計算機科学実験及び演習 4
コンピュータグラフィックス
課題 1

工学部情報学科 3 回生 1029255242
勝見久央

作成日: 2015 年 10 月 21 日

1 概要

本実験課題では 3D ポリゴンデータを透視投影によって投影した PPM 画像を生成するプログラムを C 言語で作成した.

2 要求仕様

作成したプログラムが満たす仕様は以下の通りである.

- ポリゴンデータはプログラム内部で与え、頂点座標をランダムに生成する.
- PPM 画像の大きさは 256×256
- カメラ位置は (x,y,z) = (0.0, 0.0, 0.0)
- カメラ方向は (x,y,z) = (0.0, 0.0, 1.0)
- カメラ焦点距離は 256.0
- ポリゴンには拡散反射を施す

3 プログラムの仕様

3.1 留意点

座標、RGB 値等は、データ型を double として計算し、RGB 値については、出力時に一旦 round 関数を用いて丸めて int 型に変換した後、char 型に変換して出力した. 各点の座標については、double 型のサイズ 3 の配列に xyz 座標を格納して処理した. プログラム内には頂点座標の例として 2 パターン分もコメントで記述している. 大文字アルファベットの定数はマクロである.

3.2 各種定数

3.2.1 ppm

次の定数は ppm ファイル生成のための定数である.

- FILENAME
  ファイル名を指定. ここでは image.ppma としている.
- MAGICNUM
  ppm ファルのヘッダに記述する識別子. P3 を使用.
- WIDTH, HEIGHT, WIDTH_STRING, HEIGHT_STRING
  出力画像の幅、高さ. ともに 256 とする. STRING は文字列として使用するためのマクロ. 以降も同様.
- MAX, MAX_STRING
  RGB の最大値. 255 を使用.

3.2.2 ポリゴンデータ

次の定数はポリゴンデータとして定める定数である.

- VER_NUM
  ポリゴンの頂点数
- SUR_NUM
  ポリゴンを生成する三角形平面の数
- ver[VER_NUM][3]
  VRML の Coordinate ノードの point フィールドの値. ポリゴンを形成する各点の座標を格納する. 点 i の座標は (sur[i][0], sur[i][1], sur[i][2]) である. 初期化は main 関数内で行う. double 型 2 次元配列.
- sur[SUR_NUM][3]
  VRML の IndexedFaceSet ノード内の coordIndex フィールドの値. ポリゴンを形成する三角形を指定する. 三角形は点 sur[i][0]、sur[i][1]、sru[i][2] の 3 点からなる.int 型 2 次元配列.
- diffuse_color[3]
  VRML の Material ノードの diffuseColor フィールドの値. 配列の先頭から順に RGB 値を表す. double 型配列.

3.2.3 環境設定

次の定数は光源モデルなどの外部環境を特定する定数である.

- FOCUS
  カメラの焦点距離. 256.0 と指定.
- light_dir[3]
  光源方向ベクトル.doubel 型配列.
- light_rgb[3]

光源の明るさを正規化した RGB 値にして配列に格納したもの．double 型配列．

3.2.4 その他

- image[HEIGHT][WIDTH]
  描画した画像の各点の画素値を格納するための領域．領域確保のみで初期化は関数内で行う．double 型の 3 次元配列．
- projected_ver[VER_NUM][2]
  画像平面上に投影された各点の座標を格納するための領域．領域確保のみで初期化は関数内で行う．また、最初に main 関数内で全ての点が黒になるように初期化を行ってから、シェーディングの結果をその上に反映させていく形をとる．double 型 2 次元配列．

3.3 関数外部仕様

3.3.1 double func1(double *p, double *q, double y)

double 型 2 次元配列で表された 2 点 p、q の座標と double 型の値 y を引数に取り、直線 pq と直線 y=y の交点の x 座標を double 型で返す関数．ラスタライズの計算を簡素化するために三角形を分割する際に主に用いる．

3.3.2 int lineOrNot(double *a, double *b, double *c)

double 型 2 次元配列で表された 3 点 a、b、c が一直線上にあるかどうかを判別する関数．一直線上にある場合は int 型 1 を返し、それ以外のときは int 型 0 を返す．後述の関数 shading の中で用いる．

3.3.3 void perspective_pro()

大域変数で与えられた頂点座標 ver[VER_NUM][3] の各点に対して透視投影（pespective project）を行い、その結果をグローバル領域に確保された領域 projected_ver[VER_NUM][2] に書き込む．

3.3.4 void shading(double *a, double *b, double *c, double *n)

画像平面上に投影された duble 型 2 次元配列で与えられた 3 点 a、b、c に対してシェーディングを行う関数．さらに、シェーディング時に拡散反射をコンスタントシェーディングで適用するために必要になる、3 点 a、b、c から成るポリゴンの面の法線ベクトルを double 型 3 次元配列の形にした n を引数にとる．シェーディングの結果はグローバル領域にある image[HEIGHT][WIDTH] に書き込む．

3.4 各関数のアルゴリズムの概要

3.4.1 double func1(double *p, double *q, double y)

2 点 p、q を通る直線の方程式を求めて、直線 y=y との交点を計算する．なお直線 pq が x 軸に平行の時はエラーが発生する．

3.4.2 int lineOrNot(double *a, double *b, double *c)

まず最初に 3 点 a、b、c の x 座標が全て同じであるかどうかを判定し、同じであれば一直線上にあると判定する. 同じでなければ、次に点 c の座標を直線 ab の方程式に代入し、等号が成立するかどうかで一直線上にあるかどうかを判定する。

3.4.3 void perspective_pro()

実験資料の数式通りに全ての頂点に対して透視投影を適用する. なお、このプログラムでは、撮像平面の中心が出力後の画像の原点に来るように、投影後の点の座標の xy 座標に $MAX/2$ を加えて平行移動を施している.

3.4.4 void shading(double *a, double *b, double *c, double *n)

この関数内部では処理を単純化するため、処理の最初の段階で引数として与えられた 3 点が形成する三角形の形と 3 点の位置関係を明らかにしておく必要がある. なお、3 点が一直線上にあればシェーディングの必要が無いため、この関数での処理は終了する. このため、3 点の位置関係に応じて以下の判別処理を行う.

- 与えられた 3 点 a、b、c の y 座標が小さい順に r、p、q と名前を変更する
- 与えられた 3 点からなる三角形が分割して処理可能な三角形かどうかを判別する

  ⇒分割できないときは
  - 「p と r の y 座標が同じ」もしくは「p と q の y 座標が同じ」のどちらかを判別する

  - 前の手順で場合分けを行った p と r、もしくは p と q の 2 点で x 座標が p の方が小さくなるよう、必要に応じて名前を入れ替える.
    以上により三角形は図 1 に示される、Type1 と Type2 のどちらであるかが確定する.

  ⇒分割できるときは
  - 三角形を分割し、それぞれの三角形を形成する 3 点に適当な名前をつけて再び shading 関数に引き渡す.
    これにより三角形は図 1 に示されるように Type3 に分類されるとわかる.

以上の処理を行って、三角形の形が Type1〜3 のいずれに分類されるかを判別し、それぞれを別々に処理する. また、シェーディングはコンスタントシェーディングで拡散反射を適用し、各画素の輝度値を計算していく.

4 プログラム本体

プログラム本体は次のようになった.

リスト 1 キャプション

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <math.h>
5
6
7
```

```
8  //========================================================================
9  //必要なデータ
10 #define FILENAME "image.ppm"
11 #define MAGICNUM "P3"
12 #define WIDTH 256
13 #define WIDTH_STRING "256"
14 #define HEIGHT 256
15 #define HEIGHT_STRING "256"
16 #define MAX 255
17 #define MAX_STRING "255"
18 #define FOCUS 256.0
19
20 //パターン1=======================
21 /* #define VER_NUM 5 */
22 /* #define SUR_NUM 4 */
23 /* const double ver[VER_NUM][3] = { */
24 /*      {0, 0, 400}, */
25 /*      {-200, 0, 500}, */
26 /*      {0, 150, 500}, */
27 /*      {200, 0, 500}, */
28 /*      {0, -150, 500} */
29 /* }; */
30 /* const int sur[SUR_NUM][3] = { */
31 /*      {0, 1, 2}, */
32 /*      {0, 2, 3}, */
33 /*      {0, 3, 4}, */
34 /*      {0, 4, 1} */
35 /* }; */
36 //================================
37
38
39 //パターン2=======================
40 /* #define VER_NUM 6 */
41 /* #define SUR_NUM 2 */
42 /* const double ver[VER_NUM][3] = { */
43 /*      {-200, 0, 500}, */
44 /*      {200, -100, 500}, */
45 /*      {100, -200, 400}, */
46 /*      {-100, -100, 500}, */
47 /*      {50, 200,  400}, */
48 /*      {100, 100, 500} */
49 /* }; */
50 /* const int sur[SUR_NUM][3] = { */
51 /*      {0, 1, 2}, */
52 /*      {3, 4, 5}, */
53 /* }; */
54 //================================
55
56
57
58 //パターン3（ラ ン ダ ム 座 標）=======
59 #define VER_NUM 5
60 #define SUR_NUM 4
61
62 //ランダムな座標を格納するための領域を確保
63 //頂点座標はmain関数内で格納
64 double ver[VER_NUM][3];
65
66 const int sur[SUR_NUM][3] = {
67     {0, 1, 2},
68     {0, 2, 3},
69     {0, 3, 4},
70     {0, 4, 1}
71 };
72 //================================
73
74
75 //diffuseColor
76 const double diffuse_color[3] = {0.0, 1.0, 0.0};
77
78 //光源モデルは平行光源
79 //光源方向
80 const double light_dir[3] = {-1.0, -1.0, 2.0};
81 //光源明るさ
82 const double light_rgb[3] = {1.0, 1.0, 1.0};
83 //========================================================================
84
85
86 //メモリ内に画像の描画領域を確保
```

```
87  double image[HEIGHT][WIDTH][3];
88
89  //投影された後の2次元平面上の各点の座標を格納する領域
90  double projected_ver[VER_NUM][2];
91
92
93
94  //2点p、qを結ぶ直線上のy座標がyであるような点のx座標を返す関数
95  //eg)
96  //double p[2] = (1.0, 2.0);
97  double func1(double *p, double *q, double y){
98      double x;
99      if(p[1] > q[1]){
100         x = ((p[0] * (y - q[1])) + (q[0] * (p[1] - y))) / (p[1] - q[1]);
101     }
102     if(p[1] < q[1]){
103         x = ((q[0] * (y - p[1])) + (p[0] * (q[1] - y))) / (q[1] - p[1]);
104     }
105     if(p[1] == q[1]){
106         //解なし
107         printf("\n引数が不正です.\n2点\n(%f,␣%f)\n(%f,␣%f)\nはy座標が同じです.\n"
108                 , p[0], p[1], q[0], q[1]);
109         perror(NULL);
110         return -1;
111     }
112     return x;
113 }
114
115 int lineOrNot(double *a, double *b, double *c){
116     if(a[0] == b[0]){
117         if(a[0] == c[0]){
118             return 1;
119         }
120         else{
121             return 0;
122         }
123     }
124     else{
125         if(c[1] == a[1] + ((b[1] - a[1]) / (b[0] - a[0])) * (c[0] - a[0])){
126             return 1;
127         }
128         else{
129             return 0;
130         }
131     }
132 }
133
134 void perspective_pro(){
135     for(int i = 0; i < VER_NUM; i++){
136         double xp = ver[i][0];
137         double yp = ver[i][1];
138         double zp = ver[i][2];
139         double zi = FOCUS;
140
141         double xp2 = xp * (zi / zp);
142         double yp2 = yp * (zi / zp);
143         double zp2 = zi;
144
145         //座標軸を平行移動
146         //projected_ver[i][0] = xp2;
147         //projected_ver[i][1] = yp2;
148         projected_ver[i][0] = (MAX / 2) + xp2;
149         projected_ver[i][1] = (MAX / 2) + yp2;
150     }
151 }
152
153
154 void shading(double *a, double *b, double *c, double *n){
155     //3点が1直線上に並んでいるときはシェーディングができない
156     if(lineOrNot(a, b, c) == 1){
157         //塗りつぶす点が無いので何もしない.
158     }
159     else{
160         //y座標の値が真ん中点をp、その他の点をq、rとする
161         //y座標の大きさはr <= p <= qの順
162         double p[2], q[2], r[2];
163         if(b[1] <= a[1] && a[1] <= c[1]){
164             memcpy(p, a, sizeof(double) * 2);
165             memcpy(q, c, sizeof(double) * 2);
```

```
166             memcpy(r, b, sizeof(double) * 2);
167         }
168         else{
169             if(c[1] <= a[1] && a[1] <= b[1]){
170                 memcpy(p, a, sizeof(double) * 2);
171                 memcpy(q, b, sizeof(double) * 2);
172                 memcpy(r, c, sizeof(double) * 2);
173             }
174             else{
175                 if(a[1] <= b[1] && b[1] <= c[1]){
176                     memcpy(p, b, sizeof(double) * 2);
177                     memcpy(q, c, sizeof(double) * 2);
178                     memcpy(r, a, sizeof(double) * 2);
179                 }
180                 else{
181                     if(c[1] <= b[1] && b[1] <= a[1]){
182                         memcpy(p, b, sizeof(double) * 2);
183                         memcpy(q, a, sizeof(double) * 2);
184                         memcpy(r, c, sizeof(double) * 2);
185                     }
186                     else{
187                         if(b[1] <= c[1] && c[1] <= a[1]){
188                             memcpy(p, c, sizeof(double) * 2);
189                             memcpy(q, a, sizeof(double) * 2);
190                             memcpy(r, b, sizeof(double) * 2);
191                         }
192                         else{
193                             if(a[1] <= c[1] && c[1] <= b[1]){
194                                 memcpy(p, c, sizeof(double) * 2);
195                                 memcpy(q, b, sizeof(double) * 2);
196                                 memcpy(r, a, sizeof(double) * 2);
197                             }
198                             else{
199                                 printf("エラー\n");
200                                 perror(NULL);
201                             }
202                         }
203                     }
204                 }
205             }
206         }
207 
208         //分割可能な三角形かを判定
209         if(p[1] == r[1] || p[1] == q[1]){
210             //分割できない
211 
212             //長さが1の光源方向ベクトルを作成する
213             //光源方向ベクトルの長さ
214             double length_l =
215                 sqrt(pow(light_dir[0], 2.0) +
216                      pow(light_dir[1], 2.0) +
217                      pow(light_dir[2], 2.0));
218 
219             double light_dir_vec[3];
220             light_dir_vec[0] = light_dir[0] / length_l;
221             light_dir_vec[1] = light_dir[1] / length_l;
222             light_dir_vec[2] = light_dir[2] / length_l;
223 
224             // 法線ベクトルnと光源方向ベクトルの内積
225             double ip =
226                 (n[0] * light_dir_vec[0]) +
227                 (n[1] * light_dir_vec[1]) +
228                 (n[2] * light_dir_vec[2]);
229 
230             if(0 <= ip){
231                 ip = 0;
232             }
233 
234             //2パターンの三角形を特定
235             if(p[1] == r[1]){
236                 //x座標が p <= r となるように調整
237                 if(r[0] <  p[0]){
238                     double temp[2];
239                     memcpy(temp, r, sizeof(double) * 2);
240                     memcpy(r, p, sizeof(double) * 2);
241                     memcpy(p, temp, sizeof(double) * 2);
242                 }
243                 //シェーディング処理
244                 //三角形pqrをシェーディング
```

```
245                 //y座標はp <= r
246
247                 int i;
248                 i = ceil(p[1]);
249                 for(i;
250                     p[1] <= i && i <= q[1];
251                     i++){
252
253                     //撮像平面からはみ出ていないかのチェック
254                     if(0 <= i && i <= (HEIGHT - 1)){
255                         double x1 = func1(p, q, i);
256                         double x2 = func1(r, q, i);
257                         int j;
258                         j = ceil(x1);
259
260                         for(j;
261                             x1 <= j && j <= x2 && 0 <= j && j <= (WIDTH - 1);
262                             j++){
263
264                             image[i][j][0] =
265                                 -1 * ip * diffuse_color[0] *
266                                 light_rgb[0] * MAX;
267                             image[i][j][1] =
268                                 -1 * ip * diffuse_color[1] *
269                                 light_rgb[1] * MAX;
270                             image[i][j][2] =
271                                 -1 * ip * diffuse_color[2] *
272                             light_rgb[2] * MAX;
273                         }
274                     }
275                     //はみ出ている場合は描画しない
276                     else{}
277                 }
278             }
279
280             if(p[1] == q[1]){
281                 //x座標が p < q となるように調整
282                 if(q[0] <  p[0]){
283                     double temp[2];
284                     memcpy(temp, q, sizeof(double) * 2);
285                     memcpy(q, p, sizeof(double) * 2);
286                     memcpy(p, temp, sizeof(double) * 2);
287                 }
288
289                 //シェーディング処理
290                 //三角形pqrをシェーディング
291                 //y座標はp <= q
292
293                 int i;
294                 i = ceil(r[1]);
295
296                 for(i;
297                     r[1] <= i && i <= p[1];
298                     i++){
299                     //撮像平面からはみ出ていないかのチェック
300                     if(0 <= i && i <= (HEIGHT - 1)){
301                         double x1 = func1(p, r, i);
302                         double x2 = func1(q, r, i);
303
304                         int j;
305                         j = ceil(x1);
306
307                         for(j;
308                             x1 <= j && j <= x2 && 0 <= j && j <= (WIDTH - 1);
309                             j++){
310
311                             image[i][j][0] =
312                                 -1 * ip * diffuse_color[0] *
313                                 light_rgb[0] * MAX;
314                             image[i][j][1] =
315                                 -1 * ip * diffuse_color[1] *
316                                 light_rgb[1] * MAX;
317                             image[i][j][2] =
318                                 -1 * ip * diffuse_color[2] *
319                                 light_rgb[2] * MAX;
320                         }
321                     }
322                     //はみ出ている場合は描画しない
323                     else{}
```

```
                }
            }

        }
        //分割できる
        //分割してそれぞれ再帰的に処理
        //分割後の三角形はpp2qとpp2r
        else{
            double p2[2];

            p2[0] = func1(q, r, p[1]);
            p2[1] = p[1];
            //p2のほうがpのx座標より大きくなるようにする
            if(p2[0] < p[0]){
                double temp[2];
                memcpy(temp, p2, sizeof(double) * 2);
                memcpy(p2, p, sizeof(double) * 2);
                memcpy(p, temp, sizeof(double) * 2);
            }
            //分割しても法線ベクトルは同一
            shading(p, p2, q, n);
            shading(p, p2, r, n);
        }
    }
}

int main(void){
    FILE *fp;
    char *fname = FILENAME;


    fp = fopen( fname, "w" );
    //ファイルが開けなかったとき
    if( fp == NULL ){
        printf("%sファイルが開けません.\n", fname);
        return -1;
    }

    //ファイルが開けたとき
    else{
        //頂点座標をランダムに設定==================================
        srand(10);

        ver[0][0] = 0 + (rand()%30) - (rand()%30);
        ver[0][1] = 0 + (rand()%50) - (rand()%50);
        ver[0][2] = 400 + (rand()%50) - (rand()%50);

        ver[1][0] = -200 + (rand()%50) - (rand()%50);
        ver[1][1] = 0 + (rand()%50) - (rand()%50);
        ver[1][2] = 500 + (rand()%50) - (rand()%50);

        ver[2][0] = 0 + (rand()%50) - (rand()%50);
        ver[2][1] = 150 + (rand()%50) - (rand()%50);
        ver[2][2] = 500 + (rand()%50) - (rand()%50);

        ver[3][0] = 200 + (rand()%50) - (rand()%50);
        ver[3][1] = 0 + (rand()%50) - (rand()%50);
        ver[3][2] = 500 + (rand()%50) - (rand()%50);

        ver[4][0] = 0 + (rand()%50) - (rand()%50);
        ver[4][1] = -150 + (rand()%50) - (rand()%50);
        ver[4][2] = 500 + (rand()%50) - (rand()%50);

        //=========================================================

        //描画領域を初期化=======================================
        for(int i = 0; i < 256; i++){
            for(int j = 0; j < 256; j++){
                image[i][j][0] = 0.0 * MAX;
                image[i][j][1] = 0.0 * MAX;
                image[i][j][2] = 0.0 * MAX;
            }
        }
        //=========================================================

        //ヘッダー出力
        fputs(MAGICNUM, fp);
        fputs("\n", fp);
        fputs(WIDTH_STRING, fp);
```

```
403         fputs("␣", fp);
404         fputs(HEIGHT_STRING, fp);
405         fputs("\n", fp);
406         fputs(MAX_STRING, fp);
407         fputs("\n" ,fp);
408
409         //各点の透視投影処理
410         perspective_pro();
411
412         //シェーディング
413         for(int i = 0; i < SUR_NUM; i++){
414             double a[2], b[2], c[2];
415
416             a[0] = projected_ver[(sur[i][0])][0];
417             a[1] = projected_ver[(sur[i][0])][1];
418             b[0] = projected_ver[(sur[i][1])][0];
419             b[1] = projected_ver[(sur[i][1])][1];
420             c[0] = projected_ver[(sur[i][2])][0];
421             c[1] = projected_ver[(sur[i][2])][1];
422
423             //法線ベクトルを計算
424             //投影前の3点の座標を取得
425             double A[3], B[3], C[3];
426             A[0] = ver[(sur[i][0])][0];
427             A[1] = ver[(sur[i][0])][1];
428             A[2] = ver[(sur[i][0])][2];
429
430             B[0] = ver[(sur[i][1])][0];
431             B[1] = ver[(sur[i][1])][1];
432             B[2] = ver[(sur[i][1])][2];
433
434             C[0] = ver[(sur[i][2])][0];
435             C[1] = ver[(sur[i][2])][1];
436             C[2] = ver[(sur[i][2])][2];
437
438             //ベクトルAB, ACから外積を計算して
439             //法線ベクトルnを求める
440             double AB[3], AC[3], n[3];
441             AB[0] = B[0] - A[0];
442             AB[1] = B[1] - A[1];
443             AB[2] = B[2] - A[2];
444
445             AC[0] = C[0] - A[0];
446             AC[1] = C[1] - A[1];
447             AC[2] = C[2] - A[2];
448
449             n[0] = (AB[1] * AC[2]) - (AB[2] * AC[1]);
450             n[1] = (AB[2] * AC[0]) - (AB[0] * AC[2]);
451             n[2] = (AB[0] * AC[1]) - (AB[1] * AC[0]);
452
453             //長さを1に調整
454             double length_n =
455                 sqrt(pow(n[0], 2.0) +
456                      pow(n[1], 2.0) +
457                      pow(n[2], 2.0));
458
459             n[0] = n[0] / length_n;
460             n[1] = n[1] / length_n;
461             n[2] = n[2] / length_n;
462
463             //平面iの投影先の三角形をシェーディング
464             shading(a, b, c, n);
465         }
466
467         //imageの出力
468         for(int i = 0; i < 256; i++){
469             for(int j = 0; j < 256; j++){
470                 char r[256];
471                 char g[256];
472                 char b[256];
473                 char str[1024];
474
475                 sprintf(r, "%d", (int)round(image[i][j][0]));
476                 sprintf(g, "%d", (int)round(image[i][j][1]));
477                 sprintf(b, "%d", (int)round(image[i][j][2]));
478                 sprintf(str, "%s\t%s\t%s\n", r, g, b);
479                 fputs(str, fp);
480             }
481         }
```

```
482     }
483     fclose(fp);
484
485     printf("\nppmファイル␣%s␣の作成が完了しました.\n", fname );
486     return 0;
487 }
```

5 実行例

ランダム座標の描画を行った結果が図 2、コメント内に記載されている av1.wrl の内容を出力した結果が図 3、av2 の内容を出力した結果が図 4 である.

6 問題点

問題点としては、初期段階でモジュール化をうまく考えなかったため、グローバル変数を多用する事になってしまった点、場合分けを多用しすぎてしまい、自分でもこれで必要十分なのかが把握できなくなってしまった点、などが挙げられる.

7 工夫点

工夫した点としては、コメントを随所に入れて見やすいコードを心がけた点、マクロを多用して後でプログラムに変更を加えやすいようにした点、ポリゴンデータなどの入力データは、勝手に書き換わらないよう const で宣言した点、後々の課題で使えそうな処理をモジュール化して書いた点、などが挙げられる.

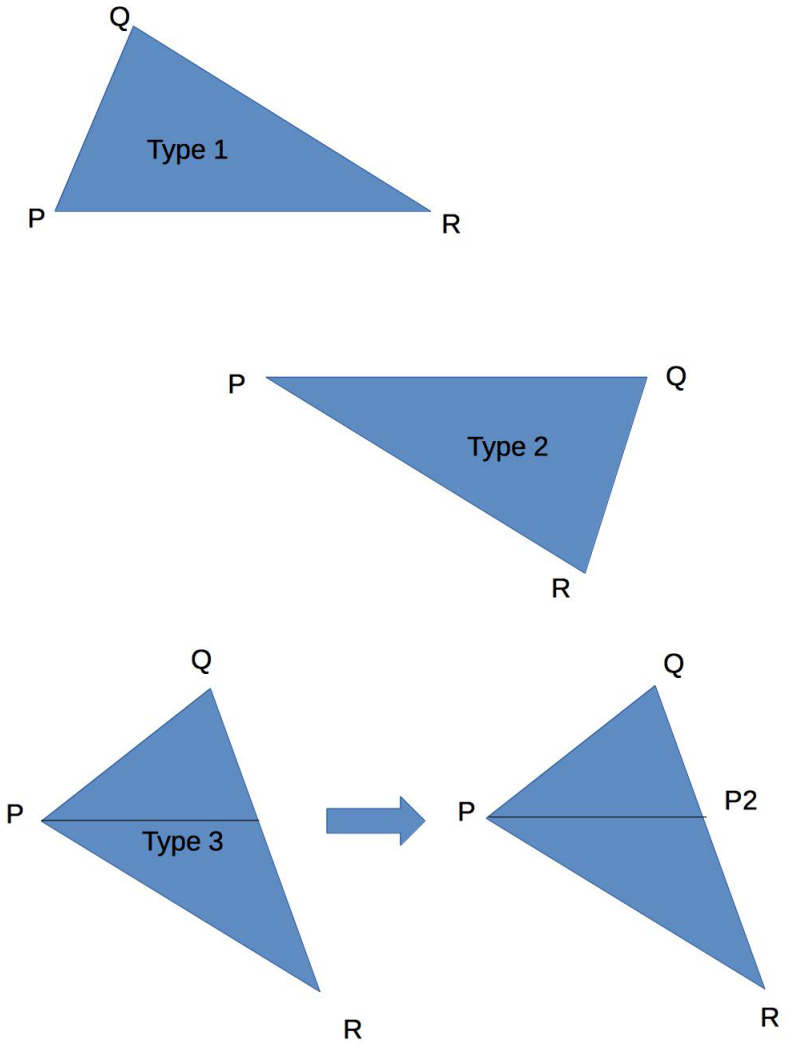

図 1　三角形の場合分け

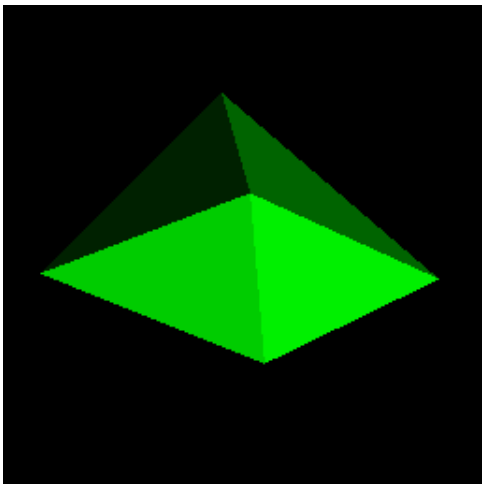

図 2　ランダム座標の出力結果

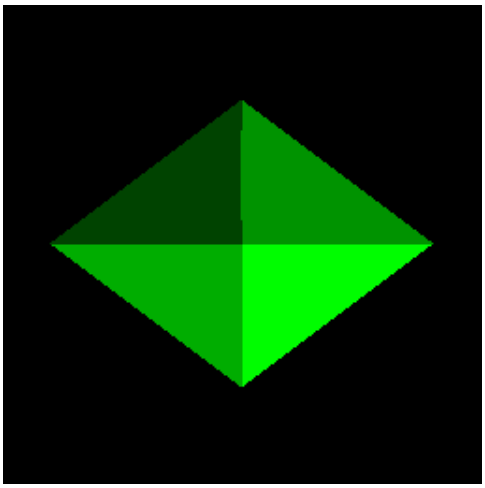

図 3　av1.wrl の出力結果

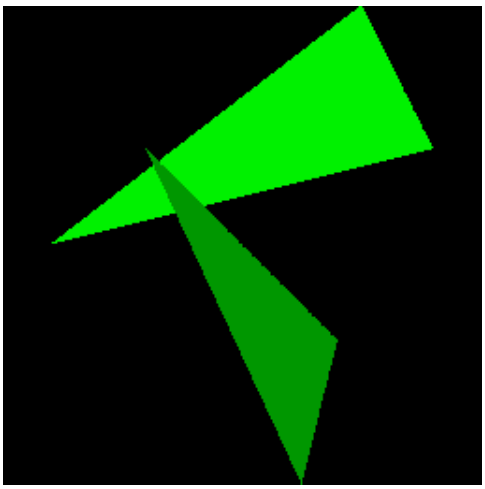

図 4　av2.wrl の出力結果