学生実験４：コンピュータグラフィックス
— 投影変換とシェーディング —

担当：亀田能成, 角所考（TA:山肩洋子）

2002 年度版 (Ver 1.02)

前書き

本書は、1997 年から 2002 年に渡って、亀田が担当した京都大学工学部情報学科計算機科学コース３年生後期配当の「学生実験４：コンピュータグラフィックス」で用いられたテキストの最終年度版である。

実験の選択肢にコンピュータグラフィックスを導入したのは、1997 年当時、家庭用ゲーム機が普及していた[1]のに、その技術的背景を説明するような授業が当時なかったからである。予備調査してみると、世間でいうコンピュータグラフィックスの演習では、線画／２次元グラフィックス／レイトレーシングなどが主流で、ゲーム機が採用したような高速化を前提とする（その代わりレイトレーシングのような美しさの追求が犠牲となっている）アルゴリズムの説明や演習はほとんどなく、いい教科書も見当たらなかった。そこで、本稿を書き上げ、それ以後５年間（2001 年度だけは訪米のため椋木先生にバトンタッチ）に渡って実験を担当した。

もう私が本実験を担当することはない。このまま本稿が消えゆくのも惜しい[2]ので、私の手を離れる前の最終版として、本稿を上梓する。

2004 年 8 月, 筑波にて
亀田能成 kameda@ieee.org

1　はじめに

本演習では、三次元データを二次元画像で表現する際の基本技術について学習し、それをプログラム演習によって確認する。ここ数年の技術の発展により、三次元物体をパソコン・ゲーム機などで簡単に表示できるようになった。そこで用いられている技法は古典的なレイトレーシング法とは異なり、高速化・ハードウェア化に適した技法が採られている。本演習ではそうした手法を学び、実際に計算機上で三次元物体の可視化を行うプログラムを製作する。

[1]例えば初代 PlayStation^{TM} は 1994 年 12 月、PlayStation2^{TM} は 2000 年 3 月。
[2]実はまだ京都大学では私の後任が本実験を続行しているが、それもあと何年のことか‥

別の言い方をすれば、これは現在の VRML ブラウザのレンダリング部分の技術を学び、ソフトウェアを製作する演習とも言える。
計算機上の三次元データを二次元画像に投影変換する方法を実現するため、以下の要素を実習で扱う。

- 撮像方法 (カメラモデル)
- 観測物体 (形状、表面の反射率、位置)
- 光源 (色、輝度、位置)
- 環境モデル (大気モデルなど)

このうち、一番目の撮像方法は幾何問題に帰着される。これについては 3 章で述べる。また、続く二項目は物理現象に関連が深く、これらを扱う処理を一般にシェーディングという。これについては 8 章で扱う。最後の項については本演習では特には取り上げない。本演習では、上記のうち、特に２、３番目を意識する。

なお、コース履修に当たって、課題などの指示は WWW に掲示する。また、更新・訂正情報や本テキストの全文も掲載する。WWW 上と本テキストで相違がある場合は、WWW 上の文書の内容を有効とする。

URL： http://www.kameda-lab.org/mirror/www.mm.media.kyoto-u.ac.jp/members/kameda/lecture/le4cg/index-j.html

2 準備

2.1 画像のデータ構造と表示

計算機内にある三次元物体を可視化するためには、計算機上のウィンドウシステムを通じて画像表示を行うことになる。
計算機で扱う画像は、画素が二次元配列で並べられたデータ構造で表すことができる。画素は物体がどのような色や明るさで見えるかを規定するが、計算機上ではこれを光の三原色である RGB の光の強さの合成で全て表す。人間が色を理解する方法は完全に解明されているわけではないが、現在においてはこの方法がもっとも妥当とされている。
本演習でも RGB それぞれを 256 段階の輝度で表す方法を採用する。これで 2^{24} 通りの色（いわゆる１６７０万色）を表現できることになる。
正確には、モニターの色温度やブライトネス・コントラスト等の調整によって表示される色は異なる。
色が重要な要素をもつ業界ではモニターの色調整から行うが、本演習ではそこまでは考慮せず、画像の生成までを対象とする。

余談その１

「2^{24} 通りの色を同時発色可能」という謳い文句が昔のパソコンの広告には見かけられた。さて、2^{24} 通りの色を**同時に観察する**ためには、当然 2^{24} 画素が必要である。簡単のために正方形のモニターを考えたとして、縦横何画素のモニターを用意すればいいのだろう？

余談その２

現在ある計算機の様々なウィンドウシステムが、すべて１６７０万色で表示しているわけではない。旧式のシステムでは、１画素あたり RGB 合わせても 8bits ないし 16bits しか割り当てていない場合があり、この状況下で１６７０万色データ形式の画像を視ようとすると、減色処理がシステムにより勝手に行われることに注意しておこう。最近でも、液晶ディスプレイは、表示色分解能としては１６７０万色も出色できないことが多い。さて、それでは、君の目はどれぐらいの色分解能があるだろうか。それを確かめる実験をするためには、どのような環境を用意し、どのようなプログラムを書けばよいだろうか。

2.2 画像フォーマット

コンピュータで扱う画像形式については現在様々な形式が提案されている。本演習ではこのうち、最も画像フォーマットが簡単な pbmplus の ppm 形式を利用する。この形式は、RGB それぞれに 8bits をとった画素あたり 24bits のカラー画像を表現できる。

ppm 形式については、man ページを参照すること。ファイルの拡張子は一般に ppm とするのが普通である。ppm 形式のファイルは、xv などで見ることができる。

```
ppm(5)                                                    ppm(5)

NAME
       ppm - portable pixmap file format

DESCRIPTION
       The  portable pixmap format is a lowest common denominator
       color image file format.  The definition is as follows:

       - A "magic number" for identifying the file type.   A  ppm
         file's magic number is the two characters "P3".

       - Whitespace (blanks, TABs, CRs, LFs).

       - A width, formatted as ASCII characters in decimal.

       - Whitespace.
```

- A height, again in ASCII decimal.

- Whitespace.

- The maximum color-component value, again in ASCII decimal.

- Whitespace.

- Width * height pixels, each three ASCII decimal values between 0 and the specified maximum value, starting at the top-left corner of the pixmap, proceeding in normal English reading order. The three values for each pixel represent red, green, and blue, respectively; a value of 0 means that color is off, and the maximum value means that color is maxxed out.

- Characters from a "#" to the next end-of-line are ignored (comments).

- No line should be longer than 70 characters.

Here is an example of a small pixmap in this format:

```
P3
# feep.ppm
4 4
15
 0  0  0    0  0  0    0  0  0   15  0 15
 0  0  0    0 15  7    0  0  0    0  0  0
 0  0  0    0  0  0    0 15  7    0  0  0
15  0 15    0  0  0    0  0  0    0  0  0
```

Programs that read this format should be as lenient as possible, accepting anything that looks remotely like a pixmap.

There is also a variant on the format, available by setting the RAWBITS option at compile time. This variant is different in the following ways:

- The "magic number" is "P6" instead of "P3".

```
- The  pixel  values are stored as plain bytes, instead of
  ASCII decimal.

- Whitespace is not allowed in the pixels area, and only a
  single  character of whitespace (typically a newline) is
  allowed after the maxval.

- The files are smaller and many times faster to read  and
  write.

Note  that  this  raw  format can only be used for maxvals
less than or equal to 255.  If you use the ppm library and
try to write a file with a larger maxval, it will automat-
ically fall back on the slower but more general plain for-
mat.
```

一般に ppm 形式では表示時にデータの初めのほうがモニター表示時に上に来て、後のほうが下に来るのが普通である。(もっとも、これは正確には画像表示ツールに依存する仕様であり、ppm 形式にこれを規定する文章はない。)

この場合、画像を X 座標、Y 座標で表現すると、Y 軸は下向きと解釈できる。ところが、後述の世界座標系では、上向きが Y 軸方向にすることが普通であり、関係が逆になることに注意する。

2.3 三次元物体の表現

計算機上で三次元世界を表現する以上、三次元物体も計算機上のデータ構造で表現しなければならない。三次元物体の表現法は、大きく分けてソリッドモデルとサーフェスモデルがある。ソリッドモデルは物体を三次元で表す。例えば、単位球を表現する場合、ソリッドモデルではこれを $x^2 + y^2 + z^2 \leq 1$ で表現する。ソリッドモデルを表現するには関数モデルや離散モデル(ボクセルという単位立方体の集合で表す方法など)などがあり、一部の CAD や数値計算シミュレーション、科学技術計算などでよく用いられる。しかし、CG に限った場合、物体内部の属性は不要である。そこで、その表面についてのデータだけを考慮するのがサーフェスモデルである。

サーフェスモデルの場合、その幾何情報を表す形態には関数モデルと離散モデルがある。関数モデルはパラメトリック曲面を構成するもので、ベジェ曲面や B-Spline 面、さらにその発展形である NURBS や、メタボール、超二次関数、さらにはフラクタクル関数に代表されるようなファンクショナルモデリングもこの範疇に属する。これらはその特性をよく理解すれば数少ないパラメータで曲面を制御でき便利であるが、その一方で各々の関数形式を越えた曲面を表現することは困難である。一方、離散モデルは一般にポリゴンモデリング、パッチモデリングと呼ばれている手法で、ある実際の曲面を微小な多角形の面の集合で表現するものである。理論上は面を微細にしていけばどのような面でも十分な精度で

近似できるという利点を持つが、他方でそのパラメータ数は膨大（原則的に頂点の数 $\times 3$ の変数）になるという欠点をもつ。現在の計算機環境では汎用性に富むポリゴンモデルが支持されていることが多い。これは、優秀なレンダリングアルゴリズムとハードウェアの出現によるところが大きいと思われる。

本演習では、三次元物体の形状を表すために、オープンな規格の一つである VRML ver 1.0 の構文の一部を利用する。

本演習で最低限用いるフォーマットは以下のみである。

- Material
  - ambientColor：環境光反射係数
  - diffuseColor：拡散反射係数
  - specularColor：鏡面反射係数
  - shininess：鏡面反射強度
- Coordinate3
- IndexedFaceSet

VRML のファイルはテキストファイルであり、通常拡張子が wrl である。次にサンプルを示す。

```
#VRML V1.0 ascii

Material{
diffuseColor    1.0 1.0 0.0
}

Coordinate3{
  point [
    0.0     1.0     2.0,
    2.0     0.0     3.0,
    2.0     2.0     2.0,
    1.5     6.0     0.5,
    1.0     0.0     0.0
  ]
}
IndexedFaceSet{
  coordIndex[
    1, 2, 4, -1,
    0, 2, 3, -1,
    0, 4, 2, -1
  ]
}
```

Coordinate3 ノード内の point フィールドのカンマで区切られたセットが X,Y,Z 座標を示し一つの頂点の座標を表現する。頂点番号は、一つ目から順に番号 $0, 1, 2, \cdots$ となる。IndexFaceSet ノード内の coordIndex フィールド内の各インデックスは-1 で区切られ、それぞれ Coordinate3 の頂点の番号を組み合わせて一つの面を表現している。本例の一行目は、頂点 1 と頂点 2 と頂点 4 とで構成される三角形の面を表す。

本演習では、この頂点の並びによって面の裏表の区別をつける。すなわち、頂点の並びが反時計回りに見える側が表（法線方向）であるとする（これは VRML 1.0 の規定ではない）。VRML 1.0 ではポリゴンが多角形でも構わないが、問題を簡単にするため、本演習ではポリゴンは全て三角形であるとする。

Material ノードについては各フィールドの意味はシェーディング処理の項で詳しく述べるが、各フィールド（上記の例では diffuseColor）に続いて $0.0 \leq 1.0$ に正規化された (R,G,B) の値で色を表現する。ただし、shininess だけは 1 要素のみが後に続く。

余談

VRML 1.0 では 3 点以上のポリゴンを認めるが、この場合、頑健な実装を行おうとすると実装側には大変な負担が強いられる。それは、それらが同一平面上にあることが保証されなくからである。さて、同一平面にないとわかったときにそれを分割して三角形ポリゴンの集合で表現しなおそうとしてみよう。いったいどんなアルゴリズムなら、考えられるあらゆる不健全なデータに対応可能であろうか。

3　投影変換

私たちが三次元世界を視るとき、その情報は目の水晶体を通し網膜上に結ばれた像から得ている。基本的にはカメラの撮像原理も同様である。このような撮像系として代表的なものに、並行投影と透視投影がある。

まず、三次元世界全体を考えよう。三次元世界を表現するために、三軸の直交座標系を採用する。その軸は右手系の順序で X,Y,Z とする。そして、その世界の中には物体が存在する。（光源も存在するが、ここでは幾何的な問題に集中するために考えない。これについては 8 章で述べる。）

三次元世界の画像を生成するとは、この三次元世界を二次元平面に投影変換する問題に他ならない。ただし、得る画像の大きさは有限なので、投影する際に空間のどの部分を投影するかを意識する必要がある。以後、この投影される平面のことを画像平面、画像平面上で実際に画像として可視化する領域を撮像領域と呼ぶ。

また、これ以降は座標系は同次座標系[3]で表現する。同次座標系は空間を表す三要素のベクトルに一要素を加えたもので、空間の射影問題を解くときに都合が良い。三次元ユークリッド空間上の一点 $(x_i, y_i, z_i)^T$ は同次座標系では $(x_i, y_i, z_i, 1.0)^T$ で表現できる。また、同次座標 $(x_a, y_a, z_a, \gamma)^T$ は各要素を γ で割れば、その三次元ユークリッド空間上の点の位

[3]斉次座標系とも呼ばれる。本演習ではカメラないし対象物体の運動を考えないので、同次座標系を導入することの利点は少ない。この座標系に従えば、移動と回転の複合を一つの行列にまとめられたり、透視投影変換を一つの行列で表現することなどが可能になる。

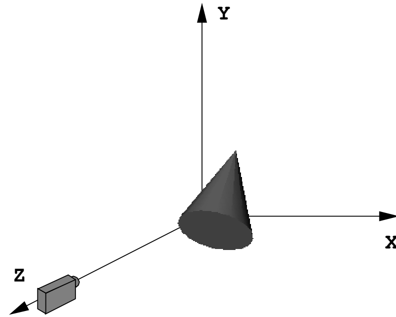

図 1: 三次元世界中の物体とカメラ

置となる。

3.1 並行投影

並行投影（直交投影、正射影とも呼ばれる）とは、空間内にあるベクトルを考え、それに平行に物体上の一点を画像平面 I 上の一点へ投影する方法である。これについては、図2で理解したほうがはやいだろう。ここでは簡単のため、視線方向は Z 軸負の方向に向いているものとし、画像平面は $z = z_I$ で規定される平面とする。

このとき、空間内の一点 $P = (x_p, y_p, z_p, 1)^T$ は、画像平面 I 上の一点 $P' = (x'_p, y'_p, z_I, 1)^T$ に投影される。これは、

$$\begin{pmatrix} x'_p \\ y'_p \\ z_I \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & z_I \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} \tag{1}$$

で表現される。ただし、点 P を画像平面 I にプロットする際には z_I に意味はない。また、図2において $z_n \leq z \leq z_f$ を満たす z 値を持つ物体のみが投影対象となり、画像上に表される。また、画像平面上の撮像領域のみが画像として可視化される。

この方法は非常にアルゴリズムが簡単であり計算量も少ないという利点があるが、古代の日本画のように、奥にある物体と手前にある物体とが同じ大きさになるという致命的な欠点がある。このため、実際の CG に用いられるよりは CAD などの見取り図を描くときのように寸法の正確さが求められる場合に利用されることが多い。

3.2 透視投影

透視投影（遠近投影とも呼ばれる）を考えるとき、一番分かりやすいのがピンホールカメラを思い浮かべることである（図3を参照)。簡単のため、いまカメラの焦点が点 $(0, 0, 0)$ に

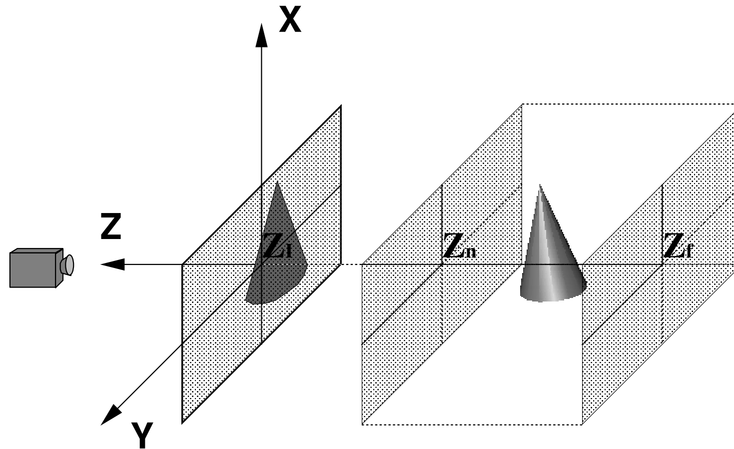

図 2: 平行投影

あってその方向は $(0,0,-1)$ を向いているものとする。また、画像平面 I が $z = z_I (z_I < 0)$ に設定されるものとする。実際のピンホールカメラでは画像平面は $z = -z_I$ になるが、計算理論上は焦点のどちら側に画像平面 I があっても変わりはない。

このとき、空間内の一点 $P = (x_p, y_p, z_p, 1)^T$ は、画像平面上の一点 $P' = (x'_p, y'_p, z_I, 1)^T$ に投影されるのだが、この P' を求めるために、まず

$$P'' = \begin{pmatrix} x_p \\ y_p \\ z_p \\ \frac{z_p}{z_I} \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{z_I} & 0 \end{bmatrix} \begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} \tag{2}$$

という行列で点 P'' を求める。P'' は P' と同一のユークリッド座標に射影される点である。本式の結果の P'' は同時座標表現なので、画像平面 I 上の座標を得るために第四要素 $\frac{z_p}{z_I}$ で P'' の全項を割ってみると、次式のように (x'_p, y'_p, z_I) が求められる。

$$\frac{1}{\frac{z_p}{z_I}} P'' = \begin{pmatrix} x_p \\ y_p \\ z_p \\ \frac{z_p}{z_I} \end{pmatrix} = \begin{pmatrix} \frac{z_I}{z_p} x_p \\ \frac{z_I}{z_p} y_p \\ z_I \\ 1 \end{pmatrix} = \begin{pmatrix} x'_p \\ y'_p \\ z_I \\ 1 \end{pmatrix} = P' \tag{3}$$

画像平面上には撮像領域が設定されるので、透視投影では画像に投影される範囲は図 3 に示されるような四角錐の内部だけとなる。$|z_I|$ の距離は焦点距離とも呼ばれる。撮像領域の大きさを一定とした場合、焦点距離を短くとればパースペクティブはきつくなり、遠近法が強調された画像となる。

余談

物体が遠くにある場合、その像が撮像領域中で大きくなるようにするためには、三つの方法がある。一つはカメラを物体に近づける方法であり、一つは焦点距離を大きくする方

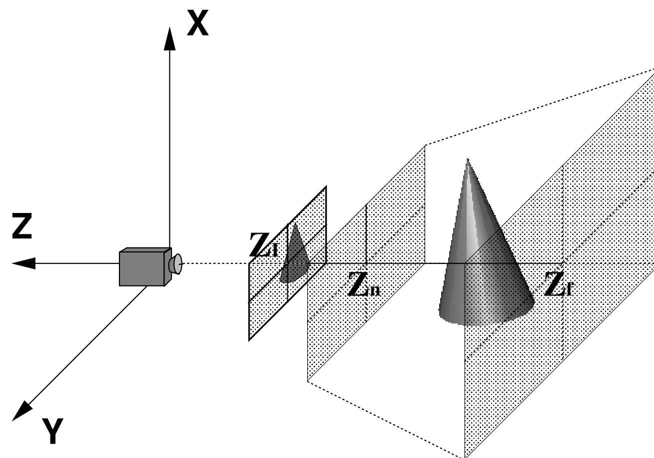

図 3: 透視投影

法である。さて、もう一つの方法はどのようなものが考えられるだろうか。ちなみにこの方法は技術ヲタクの間では非常に嫌われるアプローチである。その理由も考えてみよう。

4　三次元の移動・回転・拡大縮小

前節では、カメラ焦点位置をそのカメラ座標系の原点、カメラ方向を (0,0,-1) としたが、一般にはカメラ焦点位置・カメラ方向は世界座標系の上に設定され、ＣＧモデルも世界座標系の上に設定される。前節の投影変換を可能にするためには、世界座標系に設定されたＣＧモデルをカメラ座標系で表現しなくてはならない。

一般に、三次元座標系の移動、回転、拡大縮小は同時座標表現を用いると正方行列の演算のみで可能になる。これらの行列表現は全て逆行列を持つことに注意しよう。つまり、全ての移動・回転・拡大縮小はどのような組み合わせもあらかじめ唯一つの行列として表現しておくことが可能であり、その逆演算はその逆行列のみを求めればよいことになる。

下記で T は (t_x, t_y, t_z) の移動を表現し、R_x, R_y, R_z は X,Y,Z 軸回りの $\theta_x, \theta_y, \theta_z$ 回転、S は X,Y,Z 軸方向への (s_x, s_y, s_z) 倍の拡大縮小を表す[4]。

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) & 0 \\ 0 & \sin(\theta_x) & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

[4]移動の正負、回転の正負に注意すること

$$R_y = \begin{bmatrix} \cos(\theta_x) & 0 & \sin(\theta_x) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_x) & 0 & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

$$R_z = \begin{bmatrix} \cos(\theta_x) & -\sin(\theta_x) & 0 & 0 \\ \sin(\theta_x) & \cos(\theta_x) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7}$$

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8}$$

5　処理ポリゴン数の削減

一般に、三次元モデルのポリゴン数は膨大であり、全てのポリゴンを処理するには莫大な計算量が必要となる。そこで、レンダリングに必要がないポリゴンを処理対象からレンダリング処理の早い段階で効率よく外すことが重要になる。

三次元モデルのデータの中には、平行投影／透視投影時に撮影されないポリゴンが含まれる。画像平面上の撮像領域から完全に逸脱した場所に投影されるポリゴンはレンダリングする必要がないので、処理の対象から削除する。

また通常は、物体の表側のみしか可視化する必要がない。もちろん、半透明な物体がある場合は別であるが、本演習では扱わない。そこで、面が視点に表を向けているかどうかの判定をして、表を向いていない面を処理の対象から削除する。この判定は、そのポリゴンの法線 $\vec{n_p}$ とそのポリゴンへの視線ベクトル $\vec{e_p}$ との成す角度が $\pi/2$ 未満であるかどうかを調べる。二つの成す角度の値が必要というわけではないので、内積を使うと、判定だけれあれば次のように少ない計算コストで行える。

$$\vec{n_p} \cdot \vec{e_p} \geq 0 \tag{9}$$

この方法は “cull face” または “backfacing” といい、3D ポリゴンブラウザではこれを切り替えられるようになっている場合もある。

余談

もちろん、一つの面について裏から見ても面が見えるようにするほうがよい場合も考えられる。何らかの状況下で、物体内部にカメラが侵入していってしまった場合、その侵入された物体のポリゴンはカメラに対して裏面を見せてしまうことになり、backfacing 下ではそれらの面はいきなり消えてしまうように見える。裏面でも表示する計算コストと、カメラが物体内部に侵入しない（ないし少なくとも侵入したことを判定する）計算コストとの兼ね合いが重要になる。

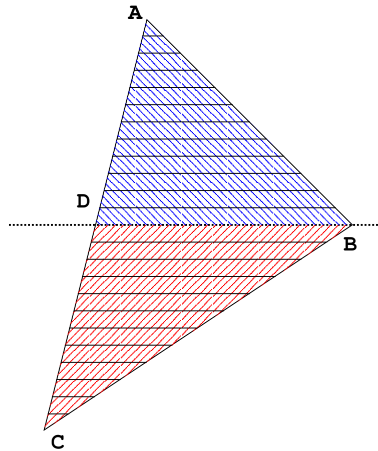

図 4: 三角形の Y 軸方向の分解

6　ラスタライズとクリッピング

空間上の三角形ポリゴンは、3 章の投影変換により画像平面上の三角形に変換される。これを実際に画像上に描画するためには、この実数表現である画像平面上の三角形を、量子化された画素に変換しなくてはならない。これを実現する方法は幾らでも考えられるが、商業性を考慮する場合は、リアルタイム CG 表示システムを念頭におくと、ハードウェア構成設計やその高速化を意識したアルゴリズムにする必要がある。つまり、単純な計算に分解しやすいアルゴリズムや並列性の高いアルゴリズムを考えることが、チップ製造の容易さや高速性への適用性に結び付くのである。

本実験では、これをラスタライズとクリッピングという技法を用いて実現する。これがなぜ商業性を考える上で有利なのかは各自プログラム作成を行いながら考察してもらいたい。

さて、三角形を画素に分解するのがラスタライズで、これは 1 画素が画像平面上の三角形に対してどれほどの大きさになるかを計算し、三角形を X 軸方向に沿ってラスタ分解するものである。この X 軸方向に沿った 1 列をスキャンラインという。さらに、スキャンラインごとに描画画素を決定する。

ラスタライズを簡単にする一つの方法は、画像平面上の三角形の頂点を Y 座標の値によってソートし、真中の値をもつ頂点によって三角形を二つに分割する方法であろう。図 4 において、三角形 ABC を三角形 ADB と三角形 CDB に分解する。

各三角形 ADB,CDB の各頂点の座標の計算は透視投影（並行投影）で計算できる。また、図 5 の辺 PQ 上の点 R の値は頂点どうしの内分比によって簡単に求められる。この辺 PQ がスキャンラインである。点 A の座標を $(x_a, y_a)^t$ などと表すと、計算式は以下のようになる。

$$\begin{pmatrix} x_p \\ y_p \end{pmatrix} = (1-s)\cdot\begin{pmatrix} x_a \\ y_a \end{pmatrix} + s\cdot\begin{pmatrix} x_d \\ y_d \end{pmatrix} \tag{10}$$

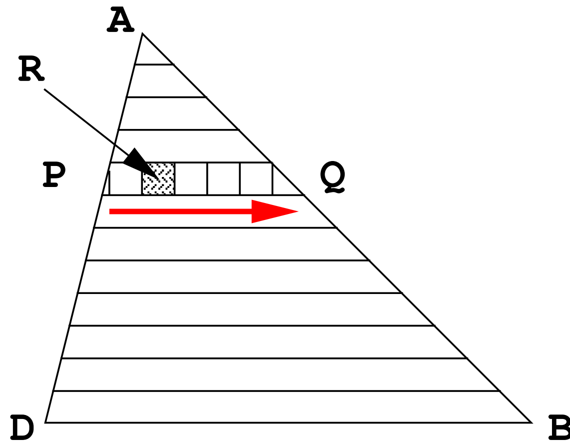

図 5: 三角形のラスタライズ

$$\begin{pmatrix} x_q \\ y_q \end{pmatrix} = (1-s)\cdot\begin{pmatrix} x_a \\ y_a \end{pmatrix} + s\cdot\begin{pmatrix} x_b \\ y_b \end{pmatrix} \tag{11}$$

$$s = \frac{AP}{AD} = \frac{AQ}{AB} \tag{12}$$

$$\begin{pmatrix} x_r \\ y_r \end{pmatrix} = (1-t)\cdot\begin{pmatrix} x_p \\ y_p \end{pmatrix} + t\cdot\begin{pmatrix} x_q \\ y_q \end{pmatrix} \tag{13}$$

$$t = \frac{PR}{PQ} \tag{14}$$

また、この画素計算において、画像の大きさを保持しておいて、一部が画像からはみ出すようなポリゴンの場合には、画像からはみだす部分については処理を行わないようにする。この処理をクリッピングという。

なお、本章での実数表現の三角形を量子化された画素上へ変換する処理は、工夫を凝らさないと一般にジャギーといわれる現象を引き起こす。本演習では学生の諸君はジャギーの発生に対して対策を施さなくともよい。

7 隠面処理とZバッファ

表を向いていて撮像領域に投影されるポリゴンをラスタライズして画素に変換していっても、実際にどの部分が可視状態か（どの面のどの部分が視点に一番近いか）を考慮しなくては、正常なコンピュータグラフィックスにならない。描画する点が他のポリゴンによって隠蔽されていないかどうかを確認する処理を隠面処理という。これを判定する方法については様々なアルゴリズムが考案されている。ここではZバッファ法について説明する。

Zバッファ法は隠面処理アルゴリズムの中ではもっとも簡単な部類に属する。まず、画素毎に $n$bits のZバッファを用意する。物体や投影パラメータ、ハードウェアにもよるが、10bits から 32bits 程度取られていることが多い。Zバッファには、現在描画されている画素の、もとの空間における z 座標値が保存される。

本演習では、カメラを原点に設定し、奥行きの z 座標は遠いほど負になるように座標系をとっていることに注意する。本演習では、z 値が小さいほど遠くにある物体である。

まず、処理の最初には最も小さな値（大きなマイナス値）を z 値に代入しておく。あとは、ポリゴンごとにその内部の点を判定するとき、その点の z 座標を調べる。もし現在の z 座標より大きければ、その点をその画素値に採用し Z バッファの値を更新する。Z バッファの比較を高速化するためには透視投影の四角錐に上底と下底を設け、その間の距離を $n$bits で表現できるように正規化して整数比較するのがよい（実数での大小比較より整数での大小比較のほうが計算量がかからない）。なお、本演習ではプログラム実装時には、実装の簡便さのためから単純に実数値比較してよい。

なお、Z バッファの値の比較は、ラスタライズと同時に行えば効率がよい。これは、プログラムのループ構造は全く同一でよいからである。

Z バッファ法はポリゴンの処理順序（データの登場順序）を問わず、しかも画素単位で計算結果が求められる良い方法であるが、難点はメモリを大量に消費することである。ハードウェア化に適した手法なので、現在の CG ハードウェアのほとんどが本方式を採用している。

余談その 1

意図的に Z バッファ法を使用しなければ、どんなＣＧになるであろうか。また、故意に z 値の評価を入れ替えて（遠いほど描画されるようにする）画像を生成すると、どうなるであろうか。ただし、後者は backfacing をしないようにしておかないと、何も描画されない可能性がある。そのようなＣＧに人間は堪えられるだろうか。

また、画素ごとにいちいち Z バッファで比較するのは馬鹿らしい（＝処理の無駄）であるという考え方もある。では、他にどうする方法が考えられるだろうか。

余談その 2

実は本節で述べた Z バッファの計算方法は、正しい Z バッファ値を与えない。ところが、実際にプログラムを作ってみても、「おかしい」画像が生成されることはほとんどない。さて、カメラとポリゴンがどういう位置関係になるとこの歪みが顕著になるだろうか。

8　シェーディング

さて、物体のどの点が画像上のどの位置に来るかという問題については 3 章および 6 章で解決したが、ではその点にどのような色 (画素値) を設定すればよいのであろうか。光と物体の特性と視点との関係からこれを決定するのがシェーディングである。

これを行うためには、光源の特性と物体の光の反射に関する属性、それに環境モデル（大気）が重要になる。本演習では大気は透明であると仮定するので、環境モデルについては考慮しない。

8.1 光源モデル

光源には、大別して平行光線と点光源がある。

平行光線は太陽のようなもので、無限遠に位置した点光源と見なすこともできる。平行光線の場合は、光源方向と光の強さのみがモデルとして必要である。

一方、点光源はある一点から光が発せられ、その位置と物体との距離が明るさに大きく影響する。点光源の場合、点光源のもとの強さを I、物体のある一点に到達する光の強さを I_p、そこと点光源との距離を d とすると、$I_p \propto \frac{I}{d^2}$ の関係がある。点光源を拡張すれば、蛍光灯のような線光源やパネルのような面光源を規定することも可能である。

以上のことから、本演習では以下のうち一種類以上の光源を利用すること。

1. 平行光源
   - 減衰しない
   - 光の強さは $0.0 \leq 1.0$ に正規化された (R,G,B) で表現する
2. 点光源
   - $I_p \propto \frac{I}{d^2}$ で減衰する（適宜変更した減衰モデルを用いてもよい）
   - 点光源の配置位置において、光の強さは $0.0 \leq 1.0$ に正規化された (R,G,B) で表現する

ただし、光源自身を画像に投影することは一般にそのモデル化が困難であるので、光源を画像に表示することは珍しい。そのような場合は、シェーディング計算上の光源モデルと別に見え方のモデルを用意してある場合が多い。

8.2 反射

光のエネルギーが物体の表面に到達すると、吸収／反射／透過（の幾つか）が同時に起こる。本節以降では、このうち反射について考えていく。

現在、光が物体表面で反射する場合は、拡散反射と鏡面反射の二種類でモデル化できると考えられている。これを二色性反射モデルという。

8.2.1 拡散反射

拡散反射とは、光が物体表面の微細構造の中に入り込み、ランダムな反射を繰り返したあと再び表面から外に向かっていく光の様子をモデル化したものである。今、入射光が方向 $\vec{i}$ で強さ I_l とし、物体の法線ベクトルを $\vec{n}$ とする[5]。ただしベクトルは全て単位ベクトルとする。Lambert の余弦法則により、視点方向 $\vec{e}$ への反射光の強さ I_d は

$$I_d = -(\vec{i} \cdot \vec{n}) k_d I_l \tag{15}$$

[5] 三角形の法線は外積を用いて求めると簡単に求められる

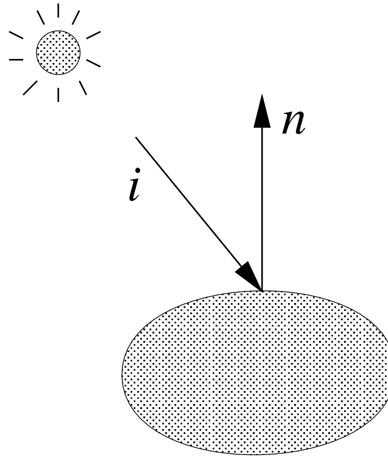

図 6: 拡散反射

で計算できる。k_d は拡散反射係数 (VRML:diffuseColor) で $0 \leq k_d \leq 1$ を満たす物体に固有の値である。また、$-(\vec{i} \cdot \vec{n})$ は $\vec{i}$ と $\vec{n}$ の成す角度の余弦であり、$-(\vec{i} \cdot \vec{n})$ が正値でないときは反射はないとする。この I_d は視線方向に依存しない。すなわち、視点を変えてどの方向からみても同じ明るさとなる。これは特に布や木目などに顕著な傾向とされている。

8.2.2 鏡面反射

鏡面反射は、金属表面を観察したときに見えるハイライト部分の様子をモデル化したものである。前節と同じように変数をとる。鏡面反射は複雑な物理的属性に基づくが、本演習では OpenGL でモデル化されている比較的簡単な鏡面反射モデルを用いる。視点方向への反射光の強さ I_s は次式で表現される。

$$I_s = (\vec{s} \cdot \vec{n})^m k_s I_l \tag{16}$$

$$\vec{s} = \frac{\vec{e} - \vec{i}}{|\vec{e} - \vec{i}|} \tag{17}$$

ここで、I_l は入射光の強さ、$\vec{e}$ は物体上の一点 P から視線への単位方向ベクトル、$\vec{i}$ は光源から点 P までの入射光の単位方向ベクトル、$\vec{n}$ は物体の面の単位法線ベクトル、k_s は物体の鏡面反射における係数 (VRML:specularColor)、$m \geq 0$（実際には 1.0 以上でないとほとんど意味はない）は鏡面反射の強度 (VRML:shiness) を示す。これも拡散反射同様、$(\vec{s} \cdot \vec{n})$ が正値でないときは反射はないとするのが妥当である。

8.2.3 環境光・環境反射

原則的には拡散反射と鏡面反射で CG の計算は可能であるが、現実世界で問題になるのは入射光がどこから来るかである。モデルで定義する明示的な平行光源や点光源は当然入射光の有力成分であるが、その他にもそれらの光源からの光が他の物体で反射した後で別

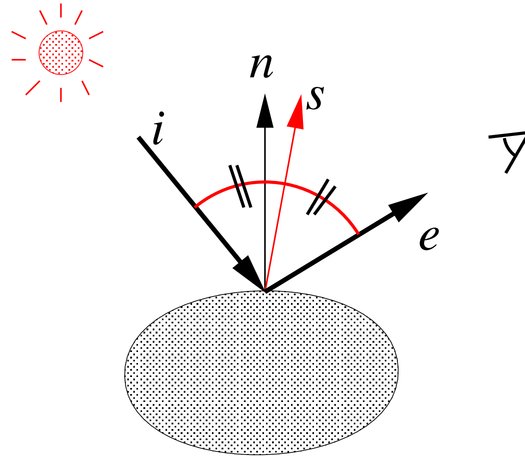

図 7: 鏡面反射

の物体の表面に到達していることが考えられる。一般にこれを相互反射という。これを厳密に計算する様々な方法も提案されているが、これらはどれも膨大な計算コストを必要とする。

そこで、この相互反射を非常に粗い強引な近似で表現するのが環境光・環境反射である。環境光の強さ I_e を次式で表す。

$$I_e = k_a I_a \tag{18}$$

ここで、I_a は環境光強度といい、光源モデルの特殊なパラメータとして設定する。また、k_a は物体に固有の環境光拡散反射係数 (VRML:ambientColor) で $0 \leq k_a \leq 1$ である。

8.2.4 反射のまとめ

拡散反射、鏡面反射、環境反射はそれぞれ RGB 独立に計算し、また複数の光源がある場合はその総和が最終的な物体の色となる。CG で画像を作成する際には、RGB 各 8bits の量子化範囲を逸脱しないように調整しなくてはならない。

8.3 ポリゴンに対するシェーディング

前節までで述べた光と反射の計算モデルは、全て物体上の一点を対象にしたものであった。しかしながら、物体上の全ての点について上記の各計算を行うと計算量は膨大なものになる。ところで、本演習のように三次元物体を表現している場合、物体の表現単位はポリゴンである。そこで、前節までの計算モデルをいかにポリゴンに適用して各画素の色を決定するかについて述べる。

8.3.1 コンスタントシェーディング

最も簡単な方法は、一つのポリゴンについてただ一つの面法線ベクトルと光源ベクトル、視点ベクトルを用い、そのポリゴンに対応する領域全てを単一色 (一定色:Constant color) で塗りつぶす方法である。反射モデルの計算に出てくる三角関数の計算がポリゴンごとにしか行われないので、計算量は少なくて済む。コンスタントシェーディングで生成される CG 画像は微小面の集合体のように見える。

もちろん実際にはポリゴンは大きさをもつので、その範囲内の各点での視線ベクトルは並行でない (透視投影の場合) し、光源ベクトルも並行ではない (点光源の場合)。しかし、一般に画像に対してポリゴンの投影される領域は比較的小さいので、本方法では無視する。

8.3.2 グーローシェーディング

ポリゴンモデルはたいていの場合、より滑らかな物体を平面で近似して表現しているので、その最終結果も滑らかに色が移り変わるように見えるほうが望ましい場合がある。これを実現するのがグーローシェーディング (Gouraud Shading) である。

原理的には、各面の輝度値 (色) を計算したあと、隣り合う面どうしの輝度値を用いて双一次関数で輝度値を補間し、滑らかな輝度値の変化になるようにする方法である。

実際の計算は次のようになる。まず各頂点において面の法線ベクトルを求める。頂点の法線ベクトルは、その頂点に隣接する面全ての面法線ベクトルの平均をとる[6]。その法線ベクトルをもとに、頂点の輝度値 (色) を反射モデルの計算から決定する。いま、投影されたポリゴン ABC について、$\overline{AB}$ と $\overline{AC}$ 上の点を P, Q、$\overline{PQ}$ 上の求める点を R とする。

点 P, Q での輝度値 $I(P), I(Q)$ は、次式で表される。

$$I(P) = (1-s) \cdot I(A) + s \cdot I(B) \tag{19}$$

$$s = \frac{AP}{AB} \tag{20}$$

$$I(Q) = (1-t) \cdot I(A) + t \cdot I(C) \tag{21}$$

$$t = \frac{AQ}{AC} \tag{22}$$

これをもとに、点 R の輝度値 $I(R)$ は次式で計算される。

$$I(R) = (1-u) \cdot I(P) + u \cdot I(Q) \tag{23}$$

$$u = \frac{PR}{PQ} \tag{24}$$

点 P, Q を同一スキャンラインになるように s, t を決定すれば、ラスタライズアルゴリズムと同一走査で点 R の値を計算しシェーディングしていくことが可能になる。

8.3.3 フォーンシェーディング

グーローシェーディングはコンスタントシェーディングに比べて計算量の増加が少ない割に滑らかな外観を与えるので三次元ポリゴンブラウザにはよく用いられるが、問題点も

[6]平均以外にも、立体角などで荷重平均をとる方法などが考えられる

ある。その一つが、双一次補間の対象が輝度値であるので、必ずしも面の法線ベクトルの向きを反映しないことである。これの解決を目指したのがフォーンシェーディング (Phong Shading) である。

フォーンシェーディングでは双一次補間の対象を輝度値の代わりに法線ベクトルとする。あとはグーローシェーディングとほぼ同様であるが、各点において法線ベクトルの値が全て異なるため、各点毎に反射モデルに基づく計算が必要になる。計算量は増大するが、特に鏡面反射などが多い場合には効果を発揮する。

8.4 テキスチャマッピング

これまでは、反射係数はポリゴンごとに 1 組だけが与えられていた。しかし、細かい模様が単一部分平面上に載っているような現実の物体をモデル化するときに、この方法ではその模様中の同色領域ごとにポリゴンを用意することになる。これではモデルデータが大きくなるだけでなく法線ベクトルの計算などに無駄が発生する。

そこで、ポリゴンごとに反射係数を 1 組与える代わりに、そのポリゴンに合わせた反射係数マップを与えることを考える。この反射係数マップが、テキスチャマップと呼ばれる。

テキスチャマッピングの問題点は、ラスタライズの過程で常にテキスチャマップを参照しなくてはならないことである。そのため、テキスチャマップを格納するメモリには非常な高速性が要求される。また、ポリゴンがどのような大きさで表示されるかが予測できない場合、どのようなテキスチャマップの解像度までを用意しておかなくてはならないかという検討が問題になる。

参考文献

[1] David F. Rogers, 山口富士夫監修,“実践コンピュータグラフィックス 基礎手続きと応用”, 日刊工業新聞社,ISBN4-526-02143-1,C3034

[2] David F.Rogers & J.Alan Adams, 山口富士夫監修,“コンピュータグラフィックス 第二版”, 日刊工業新聞社,ISBN4-526-03288-3,C3050

[3] 金谷健一, “画像理解 —3 次元認識の数理—”, 森北出版, ISBN4-627-82140-9

[4] Jakkie Neider & Tom Devis & Mason Woo, “Open GL^{TM} Programming Guide (日本語版)”, 星雲社,ISBN4-7952-9645-6,C3055

[5] Linux Magazine PS2 Linux プログラミング, vol.4, No.2 - vol4. No.10 (2002 年 2 月号 - 2002 年 10 月号), 2002.

[6] 日経 CG, vol.115-118, (1996 年 4-6 月号,9 月号),ISSN0912-1609