Chapter 12

多面体とフラクタル立体

12.1　多面体

平面のみで構成される閉じた 3 次元形状が**多面体** (polyhedron) である．多面体の稜線は 2 平面の交線であるから直線分となる．多面体の面は平面上で直線の稜線によって囲まれていることから多角形となる．多面体を構成する頂点，隣り合った 2 頂点のペアからなる稜線，多面体を構成する多角形の頂点列（あるいは稜線列）によって，多面体を表現することができる．

多面体の中でも，同一の正多角形のみで構成され，かつすべての頂点の周りに同じ数の面（稜線）があるものを正多面体 (regular polyhedron, Platonic solid) と呼ぶ．正多面体は，正四面体，立方体（正六面体），正八面体，正十二面体，正二十面体の 5 種類しか存在しない．外接球の半径が $\sqrt{3}$ ないし $\sqrt{\phi\sqrt{5}}$（正二十面体のときのみ）となる標準的な正多面体を図 12.1 ならびに表 12.1 に示す．ただし，ϕ は黄金比で $\phi = (1+\sqrt{5})/2$ である．

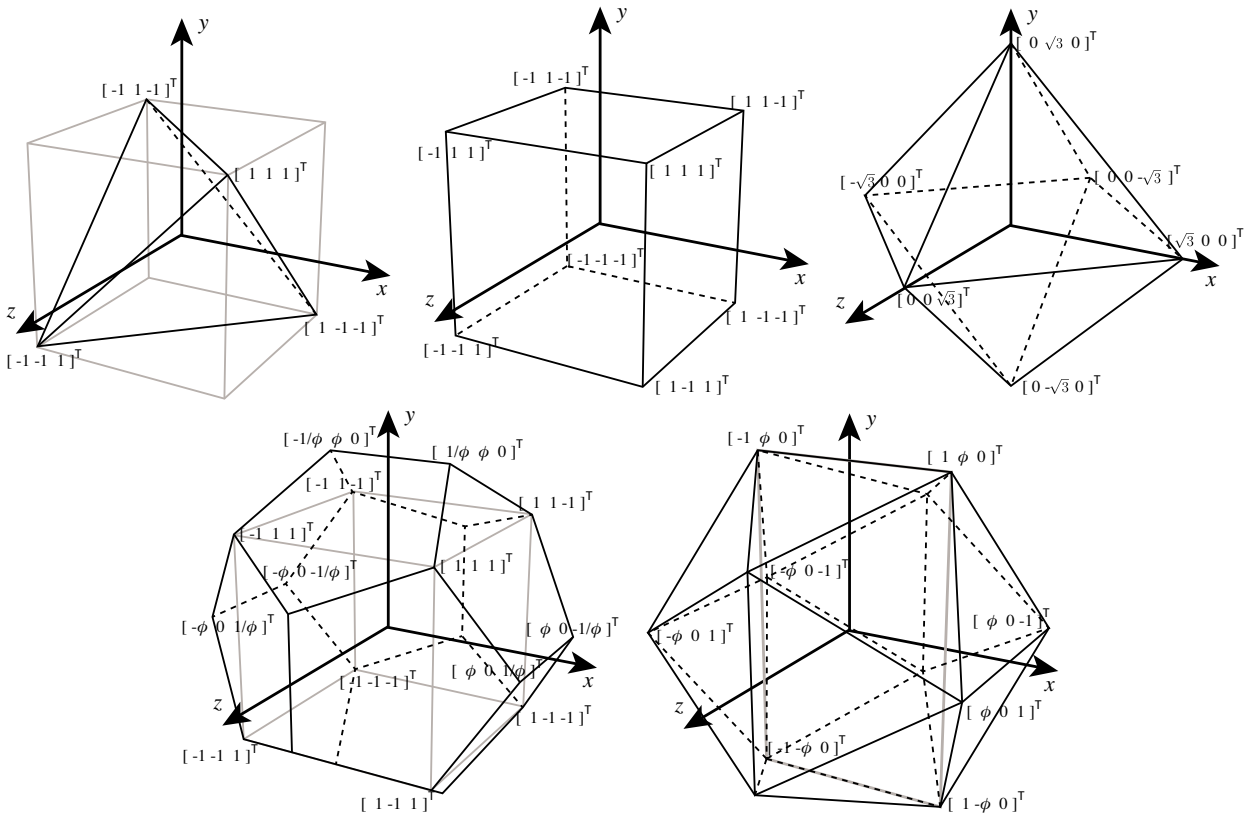

Figure 12.1: 正多面体．左上から，正四面体，立方体，正八面体，正十二面体，正二十面体

Table 12.1: 図 12.1 の正多面体

名称	正四面体	立方体	正八面体	正十二面体	正二十面体
面	正三角形	正方形	正三角形	正五角形	正三角形
頂点周りの面数	3	3	4	3	5
面数	4	6	8	12	20
頂点数	4	8	6	20	12
稜線数	6	12	12	30	30
頂点座標	$[\ 1\quad 1\quad 1\ ]^{\mathsf{T}}$ $[-1\quad 1\quad -1]^{\mathsf{T}}$ $[-1\quad -1\quad 1\ ]^{\mathsf{T}}$ $[\ 1\quad -1\quad -1]^{\mathsf{T}}$	$[\pm1\ \ \pm1\ \ \pm1]^{\mathsf{T}}$	$[\pm\sqrt{3}\quad 0\quad 0\ ]^{\mathsf{T}}$ $[\ 0\quad \pm\sqrt{3}\quad 0\ ]^{\mathsf{T}}$ $[\ 0\quad 0\quad \pm\sqrt{3}]^{\mathsf{T}}$	$[\ \pm1\quad \pm1\quad \pm1\ ]^{\mathsf{T}}$ $[\pm1/\phi\ \ \pm\phi\quad 0\ ]^{\mathsf{T}}$ $[\ 0\quad \pm1/\phi\ \ \pm\phi\ ]^{\mathsf{T}}$ $[\ \pm\phi\quad 0\quad \pm1/\phi]^{\mathsf{T}}$	$[\pm1\ \ \pm\phi\quad 0\ ]^{\mathsf{T}}$ $[\ 0\quad \pm1\ \ \pm\phi]^{\mathsf{T}}$ $[\pm\phi\quad 0\quad \pm1]^{\mathsf{T}}$
稜線長	$2\sqrt{2}$	2	$\sqrt{6}$	$2/\phi$	2
外接球の半径	$\sqrt{3}$	$\sqrt{3}$	$\sqrt{3}$	$\sqrt{3}$	$\sqrt{\phi\sqrt{5}}$

12.2 ディスプレイリスト/行列スタック/ダブルバッファ

OpenGL では高速描画を実現するために，描画命令のコンパイル機能が提供されている．たとえば，頂点数や座標変換などが多く複雑な形状の場合には，形状データの転送や変換計算のオーバヘッドが大きくなる．そこで，形状データなどをディスプレイリストという形式で，最適化コンパイルすることによって描画を高速化できる．

第 11 章「同次座標と行列計算」でも紹介したように，表示対象となる立体の位置や姿勢は `GL_MODELVIEW` 行列によって制御される．複数部品によって構成されるような形状に対して変換を施す場合には，行列をスタックに一時保存することによって実現する．OpenGL の行列計算では，各部品に作用させる順序とは逆の順に積をとるので，対象立体全体の位置や姿勢を決める変換行列を先に計算し，個別部品の局所的な配置を指定する行列を後から適用することになる．したがって，立体全体に施される行列をスタックに複製保存することで，個別部品の指定が楽になる．たとえば，部品 A と B からなる立体の変換行列 (`GL_MODELVIEW`) は，次のように処理される．

順序	処理内容	`GL_MODELVIEW` 行列	スタック内の行列
1.	恒等行列 I の設定	I	(なし)
2.	立体全体の変換行列 M_{W}	M_{W}	(なし)
3.	スタックへのプッシュ	M_{W}	M_{W}
4.	部品 A の局所配置行列 M_{A}	$M_{\mathrm{W}}M_{\mathrm{A}}$	M_{W}
5.	部品 A の描画	$M_{\mathrm{W}}M_{\mathrm{A}}$	M_{W}
6.	スタックからのポップ	M_{W}	(なし)
7.	スタックへのプッシュ	M_{W}	M_{W}
8.	部品 B の局所配置行列 M_{B}	$M_{\mathrm{W}}M_{\mathrm{B}}$	M_{W}
9.	部品 B の描画	$M_{\mathrm{W}}M_{\mathrm{B}}$	M_{W}
10.	スタックからのポップ	M_{W}	(なし)

このようにすることで，$M_{\mathrm{W}}M_{\mathrm{A}}$ すなわち「全体の変換行列部品 × A の局所配置行列」が部品 A に，$M_{\mathrm{W}}M_{\mathrm{B}}$ すなわち「全体の変換行列 × 部品 B の局所配置行列」が部品 B に，それぞれ適用される．

対象シーンが複雑になると描画バッファ（メモリ）への書き込みにも時間を要する．結果として，画像の部分を少しずつ描き足していく過程が表示されることになる．それよりは少し時間がかかったとしても描画結果を完成させたものを一気に表示したほうが好ましい．そこで描画バッファを 2 枚用意しておき，2 枚目の（表示されていない）描画バッファで描画を行い，描画が完了したところでバッファを切り替えることによって描画結果を瞬時に表示できるようになる．このダブルバッファは，特にアニメーションやインタラクティブなプログラムで有効である．シングルバッファの場合の，`glFlush` に相当する命令と考えても良い．

GL ライブラリ： OpenGL 命令のインタフェース

`glGenLists`　ディスプレイリストの作成関数
: 引数で指定した個数のディスプレイリストを作成する．作成したディスプレイリストの先頭番号を返す．

`glNewList`　ディスプレイリストの定義開始関数
: 引数で指定した番号のディスプレイリストの定義を開始する．

`glEndList`　ディスプレイリストの定義終了関数
: `glNewList` から指定された一連の描画命令をまとめてディスプレイリストとする．

`glCallList`　ディスプレイリストの実行関数
: 引数で指定した番号のディスプレイリストを実行する．

`glPushMatrix`　行列の複製保存関数
: 設定対象行列を保存用のスタックにコピーする．

`glPopMatrix`　行列の回復関数
: 設定対象行列を保存用のスタックから回復する．

GLUT ライブラリ： OS とのインタフェース

`glutSwapBuffers`　ダブルバッファの切替え関数
: 表示されていないバッファへの書き込みを進めて，書き込みが完了したら描画バッファを切り替える．なお，ダブルバッファを利用する場合には，`glutInitDisplayMode` において `GLUT_DOUBLE` を指定しておく必要がある．

12.3　各種立体を描画するプログラム

行列スタックを利用することで，立体の姿勢や配置を自由に指定できるようになる．ここでは第 13 章で紹介する対話的に立体を回転するプログラムへの導入と，多面体やフラクタル立体などを描画するプログラムについて説明する．

例１： 角度指定による立体の描画 — `myGLCanvas.py`

MyGLCancas クラスは，3 次元立体を描画するためのクラスで，描画内容を記述した `display` メソッドを持つ多面体やフラクタル立体を与えて利用する．全体的な構成は，11.3 節の例２「角度指定による立方体の描画」（`cubeAngle.py`）とほぼ同じになっている．

まず，`window` 関数としてまとめられていた描画ウィンドウの設定や生成は，初期化メソッドである `__init__` に組み込まれている．`init` メソッドは，これまでと同様に背景色や隠線・隠面消去に関する初期化を行なうとともに，ディスプレイリスト `self.objectID` を作成している．このディスプレイリストの中で多面体やフラクタル立体などの `display` メソッドを実行することで，実際の描画内容が確定される．

`reshape` メソッドは，`cameraInit` と `positionInit` の 2 つのメソッドから構成されているが，実際には，`cubeAngle` モジュールの `reshape` 関数とほとんど同じであり，z 軸方向に `depth` 分 (デフォルトでは -10) の平行移動が指定されている．モデル変換の回転行列は `reshape` メソッドではなく，`display` メソッド内で適用していて，`self.objectID` で指定される描画対象（ディスプレイリスト）を z, y, x 軸に関して，この順序で `rotZ`，`rotY`，`rotX` だけ回転している．このため `display` メソッドによって立体が表示されるごとに回転行列が適用されることになる．そこで，`display` メソッドの最初に `glPushMatrix`，最後に `glPopMatrix` を用いることで，`display` メソッドの終了時には回転行列が適用される以前の状態に戻している．`glPushMatrix` と `glPopMatrix` をコメントアウトして実行し，ウィンドウのアイコン化を繰り返すと，`glRotated` 関数が複数回重ねて実行されるので，何が起きるか確認してみるとよいだろう．

```
import sys                              # sys モジュールの import
from OpenGL.GL import *                 # GL モジュールの import
```

```
from OpenGL.GLU import *                # GLU モジュールの import
from OpenGL.GLUT import *               # GLUT モジュールの import

class MyGLCanvas(object):               # MyGLCanvas クラスの定義
  def __init__(self, width = 500, height = 500): # 初期化メソッド
    '''
    width, height - ウィンドウの幅と高さ, 省略時 500, 500
    OpenGL ウィンドウの作成と初期化を行う
    '''
    self.width, self.height = width, height # OpenGL ウィンドウの幅と高さ
    self.fieldOfView, self.near, self.far = (25, 1, 20)
                                        # カメラの設定 画角, 前方/後方クリッピング面
    self.depth, self.rotX, self.rotY, self.rotZ = (-10, 20, -30, 0)
                                        # 平行移動量, x 軸/y 軸/z 軸回りの回転角
    self.objectID = 0                   # ディスプレイリストの番号
    glutInit(sys.argv)                  # GLUT の初期化
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH) # 表示モードの指定
    glutInitWindowSize(width, height)   # ウィンドウサイズの指定
    glutInitWindowPosition(0, 0)        # ウィンドウ位置の指定
    glutCreateWindow(b'OpenGL')         # ウィンドウの作成

  def init(self, dispObj):              # OpenGL の初期化
    '''
    OpenGL を初期化する
    '''
    glClearColor(0, 0, 0, 1)            # 背景色の指定
    glEnable(GL_DEPTH_TEST)             # 奥行きテストの利用
    glEnable(GL_CULL_FACE)              # 背面除去の利用
    self.objectID = glGenLists(1)       # ディスプレイリストの作成
    glNewList(self.objectID, GL_COMPILE) # ディスプレイリストへの登録開始
    dispObj.display()                   # 描画命令
    glEndList()                         # ディスプレイリストへの登録終了

  def argsInit(self, args):             # シェル引数などの初期化
    '''
    シェル引数やキーボード入力に合わせて初期化する
    '''
    if len(args) == 1 and args[0] != '': # 文字列が 1 つの場合
      self.fieldOfView    = float(args[0]) # 画角の設定
    if len(args) == 2:                  # 文字列が 2 つの場合
      self.near, self.far = (float(args[0]), float(args[1]))
                                        # 前方/後方クリッピング面の設定
    if len(args) == 3:                  # 文字列が 3 つの場合
      self.rotX, self.rotY, self.rotZ = \
                            (float(args[0]), float(args[1]), float(args[2]))
                                        # x 軸/y 軸/z 軸回りの回転角の設定

  def reshape(self, width, height):    # ウィンドウのサイズ変更に伴うコールバックメソッド
    '''
    width  - 変更後の OpenGL ウィンドウの幅
    height - 変更後の OpenGL ウィンドウの高さ
    ウィンドウサイズ変更に伴う処理を行う
    '''
    self.cameraInit(width, height)      # カメラ (レンズ・フィルム) の設定
    self.positionInit()                 # 初期位置の設定

  def cameraInit(self, width, height):  # カメラ (レンズ・フィルム) の設定
    '''
    width  - 変更後の OpenGL ウィンドウの幅
    height - 変更後の OpenGL ウィンドウの高さ
    カメラのレンズやフィルムに対応する設定を行う
    '''
    self.width, self.height = width, height
    aspect = width / height             # スクリーン面の縦横比
    glViewport(0, 0, width, height)     # ビューポートの設定
    glMatrixMode(GL_PROJECTION)         # 投影変換行列の設定開始
```

```
    glLoadIdentity()                        # 恒等行列での初期化
    gluPerspective(self.fieldOfView, aspect, self.near, self.far)
                                            # 透視投影変換行列の設定
    glMatrixMode(GL_MODELVIEW)              # モデル変換行列の設定開始
    glLoadIdentity()                        # 恒等行列での初期化

  def positionInit(self):                   # 初期位置の設定
    '''
    初期位置の設定を行う
    '''
    glTranslated(0, 0, self.depth)          # 平行移動（z 軸）

  def display(self):                        # 描画要求に伴うコールバックメソッド
    '''
    描画要求に伴う処理を行う（描画内容の定義）
    '''
    glPushMatrix()                          # 行列の複写（待避）
    glRotated(self.rotX, 1, 0, 0)           # 回転（x 軸）
    glRotated(self.rotY, 0, 1, 0)           # 回転（y 軸）
    glRotated(self.rotZ, 0, 0, 1)           # 回転（z 軸）
    self.coredisplay()                      # 描画の本体
    glPopMatrix()                           # 複写行列の回復

  def coredisplay(self):                    # 描画の本体
    '''
    実際の描画処理を行う
    '''
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) # 背景のクリア
    glCallList(self.objectID)               # ディスプレイリストの呼出し
    glutSwapBuffers()                       # ダブルバッファの切り替え

  def loop(self):                           # コールバックメソッドの設定とループ起動
    '''
    reshape と display コールバックメソッドを設定し，ループを起動する
    '''
    glutReshapeFunc(self.reshape)           # reshape コールバックメソッドの登録
    glutDisplayFunc(self.display)           # display コールバックメソッドの登録
    glutMainLoop()                          # GLUT のメインループ起動

def getArgs():                              # シェル引数/キーボード入力による文字列の取得
  '''
  シェル引数やキーボード入力によって文字列を取得する
  '''
  if len(sys.argv) > 1:                     # シェル引数がある場合
    args = sys.argv[1:]                     # 第 1 引数以降の文字列
  else:                                     # シェル引数がない場合
    args = input('FOV / near far / rotX rotY rotZ / [] -> ').split(' ')
  return args                    # 画角，前方/後方クリッピング面，x 軸/y 軸/z 軸回りの回転角
```

例２： 描画多面体 — polyhedron.py

これまでの第 10 章や第 11 章のプログラムでは，立方体の描画には cubePosition モジュールの display 関数を用いていた．本章以降，多面体の描画には Polyhedron クラスの display メソッドを用いる．Polyhedron クラスは多面体を描画するための抽象クラスで，display メソッドは定義されているが，多面体を具体的に構成する頂点座標値，面の頂点番号列，稜線の頂点番号列，各面の描画色などのデータは空のままである．実際に描画される多面体は，この Polyhedron クラスを継承し，頂点座標値などの情報を与えることによって定義される．

```
from OpenGL.GL import *                     # GL モジュールの import

class Polyhedron(object):                   # Polyhedron クラスの定義
  def __init__(self, vertices = (), faces = (), edges = (), colors = ()):
                                            # 初期化メソッド
    '''
    vertices - 頂点座標値，省略時 空タプル
    faces    - 面の頂点番号列，省略時 空タプル
```

```
    edges    - 稜線の頂点番号列，省略時 空タプル
    colors   - 面の描画色，省略時 空タプル
    多面体を初期化する
    '''
    self.vertices, self.faces, self.edges, self.colors = \
                  (vertices, faces, edges, colors)
                              # 頂点座標値，面の頂点番号列，稜線の頂点番号列，面の描画色

  def display(self):                      #（多面体の）描画メソッド
    '''
    多面体の描画を行う
    '''
    for i in range(len(self.faces)):      # 多角形描画の反復（面番号 i)
      glBegin(GL_POLYGON)                 # 多角形描画の開始
      glColor3dv(self.colors[i])          # 描画色の指定
      for j in range(len(self.faces[i])): # 面頂点の反復（頂点番号 j)
        glVertex3dv(self.vertices[self.faces[i][j]]) # 頂点座標値の指定
      glEnd()                             # 多角形描画の終了
```

例３： 描画立方体 — cube.py

例１「角度指定による立体の描画」の MyGLCanvas クラスと，例２の「描画立方体」を継承した描画立方体の Cube クラスとを組合せて，11.3 節の例２「角度指定による立方体の描画」と同じ振舞いをするプログラムを実現している．Cube クラスの初期化メソッド __init__では，頂点座標値，面の頂点番号列，稜線の頂点番号列，面の描画色の各データに実際の値を埋め込んでいる．

```
from myGLCanvas import MyGLCanvas, getArgs # myGLCanvas モジュールの import
from polyhedron import Polyhedron         # polyhedron モジュールの import

class Cube(Polyhedron):                   # Cube クラスの定義
  def __init__(self):                     # 初期化メソッド
    '''
    立方体を初期化する
    '''
    super().__init__(                     # Polyhedron クラスの初期化メソッド
            ((-1, -1, -1), ( 1, -1, -1), ( 1,  1, -1), (-1,  1, -1),
             (-1, -1,  1), ( 1, -1,  1), ( 1,  1,  1), (-1,  1,  1)), # 頂点座標値
            ((1, 2, 6, 5), (2, 3, 7, 6), (4, 5, 6, 7),
             (0, 4, 7, 3), (0, 1, 5, 4), (0, 3, 2, 1)),        # 各面の頂点番号列
            ((0, 1), (1, 2), (2, 3), (3, 0), (0, 4), (1, 5),
             (2, 6), (3, 7), (4, 5), (5, 6), (6, 7), (7, 4)),  # 各稜線の頂点番号列
            ((  0,   1,   1), (  1,   0,   1), (  1,   1,   0),
             (  0, 0.5, 0.5), (0.5,   0, 0.5), (0.5, 0.5,   0)) ) # 各面の描画色

def main():                               # main 関数
  canvas = MyGLCanvas()                   # MyGLCanvas の作成
  dispObj = Cube()                        # Cube オブジェクトの作成
  canvas.init(dispObj)                    # OpenGL の初期化
  canvas.argsInit(getArgs())              # シェル引数/キーボード入力による文字列の取得
  canvas.loop()                           # コールバックメソッドの設定とループ起動

if __name__ == '__main__':                # 起動の確認（コマンドラインからの起動)
  main()                                  # main 関数の呼出
```

例４： フラクタル立体のプログラム — fractal.py

第 6 章「フラクタル図形」で説明したように，フラクタル形状は自己相似性があり，縮小写像の反復によって生成できる．つまり，縮小写像を施した部分形状を描画する操作を，再帰的に適用することで生成される．このような部分の組合せには，glPushMatrix と glPopMatrix が利用できる．以下のプログラムは，基本立体に対して，縮小と平行移動を組み合せて構成されるフラクタル立体の抽象クラスである．シェル引数ないしキーボード入力で再帰回数を指定する必要があるため，文字列を取得するための getArgs 関数を定義してある．

```
import sys                                # sys モジュールの import
```

```
from OpenGL.GL import *                    # GL モジュールの import

class Fractal(object):                     # Fractal クラスの定義
  def __init__(self, dispObj, scale, vecs, times): # 初期化メソッド
    '''
    dispObj - 基本立体
    scale   - 縮小率
    vecs    - 平行移動ベクトル
    times   - 縮小写像の再帰回数
    フラクタル立体を初期化する
    '''
    self.primitive, self.scale, self.vecs, self.times = \
          (dispObj, scale, vecs, times) # 基本立体，縮小率，平行移動ベクトル，再帰回数

  def display(self):                       # フラクタル立体の描画メソッド
    '''
    フラクタル立体の描画を行う
    '''
    self.drawFractal(self.times)           # フラクタル立体の描画

  def drawFractal(self, t):                # フラクタル立体の描画
    '''
    t - 残りの再帰回数
    与えられた再帰回数のフラクタル立体を描画する
    '''
    if t > 0:                              # 再帰回数が 1 回以上
      for i in range(len(self.vecs)):      # 縮小写像の数だけ反復
        glPushMatrix()                     # 縮小写像の複写（待避）
        glTranslated(self.vecs[i][0], self.vecs[i][1], self.vecs[i][2])
                                           # 平行移動適用
        glScaled(self.scale, self.scale, self.scale) # (拡大) 縮小の適用
        self.drawFractal(t-1)              # 再帰呼出し
        glPopMatrix()                      # 複写行列の回復
    else:                                  # これ以上再帰せず
      self.primitive.display()             # 基本図形の描画

def getArgs():                             # シェル引数/キーボード入力による文字列の取得
  '''
  シェル引数やキーボード入力によって文字列を取得する
  '''
  if len(sys.argv) > 1:                    # シェル引数がある場合
    args = sys.argv[1:]                    # 第 1 引数以降の文字列
  else:                                    # シェル引数がない場合
    args = input('times [FOV / near far / rotX rotY rotZ] -> ').split(' ')
                                 # 画角，前方/後方クリッピング面，x 軸/y 軸/z 軸回りの回転角
  return (int(args[0]), args[1:])          # 再帰回数と初期化文字列を返す
```

例５： メンガースポンジ — mengerSponge.py

メンガースポンジはシェルピンスキーカーペットの 3 次元版とも言えるフラクタル立体である．これは立方体を基本立体としており，1 段階ごとに 1/3 に縮小された合計 20 個の形状が，8 頂点と 12 稜線の位置に配置される．立方体の中心が原点にあるので，1/3 の縮小によって，中心を共有した大きさ 1/3 の形状ができる．この形状の頂点を共有するように移動するには，原点から各頂点までの $(1 - 1/3)$ 倍の位置に平行移動すれば良い．稜線についても同様で，原点から各稜線の中点までの $(1 - 1/3)$ 倍の位置に平行移動する．図 12.2 に 4 回の再帰を施した結果を示す．

```
import numpy as np                         # numpy モジュールの import (np で)
from myGLCanvas import MyGLCanvas          # myGLCanvas モジュールの import
from fractal import Fractal, getArgs       # fractal モジュールの import
from cube import Cube                      # cube モジュールの import

class MengerSponge(Fractal):               # MengerSponge クラスの定義
  def __init__(self, times):               # 初期化メソッド
    '''
    times - 再帰回数
```

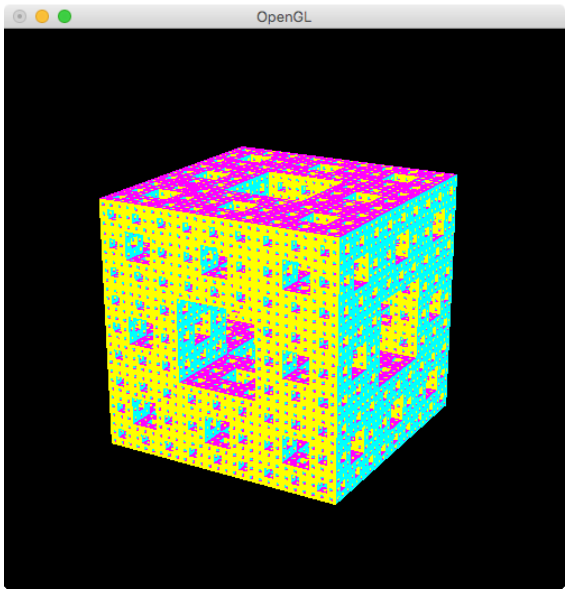

Figure 12.2: メンガースポンジ（再帰を 4 回施した結果）

```
    メンガースポンジを初期化する
    '''
    cube = Cube()                          # 基本立体（立方体）
    nv   = len(cube.vertices)              # 頂点数
    ne   = len(cube.edges)                 # 稜線数
    SCL  = 1 / 3                           # 縮小率
    vecs = []                              # 平行移動ベクトル用リストの初期化
    for i in range(nv):                    # 頂点数分の反復
      vecs.append(np.array(cube.vertices[i]) * (1-SCL)) # 頂点への平行移動ベクトル
    for i in range(ne):                    # 稜線数分の反復
      mid = (np.array(cube.vertices[cube.edges[i][0]]) +
             np.array(cube.vertices[cube.edges[i][1]])) / 2
                                           # 稜線中点
      vecs.append(mid * (1-SCL))           # 稜線中点への平行移動ベクトル
    super().__init__(cube, SCL, vecs, times) # Fractal オブジェクトの初期化

def main():                                # main 関数
  times, args = getArgs()   # シェル引数/キーボードによる文字列（再帰回数/投影設定）の取得
  canvas = MyGLCanvas()                    # MyGLCanvas の作成
  dispObj = MengerSponge(times)            # MengerSponge オブジェクトの作成
  canvas.init(dispObj)                     # OpenGL の初期化
  canvas.argsInit(args)                    # 投影設定の初期化
  canvas.loop()                            # コールバックメソッドの設定とループ起動

if __name__ == '__main__':                 # 起動の確認（コマンドラインからの起動）
  main()                                   # main 関数の呼出
```

章末課題

正多面体の表示

例 3「描画立方体」のプログラムを参考に，正多面体を表示するプログラムを作成してみよ．

フラクタル立体の表示

例 5「メンガースポンジ」のプログラムを参考に，フラクタル立体を表示するプログラムを作成してみよ．フラクタル立体の例としては，図 12.3 に挙げるものなどが考えられる．各ステップにおけるスケールの変化は，メンガースポンジは 1/3 であるが，シェルピンスキー四面体と八面体片は 1/2，十二面体片は $1/(2+\phi)$，二十

面体片は $1/(1+\phi)$ となる（ただし，ϕ は黄金比で $\phi = (1+\sqrt{5})/2$ ）．また，それぞれのフラクタル立体のハウスドルフ次元を計算せよ．

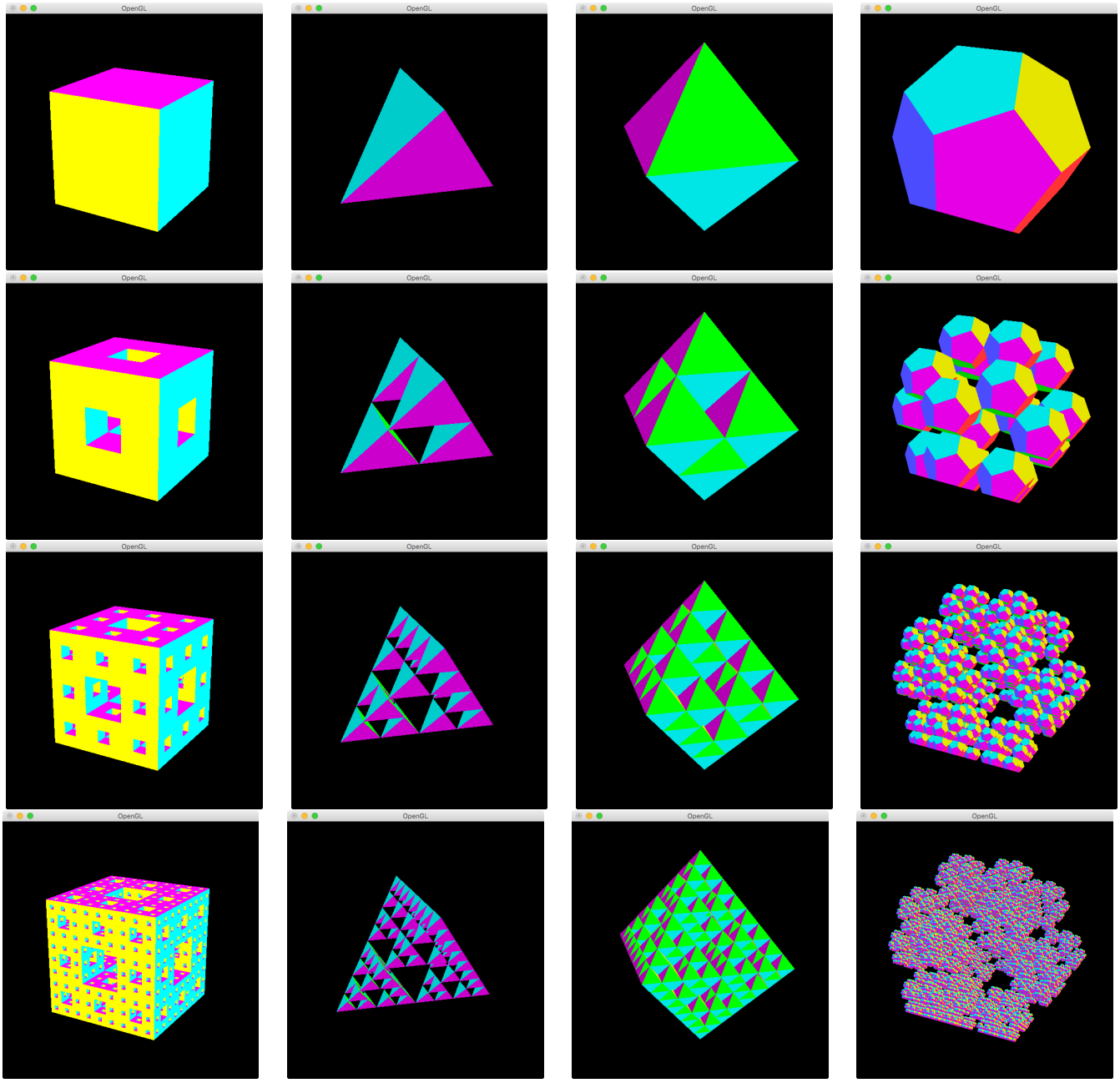

Figure 12.3: フラクタル立体の例．左から右方向の順に，メンガースポンジ，シェルピンスキー四面体，八面体片，十二面体片 (口絵参照)