);
64      gluCylinder(myQuad, RADIUS_B, WIDTH_L, HEIGHT_B, 12, 5);
65      glPopMatrix();
66      glEndList();
67      glNewList(ID_L, GL_COMPILE);
68      glPushMatrix();
69      glRotatef(-90.0, 1.0, 0.0, 0.0);
70      materials(&redPlasticMaterials);
71      gluCylinder(myQuad, WIDTH_L, WIDTH_L, HEIGHT_L, 12, 5);
72      glPopMatrix();
73      glEndList();
74      glNewList(ID_U, GL_COMPILE);
75      glPushMatrix();
76      glRotatef(-90.0, 1.0, 0.0, 0.0);
77      materials(&greenPlasticMaterials);
78      gluCylinder(myQuad, WIDTH_U, 0.0, HEIGHT_U, 12, 5);
79      glPopMatrix();
80      glEndList();
81  }
82
83  int main(int argc, char** argv) {
84      glutInit(&argc, argv);
85      glutInitWindowSize(512, 512);
86      glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
87      glutCreateWindow(argv[0]);
88      glutDisplayFunc(display);
89      glutReshapeFunc(resize);
90      init();
91      glutMainLoop();
92      return 0;
93  }
```

参考文献・URL

[1] OpenGL, http://www.opengl.org/

[2] Nate Robins, “GLUT for Win32”, http://www.xmission.com/~nate/glut.html

[3] freeglut – The Free OpenGL Utility Toolkit, http://freeglut.sourceforge.net/

[4] Edward Angel，滝沢徹・牧野祐子訳，“OpenGL 入門–やさしいコンピュータグラフィックス–”，ピアソン・エデュケーション，2002，定価 3150 円

[5] 床井浩平，“GLUT による OpenGL 入門”，工学社，2005，定価 1995 円.

[6] 和歌山大・床井浩平，“GLUT による「手抜き」OpenGL 入門”，
http://www.center.wakayama-u.ac.jp/~tokoi/opengl/libglut.html

[7] Paul Rademacher，Nigel Stewart，“GLUI User Interface Library”，http://www.cs.unc.edu/%7Erademach/glui/

[8] プログラミング演習 IV サポートページ，https://www2.st.nagaoka-ct.ac.jp/~ataka/prog4/

[9] 視覚情報処理研究室（高橋研），http://www.nagaoka-ct.ac.jp/ec/labo/visu/

付録 A　OpenGL の名前付けルール

本演習で扱う（広義の）OpenGL 関数は次の 3 種類に分類できる：

- **（狭義の）OpenGL 関数**：OpenGL のコア（core）機能が提供する関数で，関数名が `gl` で始まる．基本的かつ重要な描画機能を提供する．
- **GLU 関数**：OpenGL Utility Library が提供する関数で，関数名は `glu` で始まる．OpenGL プログラミングを簡便にする機能を提供する．
- **GLUT 関数**：OpenGL Utility Toolkit Library が提供する関数で，関数名は `glut` で始まる．ウィンドウシステム上で OpenGL プログラミングを行う GUI 機能を提供する．

OpenGL では，プログラマが自分に都合が良いデータ型を用いて座標や色成分の指定を行うことができるように，様々なデータ型に対応した同じ機能を提供する関数を用意している．データ型としては C 言語の `short`，`int`，`float`，`double` 型に対応している（ただし，先頭に `GL` を付けた型として独自に定義されている）．これらの規則と，ある程度の英語力があると，関数名からその機能や，別のデータ型用の関数を推測することができる．

例えばリスト 4 で 2 次元平面上の座標を指定している関数 `glVertex2d` については，まず先頭 2 文字の `gl` から狭義の OpenGL 関数であることがわかる．続く `Vertex` は「頂点」という意味である．数字の `2` は 2 次元であることに対応し，引数として少なくとも 2 個の座標値を渡していることが推測できる．同様の OpenGL 関数で利用される数字（次元数）には，このほか `3` と `4` がある．最後の `d` は引数のデータ型が `GLdouble` であることを意味する．データ型の識別子としては，ほかに `s`，`i`，`f` があり，それぞれ `GLshort`，`GLint`，`GLfloat` に対応する．多次元のデータを一まとめに扱うには，配列が便利な場合も多い．配列により 2 次元の頂点座標を指定する場合は `glVertex2dv` のように，データ型の識別子の後に `v` を付けた関数が用意されている．これらをまとめて，次のように表記することがある．

```
glVertex{234}{sifd}(...);
glVertex{234}{sifd}v(...);
```

付録 B　座標変換

B.1　スケーリング（scaling）

空間を x 軸方向に s_x 倍，y 軸方向に s_y 倍，z 軸方向に s_z 倍する変換は，次式で表される：

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{7}$$

この式の 3×3 の行列は，非対角成分が 0 であるので，**対角行列（diagonal matrix）**と呼ばれ，次のように略記される：

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} = \mathrm{diag}(s_x,\ s_y,\ s_z)$$

B.2　平行移動・並進（translation）

x 軸に沿って t_x，y 軸に沿って t_y，z 軸に沿って t_z だけ平行移動する変換は，次式で表される：

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \tag{8}$$

ベクトル $\boldsymbol{t} = (t_x, t_y, t_z)$ を**並進ベクトル**と呼ぶ．

B.3　回転（rotation）

空間中に回転軸となる直線を定め，その回りにある角度だけ回転する変換は，次式で表される：

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{9}$$

行列 $\boldsymbol{R} = (r_{ij})$ を**回転行列**と呼ぶ．回転行列は**正規直交行列**である．正規直交行列では，$\boldsymbol{R}^{-1} = \boldsymbol{R}^T$ が成立する（T は転置を表す）．

■**例**：　z 軸まわりに α だけ回転させる回転行列：

$$\boldsymbol{R}(0,0,\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

B.4　変換の合成

回転の合成は，行列の積で表現できる．例えば，

$$\boldsymbol{R}(0,0,\alpha_1+\alpha_2) = \boldsymbol{R}(0,0,\alpha_2)\boldsymbol{R}(0,0,\alpha_1)$$

では，任意の順序で平行移動と回転を行う場合は，合成できるだろうか？

付録 C　同次座標表現

同次座標表現（homogeneous coordinates）を用いると，座標変換の合成を同次座標変換行列の積で表

すことができる．三次元座標 (x,y,z) の同次座標表現を次のように定義する：

$$(x,\ \ y,\ \ z,\ \ s)$$

ここで，(x,y,z) が点の座標（位置ベクトル）を表すとき $s=1$，(x,y,z) がベクトルの成分を表すとき $s=0$ とする．

C.1 スケーリング

式 (7) を同次座標表現で表すと次式となる．

$$\begin{pmatrix} x' \\ y' \\ z' \\ s \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ s \end{pmatrix} \tag{10}$$

C.2 平行移動・並進

式 (8) を同次座標表現で表すと次式となる．

$$\begin{pmatrix} x' \\ y' \\ z' \\ s \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ s \end{pmatrix} \tag{11}$$

C.3 回転

式 (9) を同次座標表現で表すと次式となる．

$$\begin{pmatrix} x' \\ y' \\ z' \\ s \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ s \end{pmatrix} \tag{12}$$

付録 D　法線ベクトル

右手系の座標系では以下の手順で，反時計回りに向きがつけられた 3 頂点 $P_i(x_i,y_i,z_i)$，$i=1,2,3$ から，三角形 $\triangle P_1P_2P_3$ の表向きの単位法線ベクトル $\overrightarrow{n}$ を決定できる．

1. 次の 2 ベクトルを定める：

$$\overrightarrow{P_1P_2} = (x_2-x_1, y_2-y_1, z_2-z_1) \tag{13}$$

$$\overrightarrow{P_2P_3} = (x_3-x_2, y_3-y_2, z_3-z_2) \tag{14}$$

2. 2 ベクトル $\overrightarrow{P_1P_2}$，$\overrightarrow{P_2P_3}$ の外積 $\overrightarrow{N}$ を求める．

$$\overrightarrow{N} = \overrightarrow{P_1P_2} \times \overrightarrow{P_2P_3} \tag{15}$$

3. 単位法線ベクトル $\overrightarrow{n}$ を求める．

$$\overrightarrow{n} = \frac{\overrightarrow{N}}{|\overrightarrow{N}|} \tag{16}$$

ベクトル演算を行う関数を自作し，分割コンパイルできるようにしておくとプログラム開発の手間が軽減できる．例えばリスト 28 に示す関数プロトタイプ宣言に対応するユーザ定義関数を自作しておいたとすると，リスト 21 の正四面体の，第 1 面の法線ベクトルはリスト 29 で計算できる（かもしれない）．

リスト 28　ベクトル演算関数のヘッダファイル

```
/* 3D vector */
#if !defined(VECT_H)
#define VECT_H

/* w = v1 + v2 */
void vect_add(
  GLdouble v1[], GLdouble v2[],
  GLdouble w[]
);
/* w = v1 − v2 */
void vect_sub(
  GLdouble v1[], GLdouble v2[],
  GLdouble w[]
);
/* w = s*v */
void vect_scale(
  GLdouble s, GLdouble v[],
  GLdouble w[]
);
/* |v|を返却 */
GLdouble vect_norm(GLdouble v[]);
/* v1とv2の内積を返却 */
GLdouble vect_innerproduct(
  GLdouble v1[], GLdouble v2[]
);
/* w = v1 x v2(外積) */
void vect_outerproduct(
  GLdouble v1[], GLdouble v2[],
  GLdouble w[]
);
#endif /* VECT_H */
```

リスト 29　法線ベクトル計算例

```
#include "vect.h"

GLdouble n;
GLdouble v1[3],v2[3];
GLdouble tmp[3];

vect_sub(vP[1], vP[0], v1);
vect_sub(vP[2], vP[1], v2);
vect_outerproduct(v1, v2, tmp);
n = vect_norm(tmp);
vect_scale(1.0/n, tmp, nV);
```