Chapter 17

シェーディング

17.1 光の反射モデル

これまでのプログラムでは描画色を直接に指定していたが，実際の環境では対象の材質や光の当たり方で見た目の色が変化する．たとえば映画館では白いスクリーンに色のついた光が当たることによって，映画の画面が写し出される．仮に赤いスクリーンであれば赤のモノクロ画面になるし，黒いスクリーンであれば単に真っ黒な画面となってしまう．光の反射を考慮して色をつける作業を**シェーディング** (shading) または**陰付け**と呼ぶ．OpenGL などのコンピュータグラフィクスでは，光の反射特性を**環境成分** (ambient component)，**拡散成分** (diffuse component)，**鏡面成分** (specular component) の 3 つの成分に分けて考える．これは物理現象を正確に表すものではないが，現実の反射をそれらしく表せるモデルとなっている．なお，これらは光の色を表す反射モデル (reflection model) であり，実際の計算では各成分ごとに RGB 値を算出することになる．

環境成分 R_a とは，間接光によるぼんやりとした明るさに由来する光の成分である．現実の世界では，直接的な光源だけではなく，他の物質からの反射によって空間は光で満ちている．たとえば，部屋の机の下など，照明によって直接照らされていない場所も真っ暗ではないのは，この間接光によるものである．反射光の環境成分は，次式で与えられる．

$$R_a = I_a k_a \tag{17.1}$$

ただし，I_a は入射光の環境成分，k_a は材質の環境成分の反射率である．

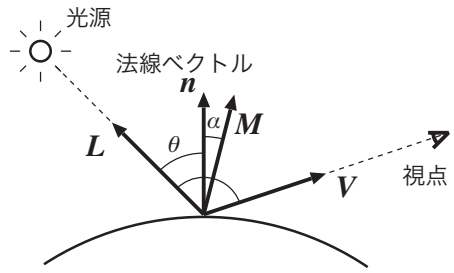

Figure 17.1: 反射モデル

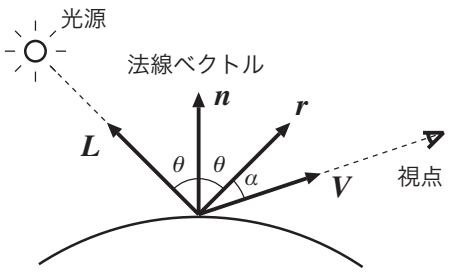

Figure 17.2: フォンの鏡面反射モデル

拡散成分 R_d とは，映画館のスクリーンや白い紙のように，反射光の強さが入射光の当たり方，つまり入射する光の強さと方向によってのみ定まる光の成分であり，観察する位置には依存しない．光源への方向ベクトル $\boldsymbol{L}$ と面の法線ベクトル $\boldsymbol{n}$ のなす角 θ の余弦 $\cos\theta$ によって定まる (図 17.1)．2 つのベクトル $\boldsymbol{L}, \boldsymbol{n}$ が正規化されており単位ベクトルであれば，余弦の値はベクトルの内積として計算される．このような反射はランバート (Lambert) 反射ないしは完全拡散反射と呼ばれ，次式で与えられる．

$$R_d = I_d k_d \cos\theta = I_d k_d \boldsymbol{n} \cdot \boldsymbol{L} \tag{17.2}$$

ただし，I_d は入射光の拡散成分，k_d は材質の拡散成分の反射率である．

鏡面成分 R_s とは，金属やプラスチックなどの光沢に相当するもので，反射光の強さが入射光の当たる向きと観察する方向に依存する．光源への方向ベクトルと面の法線ベクトル $\boldsymbol{n}$ によって決まる正反射の方

向を中心に比較的狭い領域で強くなる．そこで R_s は，光源方向 $\boldsymbol{L}$ と視点方向 $\boldsymbol{V}$ との中間方向のベクトル $\boldsymbol{M} = (\boldsymbol{L} + \boldsymbol{V})/|\boldsymbol{L} + \boldsymbol{V}|$ と法線ベクトル $\boldsymbol{n}$ のなす角 α の余弦 $\cos\alpha$ のベキ乗によって与えられる (図 17.1)．2 つのベクトルが正規化されていれば，次式で計算される．

$$R_s = I_s k_s \cos^n\alpha = I_s k_s (\boldsymbol{n} \cdot \boldsymbol{M})^n \tag{17.3}$$

ただし，I_s は入射光の鏡面成分，k_s は材質の鏡面成分の反射率である．また n は減衰を表す係数であり，この値が大きくなると光沢によって明るくなる部分が小さくなる．これは，後述する OpenGL の GL_SHININESS に相当する．なお鏡面反射の性質として，一般に光源の色と同じ色を反射するので，鏡面反射の場合には RGB 成分に同じ値を用いる．最終的な反射光は，(17.1)〜(17.3) 式の 3 成分を足し合わせたものとなり，RGB ごとに次式で計算される．

$$R_a + R_d + R_s = I_a k_a + I_d k_d \boldsymbol{n} \cdot \boldsymbol{L} + I_s k_s (\boldsymbol{n} \cdot \boldsymbol{M})^n \tag{17.4}$$

なお，上述の鏡面成分は OpenGL での計算法であるが，コンピュータグラフィクスでよく利用される鏡面成分の計算法としてフォン (Phong) のモデルもある．フォンのモデルでは，図 17.2 のように入射光に対する正反射の方向を考え，その正反射方向 $\boldsymbol{r}$ と視線方向 $\boldsymbol{V}$ とがなす角 α の余弦 $\cos\alpha$ のベキ乗によって反射光の強さを求める．すなわち，次式のようになる．

$$R_s = I_s k_s \cos^n\alpha = I_s k_s (\boldsymbol{r} \cdot \boldsymbol{V})^n$$

なお正反射方向 $\boldsymbol{r}$ は $\boldsymbol{L} + \boldsymbol{r} = 2\boldsymbol{n}\cos\theta$ となることから，

$$\boldsymbol{r} = 2\,(\boldsymbol{L} \cdot \boldsymbol{n})\,\boldsymbol{n} - \boldsymbol{L}$$

によって計算される．

17.2 スムーズシェーディング

球面などの曲面を描画する場合には，細かな多面体の集合によって曲面を近似して描画する．しかし，人間の目にはシュブルール錯視によって隣接する輝度の差が強く知覚されるため，かなり細かく近似しても凹凸のある多面体として見えてしまう．そこで利用される手法が**スムーズシェーディング** (smooth shading) である．反射光の輝度は面の法線ベクトルによって決定されるので，スムーズシェーディングでは，法線を面ではなく各頂点に与えて，多角形の内部の輝度を滑らかに変化させることで曲面のように感じさせる．なおスムーズシェーディングに対して，多面体の各面の内部を 1 色で塗りつぶす手法を，**フラットシェーディング** (flat shading) や**コンスタントシェーディング** (constant shading) と呼ぶ．

スムーズシェーディングには，各頂点の法線を補間して面上のすべての点で輝度を計算するフォン (Phong) 補間と，各頂点で輝度を計算してそれを補間するグロー (Gouraud) 補間の 2 通りがある．画像の質としては光沢が鮮やかに出る点でフォン補間の方が優れているが，計算速度の観点ではグロー補間の方が優れており，OpenGL の固定シェーダと呼ばれる標準設定では，後者のグロー補間が行なわれる．実際には，頂点ごとに輝度を計算した後に，第 16 章の「色の補間と頂点配列」で説明した色の補間機能によってグロー補間を実現している．なおフォン補間はピクセルシェーダを利用したピクセル単位描画によって実現できる．

17.3 PyOpenGL におけるシェーディング

OpenGL では (17.4) 式の計算をプログラム化する必要はなく，グラフィクスハードウェア内部でシェーディング計算が行なわれる．プログラムでは，光源の位置や性質，反射特性，法線ベクトルの指定を行なえばよい．

GL ライブラリ： OpenGL 命令のインタフェース

`glEnable` 機能の利用可能関数
: 引数で指定した機能を利用可能にする．シェーディングを行なう際は，GL_LIGHTING と指定する．

`glShadeModel` シェーディングモデルの指定関数
: 引数として GL_FLAT，GL_SMOOTH などを与えて，シェーディングモデルを指定する．

glLightfv　光源設定関数
　光源 (照明) の設定を行なう．引数として，光源の番号 GL_LIGHT$i(i = 0, 1, 2, \cdots)$，光源の種類 GL_AMBIENT，GL_DIFFUSE，GL_SPECULAR など，光源の指定値配列とそのオフセットを指定する．複数の光源 GL_LIGHTi は glEnable 関数で光源を指定することにより，実際に点灯される光源をコントロールできる．

glMaterialfv　反射特性設定関数
　反射特性の設定を行なう．引数として，面の表裏 GL_FRONT，GL_BACK，GL_FRONT_AND _BACK，反射特性の種類 GL_AMBIENT，GL_DIFFUSE，GL_SPECULAR，GL_SHININESS など，反射特性の指定値配列とそのオフセットを指定する．

glNormal3dv　法線指定関数
　頂点の法線を指定する．引数として，法線ベクトルの配列とそのオフセットなどを与える．法線ベクトルには，通常，単位ベクトルを与えるが，glEnable 関数で GL_NORMALIZE を指定すれば，内部で自動的に正規化するので単位ベクトルでなくても構わなくなる．

17.4　シェーディングのプログラム

例１： シェーディングによる描画 — myShadeCanvas.py
　15.3 節，例３の MyTranslateCanvas クラスを拡張して，シェーディングを実現するプログラムにしている．実際に物体をシェーディングする場合には，この MyShadeCanvas クラスに，反射率を設定した描画オブジェクトを与えれば良い．光源に関する数値は，すべてコンストラクタの__init__メソッドに記述されている．このうち，光源の色や反射計算に関わる指定は init メソッドで，光源の位置に関する指定は display メソッドで実行されている．

光源の色や反射に関する情報は 2 通り，光源の位置に関する情報も 2 通りが用意されている．これらはキーボードから b’l’ ないし b’p’ が入力されると，対応する self.lid ないし self.pid の値が 1 つズレることによって，切り替えられる．なお，b’l’ や b’p’ となっているのは，PyOpenGL が受け渡す文字列が，bytes 型であって String 型ではないためである．

光源の色や反射については，光源 GL_LIGHT0 と GL_LIGHT1 として，それぞれ，環境成分の RGB 値は (0.1　0.1　0.1) と (0.3　0.1　0.1)，拡散成分の RGB 値は (1.0　1.0　1.0) と (1.0　0.1　0.1)，鏡面成分の RGB 値は (0.9　0.9　0.9) と (0.9　0.2　0.2) となっており，GL_LIGHT0 は概ね白色の光源，GL_LIGHT1 は概ね赤色の光源であり，最初に init メソッドにおいて glLightfv 関数を用いて指定されている．また，実際の光源の切り替えは，glDisable(GL_LIGHTi) 関数と glEnable(GL_LIGHTi) 関数によって実現されている．

光源位置は，頂点情報と同様に，GL_MODELVIEW 行列の影響を受けて移動する．しかしこのプログラムでは，display メソッドにおいて，GL_MODELVIEW 行列に恒等行列が設定される glLoadIdentity 関数 (第 15 章の例３「回転と平行移動のプログラム」を参照せよ) の直後において，glLightfv 関数によって指定されるため，視点座標系での位置として固定されている．結果として，マウス入力に伴って GL_MODELVIEW 行列が変更されても，光源位置は視点に対して固定されており，相対的に動かないことを注意してもらいたい．なお，光源位置には $[5\ \ 5\ \ 0\ \ 1]^{\mathsf{T}}$ と $[5\ \ 5\ \ 0\ \ 0]^{\mathsf{T}}$ とがある．後者は w 成分が 0 になるために無限遠の方向 $[5\ \ 5\ \ 0\ \ 0]^{\mathsf{T}}$ からの平行光線の光源になる．

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

class MyShadeCanvas(MyTranslateCanvas):    # MyShadeCanvas クラスの定義
  def __init__(self):                      # 初期化メソッド
    '''
    陰影描画する OpenGL ウィンドウを初期化する
    '''
    super().__init__()                     # MyTranslateCanvas クラスの初期化メソッド
    self.lid, self.pid = (0, 0)            # 光源番号と光源位置番号の初期化
```

```
    self.light = (GL_LIGHT0, GL_LIGHT1)  # 光源番号（以下，環境/拡散/鏡面成分と対応）
    self.ambient = ((0.1, 0.1, 0.1, 1.0), (0.3, 0.1, 0.1, 1.0))  # 環境成分
    self.diffuse = ((1.0, 1.0, 1.0, 1.0), (1.0, 0.1, 0.1, 1.0))  # 拡散成分
    self.specular = ((0.9, 0.9, 0.9, 1.0), (0.9, 0.2, 0.2, 1.0)) # 鏡面成分
    self.lightPosition = ((5.0, 5.0, 0.0, 1.0), (5.0, 5.0, 0.0, 0.0))
                                        # 光源位置（w 成分による平行光源の実現）

  def init(self, dispObj):              # OpenGL の初期化
    '''
    OpenGL & light を初期化する
    '''
    super().init(dispObj)               # OpenGL の初期化（MyGLCanvas）
    glEnable(GL_LIGHTING)               # 陰影描画の利用
    glEnable(GL_NORMALIZE)              # 法線ベクトルの正規化
    for i in range(len(self.light)):    # 各光源番号の設定
      glLightfv(self.light[i], GL_AMBIENT, self.ambient[i])   # 環境成分の設定
      glLightfv(self.light[i], GL_DIFFUSE, self.diffuse[i])   # 拡散成分の設定
      glLightfv(self.light[i], GL_SPECULAR, self.specular[i]) # 鏡面成分の設定
    glEnable(self.light[self.lid])      # 初期光源番号の利用

  def keyboard(self, key, x, y):        # キーボード入力に伴うコールバックメソッド
    '''
    key    - 入力されたキーの文字（bytes 型）
    x, y   - キーボード入力時のマウスカーソル位置
    キーボード入力に伴う処理を行う
    '''
    if key == b'l':                     # 入力キーが b'l' の場合
      glDisable(self.light[self.lid])   # 現行の光源番号の無効化
      self.lid = (self.lid + 1) % len(self.light) # 光源番号の計算（1 つずらす）
      glEnable(self.light[self.lid])    # 新しい光源番号の利用
    elif key == b'p':                   # 入力キーが b'p' の場合
      self.pid = (self.pid + 1) % len(self.lightPosition)
                                        # 光源位置番号の計算（1 つずらす）
    self.display()                      # 描画（glutPostRedisplay() も可）

  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）
    '''
    glLoadIdentity()                    # 恒等行列で初期化
    glLightfv(self.light[self.lid], GL_POSITION, self.lightPosition[self.pid])
    glMultMatrixd(self.state.setMatrix(self.offset)) # モデル変換行列の設定
    self.coredisplay()                  # 描画の本体

  def loop(self):                       # コールバックメソッドの設定とループ起動
    '''
    keyboard とその他のコールバックメソッドを設定し，ループを起動する
    '''
    glutKeyboardFunc(self.keyboard)     # keyboard コールバックメソッドの登録
    super().loop()                  # その他のコールバックメソッド登録とメインループ起動
```

例２： シェーディング描画多面体 — shadedPolyhedron.py

多面体をシェーディング描画するための抽象クラス ShadedPolyhedron を定義している．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，面の法線ベクトル，反射特性などのデータは未定義となっている．実際にシェーディングされる多面体は，この ShadedPolyhedron クラスを拡張し，頂点座標値や反射特性などの情報を与えて定義される．なお，面の法線ベクトルは面の最初の 3 頂点を用いて，差分ベクトルの外積によって計算している．

```
import numpy as np                      # numpy モジュールの import（np で）
from OpenGL.GL import *                 # GL モジュールの import

class ShadedPolyhedron(object):         # ShadedPolyhedron クラスの定義
  def __init__(self, vertices = (), faces = (),
               diffuse = (), specular = (), shininess = 0):
```

```
                                        # 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
    diffuse   - 拡散成分，省略時 空タプル
    specular  - 鏡面成分，省略時 空タプル
    shininess - 鏡面反射の減衰係数，省略時 0
    陰影多面体を初期化する
    '''
    self.vertices, self.faces, self.diffuse, self.specular, self.shininess = \
                  (vertices, faces, diffuse, specular, shininess)
              # 頂点座標値，面の頂点番号列，稜線の頂点番号列，面の拡散成分/鏡面成分/減衰係数

  def display(self):                    # (多面体の) 陰影描画メソッド
    '''
    多面体の陰影描画を行う
    '''
    self.material()                     # 反射率の設定
    glShadeModel(GL_FLAT)               # フラットシェーディングの利用
    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
      glNormal3dv(normal(self.vertices[self.faces[i][0]],
                         self.vertices[self.faces[i][1]],
                         self.vertices[self.faces[i][2]])) # 法線ベクトルの指定
      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

  def material(self):                   # 反射率の設定メソッド
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, self.diffuse)
                                        # 環境成分と拡散成分の設定
    glMaterialfv(GL_FRONT, GL_SPECULAR, self.specular)  # 鏡面成分の設定
    glMaterialf(GL_FRONT, GL_SHININESS, self.shininess) # 減衰係数の設定

def normal(p0, p1, p2):                 # 法線ベクトルの計算関数
  '''
  p0, p1, p2 - 3 点の座標値
  与えられた 3 点の張る三角形の法線ベクトルを計算する
  '''
  p0, p1, p2 = (np.array(p0), np.array(p1), np.array(p2)) # numpy.array への変換
  return tuple(np.cross(p1 - p0, p2 - p0)) # (p1-p0) と (p2-p0) の外積
```

例３： 立方体のシェーディング描画 — shadedCube.py

例 2 のシェーディング描画多面体を実装した立方体．頂点座標値，面の頂点番号列，面の法線ベクトル，反射特性などのデータに実際の値を埋め込んでいる．反射特性としては，環境成分ならびに拡散成分が (0.8 0.8 0.2)，鏡面成分が (0.9 0.9 0.9)，減衰係数が $n = 100$ と指定されている．このため物体は，光源 GL_LIGHT0 の下では，図 17.3 のように黄色味がかった色となる．一方，光源 GL_LIGHT1 では，赤味がかった色となることが確認できる．なお，鏡面反射光は光源の色をそのまま反射し，かなり狭い範囲で光沢が現れる．

```
from myShadeCanvas import MyShadeCanvas  # myShadeCanvas モジュールの import
from shadedPolyhedron import ShadedPolyhedron # shadedPolyhedron モジュールの import

class ShadedCube(ShadedPolyhedron):     # ShadedCube クラスの定義
  def __init__(self):                   # 初期化メソッド
    '''
    陰影描画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
            (0.8, 0.8, 0.2, 1.0),       # 拡散成分
            (0.9, 0.9, 0.9, 1.0),       # 鏡面成分
```

```
            100)                          # 鏡面反射の減衰係数

def main():                               # main 関数
  canvas = MyShadeCanvas()                # MyShadeCanvas の作成
  dispObj = ShadedCube()                  # ShadedCube オブジェクトの作成
  canvas.init(dispObj)                    # OpenGL 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
```

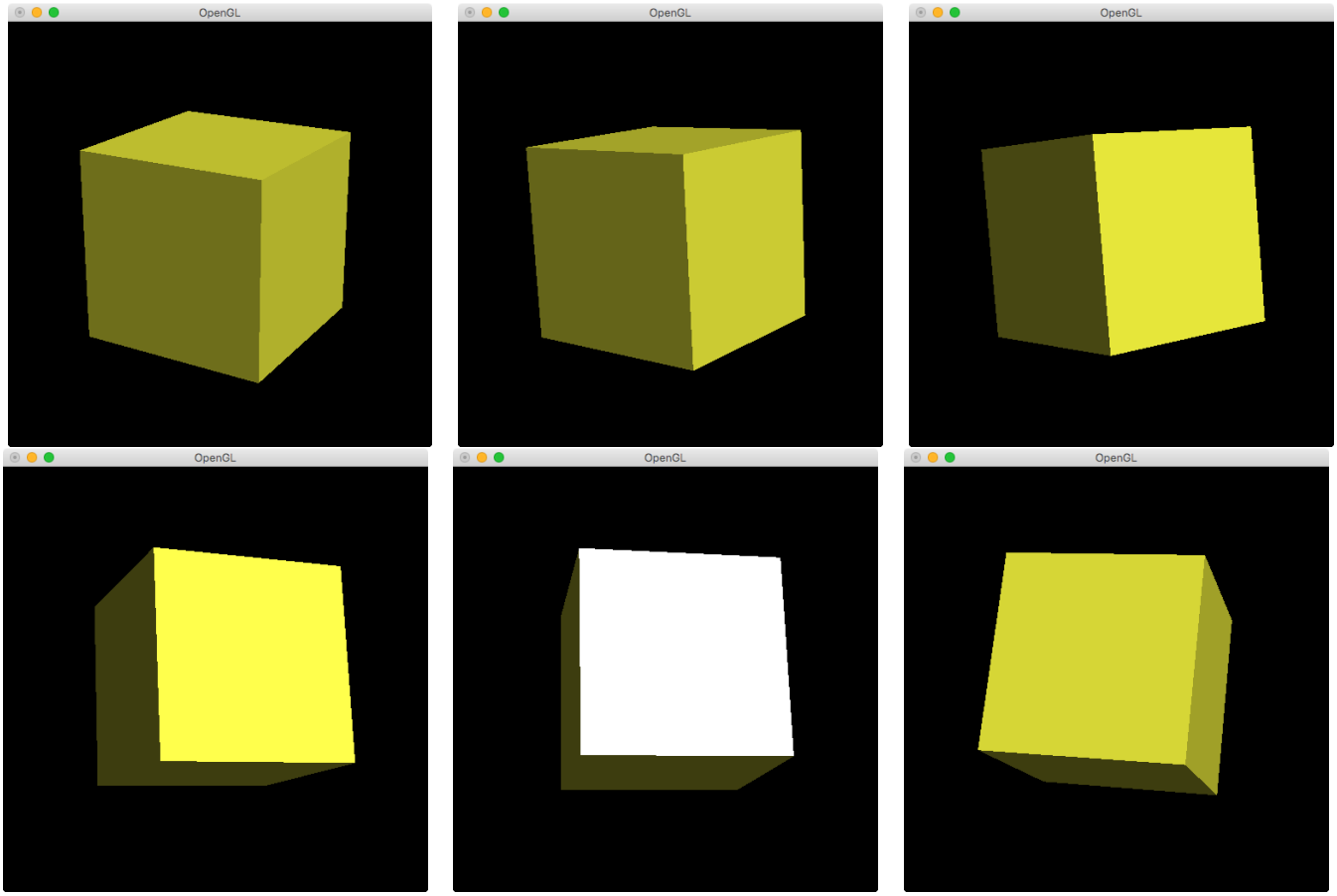

Figure 17.3: 立方体のシェーディング描画

例４： 正二十面体のシェーディング描画 — shadedIcosahedron.py

例２のシェーディング描画多面体を実装した正二十面体．反射特性としては，環境成分ならびに拡散成分が (0.8 0.3 0.8)，鏡面成分が (0.9 0.9 0.9)，減衰係数が $n = 100$ と指定されている．このため，物体はマゼンタ系の色となる．鏡面反射光は光源の色をそのまま反射し，かなり狭い範囲に光沢が現れる．実行結果を図 17.4 に示す．

```
from myShadeCanvas import MyShadeCanvas  # myShadeCanvas モジュールの import
from shadedPolyhedron import ShadedPolyhedron # shadedPolyhedron モジュールの import

class ShadedIcosahedron(ShadedPolyhedron): # ShadedIcosahedron クラスの定義
  def __init__(self):                     # 初期化メソッド
    '''
    陰影描画正 20 面体を初期化する
    '''
    PHI = (1 + 5**0.5) / 2                # 黄金比
    INP = 1 / PHI                         # 黄金比の逆数
```

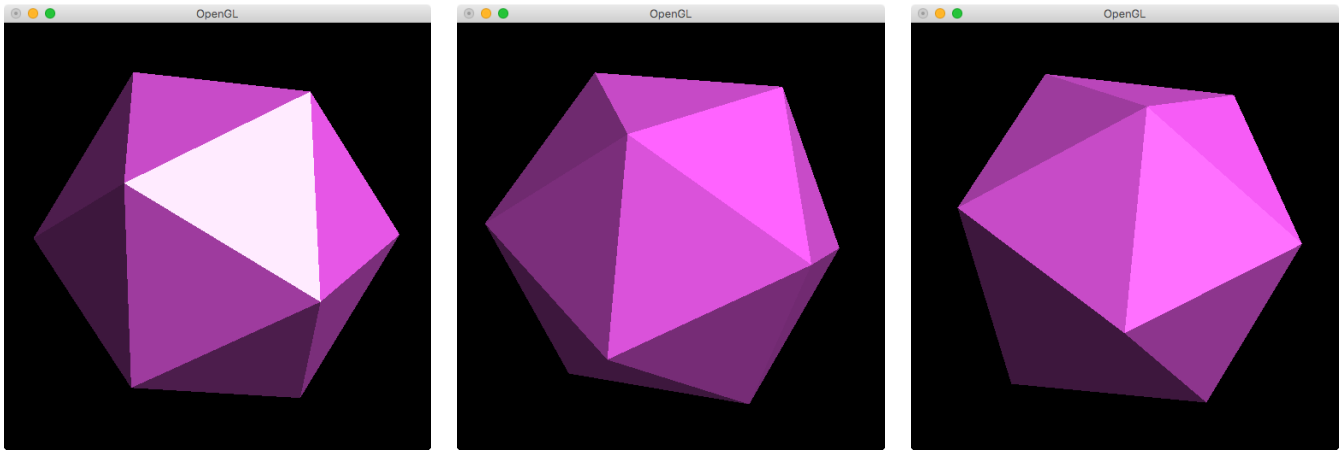

Figure 17.4: 正二十面体のシェーディング描画

```
    self.RCS = (PHI * 5**0.5)**0.5          # 外接球の半径（細分割に利用）
    super().__init__(                       # Polyhedron クラスの初期化メソッド
      ((   0,    1,  PHI), (   0,   -1,  PHI), (   0,   -1, -PHI),
       (   0,    1, -PHI), ( PHI,    0,    1), ( PHI,    0,   -1),
       (-PHI,    0,   -1), (-PHI,    0,    1), (   1,  PHI,    0),
       (  -1,  PHI,    0), (  -1, -PHI,    0), (   1, -PHI,    0)), # 頂点座標値
      ((0,  1,  4), (0,  4,  8), (0,  8,  9), (0,  9,  7), (0,  7,  1),
       (1, 10, 11), (1, 11,  4), (4, 11,  5), (4,  5,  8), (8,  5,  3),
       (8,  3,  9), (9,  3,  6), (9,  6,  7), (7,  6, 10), (7, 10,  1),
       (2, 10,  6), (2,  6,  3), (2,  3,  5), (2,  5, 11), (2, 11, 10)),
                                            # 各面の頂点番号列
      (0.8, 0.3, 0.8, 1.0),                 # 拡散成分
      (0.9, 0.9, 0.9, 1.0),                 # 鏡面成分
      100)                                  # 鏡面反射の減衰係数

def main():                                 # main 関数
  canvas = MyShadeCanvas()                  # MyShadeCanvas の作成
  dispObj = ShadedIcosahedron()             # ShadedIcosahedron オブジェクトの作成
  canvas.init(dispObj)                      # OpenGL 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
```

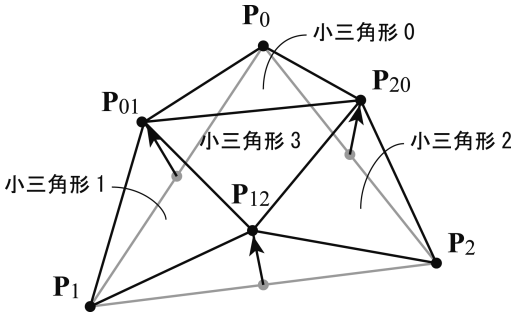

Figure 17.5: 三角形の分割

例５： 多面体近似球面のシェーディング描画 — `shadedSubdivision.py`

例４の「正二十面体のシェーディング描画」から，三角形を再帰的に分割して球面を近似する．`subdivide` メソッドは，引数として三角形の頂点座標と再帰のレベルをとる．再帰レベルが正の値であれば，第 6 章「フラクタル図形」で紹介したシェルピンスキーガスケットと同様の方法で，図 17.5 のように 1 つの三角形を 4 つの小三角形に分割する．このとき，三角形の各辺を 2 等分する頂点を正二十面体の外接球

面にまで持ち上げることで，すべての頂点が球面上に乗るようにしている．再帰レベルが 0 であれば，triangle メソッドによって三角形をフラットシェーディングで描画する．この際，三角形の法線ベクトルは，球の中心である原点からその三角形の重心に向かうベクトルで代用されている．

実行結果を図 17.6 に示す．分割の再帰レベルは実行時のシェル引数ないしキーボード入力によって指定できる．分割回数を増やすことで形は球面に近づいていくが，シュブルール錯視の効果 (隣接した色の差分を強調する効果) によって，相当に細かい面を用いても凹凸が見えてしまっている．

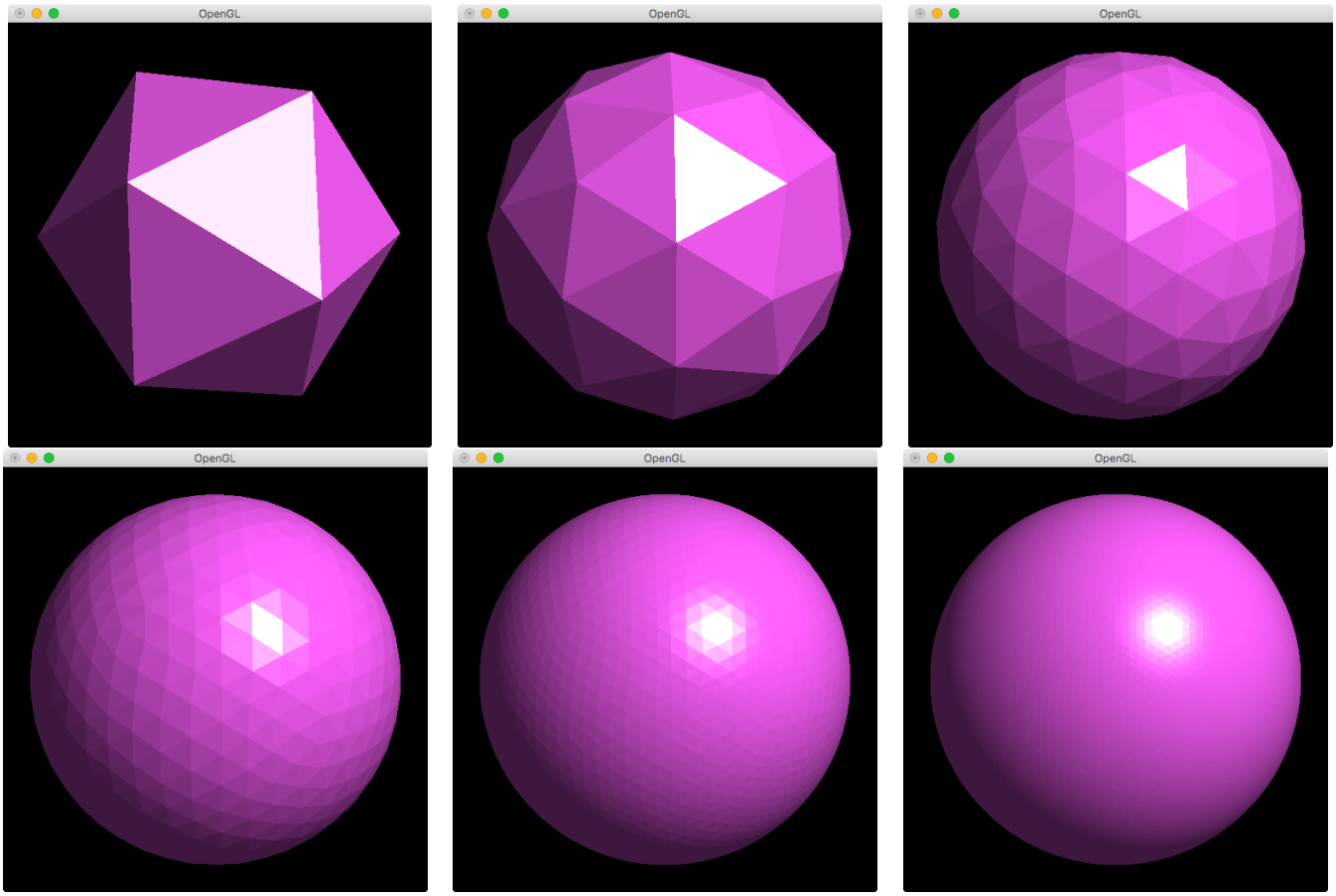

Figure 17.6: 近似球面のシェーディング描画．実行時に与えた引数は，左上から右の方向に，それぞれ「0」,「1」,「2」,「3」,「4」,「5」である．1 回の分割ごとに面の数は 4 倍になるので，それぞれ 20 面体，80 面体，320 面体，1280 面体，5120 面体，20480 面体になっている

```
import sys                                  # sys モジュールの import
import numpy as np                          # numpy モジュールの import (np で)
from OpenGL.GL import *                     # GL モジュールの import
from myShadeCanvas import MyShadeCanvas     # myShadeCanvas モジュールの import
from shadedIcosahedron import ShadedIcosahedron
                                            # shadedIcosahedron モジュールの import

class ShadedSubdivision(ShadedIcosahedron): # ShadedSubdivision クラスの定義
  def __init__(self, times):               # 初期化メソッド
    '''
    times - 再帰回数
    細分割正 20 面体を初期化する
    '''
    super().__init__()                      # ShadedIcosahedron クラスの初期化メソッド
    self.times = times                      # 再帰回数の設定

  def display(self):                        # (細分割正 20 面体の) 陰影描画メソッド
    '''
```

```
    細分割正 20 面体の陰影描画を行う
    '''
    self.material()                         # 反射率の設定
    for i in range(len(self.faces)):        # 多角形描画の反復 (面番号 i)
      self.subdivide(np.array(self.vertices[self.faces[i][0]]),
                     np.array(self.vertices[self.faces[i][1]]),
                     np.array(self.vertices[self.faces[i][2]]),
                     self.times)            # 3 頂点の numpy.array へ変換し，細分割描画

  def subdivide(self, p0, p1, p2, l):       # 細分割描画メソッド
    '''
    p0, p1, p2 - 三角形の 3 頂点
    l          - 再帰のレベル
    再帰レベルに応じて三角形を細分割ないし陰影描画する
    '''
    if l > 0:                               # 再帰レベルが 0 より大きい場合
      p01 = self.split(p0, p1)              # 稜線 p0, p1 を分割
      p12 = self.split(p1, p2)              # 稜線 p1, p2 を分割
      p20 = self.split(p2, p0)              # 稜線 p2, p0 を分割
      self.subdivide(p0, p01, p20, l-1)     # 三角形 p0, p01, p20 の細分割描画
      self.subdivide(p1, p12, p01, l-1)     # 三角形 p1, p12, p01 の細分割描画
      self.subdivide(p2, p20, p12, l-1)     # 三角形 p2, p20, p12 の細分割描画
      self.subdivide(p01, p12, p20, l-1)    # 三角形 p01, p12, p20 の細分割描画
    else:                                   # 再帰レベルが 0 の場合
      self.triangle(p0, p1, p2)             # 三角形の陰影描画

  def split(self, p0, p1):                  # 稜線の分割メソッド
    '''
    p0, p1 - 稜線の 2 頂点
    稜線を分割して，外接球面上に移動する
    '''
    mid = p0 + p1                           # 稜線の中点方向のベクトル
    return mid * self.RCS / (mid@mid)**0.5 # 外接球面上に移動する

  def triangle(self, p0, p1, p2):           # 三角形の陰影描画メソッド
    '''
    p0, p1, p2 - 三角形の 3 頂点
    三角形をフラットシェーディングで陰影描画する (スムーズシェーディングでオーバライドされる)
    '''
    glShadeModel(GL_FLAT)                   # フラットシェーディングの利用
    glBegin(GL_POLYGON)                     # 多角形描画の開始
    glNormal3dv(tuple(p0 + p1 + p2))        # 法線ベクトル (簡易計算=重心方向) の指定
    glVertex3dv(tuple(p0))                  # 頂点座標値 p0 の指定
    glVertex3dv(tuple(p1))                  # 頂点座標値 p1 の指定
    glVertex3dv(tuple(p2))                  # 頂点座標値 p2 の指定
    glEnd()                                 # 多角形描画の終了

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 -> ').split(' ')    # 再帰回数
  return int(args[0])                       # 再帰回数を返す

def main():                                 # main 関数
  canvas = MyShadeCanvas()                  # MyShadeCanvas の作成
  dispObj = ShadedSubdivision(getArgs())    # ShadedSubdivision オブジェクトの作成
  canvas.init(dispObj)                      # OpenGL 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
```

例６： 近似球面のスムーズシェーディング描画 — shadedSphere.py

例５の「多面体近似球面のシェーディング描画」に，三角形の描画にスムーズシェーディングを利用することで，近似球面を描画するプログラムである．triangle メソッドで三角形を描画する際に，スムーズシェーディングを指定し，各頂点の座標と法線ベクトルを与えている．その際，原点を中心とする球面上の各頂点の法線ベクトルとして，当該の頂点座標をそのまま利用している．実行結果を図 17.7 に示す．図からもわかるように，比較的少ない分割回数でも十分に滑らかな球に見える．

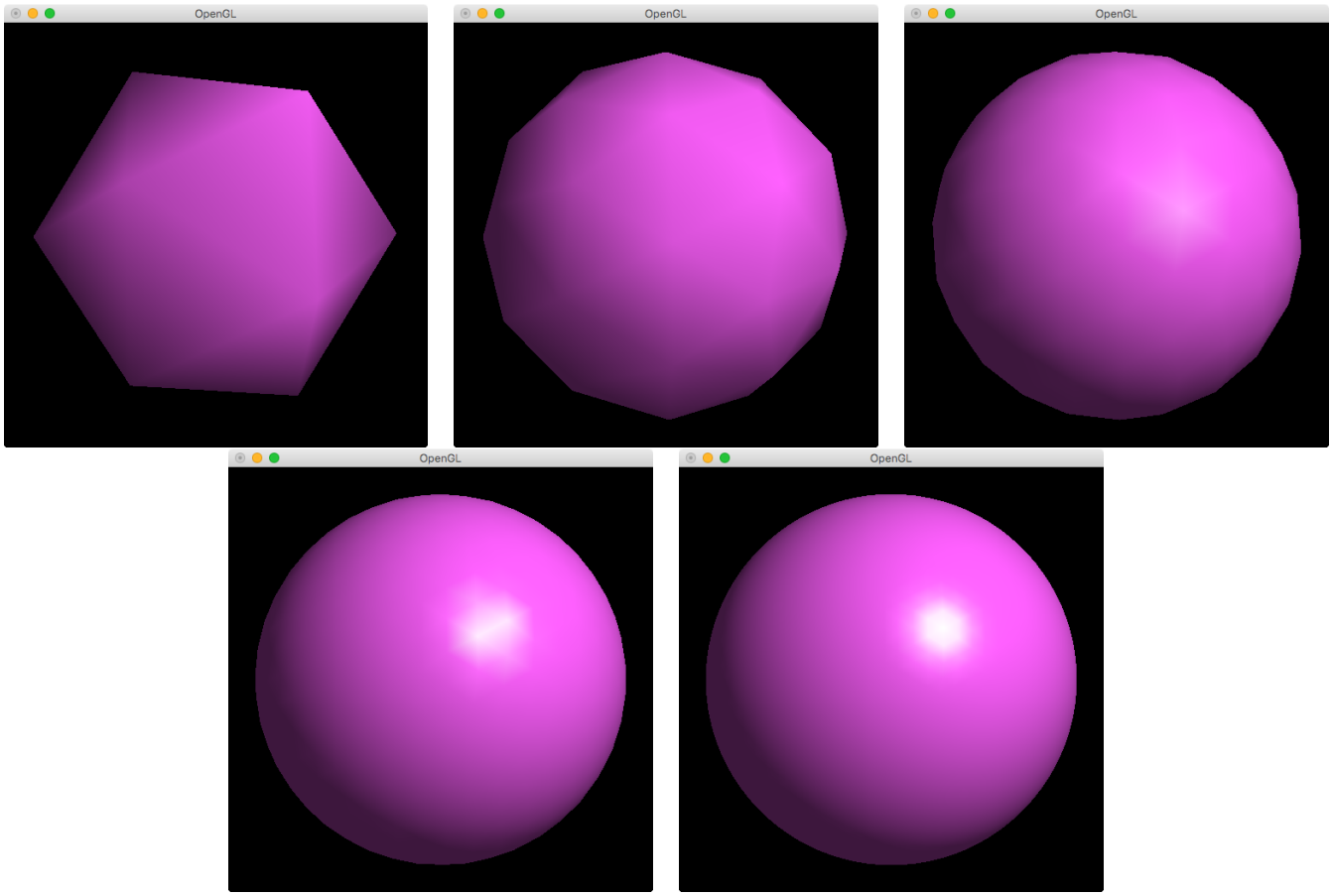

Figure 17.7: 近似球面のスムーズシェーディング描画．実行時に与えた引数は，左上から右の方向に，それぞれ「0」,「1」,「2」,「3」,「4」である．スムーズシェーディングによって面が滑らかに感じられる

```
from OpenGL.GL import *                    # GL モジュールの import
from myShadeCanvas import MyShadeCanvas    # myShadeCanvas モジュールの import
from shadedSubdivision import ShadedSubdivision, getArgs
                                           # shadedSubdivision モジュールの import

class ShadedSphere(ShadedSubdivision):     # ShadedSphere クラスの定義
  def __init__(self, times):               # 初期化メソッド
    '''
    球面を初期化する
    '''
    super().__init__(times)                # ShadedSubdivision クラスの初期化メソッド

  def triangle(self, p0, p1, p2):          # 三角形の陰影描画メソッド
    '''
    p0, p1, p2 - 三角形の 3 頂点
    三角形をスムーズシェーディングで陰影描画する
    '''
    glShadeModel(GL_SMOOTH)                # スムーズシェーディングの利用
    glBegin(GL_POLYGON)                    # 多角形描画の開始
    glNormal3dv(tuple(p0))                 # 法線ベクトル (=点 p0) の指定
```

```
        glVertex3dv(tuple(p0))                  # 頂点座標値 p0 の指定
        glNormal3dv(tuple(p1))                  # 法線ベクトル (=点 p1) の指定
        glVertex3dv(tuple(p1))                  # 頂点座標値 p1 の指定
        glNormal3dv(tuple(p2))                  # 法線ベクトル (=点 p2) の指定
        glVertex3dv(tuple(p2))                  # 頂点座標値 p2 の指定
        glEnd()                                 # 多角形描画の終了

    def main():                                 # main 関数
      canvas = MyShadeCanvas()                  # MyShadeCanvas の作成
      dispObj = ShadedSphere(getArgs())         # ShadedSphere オブジェクトの作成
      canvas.init(dispObj)                      # OpenGL 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

フラクタル立体のシェーディング描画

基本的な立体がシェーディングできれば，フラクタル立体もシェーディングできる．たとえば，第 12 章の例５「メンガースポンジ」で，primitive を Cube から例３「立方体のシェーディング描画」の ShadedCube に置き換えることで，図 17.8 左のようにメンガースポンジをシェーディングできる．正二十面体を基本立体とする正二十面体片なども，同様にシェーディングできる（図 17.8 右）．なお，シェーディング描画だけでは，他の面によって遮られて光の当たらない影（shadow）を扱えないため，穴の奥にも光が当たっている．

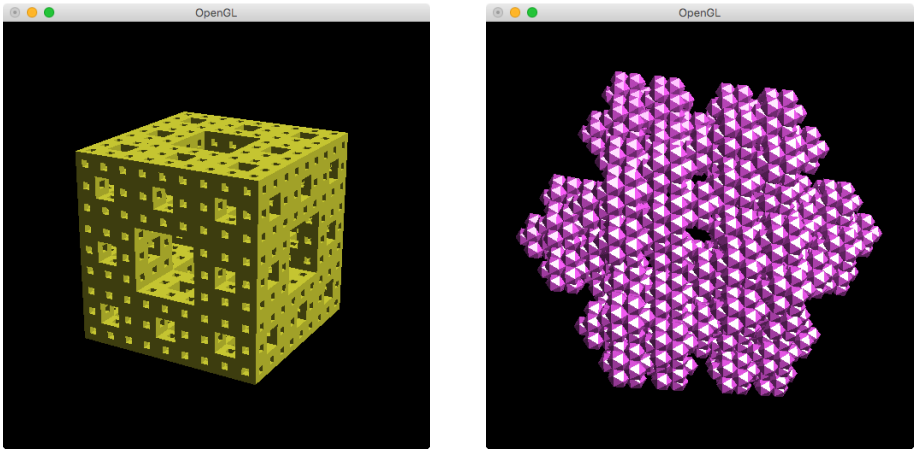

Figure 17.8: フラクタル立体のシェーディング．左はメンガースポンジのシェーディング結果（再帰は 3 回），右は正二十面体片のシェーディング結果 (再帰は 3 回)

円柱と円錐のシェーディング描画

円柱と円錐を作成し，シェーディング描画するプログラムを作成せよ．ただし，円柱ならびに円錐の側面 (円柱面と円錐面の部分) は，図 17.9 のようにスムーズシェーディングをかけること．なお，多角柱や多角錐の描画にあたっては，例６の「近似球面のスムーズシェーディング描画」のように再帰的なプログラムにする必要はない．

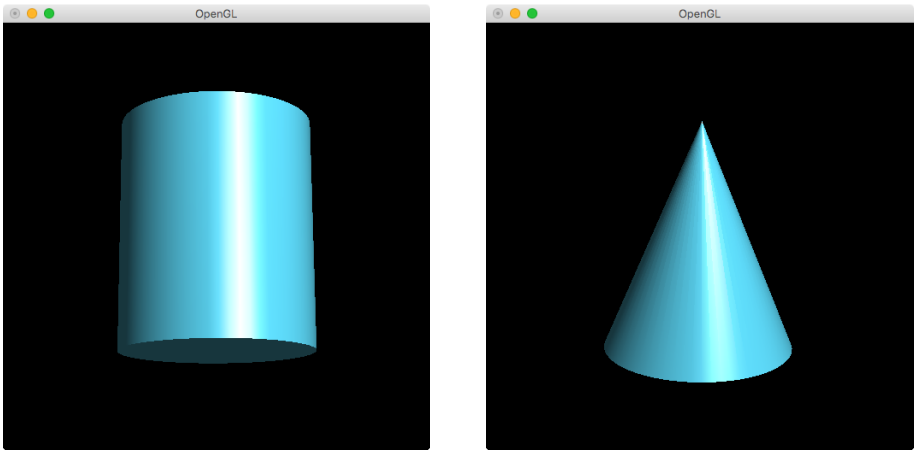

Figure 17.9: 円柱と円錐のシェーディング