OpenGL

- typedef unsigned int GLenum
- typedef unsigned int GLbitfield
- typedef int GLint
- typedef int GLsizei
- typedef unsigned int GLuint
- typedef float GLfloat
- typedef float GLclampf
- typedef double GLdouble
- typedef void GLvoid
- void glutInit(int *argcp, char **argv)
  - GLUTライブラリを初期化します.
  - 「argc」と「argv」はmain関数の引数，すなわちコマンドライン引数を渡します．これらの引数は，コマンドラインのオプション指定時に用いられます.
- void glutInitDisplayMode(unsigned int mode)
  - ディスプレイの表示モードを設定します.
  - 「glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE | GLUT_DEPTH)」のように書くと，「RGBAカラーモデル」で「ダブルバッファ」を使い，「デプスバッファ」も使うという指定になります.
- void glutInitWindowSize(int width, int height)
  - ウィンドウの初期サイズを設定します.
  - 「width」はウィンドウの幅，「height」はウィンドウの高さになります.
- void glutInitWindowPosition(int x, int y)
  - ウィンドウの左上の位置を指定する．引数は共にピクセル値.
- int glutCreateWindow(char *title)
  - ウィンドウを生成する．引数はそのウィンドウの名前となる.
- void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha)
  - 「glClear(GL_COLOR_BUFFER_BIT)」でウィンドウを塗りつぶす際の色を指定します.
  - 「red」「green」「blue」はそれぞれ「赤」「緑」「青色」の成分の強さを示すGLclampf型（float型と等価）の値で，0～1の間の値をもちます．1が最も明るく，この3つに(0,0,0)を指定すれば「黒色」になり，(1,1,1)を指定すれば「白色」になります.
  - 最後の「alpha」は「α値」と呼ばれ，OpenGLでは不透明度として扱われます（0で透明，1で不透明）．ここではとりあえず「1」にしておいてください.
- void glutMainLoop(void)
  - GLUTのイベントが発生するまで，待機状態になります.
- void glutSwapBuffers(void)
  - 描画の最後で記述する．この関数が実行されると，バックバッファの内容がフロントバッファに転送される.
- void glClear(GLbitfield mask)
  - 「mask」に指定したバッファのビットを初期化します.
  - 「glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)」と指定すると「カラーバッファ」と「Zバッファ」が初期化されます.
- void glEnable(GLenum cap)
  - GLenum型の引数「cap」に指定した機能を使用可能にします.
  - 「glEnable(GL_DEPTH_TEST)」を実行すると，それ以降「Zバッファ」を使います.
  - 「glEnable(GL_LIGHTING)」を実行すると，それ以降「陰影付け」の計算をします.
  - 「glEnable(GL_LIGHT0)」を実行すると，0番目の光源を点灯します.

OpenGL

- void glutDisplayFunc(void (*)(void))
  - 引数は開いたウィンドウ内に描画する関数へのポインタです．ウィンドウが開かれたり，他のウィンドウによって隠されたウィンドウが再び現われたりしてウィンドウを再描画する必要があるときに，この関数が実行されます．したがって，この関数内で図形表示を行います．
- void glutReshapeFunc(void (*)(int width, int height))
  - 引数には，ウィンドウがリサイズされたときに実行する関数のポインタを与えます．この関数の引数にはリサイズ後のウィンドウの幅と高さが渡されます．
- void glutTimerFunc(unsigned int millis, void (*)(int value), int value)
  - 指定された時間に呼び出されるコールバック関数を登録します．異なる時間のコールバック関数を複数用意できます．
  - 「millis」は呼び出される時間をミリ秒で指定します．少なくとも「millis」ミリ秒後にコールされるようになります．
  - 第3引数の「value」は登録したタイマーコールバック関数に渡されます．
- void glutKeyboardFunc(void (*)(unsigned char key, int x, int y))
  - 引数には，キーがタイプされたときに実行する関数のポインタを与えます．この関数の引数「key」には，タイプされたキーのASCIIコードが渡されます．また，「x」と「y」にはキーがタイプされたときのマウスの位置が渡されます．
- void glutPostRedisplay(void)
  - ウィンドウを再描画します．glutDisplayFunc()で登録したコールバック関数が呼び出されます．
- void glViewport(GLint x, GLint y, GLsizei width, GLsizei height)
  - 「ビューポート」を設定します．「ビューポート」とは，開いたウィンドウの中で，実際に描画される領域のことをいいます．正規化デバイス座標系の2点(-1,-1)，(1,1)を結ぶ線分を対角線とする矩形領域がここに表示されます．最初の2つのGLint型（int型と等価）の引数「x,y」にはその領域の左下隅の位置，後の2つのGLsizei型（int型と等価）の「width」と「height」には，それぞれ幅と高さをデバイス座標系の値，すなわちディスプレイ上の画素数で指定します．glutReshapeFuncで指定されたコールバック関数の引数「width,height」にはそれぞれウィンドウの幅と高さが入っていますから，glViewport(0,0,width,height)はリサイズ後のウィンドウの全面を表示領域に使うことになります．

OpenGL

- void glMatrixMode(GLenum mode)
  - 設定する変換行列を指定します．引数「mode」が「GL_MODELVIEW」なら「モデルビュー変換行列」を指定し，「GL_PROJECTION」なら「透視変換行列」を指定します．
- void glLoadIdentity(void)
  - これは変換行列を初期化します．座標変換の合成は行列の積で表されますから，この関数を使って変換行列に初期値として単位行列を設定します．
- void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)
  - 変換行列に透視変換の行列を乗じます．
  - 最初の引数「fovy」はカメラの画角であり，「度」で表します．これが大きいほど広角（透視が強くなり，絵が小さくなります）になり，小さいほど望遠レンズになります．
  - 2つ目の引数「aspect」は画面のアスペクト比（縦横比）であり，「1」であればビューポートに表示される図形のx方向とy方向のスケールが等しくなります．
  - 3つ目の引数「zNear」と4つ目の引数「zFar」は表示する奥行き方向の範囲で，「zNear」は手前（前方面），zFarは後方（後方面）の位置を示します．この間にある図形が描画されます．
- void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz)
  - この最初の3つの引数「eyex, eyey, eyez」は視点の位置，次の3つの引数「centerx, centery, centerz」は目標の位置，最後の3つの引数「upx, upy, upz」は，ウィンドウに表示される画像の「上」の方向を示すベクトルです．
- void glPushMatrix(void)
  - glMatrixMode()で指定している現在の変換行列を保存します．
- void glPopMatrix(void)
  - glPopMatrix()で保存した変換行列を復帰します．したがって，「glPushMatrix()」を呼び出した後，glTranslated()やglRotated()あるいはgluLookAt()などを使って変換行列を変更しても，「glPopMatrix()」を呼び出すことによって，それ以前の変換行列に戻すことができます．
- void glTranslated(GLdouble x, GLdouble y, GLdouble z)
  - 変換行列に平行移動の行列を乗じます．引数はいずれもGLdouble型で，3つの引数「x」「y」「z」には現在の位置からの相対的な移動量を指定します．

OpenGL

- void glBegin(GLenum mode)
  void glEnd(void)
  - 「glBegin」と「glEnd」の間に指定した頂点座標を使って，描画を行います．
  - 描画内容は「mode」に指定します．「mode」には「GL_LINE_STRIP」や「GL_LINES」や「GL_TRIANGLES」や「GL_QUADS」や「GL_POLYGON」などが指定できます．
- void glVertex3d(GLdouble x, GLdouble y, GLdouble z)
  void glVertex3dv(GLdouble *v)
  - 3次元の座標値を設定します．
  - 引数はGLdouble型の (x, y, z)で指定します．
  - 引数として3つの要素をもつGLdouble型の配列vで指定する関数も用意されています．
- void glNormal3d(GLdouble nx, GLdouble ny, GLdouble nz)
  void glNormal3dv(GLdouble *v)
  - 単位法線ベクトルを設定します．
  - 引数はGLdouble型の(nx, ny, nz)，または，3つの要素をもつGLdouble型の配列vで指定します．
- void glMaterialfv(GLenum face, GLenum pname, GLfloat *params)
  void glMaterialf(GLenum face, GLenum pname, GLfloat param)
  - 表面属性を定義する．
  - 「face」に「GL_FRONT」を指定すると，ポリゴンの表面のみに属性を設定します．
  - 「pname」に「GL_DIFFUSE」を指定すると，「params」でfloat型の配列を指定することで，材質の拡散RGBA値を設定できます．その際はglMaterialfvを使います．
  - 「pname」に「GL_SPECULAR」を指定すると，「params」でfloat型の配列を指定することで，材質の鏡面RGBA値を設定できます．その際はglMaterialfvを使います．
  - 「pname」に「GL_SHININESS」を指定すると，「param」でfloat型の数値を指定することで，鏡面係数を設定できます．その際はglMaterialfを使います．
- void glLightfv(GLenum light, GLenum pname, GLfloat *params)
  - 光源の属性と位置などを設定します
  - lightには設定する光源の番号に応じてGL_LIGHT0～GL_LIGHT7のいずれかを指定します
  - pnameにGL_POSITIONを指定すると，paramsでfloat型の配列を指定することで，光源の位置(x,y,z,w)を設定できます．点光源はw=1として(x,y,z)で座標を指定します．平行光源はw=0として(x,y,z)で方向ベクトルを指定します．
  - pnameにGL_DIFFUSEを指定すると，paramsでfloat型の配列を指定することで，拡散光のRGBA値を設定できます．
  - pnameにGL_SPECULARを指定すると，paramsでfloat型の配列を指定することで，鏡面光のRGBA値を設定できます．

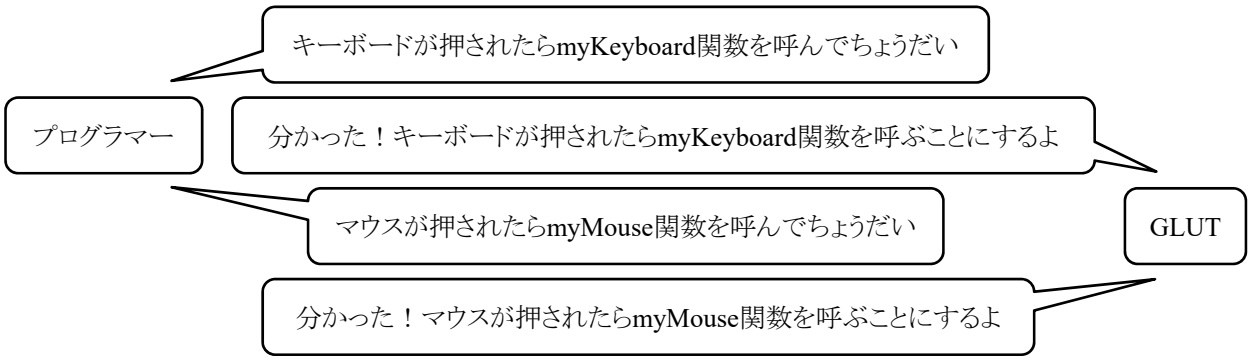
OpenGL

- void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks)
  - GLUTが提供する3D基本オブジェクトの1つ. 球. 球の中心は, 初期値では原点にある.
  - radius 球の半径
  - slices 緯度方向の分割数
  - stacks 経度方向の分割数

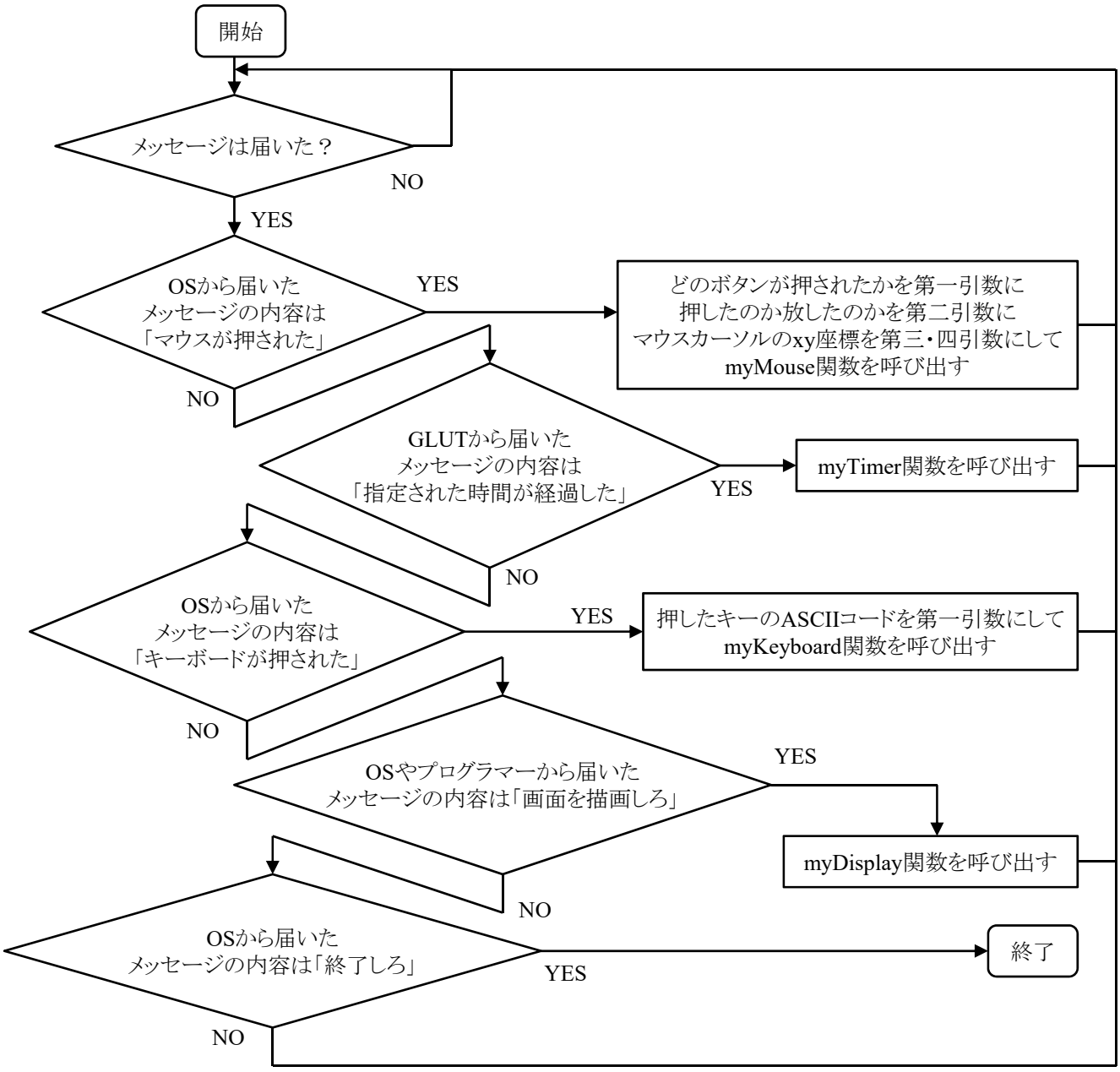
イベントドリブン型プログラミング

- glutMainLoopは無限ループしているだけで，メッセージが届くのをひたすら待ち続けます
- メッセージを受け取ったら，メッセージに応じた処理をして，また再び無限に待機します

コールバック関数の指定



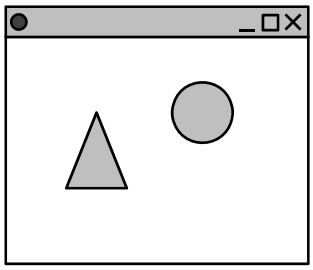
ループ中の動作の例



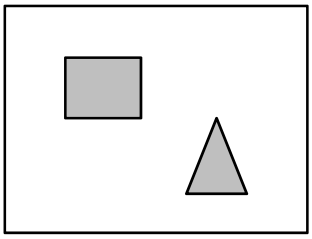
画面の描画

- glClearで画面を消去してglutSwapBuffersで描画します

描画の流れ

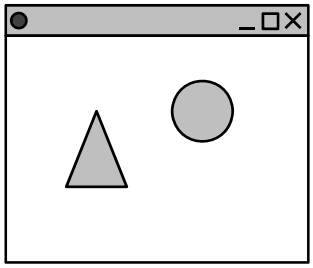


ウィンドウ表示用のバッファ



メモリ内にあるバッファ

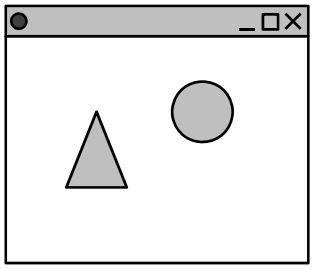
glClear(GL_COLOR_BUFFER_BIT);
画面消去



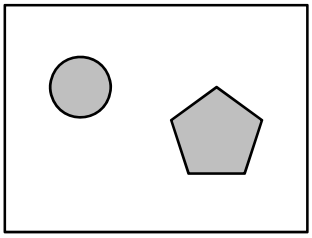
ウィンドウ表示用のバッファ

メモリ内にあるバッファ

glBegin～glEnd
図形の描画

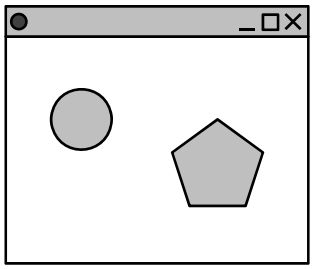


ウィンドウ表示用のバッファ

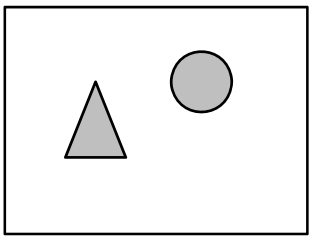


メモリ内にあるバッファ

glutSwapBuffers();
ウィンドウに表示



ウィンドウ表示用のバッファ



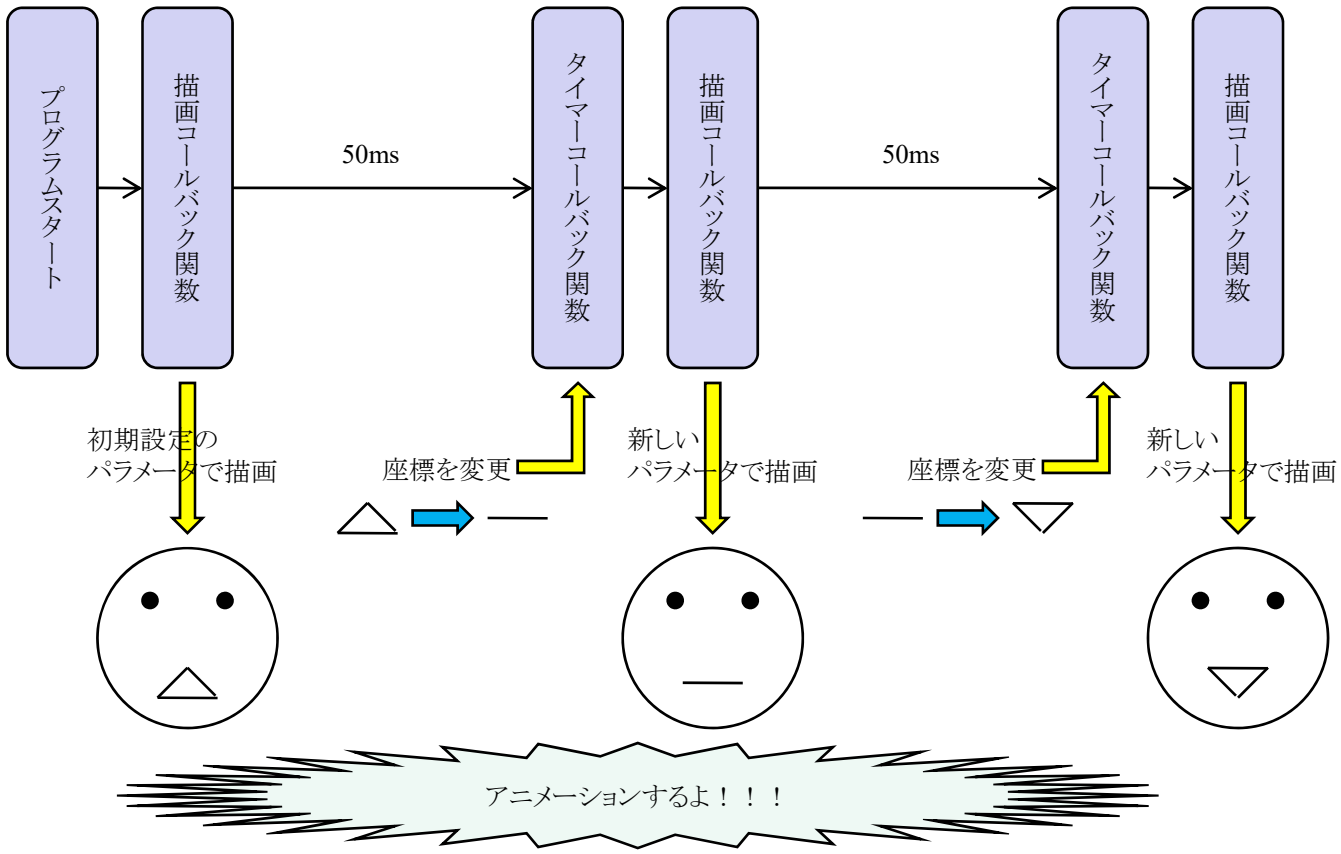
メモリ内にあるバッファ

```
void ディスプレイコールバック関数()
{
 // 画面消去
 glClear(GL_COLOR_BUFFER_BIT);

 // 図形の描画
 glBegin(GL_QUADS);
 glVertex3d(-1.0, -1.0, 0.0);
 glVertex3d(1.0, -1.0, 0.0);
 glVertex3d(1.0, 1.0, 0.0);
 glVertex3d(-1.0, 1.0, 0.0);
 glEnd();

 // ウィンドウに表示
 glutSwapBuffers();
}
```

タイマー

- void glutTimerFunc(unsigned int msecs, void (*func)(int value), int value)
  - 指定された時間に呼び出されるコールバック関数を登録します．異なる時間のコールバック関数を複数用意できます．
  - 「msecs」は呼び出される時間をミリ秒で指定します．少なくとも「msecs」ミリ秒後にコールされるようになります．
  - 第3引数の「value」は登録したタイマーコールバック関数に渡されます．
- void glutPostRedisplay(void)
  - ウィンドウ内の再描画を行います．より正確には，現在のウィンドウをマークして，「display関数」を呼び出します．

```
glutTimerFunc(50,myTimer,1); // 50ミリ秒後にmyTimer関数を呼べ
void myTimer(int value) {
    // 描画オブジェクトの座標や各種パラメータを変更する処理を行う
    glutTimerFunc(50,myTimer,1); // 再び50ミリ秒後にmyTimer関数を呼べ
    glutPostRedisplay(); // ウィンドウの内容を再描画しろ（新しい座標やパラメータで描画）
}
```



点，線，ポリゴンの描画

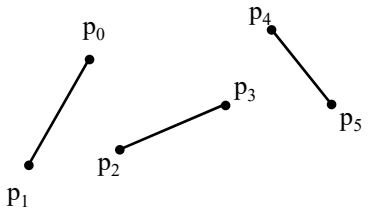
- 点，線，ポリゴンを描くのに，次のようにglBegin()とglEnd()および，glVertex*()を用いる．
- glBegin(mode);
  glVertex*(p0);
  glVertex*(p1);
  .....
  glVertex*(pn);
  glEnd();
- modeは描く図形の種類を指定し，p0，p1，…，pnは座標位置を意味する．

modeの種類

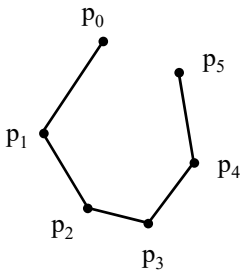
- GL_LINE_STRIP　最初の頂点から最後の頂点まで線分を連結して描画する．
- GL_LINES　二つの頂点を結んだ直線を生成する．
- GL_TRIANGLES　三つ一組の頂点を，それぞれ独立した三角形として描画する．
- GL_QUADS　四つ一組の頂点を，それぞれ独立した四角形として描画する．
- GL_POLYGON　単独の凸ポリゴンを描画する．

ポリゴン描写の注意点

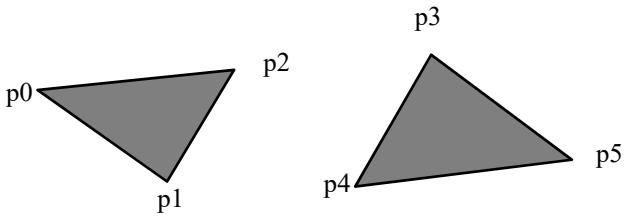
- ポリゴン（polygon）とは多角形の意味
- 頂点座標は反時計回りに設定すること
  - ポリゴンの表面と裏面を区別するため
  - 視点から見て頂点座標が反時計回りに配置されているポリゴンは表面，時計回りの場合は裏面と約束されている



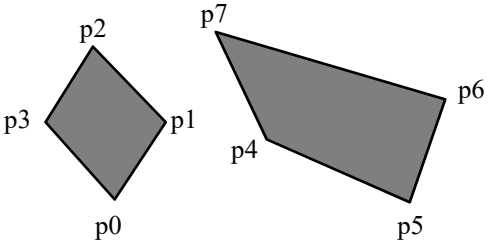
GL_LINES



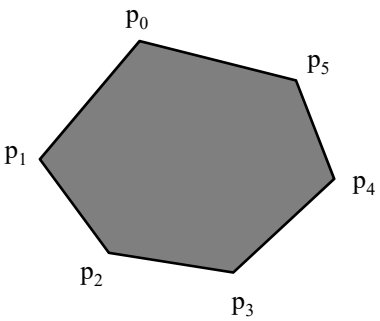
GL_LINE_STRIP



GL_TRIANGLES



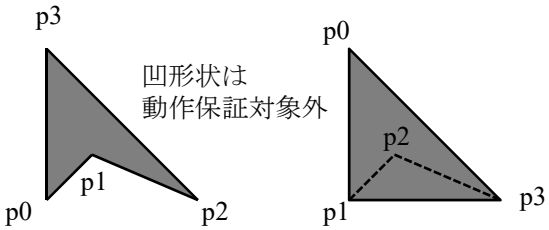
GL_QUADS



GL_POLYGON

四角形の頂点の順番

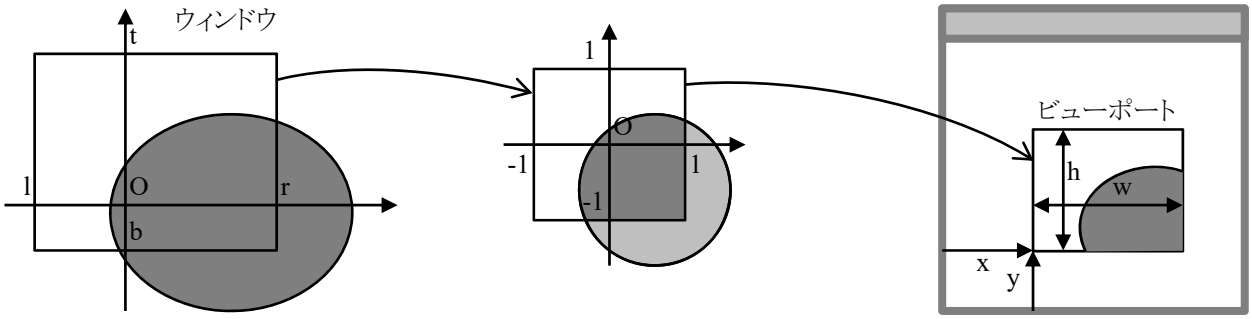
- 4つの頂点の順番によって表示される図形が異なる



- 右の例は，p0→p1→p2→p3の順番だからおかしくなるのであって，p0→p2→p1→p3の順番でglVertex?d関数を呼び出せば，正しく描画される

- OpenGLの仕様とは異なる使い方をした場合（この場合，凸多角形にしか対応していないという仕様を無視して，非凸多角形を描画しようとした場合）の動作は保証されない．
  http://monobook.org/wiki/OpenGL
- 通常，凹多角形は三角形に分割して，凸多角形で表現するのが普通（検索キーワード：三角形分割）

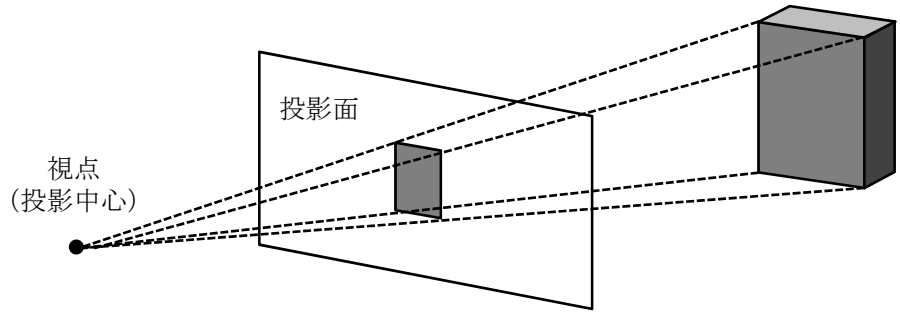
ウィンドウとビューポート



ワールド座標系　　正規化デバイス座標系　　デバイス座標系（ウィンドウ）
（左下が原点）

透視投影 (perspective projection)

- 視点と投影面を指定
- 人間の見え方に近い



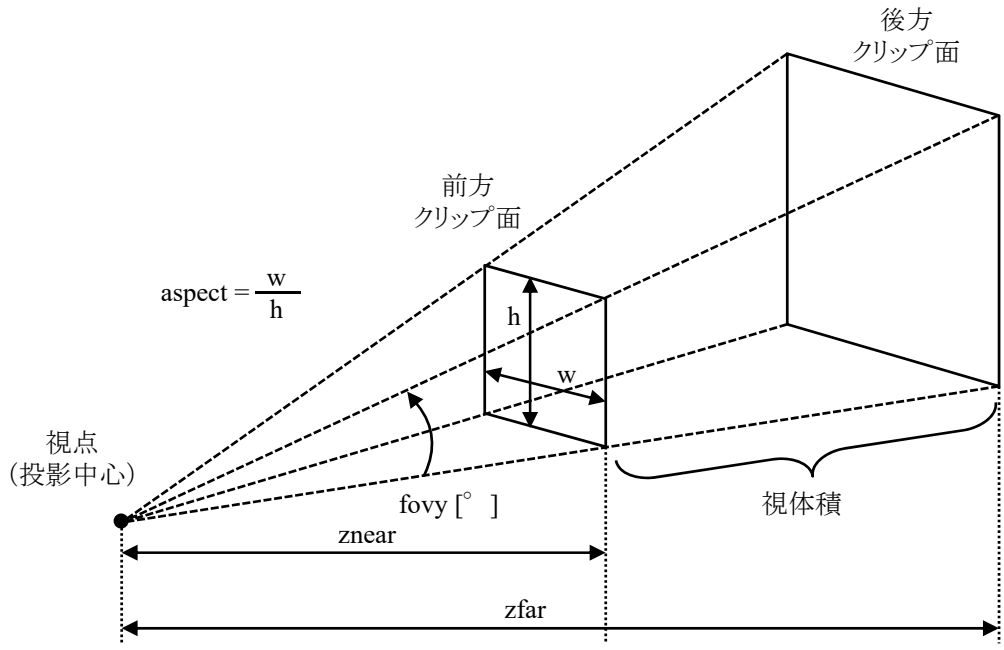
視体積

- オブジェクトが見える領域を視体積(view volume)という
- この錐台の外にあるオブジェクトは表示されない

関数

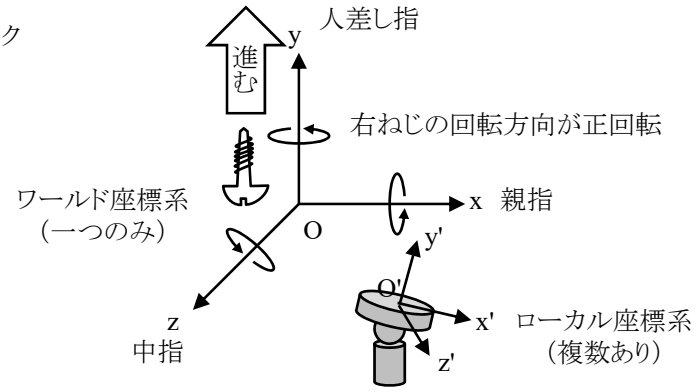
- void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)
  - 変換行列に透視変換の行列を乗じます．
  - 最初の引数「fovy」はカメラの画角であり，「度」で表します．これが大きいほど広角（透視が強くなり，絵が小さくなります）になり，小さいほど望遠レンズになります．
  - 2つ目の引数「aspect」は画面のアスペクト比（縦横比）であり，「1」であればビューポートに表示される図形のx方向とy方向のスケールが等しくなります．
  - 3つ目の引数「zNear」と4つ目の引数「zFar」は表示する奥行き方向の範囲で，「zNear」は手前（前方面），zFarは後方（後方面）の位置を示します．この間にある図形が描画されます．

透視投影の視体積



右手系

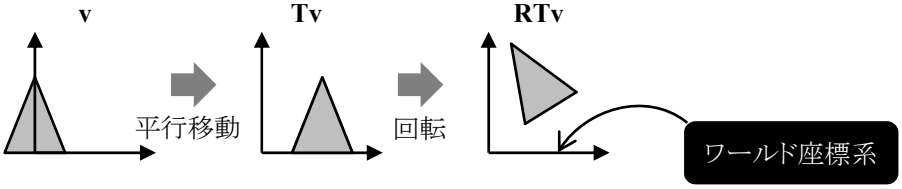
- ワールド座標系：ユーザが描く図形やオブジェクト全てに対して共通な座標系である
- ローカル座標系：各オブジェクト固有の座標系であり，ワールド座標系の中に複数存在する



モデリング変換

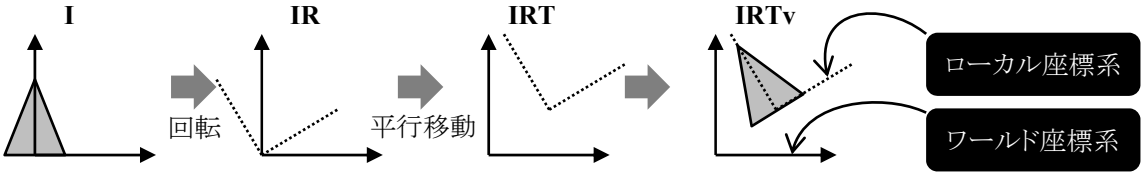
- OpenGLは，オブジェクトの平行移動(translation)，回転(rotation)，スケーリング(scaling)の三つのモデリング変換を用意している
- OpenGLは，2次元や3次元の頂点を扱うが，内部では統一的に4次元ベクトル(x, y, z, w)を用いる．これを同次座標という．
- OpenGLでは，点$\mathbf{v}=(x\ y\ z\ 1)^{\mathrm{T}}$を点$\mathbf{v}'=(x'\ y'\ z'\ 1)^{\mathrm{T}}$に変換する一般式は$\mathbf{v}'=\mathbf{T}\mathbf{v}$で表される
- $\mathbf{T}$は4×4の行列で，変換行列を表す
- 変換行列には，モデルビュー行列(model view matrix)と投影行列(projection matrix)がある
- OpenGLでは，ローカル座標系の考え方をすると，プログラムの処理順序に合う

ワールド座標系の考え方




```
glLoadIdentity();
glRotate*();
glTranslate*();
glWireCone();
```

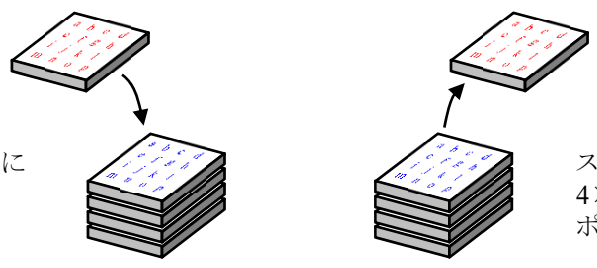
ローカル座標系の考え方



スタック

- OpenGLでは，変換行列をスタック領域に格納することができる

PUSH（プッシュ）　POP（ポップ）



スタック（積み重ねたもの）に
4×4行列を
プッシュ（押し込む）する

スタック（積み重ねたもの）から
4×4行列を
ポップ（引き出す）する

平行移動

glVertex?d関数（頂点の座標を変更する場合）

- glVertex?d関数で頂点の座標を指定できるので，この数値を変えることで図形を移動できます

例1
```
glBegin(GL_QUADS);
glVertex3d(posx-0.5, posy-0.5, 0.1);
glVertex3d(posx+0.5, posy-0.5, 0.1);
glVertex3d(posx+0.5, posy+0.5, 0.1);
glVertex3d(posx-0.5, posy+0.5, 0.l);
glEnd();
```

glTranslated関数（座標系を変更する場合）

- glTranslated関数でローカル座標系を平行移動できるので，ローカル座標系を移動させたあと，図形の描画命令を書けばいいわけです
  - ローカル座標系から見れば図形は移動していませんが，ワールド座標系から見れば図形が移動しているかのように見えます

例1
```
glPushMatrix();
glTranslated(posx, posy, 0.5);
glutSolidSphere(1.0, 10, 10);
glPopMatrix();
```

例2
```
glPushMatrix();
glTranslated(posx, posy, 1.9);
myHuman(timestep, cycle);
glPopMatrix();
```

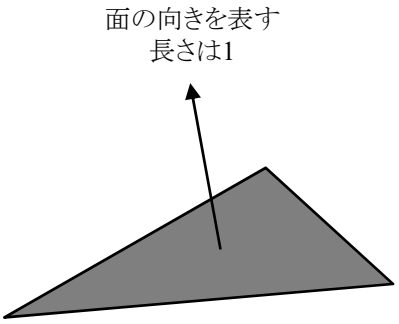
例3
```
glPushMatrix();
glTranslated(posx, posy, 0.1);
glBegin(GL_QUADS);
glVertex3d(-0.5, -0.5, 0.0);
glVertex3d(+0.5, -0.5, 0.0);
glVertex3d(+0.5, +0.5, 0.0);
glVertex3d(-0.5, +0.5, 0.0);
glEnd();
glPopMatrix();
```

法線

- 法線は，面に垂直な方向を指しているベクトルです．OpenGLでは，各ポリゴン，または各頂点に対して法線を指定できます．
- 現在の法線はglNormal*()で設定します．それ以降に呼び出されたglVertex*()で指定した頂点に現在の法線が割り当てられます．

用例

```
glBegin(GL_POLYGON);
   glNormal3d(n0x, n0y, n0z);
   glVertex3d(v0x, v0y, v0z);
   glNormal3d(n1x, n1y, n1z);
   glVertex3d(v1x, v1y, v1z);
   glNormal3d(n2x, n2y, n2z);
   glVertex3d(v2x, v2y, v2z);
   glNormal3d(n3x, n3y, n3z);
   glVertex3d(v3x, v3y, v3z);
glEnd();
```

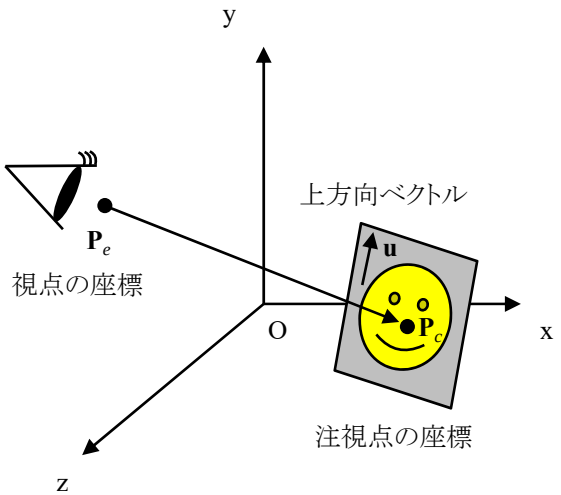


関数の説明

- void glNormal3d(GLdouble nx, GLdouble ny, GLdouble nz);
  - 法線ベクトルを設定します．

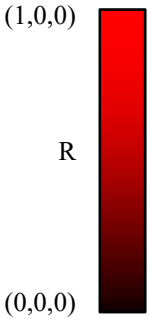
視点変換

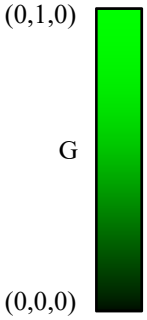
- void gluLookAt(GLdouble ex, GLdouble ey, GLdouble ez, GLdouble cx, GLdouble cy, GLdouble cz, GLdouble ux, GLdouble uy, GLdouble uz)
  - この最初の3つの引数「ex, ey, ez」は視点の位置，次の3つの引数「cx, cy, cz」は目標の位置，最後の3つの引数「ux, uy, uz」は，ウィンドウに表示される画像の「上」の方向を示すベクトルです．

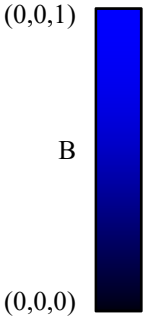


色の表現

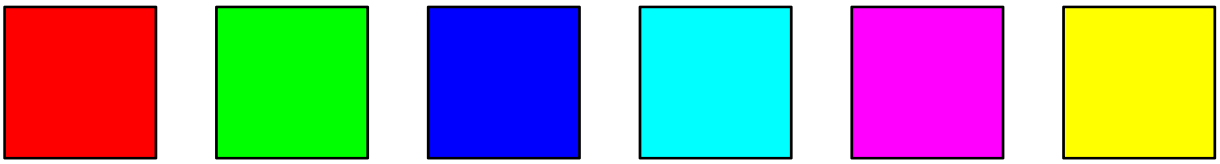
- 光の三原色で表す
- R: Red 赤, G: Green 緑, B: Blue 青
- glClearColor関数で背景色を指定，glColor3d関数で描画する図形の色を指定します
- OpenGLのこれらの関数の引数は，浮動小数点の0～1の値で表します
  - ちなみに，画面や画像ファイルなどの画素は通常，RGBそれぞれ8bitずつ，合計24bitで表します．整数で0～255の値で表します．OpenGLは0～1ですのでご注意ください．
- 0が暗くて，1が明るいです







(R,G,B)=(0,0,0)　　(R,G,B)=(0.5,0.5,0.5)　　(R,G,B)=(1,1,1)



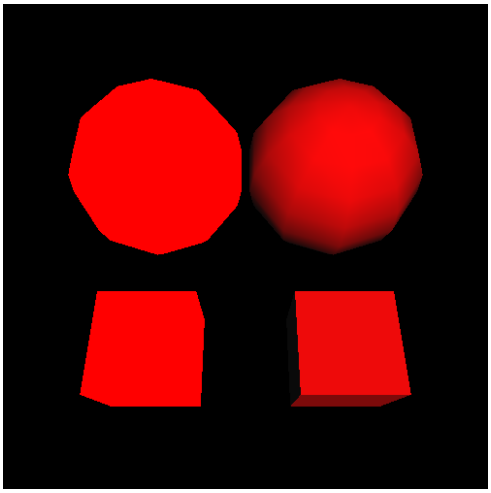
(1,0,0)　(0,1,0)　(0,0,1)　(0,1,1)　(1,0,1)　(1,1,0)

(1,0,0)　(1,0.5,0.5)　(1,1,1)　　(0,1,0)　(0.5,1,0.5)　(1,1,1)　　(0,0,1)　(0.5,0.5,1)　(1,1,1)

(1,0,0)　(1,1,0)　(0,1,0)　(0,1,1)　(0,0,1)　(1,0,1)　(1,0,0)

図形の色

→ライティングがオフの場合



←ライティングがオンの場合

自分が今作っているプログラムはライティングがオンの状態ですか？オフの状態ですか？ソースコードをよく理解して適切な方法で色を設定してください．

glColor3d関数（ライティングしない場合）

- 初期状態ではglDisable(GL_LIGHTING)，すなわち，ライティング計算をしない状態になっています．
- 陰影は計算されません．
- 図形の色はglColor3d関数で指定します．

例1
```
glColor3d(1.0, 0.0, 0.0);
glutSolidSphere(1.0, 10, 10);
```

例2
```
glColor3d(1.0, 0.0, 0.0);
glBegin(GL_QUADS);
glVertex3d(-1.0, -1.0, 0.0);
glVertex3d(1.0, -1.0, 0.0);
glVertex3d(1.0, 1.0, 0.0);
glVertex3d(-1.0, 1.0, 0.0);
glEnd();
```

glMaterialfv関数（ライティングする場合）

- glEnable(GL_LIGHTING)で，ライティング計算をする状態になります．
- glEnable(GL_LIGHT0)などで，光源を点灯する必要があります．
- 光源と法線の関係を使って陰影が計算されます．
- 図形には法線が正しく設定されていなければいけません．
- 図形の色はglMaterialfv関数で指定します．

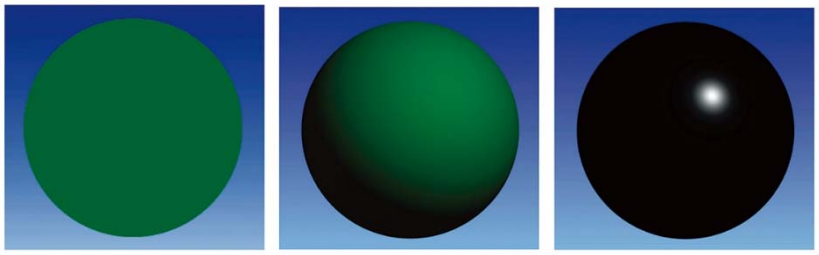
例1
```
float red[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glMaterialfv(GL_FRONT, GL_DIFFUSE, red);
glutSolidSphere(1.0, 10, 10);
```

例2
```
float red[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glMaterialfv(GL_FRONT, GL_DIFFUSE, red);
glBegin(GL_QUADS);
glNormal3d(0.0, 0.0, 1.0);
glVertex3d(-1.0, -1.0, 0.0);
glVertex3d(1.0, -1.0, 0.0);
glVertex3d(1.0, 1.0, 0.0);
glVertex3d(-1.0, 1.0, 0.0);
glEnd();
```

光源の種類

- 環境光(ambient)
  - 光の方向が特定できないもので，あらゆる方向から発しているように見える．物体への光源は，光源からの直接光の他に，空気による散乱光や他の物体表面からの2次反射，3次反射により，あらゆる方向から弱く照らされている．このため，オブジェクトの影の部分もある程度明るく見える．環境光反射成分を考慮しないと，宇宙空間で撮影した写真のように影の部分が真っ黒となる．環境光が面に当たると，全方向に均等に散乱する．
  - OpenGLでは，環境光を厳密に計算すると膨大になりすぎるため，一定の値でそのオブジェクトを照らすことにして，計算量の低減を図っている．
- 拡散光(diffuse)
  - 特定の位置・方向から来る光で，光源の成分としても使われる．しかし，一度面に当たると全方向に均等に散乱するため，布地や人肌のようなオブジェクト表面のざらつき度を表現することができる．特定の位置・方向から来る光は全て拡散光を有する．
- 鏡面光(specular)
  - 特定の方向から来る光で，面での反射は特定の方向に向かう．高品質の鏡に反射するレーザービームはほとんど鏡面反射となる．磨かれた金属・プラスチックは鏡面光の成分が大きく，チョーク・絨毯などはほとんど持たない．鏡面度は光沢の程度と理解しても差し支えない．

■図 4.23——環境光による反射，拡散反射，鏡面反射の各成分



[a] 環境光　[b] 拡散反射光　[c] 鏡面反射光

「コンピュータグラフィックス　改訂新版」2015年／公益財団法人 画像情報教育振興協会(CG-ARTS協会)

光源の有効・無効化

- glEnable(GL_LIGHTING);
  - 光源による陰影を計算する機能をオンにする
- glEnable(GL_LIGHTn);
  - 番号n=0～7の光源を点灯する
- glLightfv(GL_LIGHTn, pname, params);
  - 番号n=0～7の光源の属性を設定する
- glほにゃらら(引数);
  - 光源が点灯された状態で描画したいシーンのOpenGL命令を実行する
- glDisable(GL_LIGHTn);
  - 番号n=0～7の光源を消灯する
- glDisable(GL_LIGHTING);
  - 光源による陰影を計算する機能をオフにする

- 光源0番はデフォルトで白色光が設定されているが，光源1番以降はデフォルトで真っ暗になっているので，光源1番以降を使用する際は光源の明るさの設定を忘れずに

光源の設定

- void glLightfv(GLenum light, GLenum pname, GLfloat *params)
  - 光源の属性と位置などを設定します
  - lightには設定する光源の番号に応じてGL_LIGHT0～GL_LIGHT7のいずれかを指定します

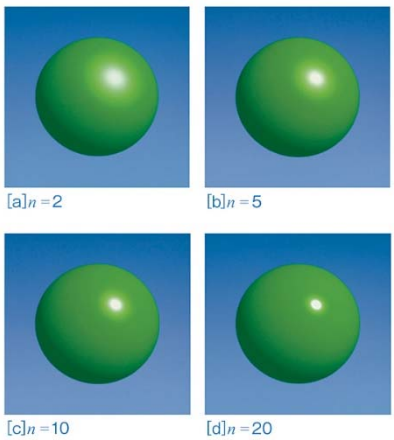
pname	params	パラメータ
GL_AMBIENT	float型の4次元配列	環境光のRGBA値
GL_DIFFUSE	float型の4次元配列	拡散光のRGBA値
GL_SPECULAR	float型の4次元配列	鏡面光のRGBA値
GL_POSITION	float型の4次元配列	光源の位置(x,y,z,w) 点光源はw=1.(x,y,z)は座標を表す 平行光源はw=0.(x,y,z)は方向ベクトルを表す

物体表面の材質の設定

- void glMaterialfv(GLenum face, GLenum pname, GLfloat *param)
  void glMaterialf(GLenum face, GLenum pname, GLfloat param)
  - 表面属性を定義する.
  - faceにGL_FRONTを指定すると，ポリゴンの表面のみに属性を設定します.
  - paramでfloat型の数値を指定する場合はglMaterialfを使い，paramでfloat型の配列を指定する場合はglMaterialfvを使う

pname	params	パラメータ
GL_AMBIENT	float型の4次元配列	材質の環境RGBA値
GL_DIFFUSE	float型の4次元配列	材質の拡散RGBA値
GL_SPECULAR	float型の4次元配列	材質の鏡面RGBA値
GL_SHININESS	float型のスカラー値	鏡面係数 0以上の値を指定する 小さい数値を指定すると鏡面反射が広がり，表面が粗い材質を表現できる 大きい数値を指定すると鏡面反射がせばまり，表面が滑らかな材質を表現できる

■図 4.33――フォンの反射モデルによるハイライトの違い



「コンピュータグラフィックス　改訂新版」2015年／公益財団法人 画像情報教育振興協会(CG-ARTS協会)

位置と速度

- 単位時間あたりで更新

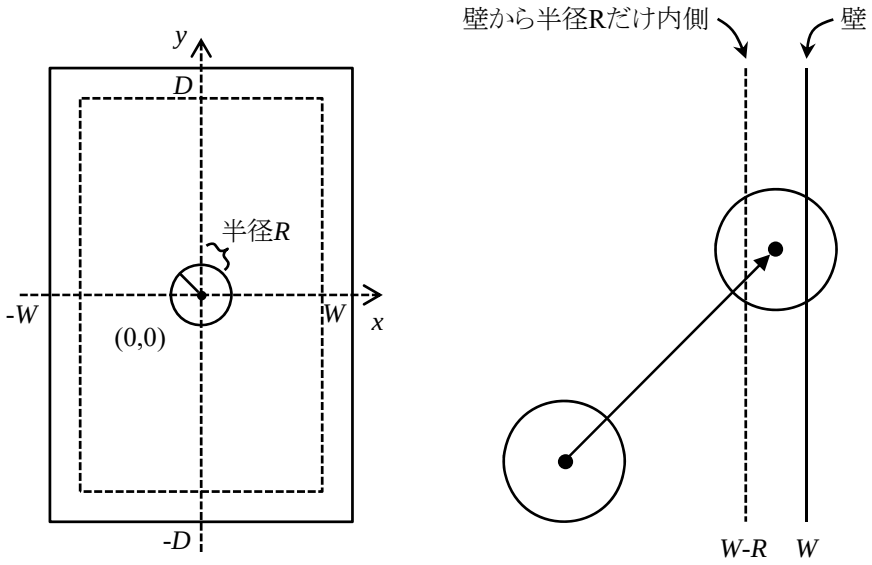
$$\frac{d\mathbf{p}}{dt} = \mathbf{v} \quad \Longrightarrow \quad \mathbf{p}(t+\Delta t) - \mathbf{p}(t) = \mathbf{v}(t)\Delta t \quad \Longrightarrow \quad \mathbf{p} \leftarrow \mathbf{p} + \mathbf{v}$$

離散化　　擬似ソースコード

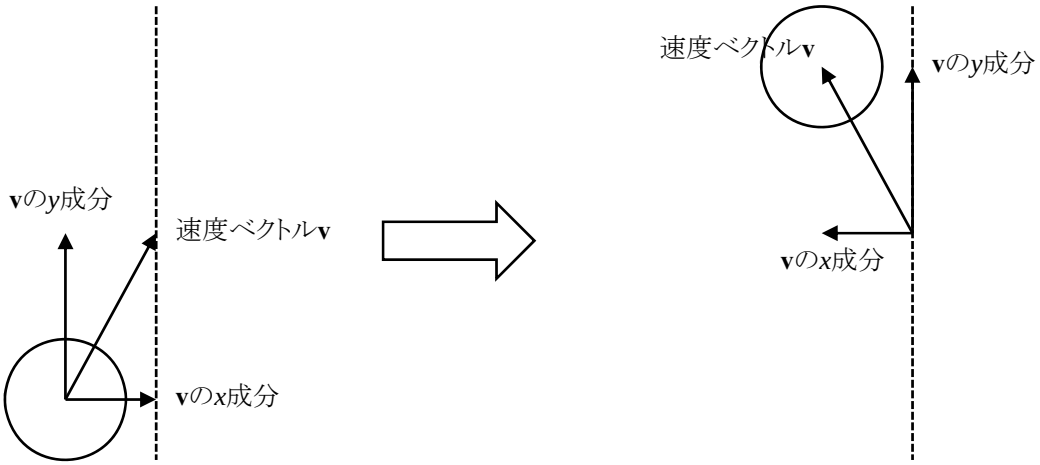
p: 位置，　v: 速度，　t: 時刻，　Δt: ある微小な時間（今回の実装では1と設定している）

壁での反射

- 位置関係は以下の通り（z座標は，変数Rで指定されている）



- 壁に垂直な速度成分の符号を反転する

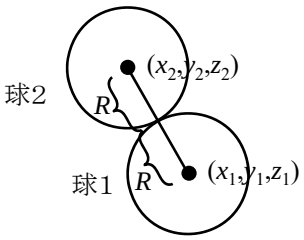


例えば，右端の壁に当たった場合
速度ベクトルのx成分だけ反転して，y成分は変更しない

球の衝突判定

- 2つの球の重心同士の距離が2R以下なら衝突したと判定する

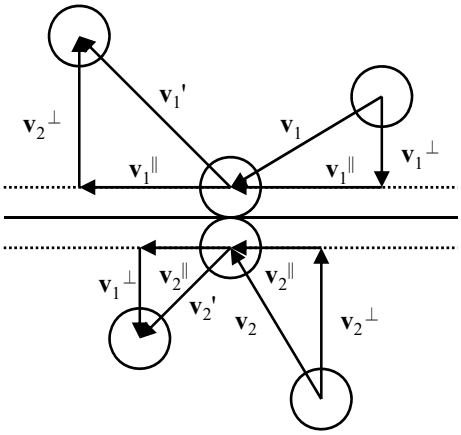
$$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2} \le 2R \quad \Longrightarrow \quad \text{衝突}$$



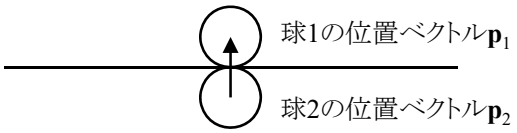
R: 半径，　(x_1,y_1,z_1): 球1の位置，　(x_2,y_2,z_2): 球2の位置
ちなみに，今回のビリヤードの場合，$(z_1-z_2)^2$は常にゼロになる

球の反射

- 球1の速度ベクトルがv1からv1'に変わり，球2の速度ベクトルがv2からv2'に変わる
- 反射面に垂直な成分を⊥記号，水平な成分を||記号で表す

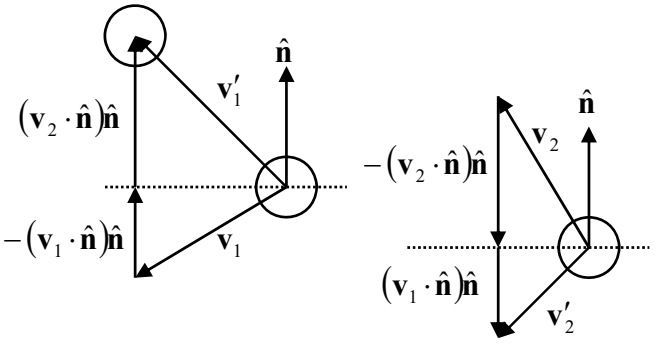


- 球1の位置ベクトルから球2の位置ベクトルを引いたベクトル，それを正規化したものが反射面の法線ベクトル



反射面の単位法線ベクトル　$\hat{\mathbf{n}} = \dfrac{\mathbf{p}_1-\mathbf{p}_2}{\|\mathbf{p}_1-\mathbf{p}_2\|}$

- 反射した球1と球2の速度ベクトルの計算は以下の通り



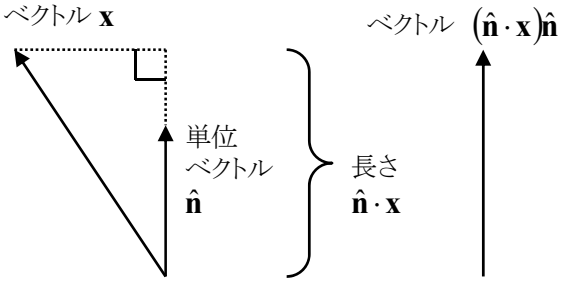
線形代数（正規化）

- ベクトル$\mathbf{x}=(x_1, x_2, x_3)$の長さを1にすることを正規化という
- ベクトル$\mathbf{x}$の長さは$\|\mathbf{x}\|$なので，その値で割り算をすると長さが1のベクトルになる
- 長さが1のベクトルを単位ベクトルという

$$\hat{\mathbf{x}} = \frac{\mathbf{x}}{\|\mathbf{x}\|} = \frac{(x_1,x_2,x_3)^T}{\sqrt{x_1^2+x_2^2+x_3^2}} = \begin{pmatrix} \dfrac{x_1}{\sqrt{x_1^2+x_2^2+x_3^2}} \\ \dfrac{x_2}{\sqrt{x_1^2+x_2^2+x_3^2}} \\ \dfrac{x_3}{\sqrt{x_1^2+x_2^2+x_3^2}} \end{pmatrix}$$

線形代数（内積）

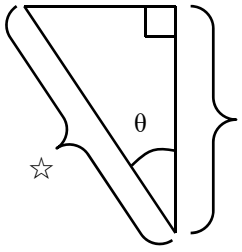
- 任意のベクトル$\mathbf{x}=(x_1, x_2, x_3)$と単位ベクトル$\mathbf{n}=(n_1, n_2, n_3)$の内積を計算すると，ベクトル$\mathbf{x}$の成分のうち$\mathbf{n}$に平行な成分を取り出すことができる



なお，なす角が鈍角の場合は長さ$\mathbf{n}\cdot\mathbf{x}$は負の値になる

内積　$\hat{\mathbf{n}}\cdot\mathbf{x} = n_1x_1+n_2x_2+n_3x_3$

- なぜなら



$\cos\theta = \dfrac{\diamondsuit}{\bigstar}$　だから

内積　$\hat{\mathbf{n}}\cdot\mathbf{x} = \|\hat{\mathbf{n}}\|\|\mathbf{x}\|\cos\theta$

単位ベクトル　$\|\hat{\mathbf{n}}\| = 1$

$\hat{\mathbf{n}}$ と $\mathbf{x}$ のなす角　θ

プロトタイプ

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <GL/glut.h>

int winw, winh;
const int delay = 15;
const int BALLNUM = 3;
const double W = 4;
const double D = 6;
const double R = 0.3;

double ballP[BALLNUM][3] = {
    { 0.0, -5.0, R },
    { -0.5,  1.0, R },
    { 1.0,  2.0, R }
};
double ballV[BALLNUM][3] = {
    { 0.0, 0.2, 0.0 },
    { 0.0, 0.0, 0.0 },
    { 0.0, 0.0, 0.0 }
};
double wallN[4][3] = {
    { 0.0,  1.0, 0.0 },
    { -1.0,  0.0, 0.0 },
    { 0.0, -1.0, 0.0 },
    { 1.0,  0.0, 0.0 }
};
double wallP[8][3] = {
    { -W, -D, 0.0 },
    { -W, -D, 0.5 },
    { W, -D, 0.0 },
    { W, -D, 0.5 },
    { W,  D, 0.0 },
    { W,  D, 0.5 },
    { -W,  D, 0.0 },
    { -W,  D, 0.5 }
};

void myGround()
{
    int i;

    glBegin(GL_QUADS);

    glNormal3d(0.0, 0.0, 1.0);
    glVertex3d(-W, -D, 0.0);
    glVertex3d(W, -D, 0.0);
    glVertex3d(W, D, 0.0);
    glVertex3d(-W, D, 0.0);

    for (i = 0; i < 4; i++) {
        glNormal3dv(wallN[i]);
        glVertex3dv(wallP[(i * 2 + 0) % 8]);
        glVertex3dv(wallP[(i * 2 + 1) % 8]);
        glVertex3dv(wallP[(i * 2 + 3) % 8]);
        glVertex3dv(wallP[(i * 2 + 2) % 8]);
    }

    glEnd();
}
```

プロトタイプ

```
void myDisplay()
{
    static float lightPos[4] = { 0.0f, 0.0f, 5.0f, 1.0f };
    int i;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (double)winw / (double)winh, 0.1, 100.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, -10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);

    myGround();
    for (i = 0; i < BALLNUM; i++) {
        glPushMatrix();
        glTranslated(ballP[i][0], ballP[i][1], ballP[i][2]);
        glutSolidSphere(R, 20, 20);
        glPopMatrix();
    }

    glutSwapBuffers();
}

void myTimer(int value)
{
    int i, j;
    // 必要とあらば変数を定義する

    if (value == 1) {
        for (i = 0; i < BALLNUM; i++) {
            //［課題］速度に応じてボールの位置を更新
            //［課題］ボールと壁が当たった場合の処理をする
        }

        for (i = 0; i < BALLNUM; i++) {
            for (j = i + 1; j < BALLNUM; j++) {
                //［課題］ボールiとボールjが当たった場合の処理をする
            }
        }

        glutTimerFunc(delay, myTimer, 1);
        glutPostRedisplay();
    }
}

void myKeyboard(unsigned char key, int x, int y)
{
    if (key == 0x1B) exit(0);
}

void myReshape(int width, int height)
{
    winw = width;
    winh = height;
    glViewport(0, 0, winw, winh);
}

void myInit(char* progname)
{
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    glutCreateWindow(progname);
}
```

プロトタイプ

```
int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    myInit(argv[0]);
    glutKeyboardFunc(myKeyboard);
    glutTimerFunc(delay, myTimer, 1);
    glutReshapeFunc(myReshape);
    glutDisplayFunc(myDisplay);
    glutMainLoop();
    return 0;
}
```

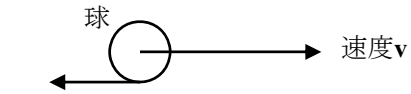
実際に実装してみたら球が壁や球に食い込むんだけど？

- ここまでの資料の通りに実装したものは球が壁や球に食い込むことがある
  - それを解決したい人は次ページ以降の発展課題にもぜひ挑戦を

床からの摩擦

- 単位時間あたりで更新

運動方程式 $\mathbf{F} = m\mathbf{a}$　　m: 質量，　F: 力，　a: 加速度　$\mathbf{a} = \dfrac{d\mathbf{v}}{dt}$



μ: 動摩擦係数，　N: 垂直抗力，　m: 質量，　g: 重力加速度

$F = -\mu N = -\mu mg$
動摩擦力

ベクトルの向きを考慮すると動摩擦力は　$\mathbf{F} = -\mu mg \dfrac{\mathbf{v}}{\|\mathbf{v}\|}$

$$\Longrightarrow m\frac{d\mathbf{v}}{dt} = -\mu mg\frac{\mathbf{v}}{\|\mathbf{v}\|} \Longrightarrow \mathbf{v}(t+\Delta t) - \mathbf{v}(t) = -\mu g\frac{\mathbf{v}(t)}{\|\mathbf{v}(t)\|}\Delta t \Longrightarrow \mathbf{v} \leftarrow \mathbf{v} - \tilde{\mu}\frac{\mathbf{v}}{\|\mathbf{v}\|}$$

離散化　　擬似ソースコード

比例定数　$\tilde{\mu} \equiv \mu g$

- 上記の漸化式で速度ベクトルを更新すればいい
- 動いている物体は摩擦が生じるが，静止した物体には上記の漸化式は適用しない

$$\begin{cases} \mathbf{v} \leftarrow \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \left(\|\mathbf{v}\| \le \tilde{\mu} \quad \text{の場合} \right) \\ \mathbf{v} \leftarrow \mathbf{v} - \tilde{\mu}\dfrac{\mathbf{v}}{\|\mathbf{v}\|} & \left(\|\mathbf{v}\| > \tilde{\mu} \quad \text{の場合} \right) \end{cases}$$

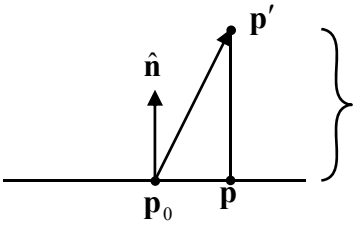
線形代数（平面）

- $\mathbf{p}_0$を含み，$\mathbf{n}$に垂直な平面の方程式

$$(\mathbf{p} - \mathbf{p}_0)\cdot\hat{\mathbf{n}} = 0$$

平面上の点は$\mathbf{p}$で表される

- $\mathbf{p}'$に最も近い平面上の点$\mathbf{p}$までの距離を計算する方法

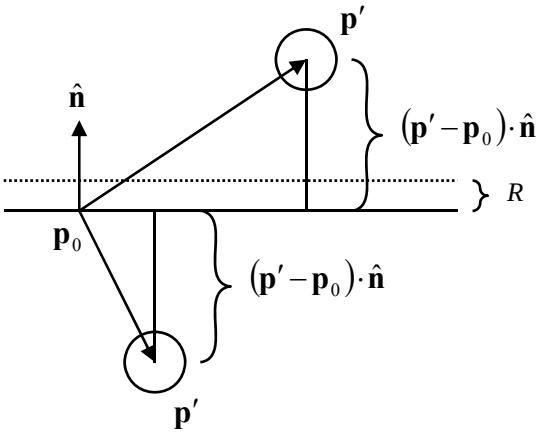


長さ　$(\mathbf{p}' - \mathbf{p}_0)\cdot\hat{\mathbf{n}}$　　法線$\mathbf{n}$と逆方向の場合は負の値になる

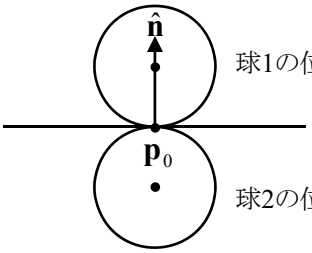
ちなみにその点$\mathbf{p}$の座標は　$\mathbf{p} = \mathbf{p}' - ((\mathbf{p}' - \mathbf{p}_0)\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}$

平面との衝突判定

- 平面との符号つき距離を求めて，その距離がR以下ならば衝突している，と判定する



- 任意の向きの壁との衝突判定に使える
- 接平面など，反射面が分かる状況であれば，衝突判定に使える
- 2つの球の場合，以下のような平面を使えばいい



球1の位置ベクトル$\mathbf{p}_1$

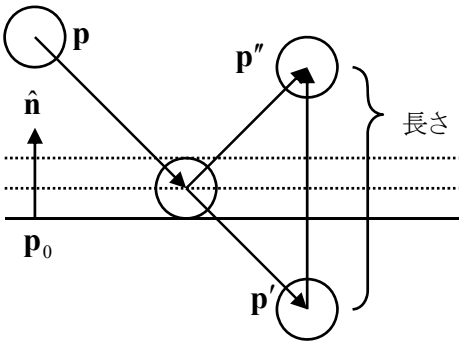
球2の位置ベクトル$\mathbf{p}_2$

反射面の単位法線ベクトル $\hat{\mathbf{n}}=\dfrac{\mathbf{p}_1-\mathbf{p}_2}{\|\mathbf{p}_1-\mathbf{p}_2\|}$

反射面に含まれる点 $\mathbf{p}_0=\dfrac{\mathbf{p}_1+\mathbf{p}_2}{2}$

平面への食い込み

- 平面の内側に食い込んだ球を平面の外側に戻す方法
- 以下ははねかえり係数が1の壁の場合の計算方法
- 球についても同様の計算が可能



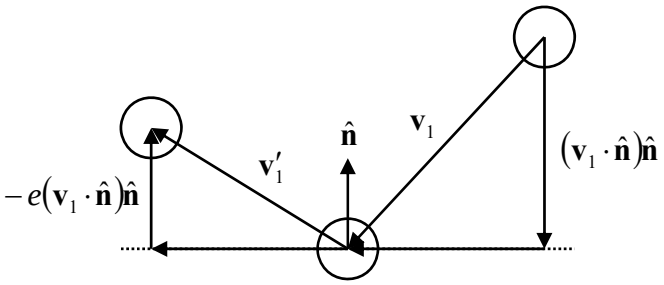
長さ $-2\left((\mathbf{p}'-\mathbf{p}_0)\cdot\hat{\mathbf{n}}\right)+2R$

点$\mathbf{p}'$を点$\mathbf{p}''$にすることで反射後の位置が計算できる

$$\mathbf{p}''=\mathbf{p}'+\left\{-2\left((\mathbf{p}'-\mathbf{p}_0)\cdot\hat{\mathbf{n}}\right)+2R\right\}\hat{\mathbf{n}}$$

壁との反射

- 壁との反射で速度が減衰する場合



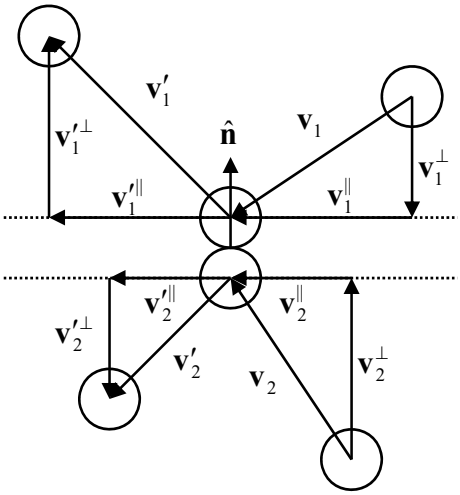
e: はねかえり係数（反発係数）（0～1の値）

反射後の速度

$$\mathbf{v}_1' = \mathbf{v}_1 - (\mathbf{v}_1\cdot\hat{\mathbf{n}})\hat{\mathbf{n}} - e(\mathbf{v}_1\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}$$

球の反射

- はねかえり係数が1以外でも対応させる場合



反射面に水平な成分は変えない

$$\mathbf{v}_1'^{\parallel} = \mathbf{v}_1^{\parallel} \qquad \mathbf{v}_2'^{\parallel} = \mathbf{v}_2^{\parallel}$$

v_1, v_2, v'_1, v'_2: 符号つきスカラー値を以下のように定義

$$\mathbf{v}_1^{\perp} = v_1\hat{\mathbf{n}} \qquad \mathbf{v}_2^{\perp} = v_2\hat{\mathbf{n}} \qquad \mathbf{v}_1'^{\perp} = v_1'\hat{\mathbf{n}} \qquad \mathbf{v}_2'^{\perp} = v_2'\hat{\mathbf{n}}$$

v_1とv_2は以下のように計算できる

$$v_1 = \mathbf{v}_1\cdot\hat{\mathbf{n}} \qquad v_2 = \mathbf{v}_2\cdot\hat{\mathbf{n}}$$

はねかえり係数の関係式　$e = -\dfrac{v_1' - v_2'}{v_1 - v_2}$

運動量保存の法則　$mv_1 + mv_2 = mv_1' + mv_2'$

これを解くと　$v_1' = \dfrac{1}{2}(1-e)v_1 + \dfrac{1}{2}(1+e)v_2$　　$v_2' = \dfrac{1}{2}(1+e)v_1 + \dfrac{1}{2}(1-e)v_2$

よって

$$\mathbf{v}_1' = \mathbf{v}_1 - (\mathbf{v}_1\cdot\hat{\mathbf{n}})\hat{\mathbf{n}} + \frac{1}{2}(1-e)(\mathbf{v}_1\cdot\hat{\mathbf{n}})\hat{\mathbf{n}} + \frac{1}{2}(1+e)(\mathbf{v}_2\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}$$

$$\mathbf{v}_2' = \mathbf{v}_2 - (\mathbf{v}_2\cdot\hat{\mathbf{n}})\hat{\mathbf{n}} + \frac{1}{2}(1-e)(\mathbf{v}_2\cdot\hat{\mathbf{n}})\hat{\mathbf{n}} + \frac{1}{2}(1+e)(\mathbf{v}_1\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}$$