Chapter 10

3次元グラフィクスの基礎

10.1　グラフィクスパイプライン

3 次元情報を 2 次元画像として表示するまでの一連の処理の流れをグラフィクスパイプラインと呼ぶ．特に形状情報に着目して，3 次元空間から 2 次元平面へと座標変換を施す過程を指す場合には，ビューイングパイプラインと呼ぶこともある．

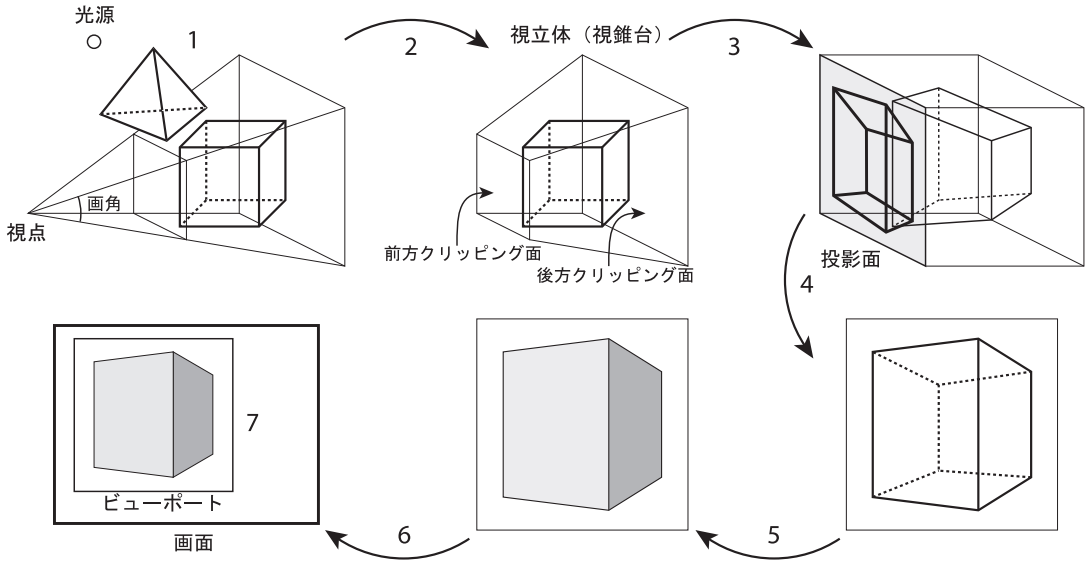

Figure 10.1: グラフィクスパイプライン

グラフィクスパイプラインの例として，図 10.1 のような 7 段階の処理が考えられる．

1. **モデル変換/視野変換** (model/view transformation)
   視点の位置や視線の向きなどに合わせて，3 次元データに平行移動や回転，拡大縮小などを施し，適当な位置，向き，大きさに変換する．

2. **クリッピング** (clipping)
   3 次元空間から表示すべき部分として，**視立体** (viewing volume) と呼ばれる領域を切り出す．透視変換が施される場合には，もとの 3 次元空間において視立体は四角錘台状になることから，**視錐台** (viewing frustum) と呼ばれる．図 10.1 のように視錐台の底面となる 2 枚の面を前方クリッピング面と後方クリッピング面と呼ぶ．

3. **投影変換** (projection transformation)
   投影変換によって，3 次元データを投影面上の 2 次元空間のデータに変換する．3 次元グラフィクスでは多くの場合，透視変換 (perspective transformation) が用いられる．

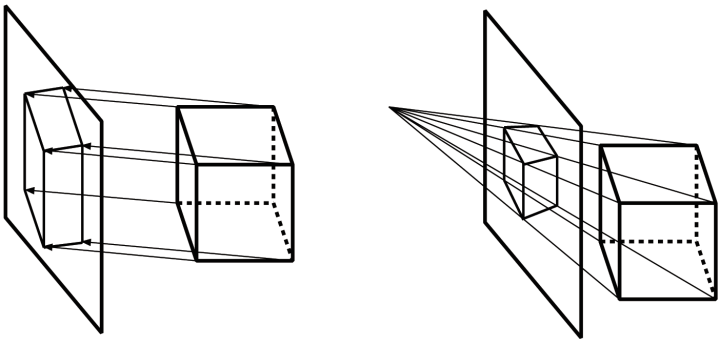

Figure 10.2: 平行投影と透視投影

4. **隠線消去/隠面消去** (hidden line/surface removal)
   複数の物体と視点との位置関係によって，物体は部分的に遮蔽されて，視点から観察されない場合もある．この遮蔽などに伴う可視・不可視を判定して，不可視部分を削除する隠線消去や隠面消去を実施する．

5. **陰影処理** (shading/shadowing)
   視点と光源，面の位置関係から明るさや色を計算する陰影処理行なう．局所的な光の反射を考慮して陰を計算するシェーディングモデルや，より大域的な光の遮閉や間接反射などによって生じる影を計算する大域照明モデルなどがある．

6. **ウィンドウ・ビューポート変換** (window-viewport transformation)
   画面上で実際に描画する領域として，ビューポートと呼ばれる矩形領域に画像を当てはめる．

7. **ラスタ化** (rasterization)
   2 次元情報を標本化して，画素情報に変換する．

なお，OpenGL などのハードウェアベースの描画パイプラインでは，1. モデル変換/視野変換 → 2. クリッピング → 3. 投影変換 → 6. ウィンドウ・ビューポート変換→ 7. ラスタ化 → 4. 隠線消去/隠面消去，という順序をとることが多い．この際,「5. 陰影処理」は「1. モデル変換/視野変換」の直後か,「7. ラスタ化」の後に行なわれる．従来型の固定シェーダの場合は前者，プログラマブルシェータを利用したピクセル単位描画の場合は後者となる．

3 次元情報を 2 次元画像として表示するための鍵となる処理は投影である．図 10.2 左のように 3 次元の点を一定方向に投影する方法，すなわち各点から平行線を延ばして 2 次元平面 (投影面) との交点を求める方法は，**平行投影** (parallel projection) と呼ばれる．平行投影の場合には，3 次元空間で平行な直線群は投影後のスクリーン上でも平行になる．特に投影方向が投影面に垂直となる場合を**直投影** (orthogonal projection) と呼ぶ．座標平面への直投影は，非常に単純な処理で実現できる．たとえば，xy 平面への直投影は 3 次元座標の x, y 成分のみを，そのまま用いればよい．

これに対して，図 10.2 右のように 3 次元の各点を視点にあたる投影中心に向けて投影する方法，すなわち視点から各点に向けて放射線状に延びる直線群を構成して投影面との交点を求める方法を**透視投影** (perspective projection) と呼ぶ．透視投影はカメラや人間の目と似た投影方法であり，自然な画像が得られる．透視投影の場合には，3 次元空間で平行な直線群は投影後のスクリーン上では，一般に平行にはならずに 1 点，すなわち**消失点** (vanishing point) で交差する．どこまでもまっすぐに延びる電車のレールを考えると，地平線で 1 点で交わるように見えることに相当する．

10.2 PyOpenGL における変換行列

OpenGL では，カメラとモデルとの位置関係を表現するモデル変換 (視野変換) 行列とカメラのレンズやフィルム面のアスペクト比 (縦横比) などを表す投影変換行列との 2 種類の行列を持っている．前者は `GL_MODELVIEW` 行列，後者は `GL_PROJECTION` 行列と呼ばれる．さらに OpenGL のユーティリティ機能として，これらの行列を生成・設定する関数が提供されており，行列を意識することなく描画できるようになっている．

GL ライブラリ： OpenGL 命令のインタフェース

`glViewport`　描画領域の指定関数
　引数として四角形領域の左上座標 `x`，`y` と大きさ `width`，`height` を与えて，`GLCanvas` 内で実際に描画される領域を指定する．

`glMatrixMode`　設定対象となる行列の指定関数
　引数として `GL_MODELVIEW` ないし `GL_PROJECTION` を与えて，以後設定対象となる行列を指定する．

`glLoadIdentity`　行列の初期化関数
　設定対象行列を恒等行列 (単位行列) に初期化する．

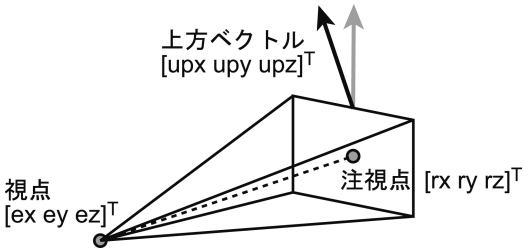

Figure 10.3: `gluLookAt` の説明図

GLU ライブラリ： OpenGL のユーティリティ命令のインタフェース
OpenGL のユーティリティ命令 (ハードウェアとは独立でソフトウェアによる関数) は，`glu` で始まる関数によって提供される．

`gluLookAt` モデル/視野変換行列の設定関数
　`GL_MODELVIEW` 行列に対して利用する関数で，視点とモデル空間との相対位置を指定する行列を生成・設定する．関数の引数は以下のとおりで，それぞれ図 10.3 のようになっている．

```
void gluLookAt(double ex, double ey, double ez,
               double rx, double ry, double rz,
               double upx, double upy, double upz);
   ex,ey,ez     視点の座標
   rx,ry,rz     注視点 (参照点) の座標
   upx,upy,upz 画面の上方向ベクトル (投影面へ投影したベクトルが画面上方になる)
```

`gluPerspective` 透視投影行列の設定関数
　`GL_PROJECTION` 行列に対して利用する関数で，透視投影の行列を生成・設定する．関数の引数は以下のとおりで，それぞれ図 10.4 のようになっている．

```
void gluPerspective(double fieldOfView, double aspect,
                    double near, double far);
   fieldOfView   y 方向 (縦方向) の視野角 (画角)
   aspect        xy のアスペクト比 (投影面の縦横比)
   near          前方クリッピング面までの距離
   far           後方クリッピング面までの距離
```

10.3　PyOpenGL による 3 次元グラフィクスのプログラム

PyOpenGL を利用して，図 10.5 のように原点を中心とし一辺の長さが 2.0 の立方体を描画する．ここで頂点番号は，プログラム中の `vertices` 配列の添字に対応している．

例１： 透視投影による立方体の描画 — `cubePosition.py`
　原点を中心として，1 辺の長さが 2 の立方体を描く．注視点は原点，すなわち立方体の中心で，視点の位置はコマンド引数ないしキーボード入力によって指定できる．引数を与えない場合には，標準の視点位置として $[4\ 3\ 7]^{\mathsf{T}}$ の位置から見た図が描画される．

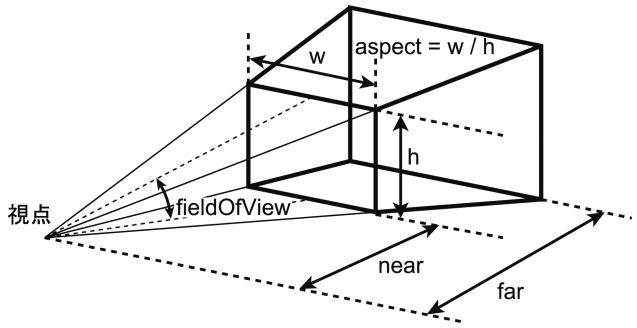

Figure 10.4: gluPerspective の説明図

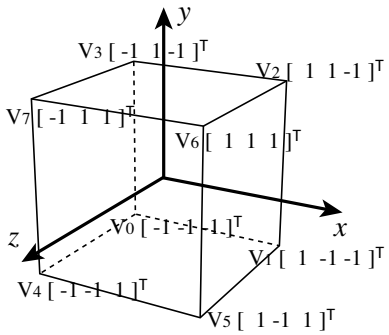

Figure 10.5: 立方体のデータ

まず，window 関数では，GLUT ライブラリを用いて，描画ウィンドウの設定と作成を行っている．次に，init 関数では，背景色の設定とともに，GL_DEPTH_TEST, GL_CULL_FACE を設定している．この 2 つの設定は，window 関数の glutInitDisplayMode における GLUT_DEPTH ならびに display 関数の glClear における GL_DEPTH_BUFFER_BIT に関連しているが，詳細については第 14 章「隠線・隠面消去」で説明する．

argsInit 関数は，視点位置に関する文字列情報を，シェル引数ないしキーボード入力から取得している．この情報は，reshape 関数で利用される．loop 関数では，GLUT ライブラリを用いて，reshape 関数と display 関数をコールバック関数として設定したうえで，実行ループを起動している．

reshape 関数は描画ウィンドウの形状が変更された際に呼び出されるコールバック関数であるが，描画ウィンドウの幅 width と高さ height に合わせて，GL_PROJECTION と GL_MODELVIEW の 2 つの変換行列を設定している．いずれも glMatrixMode で変換行列の指定したのちに，いったん glLoadIdentity で恒等行列を設定して，gluPerspective ないし gluLookAt を用いて行列を定めている．この gluLookAt では，描画にあたって y 軸の正の向きが，画面上方となるように設定されている．

display 関数では，立方体の描画が定義されている．立方体の頂点座標値は vertices 配列に格納されており，各面を構成する頂点データは faces 配列に頂点番号の列として表現されている．このとき，頂点は外から見て左回りの順で与えられていることに注意して欲しい．ここでは描画要素として，GL_QUADS を利用している．x 軸，y 軸，z 軸に垂直な各面は，それぞれシアン，マゼンタ，黄色が指定されており，各軸の正の側は明るい色，負の側は暗い色になっている．図 10.6 のように視点位置を変えて，様々な方向から眺めた画像を作ってみよう．gluLookAt では，y 軸の正の向きが画面上方となるように設定されているため，いずれの方向から見ても立方体の y 軸の面であるマゼンタの面が画面内の上下方向に現れる．

```
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

eyeX, eyeY, eyeZ = (4, 3, 7)                # デフォルトの視点位置
vertices = ((-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
faces    = (( 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
colors   = ((  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 window(width = 500, height = 500):      # ウィンドウ作成
  '''
  width  - OpenGL ウィンドウの幅
  height - OpenGL ウィンドウの高さ
  GLUT を初期化して，OpenGL ウィンドウを作成する
  '''
```

```
  glutInit(sys.argv)                        # GLUT の初期化
  glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE | 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():                                 # 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

def argsInit():                             # 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
  '''
  global eyeX, eyeY, eyeZ                   # 大域変数 eyeX, eyeY, eyeZ
  if len(sys.argv) > 3:                     # シェル引数が 3 つ以上ある場合
    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('eyeX eyeY eyeZ or [] -> ').split(' ') # 視点位置の文字列を入力
  if len(args) >= 3:                        # 視点位置 (無指定の場合は, デフォルト値)
    eyeX, eyeY, eyeZ = (float(args[0]), float(args[1]), float(args[2]))

def reshape(width, height):                 # ウィンドウのサイズ変更に伴うコールバック関数
  '''
  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
  '''
  global eyeX, eyeY, eyeZ                   # 大域変数 eyeX, eyeY, eyeZ (実際は不要)
  fieldOfView, near, far = (25, 1, 20)      # カメラの設定 画角, 前方/後方クリッピング面
  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(fieldOfView, aspect, near, 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
  gluLookAt(eyeX, eyeY, eyeZ, 0, 0, 0, 0, 1, 0) # モデル変換行列の設定

def display():                              # 描画要求に伴うコールバック関数
  '''
  描画要求に伴う処理を行う (立方体を描画する)
  '''
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) # 背景のクリア
  glBegin(GL_QUADS)                         # 四角形描画の開始
  for i in range(len(faces)):               # 四角形描画の反復 (面番号 i)
    glColor3dv(colors[i])                   # 描画色の指定
    for j in range(len(faces[i])):          # 面頂点の反復 (頂点番号 j)
      glVertex3dv(vertices[faces[i][j]]) # 頂点座標値の指定
  glEnd()                                   # 四角形描画の終了
  glFlush()                                 # 描画命令の送信

def loop():                                 # コールバック関数の設定とループ起動
  '''
  reshape と display コールバック関数を設定し, ループを起動する
  '''
  glutReshapeFunc(reshape)                  # reshape コールバック関数の登録
  glutDisplayFunc(display)                  # display コールバック関数の登録
  glutMainLoop()                            # GLUT のメインループ起動

def main():                                 # main 関数
  window()                                  # ウィンドウの作成
```

```
  init()                                  # OpenGL の初期化
  argsInit()                              # シェル引数などの初期化
  loop()                                  # コールバック関数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２： 多角形としての立方体の描画 — cubePolygon.py

例１のプログラムでは，立方体の描画にあたって描画要素に GL_QUADS を用いていた．四角形は多角形の一種であるから，描画要素に GL_POLYGON を利用することもできる．ただし，GL_POLYGON の場合には，与えられた全頂点列で１つの多角形を描くので，glBegin と glEnd の両関数を呼び出すタイミングに注意せよ．また，新たに定義した display 関数を呼び出すためには，display 関数を呼び出している loop 関数も同一ファイル内になくてはならない（cubePosition モジュールの loop 関数は利用できない）．

```
import sys                                # sys モジュールの import
from OpenGL.GL import *                   # GL モジュールの import
from OpenGL.GLUT import *                 # GLUT モジュールの import
import cubePosition as cp                 # cubePosition モジュールの import

def display():                            # 描画要求に伴うコールバック関数
  '''
  描画要求に伴う処理を行う（立方体を描画する）
  '''
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) # 背景のクリア
  for i in range(len(cp.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(cp.colors[i])              # 描画色の指定
    for j in range(len(cp.faces[i])):     # 面頂点の反復（頂点番号 j）
      glVertex3dv(cp.vertices[cp.faces[i][j]]) # 頂点座標値の指定
    glEnd()                               # 多角形描画の終了
  glFlush()                               # 描画命令の送信

def loop():                               # コールバック関数の設定とループ起動
  '''
  reshape と display コールバック関数を設定し，ループを起動する
  '''
  glutReshapeFunc(cp.reshape)             # reshape コールバック関数の登録
  glutDisplayFunc(display)                # display コールバック関数の登録
  glutMainLoop()                          # GLUT のメインループ起動

def main():                               # main 関数
  cp.window()                             # ウィンドウの作成
  cp.init()                               # OpenGL の初期化
  cp.argsInit()                           # シェル引数などの初期化
  loop()                                  # コールバック関数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

視点位置

例１「透視投影による立方体の描画」のプログラムにおいて，様々な視点からの画像を作成してみよ．

立体データの意味

例１「 透視投影による立方体の描画」のプログラムにおいて，立方体の面情報 faces を書き換えてみよ．たとえば，最初の { 1, 2, 6, 5 } を { 1, 2, 5, 6 } や { 5, 6, 2, 1 } などに変更すると何が起きるか確認してみよ．また，ここでも視点を変えて，画像を作成してみよ．

その他の立体データ

例 1「 透視投影による立方体の描画」のプログラムを参考にして，正四面体を描くプログラムを書いてみよ．正四面体の 4 頂点としては，立方体の 8 頂点のうち，ねじれの位置にある 4 頂点を用いるとよい．なお，例 1 のプログラムは描画要素に `GL_QUADS` を用いているので，三角形で構成される正四面体の描画には，そのままでは利用できないことに注意せよ．

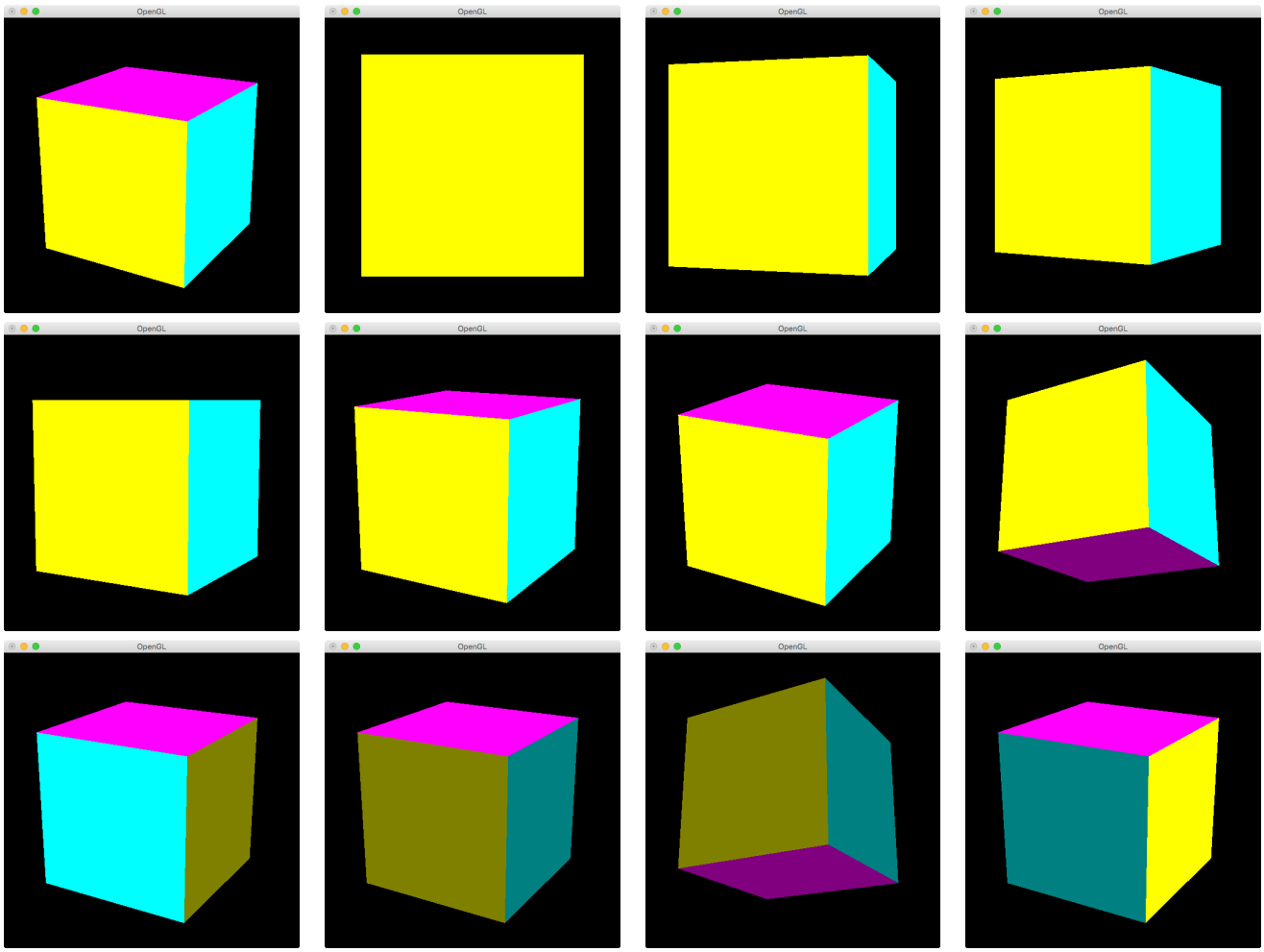

Figure 10.6: 立方体の描画．実行時に与えた引数は，左上から右方向の順に，それぞれ「(なし)」,「0 0 7」,「2 0 7」,「4 0 7」,「4 1 7」,「4 2 7」,「4 3 7」,「4 −3 7」,「7 3 −4」,「−4 3 −7」,「−4 −3 −7」,「−7 3 4」