エルゴノミクスコンピューティング実習
全天球画像処理

井村 誠孝 (m.imura@kwansei.ac.jp)

2017 年 11 月 28 日

バーチャルリアリティシステムの特徴の一つとして，ユーザがある人工的な環境に没入できる点がある．本演習では，全天球画像からユーザの視線に応じた視野画像を生成し提示する方法について学習する．

図 1　全天球画像とアプリケーション例

必要な知識は，

- 画像を扱う方法
- カメラと視界と撮影像の関係

である．

第 1 週は画像について，第 2 週は全天球画像とカメラについて学習し，第 3 週でインタフェースを整備してアプリケーションらしく仕上げる．

■■■ 演習: この項目が演習 各自，実施してください．発展と書かれている項目は，任意です．

(11/28) レポートについては，作成した全天球画像ビューワを提出していただきます．3 節のインタラクティビティの実装については，発展部分とします．

1　Processing で画像を扱う

Processing では，画像を PImage クラスで扱う．

クラスとは

オブジェクト指向プログラミングにおいて，データ (メンバと呼ぶ) とその操作手順 (メソッドと呼ぶ) をまとめたもの．

クラスはメンバ (フィールド) と呼ばれる変数と，メソッドと呼ばれる関数から構成される．
クラスは自分で作ることもできるが，まずは Processing が用意しているクラスを自在に使えるようになろう．

■■■ 演習: **PImage** クラス Processing のリファレンスを参照し，PImage クラスのメンバやメソッドについて確認せよ．

1.1　画像の表示

画像ファイルから画像を読み込み，PImage クラスに格納し，画面に表示するプログラムを以下に示す．画像ファイル test.jpg は，ソースコードと同じフォルダに置かれている必要がある．

リスト 1　画像の読み込みと表示

```
1  PImage img;
2
3  void setup() {
4    size(512, 512);
5    frameRate(60);
6    img = loadImage("test.jpg");
7  }
8
9  void draw() {
10   image(img, 0, 0);
11 }
```

image(img, x, y) 関数は img に格納された画像を (x,y) を左上角として画面に表示する．

1.2　ピクセルの色の取得

画面に表示されているピクセルの色を取得するには，get(x, y) を用いる．
一方，PImage クラスに格納されているピクセルの色を取得するには，PImage クラスのメソッド get() を用

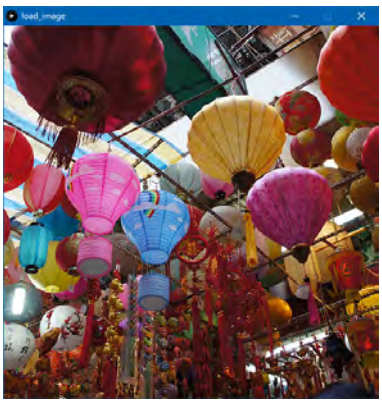

図 2　画像の表示 実行結果

いる．

以下のプログラムは，画像描画前と後で，それぞれ画面と画像の座標 (0,0) のピクセルの色を取得する．実行して画面と画像が別個であることを確認せよ．

リスト 2　画面からと PImage から，それぞれピクセルの色情報の取得

```
PImage img;

void setup() {
  size(512, 512);
  frameRate(60);
  img = loadImage("red.jpg");
}

void draw() {
  background(255);
  color colorWindow1 = get(0, 0);
  color colorImage1 = img.get(0, 0);
  image(img, 0, 0);
  color colorWindow2 = get(0, 0);
  color colorImage2 = img.get(0, 0);

  println("before drawing image");
  println("window", red(colorWindow1), green(colorWindow1), blue(colorWindow1));
  println("image", red(colorImage1), green(colorImage1), blue(colorImage1));
  println("after drawing image");
  println("window", red(colorWindow2), green(colorWindow2), blue(colorWindow2));
  println("image", red(colorImage2), green(colorImage2), blue(colorImage2));
}
```

■プログラムの説明　color 型から赤成分, 緑成分, 青成分を取り出す関数として, 関数 red(), green(), blue() が用意されている．

1.2.1　画面と画像の違いを実感する

まず，以下のサンプルは，マウスがクリックされている間，マウスが置かれている点の色で円を描くサンプルである．実行して動作を確かめよ．

リスト 3　画面と画像の違い 1

```
PImage img;

void setup() {
  size(512, 512);
  frameRate(60);
  img = loadImage("test.jpg");
}

void draw() {
  image(img, 0, 0);
  if (mousePressed == true) {
    color c = get(mouseX, mouseY);
    stroke(0);
    fill(c);
    float size = 100;
    ellipse(mouseX, mouseY, size, size);
  }
}
```

マウスがクリックされているかどうかは，boolean 型変数 mousePressed により知ることができる．またマウスの座標は mouseX と mouseY に格納される．

次に，画面への画像の描画を，毎フレームではなく初期化時に一度だけ行うように変更する．どうなるだろうか．

リスト 4　画面と画像の違い 2

```
PImage img;

void setup() {
  size(512, 512);
  frameRate(60);
  img = loadImage("test.jpg");
  image(img, 0, 0);
}

void draw() {
  if (mousePressed == true) {
    color c = get(mouseX, mouseY);
    stroke(0);
    fill(c);
    float size = 100;
    ellipse(mouseX, mouseY, size, size);
  }
}
```

更に，円の色を，画面からではなく，画像から取得するようにするとどうなるか．

リスト 5　画面と画像の違い 3

```
PImage img;

void setup() {
  size(512, 512);
  frameRate(60);
  img = loadImage("test.jpg");
  image(img, 0, 0);
}

void draw() {
  if (mousePressed == true) {
    color c = img.get(mouseX, mouseY);
    stroke(0);
    fill(c);
    float size = 100;
    ellipse(mouseX, mouseY, size, size);
  }
}
```

■■■ 演習: 画面と画像の違い　上記 3 つのプログラムを実行し，画面と画像の違いを実感せよ．

■■■ 演習: 色の取得 モザイク　マウスがクリックされている間，マウス周辺をモザイク化するプログラムを作成せよ．

for の二重ループと rect() を使う．

1.3　ピクセルの色の変更

画面に表示されているピクセルの色を変更するには，set(x, y, color) を用いる．

座標 (x, y) の色を赤 x，緑 y, 青 0 にするプログラムを以下に示す．実行してみよ．

リスト 6　画面のピクセルに描画

```
void setup() {
```

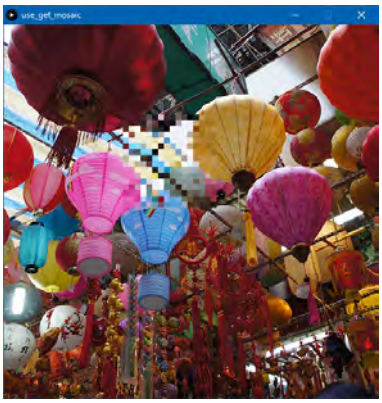

図 3　モザイク化の結果

```
  size(512, 512);
  frameRate(60);
}

void draw() {
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      color c = color(x, y, 0);
      set(x, y, c);
    }
  }
}
```

PImage クラスに格納されているピクセルの色を変更するには，PImage クラスのメソッド `set()` を用いる．読み込んだ画像をモノクロにしてから表示するプログラムを以下に示す．実行してみよ．

リスト 7　画像のピクセルに描画

```
PImage img;

void setup() {
  size(512, 512);
  frameRate(60);
  img = loadImage("test.jpg");

  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      color c = img.get(x, y);
      float v = 0.3 * red(c) + 0.6 * green(c) + 0.1 * blue(c);
      img.set(x, y, color(v, v, v));
    }
  }
}

void draw() {
  image(img, 0, 0);
}
```

注: PImage のモノクロ化はメソッド `filter()` を用いても行えるが，ここでは画素単位の処理を実感するために，画素値の直接操作を行っている．

■■■ 演習: 画素値の変更 モノクロ化　マウスがクリックされている間，マウス周辺をモノクロ化するプログラムを作成せよ．

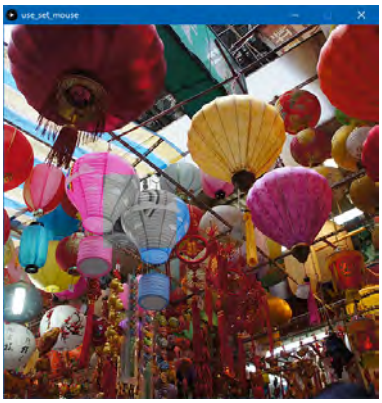

図 4　モノクロ化の結果

1.4　画像の変形

画像の拡大縮小，移動，回転などは，変換前の画像のピクセルと，変換後の画像のピクセルを対応付ける処理と考えることができる．

例えば，拡大縮小の場合，変換前の画像のサイズが (w_1, h_1) で，変換後の画像のサイズが (w_2, h_2) の場合，変換前のピクセルの座標 (x_1, y_1) と変換後のピクセルの座標 (x_2, y_2) との対応は以下のようになる．

$$x_2 = \frac{w_2}{w_1} x_1$$

$$y_2 = \frac{h_2}{h_1} y_1$$

変換前を PImage に読み込んだ画像，変換後を画面として，マウスの位置に応じた拡大・縮小を行うプログラムを以下に示す．

リスト 8　拡大・縮小

```
PImage img;

void setup() {
  size(512, 512);
  frameRate(60);
  img = loadImage("test.jpg");
}

void draw() {
  background(0);
  float scaleX = float(mouseX) / img.width;
  float scaleY = float(mouseY) / img.height;
  for (int y2 = 0; y2 < mouseY; y2++) {
    for (int x2 = 0; x2 < mouseX; x2++) {
      int x1 = int(x2 / scaleX);
      int y1 = int(y2 / scaleY);
      set(x2, y2, img.get(x1, y1));
    }
  }
}
```

■■■ 演習: 画像の回転　ピクセル単位の処理を行って画像を回転させるプログラムを作成せよ．細部の設計 (回転の中心，速度，インタラクティブ性など) は各自の好みで行ってよい．

■■■ 演習: [発展] 画像の高品質な回転　変換後のピクセルの色を，変換前の複数ピクセルの色の重み付け和から算出することによって，回転後の画像の品質を向上させよ．

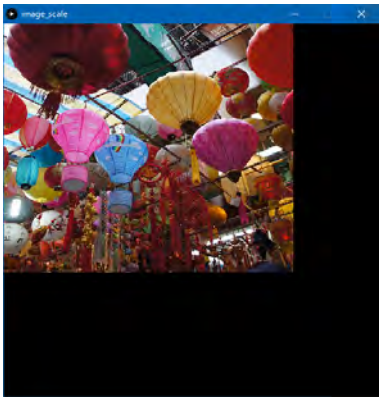

図 5　拡大縮小の結果

(a) RICOH Theta　　(b) Panono

図 6　全天球カメラの例

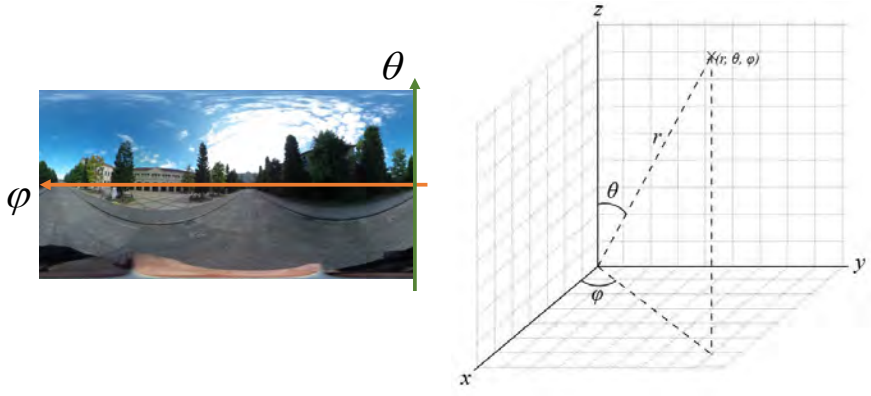

図 7　全天球画像の座標系と極座標表現

2　全天球画像

全天球画像，あるいは全方位画像は，上下左右全方向の情報を持った画像である．360 度パノラマ画像と呼ばれることもある．

全天球画像の撮影は，RICOH Theta シリーズ (図 6(a)) や Panono(図 6(b)) のような特殊なカメラを用いることで容易に行える．またスマートフォンでも指示に従って画像を複数枚撮影することで全天球画像を合成するアプリケーションがある．

得られた全天球画像は，カメラを中心とする球面上の画像を，世界地図のように展開したものとなる．画像の横軸が経度，縦軸が緯度に相当する．カメラの中心を原点とすると，原点からの視線ベクトルを極座標 r, θ, φ で表した際の，θ が縦軸，φ が横軸となる．全天球画像の座標系と極座標との対応を図 7 に示す．

注意: 全天球画像を内側から眺めるので，いわゆる極座標での φ の増加方向は画像の右から左になる．

これからわかるように，全天球画像の適切なピクセルにアクセスするためには極座標系での知識が必要となる．直交座標系 (x, y, z) と極座標系 (r, θ, φ) の変換は相互に行えるようになっておこう．

■直交座標から極座標へ

$$r = \sqrt{x^2 + y^2 + z^2} \tag{1}$$

$$\theta = \tan^{-1} \frac{\sqrt{x^2 + y^2}}{z} \tag{2}$$

$$\varphi = \tan^{-1} \frac{y}{x} \tag{3}$$

Processing では $\tan^{-1}$ の計算には `atan2()` を用いるのがよいことは以前述べた (はず).

■極座標から直交座標へ

$$x = r \sin\theta \cos\varphi \tag{4}$$
$$y = r \sin\theta \sin\varphi \tag{5}$$
$$z = r \cos\theta \tag{6}$$

2.1 全天球画像をとりあえず表示

全体の流れは以下のようになる．

1. 全天球画像を読み込む (imgSrc).
2. ウィンドウと同じサイズのカメラ画像 imgDst を作成する．
3. カメラ画像 imgDst の各ピクセルに，全天球画像 imgSrc の対応するピクセルの RGB 値をコピーする．
4. カメラ画像をウィンドウに表示する．

このうちステップ 3 が重要であるが，対応関係については後程考えるとして，ここではまず，全天球画像の中央部分を切り出して表示する雛形を示す．

リスト 9　全天球画像の中央部分を切り出して表示

```
1  PImage imgSrc;
2  PImage imgDst;
3
4  void setup() {
5    size(640, 360);
6    frameRate(60);
7
8    imgSrc = loadImage("R0010014.JPG");
9    imgDst = createImage(width, height, RGB);
10 }
11
12 void draw() {
13   imgSrc.loadPixels();
14   imgDst.loadPixels();
15   int offsetX = (imgSrc.width - imgDst.width) / 2;
16   int offsetY = (imgSrc.height - imgDst.height) / 2;
17   for (int y = 0; y < imgDst.height; y++) {
18     for (int x = 0; x < imgDst.width; x++) {
19       int x1 = offsetX + x;
20       int y1 = offsetY + y;
21       color c = imgSrc.pixels[x1 + y1 * imgSrc.width];
22       imgDst.pixels[x + y * imgDst.width] = c;
23     }
24   }
25   imgDst.updatePixels();
26   set(0, 0, imgDst);
27 }
```

■ソースコードの解説

- 画像を保持する imgSrc および imgDst はグローバルに宣言する．
- createImage() は，引数に指定されたサイズの画像を生成する．imgDst はウィンドウと同じサイズである必要があるため，ウィンドウサイズを格納しているシステム変数である width と height を指定している．第 3 引数はカラーモードである．
- PImage クラスの各ピクセルの値はフィールド pixels[] を通して読み書きできる．pixels[] は color 型の 1 次元配列である (2 次元配列ではない)．画像 img の位置 (x,y) のピクセルには，

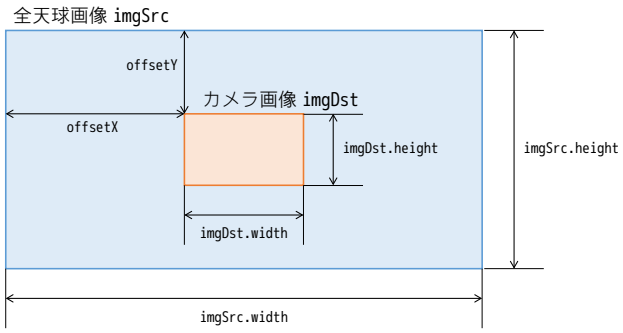

図 8　切り出す領域

`img.pixels[x + y * img.width]` でアクセスできる．

- PImage クラスの画素値を直接操作する場合，操作を開始する前にメソッド `loadPixels()` を実行する必要がある．このメソッドを呼ぶことで，フィールド `pixels[]` に正しく値が格納される．
- フィールド `pixels[]` の内容を変更した場合，メソッド `updatePixels()` を実行することで，変更が画像に反映される．
- `set()` によって，画像を画面にコピーすることができる．第 1，第 2 引数はコピー先の左上座標である．
- 切り出す領域は図 8 のようになっている．ソースコード中，`offsetX` および `offsetY` を計算している部分を確認せよ．

■■■ **演習: 画像の切り出し**　上記プログラムを実行してみよ．また，画像の左上，および，右下部分を切り出すように，`offsetX` および `offsetY` の値を変更してみよ．

このプログラムでは，全天球画像の中央部分を切り出しているが，矩形で切り出すのでは，上下中央付近であれば歪みが少ないが，仰角あるいは俯角が大きくなると歪みが大きくなる．歪みの無い画像を得るためには，カメラモデルを利用して，適切なピクセル間の対応を算出する必要がある．以下では対応の算出に必要な情報を整理する．

2.2　カメラモデル

全天球画像ビューワでは，バーチャルなカメラを球の中心に置き，そのカメラから撮影できるであろうカメラ画像を生成する．

カメラはピンホールカメラモデルで取り扱う．ピンホールカメラモデルでは，空間中の点 M は，点 M とレンズ中心 (焦点) を結ぶ直線と，画像平面との交点に投影される．実際のカメラでは画像平面はレンズよりも後ろにあるが，計算の簡単化のためにレンズ中心で点対称な位置にあるとしても数学的には差し支えない．ピンホールカメラモデルを図 9 に示す．

全天球画像ビューワでは，レンズ中心が全天球画像を仮想的に貼り付けた球の中心と一致しているものとする．

ここでは焦点面と画像平面の間が f 離れているとする．z 軸正の方向を光軸に取る．画像平面のサイズを幅 w，高さ h とし，レンズ中心から画像平面を見て右方向に x 軸正の方向を，下方向に y 軸正の方向を取る．画像平面が横方向に n_x 画素，縦方向に n_y 画素で構成されているとすると，画素 (i,j) のカメラ座標系での座標 (x_c, y_c, z_c) は，以下の式で表される．

$$x_c = \left(i - \frac{n_x}{2}\right)\frac{w}{n_x} \tag{7}$$

$$y_c = \left(j - \frac{n_y}{2}\right)\frac{h}{n_y} \tag{8}$$

$$z_c = f \tag{9}$$

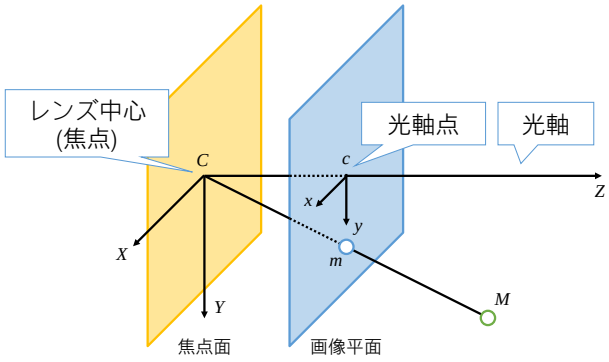

図 9　ピンホールカメラモデル

画像平面のサイズは焦点面と画像平面の間の距離 f によって変わるため，直接指定するよりは，画角で指定する方が直感的である．画像平面の水平方向画角を θ_{fov} とすると，スクリーンの幅 w は

$$w = 2f \tan \frac{\theta_{\text{fov}}}{2}$$

で表される．スクリーンのアスペクト比はウィンドウのアスペクト比と同じである．[*1]表示先 (=本演習では `imgDst`) の横画素数は n_x，縦画素数は n_y であったから，スクリーンの高さ h は

$$h = \frac{n_y}{n_x} w$$

で得ることができる．

■本節のまとめ　焦点面と画像平面間の距離 f，水平方向画角 θ_{fov} を決める．ウィンドウ (=`imgDst`) の画素数 n_x および n_y は決まっている．

→カメラ画像のサイズ w および h が求まる．

→画素 (i,j) のカメラ座標系での 3 次元座標 (x_c, y_c, z_c) が求まる．

2.3　カメラの姿勢の表現とカメラ座標系

カメラの位置および姿勢の表現について考える．位置はレンズ中心が原点に固定されている．姿勢は，光軸ベクトルおよびカメラ上方向ベクトルによって決まる．最終的に全天球画像のピクセルにアクセスすることを考えると，光軸の向き (カメラ座標系での z 軸正方向) を極座標で表現する，すなわち，θ_v および φ_v で表すのが好都合である．添字の v は viewing の意味である．カメラ上方向は，カメラ座標系では y 軸負の方向であるが，これはなるべく鉛直上向きになるようにする．すると，残りのカメラ座標系 x 軸は，水平面上にくる．カメラ座標系の様子を図 10 に示す．

θ_v および φ_v が決まると，カメラ座標系での基底ベクトルの全天球画像座標系における成分を，θ_v および φ_v で表現可能である．

$$\vec{e}_x = (\sin\varphi_v, -\cos\varphi_v, 0)^T \tag{10}$$

$$\vec{e}_y = (\cos\theta_v \cos\varphi_v, \cos\theta_v \sin\varphi_v, -\sin\theta_v)^T \tag{11}$$

$$\vec{e}_z = (\sin\theta_v \cos\varphi_v, \sin\theta_v \sin\varphi_v, \cos\theta_v)^T \tag{12}$$

■本節のまとめ　カメラの向いている方向 θ_v および φ_v (これらの値はプログラム内で指定，もしくは，マウス等でインタラクティブに変更)

→座標系の基底ベクトルの成分を θ_v および φ_v で記述

[*1] アスペクト比とは，矩形の長辺と短辺の比率を指す．

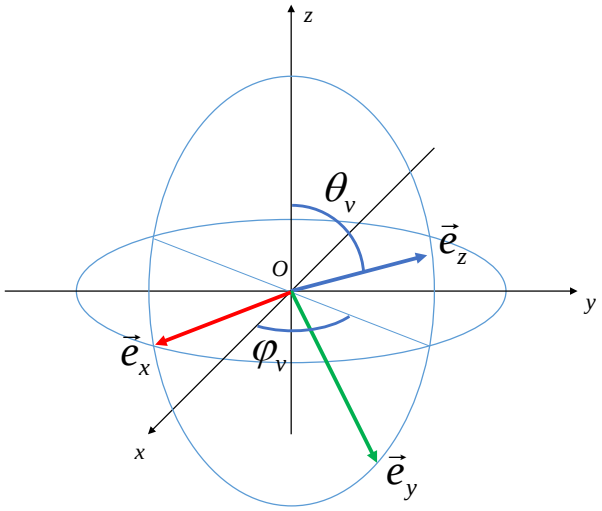

図 10　カメラ座標系

→ 2.2 節で求めた，各画素のカメラ座標系での座標値と，基底ベクトルの成分から，各画素の全天球座標系での座標値がわかる (次節で活用).

2.4　全天球画像とカメラ画像との対応関係

これまでの準備に基づいて，カメラ画像の各画素に対応する，全天球画像の画素の座標を求める手順を整理すると，以下のようになる．

1. カメラの光軸の向きは，θ_v および φ_v で表す．それぞれの変数の取り得る範囲は，$-\pi \le \theta_v \le \pi$ および $0 \le \varphi_v < 2\pi$ である．
2. カメラ座標系の基底ベクトル $\vec{e}_x, \vec{e}_y, \vec{e}_z$ の全天球座標系における成分を求める．
3. カメラ画像の各画素について，以下の処理を行って色を決定する．
   (a) カメラ座標系での座標 (x_c, y_c, z_c) を求める．
   (b) 座標を全天球座標系に直す．具体的には，

$$x_c\vec{e}_x + y_c\vec{e}_y + z_c\vec{e}_z$$

   を計算する．
   (c) 得られた座標を極座標に変換する．変換後の θ と φ が，今処理している画素の色のコピー元座標となる．
   (d) 全天球画像の画素値をカメラ画像に書き込む．

■■■ 演習: 全方位画像ビューワの作成　上記の処理を実装し，全方位画像ビューワを作成せよ．
焦点距離 f の値は実は何でもよい．θ_{fov} は 50deg くらい (Processing の各種関数に与える角度はラジアンなので注意)．視線の向き θ_v および φ_v はさしあたり $\theta_v = \pi/2$，$\varphi_v = \pi$ にしておき，適宜変更してみよ．
(θ_v は $\pi/2$ だと水平です．)

2.5　PVector クラスを用いる

Processing にはベクトルを扱うためのクラス PVector が用意されているので，適宜利用するとよい．使用方法はリファレンスマニュアルの使用例を見て理解するのがよいと思うが，主な使い方をまとめておく．

- ベクトルの生成

成分が (x1,y1,z1) のベクトルを作る

```
v1 = new PVector(x1, y1, z1);
```

- ベクトルの和

  ベクトル v2 にベクトル v1 を足して，v2 に格納する．

  ```
  v2.add(v1);
  ```

  ベクトル v2 にベクトル v1 を足して，v3 に格納する．

  ```
  v3 = PVector.add(v1, v2);
  ```

- ベクトルの定数倍

  ベクトル v1 の全成分を s 倍する．

  ```
  v1.mult(s);
  ```

- ベクトルの内積

  ベクトル v1 と v2 の内積を得る．

  ```
  float d = v1.dot(v2);
  ```

- ベクトルの正規化

  ベクトル v1 の長さを 1 にする．

  ```
  v1.normalize();
  ```

3 ユーザインタフェース

本節は，Processing が提供しているインタラクティビティについてまとめておく．

Processing でユーザからの入力状況を知るための方法は，大別して 2 種類ある．一つは，システムが用意している変数 (システム変数) の値を読み出す方法，もう一つはイベント関数を用いて各種のイベント発生時の処理を記述する方法である．

キーの状態やマウスの位置など，インタフェースの状態に応じて，毎フレーム処理を行う場合はシステム変数の利用が適している．

一方で，キーの押し下げやマウスの移動など，不定期に行われる操作 (イベント) に対応した処理は，イベントが起きた際に実行されるイベント関数を使用すると簡潔に記述できる．イベント関数の中でシステム変数やイベントに関する情報を適宜参照し，必要な情報を得て処理を進めるのが一般的である．このように生じたイベントに対応する形式で処理を進めることをイベントドリブン (イベント駆動) 方式と呼ぶ．

3.1 マウス

マウスの情報を保持しているシステム変数は以下の通りである．

- `mouseX`: X 座標
- `mouseY`: Y 座標
- `pmouseX`: 前フレームの X 座標
- `pmouseY`: 前フレームの Y 座標
- `mousePressed`: ボタンのいずれかが押されているか．boolean 型.
- `mouseButton`: どのボタンが押されているか．`LEFT` or `RIGHT` or `CENTER`

また，マウスに関連するイベント関数には，以下のものがある．

1. `mousePressed()`: マウスボタンを押したときに実行される．
2. `mouseReleased()`: マウスボタンを離したときに実行される．
3. `mouseClicked()`: マウスボタンがクリック (押されてから離された) ときに実行される．
4. `mouseMoved()`: マウスを移動させたときに実行される．
5. `mouseDragged()`: マウスを押しながら移動させたときに実行される．
6. `mouseWheel()`: マウスのホイールを回転させたときに実行される．

■■■ 演習: [発展] 全天球画像ビューワへのインタラクティビティの追加 前節で実装した全天球画像ビューワについて，見る範囲をマウスで操作できるようにせよ．例えばドラッグで視線の移動，ホイールで視野角の変更など．

3.2 キーボード

キーボードの情報を保持しているシステム変数は以下の通りである．

- `key`: 最後に押されたキーの文字 (英数字 1 文字)
- `keyCode`: 最後に押されたキーのキーコード
- `keyPressed`: キーのいずれかが押されているか．boolean 型.

また，キーボードに関連するイベント関数には，以下のものがある．

1. `keyPressed()`: キーを押したときに実行される．

2. keyReleased(): キーを離したときに実行される．
3. keyTyped(): キーを押したときに実行される．ただし，修飾キー (Shift, Ctrl, Alt など) は無視される．

3.3 イベントフロー

プログラムの流れを制御するための関数がいくつか用意されている．

1. noLoop(): draw() の繰り返し実行を止める．
2. loop(): draw() の繰り返し実行を再開する．
3. redraw(): draw() を一度だけ実行する．