学籍番号202113517

所属 情報学群情報メディア創成学類 3年

氏名 高橋 健太郎

# 課題1~3

## プログラム

#define NOMINMAX *// Windows 上で min/max のマクロを無効化するためのおまじない*

#include <stdio.h>

#include <stdlib.h>

#include <GLUT/glut.h>

#include <math.h>

#include <algorithm>

**class** Vector3d

{

**public:**

**double** x, y, z;

Vector3d() { x = y = z = 0; }

Vector3d(**double** \_x, **double** \_y, **double** \_z)

{

x = \_x;

y = \_y;

z = \_z;

}

**void** set(**double** \_x, **double** \_y, **double** \_z)

{

x = \_x;

y = \_y;

z = \_z;

}

*// 長さを1に正規化する*

**void** normalize()

{

**double** len = length();

x /= len;

y /= len;

z /= len;

}

*// 長さを返す*

**double** length() { return sqrt(x \* x + y \* y + z \* z); }

*// s倍する*

**void** scale(**const** **double** s)

{

x \*= s;

y \*= s;

z \*= s;

}

*// 加算の定義*

Vector3d operator+(Vector3d v) { return Vector3d(x + v.x, y + v.y, z + v.z); }

*// 減算の定義*

Vector3d operator-(Vector3d v) { return Vector3d(x - v.x, y - v.y, z - v.z); }

*// 内積の定義*

**double** operator\*(Vector3d v) { return x \* v.x + y \* v.y + z \* v.z; }

*// 外積の定義*

Vector3d operator%(Vector3d v) { return Vector3d(y \* v.z - z \* v.y, z \* v.x - x \* v.z, x \* v.y - y \* v.x); }

*// 代入演算の定義*

Vector3d **&**operator=(**const** Vector3d **&**v)

{

x = v.x;

y = v.y;

z = v.z;

return (\*this);

}

*// 加算代入の定義*

Vector3d **&**operator+=(**const** Vector3d **&**v)

{

x += v.x;

y += v.y;

z += v.z;

return (\*this);

}

*// 減算代入の定義*

Vector3d **&**operator-=(**const** Vector3d **&**v)

{

x -= v.x;

y -= v.y;

z -= v.z;

return (\*this);

}

*// 値を出力する*

**void** print() { printf("Vector3d(%f %f %f)\n", x, y, z); }

};

*// マイナスの符号の付いたベクトルを扱えるようにするための定義 例：b=(-a); のように記述できる*

Vector3d operator-(**const** Vector3d **&**v) { return (Vector3d(-v.x, -v.y, -v.z)); }

*// ベクトルと実数の積を扱えるようにするための定義 例： c=5\*a+2\*b; c=b\*3; のように記述できる*

Vector3d operator\*(**const** **double** **&**k, **const** Vector3d **&**v) { return (Vector3d(k \* v.x, k \* v.y, k \* v.z)); }

Vector3d operator\*(**const** Vector3d **&**v, **const** **double** **&**k) { return (Vector3d(v.x \* k, v.y \* k, v.z \* k)); }

*// ベクトルを実数で割る操作を扱えるようにするための定義 例： c=a/2.3; のように記述できる*

Vector3d operator/(**const** Vector3d **&**v, **const** **double** **&**k) { return (Vector3d(v.x / k, v.y / k, v.z / k)); }

*// 球体*

**class** Sphere

{

**public:**

Vector3d center; *// 中心座標*

**double** radius; *// 半径*

**double** cR, cG, cB; *// Red, Green, Blue 値 0.0～1.0*

Sphere(**double** x, **double** y, **double** z, **double** r,

**double** cr, **double** cg, **double** cb)

{

center.x = x;

center.y = y;

center.z = z;

radius = r;

cR = cr;

cG = cg;

cB = cb;

}

*// 点pを通り、v方向のRayとの交わりを判定する。*

*// 交点が p+tv として表せる場合の t の値を返す。交わらない場合は-1を返す*

**double** getIntersec(Vector3d **&**p, Vector3d **&**v)

{

*// A\*t^2 + B\*t + C = 0 の形で表す*

**double** A = v.x \* v.x + v.y \* v.y + v.z \* v.z;

**double** B = 2.0 \* (p.x \* v.x - v.x \* center.x +

p.y \* v.y - v.y \* center.y +

p.z \* v.z - v.z \* center.z);

**double** C = p.x \* p.x - 2 \* p.x \* center.x + center.x \* center.x +

p.y \* p.y - 2 \* p.y \* center.y + center.y \* center.y +

p.z \* p.z - 2 \* p.z \* center.z + center.z \* center.z -

radius \* radius;

**double** D = B \* B - 4 \* A \* C; *// 判別式*

if (D >= 0)

{ *// 交わる*

**double** t1 = (-B - sqrt(D)) / (2.0 \* A);

**double** t2 = (-B + sqrt(D)) / (2.0 \* A);

return t1 < t2 ? t1 : t2; *// 小さいほうのtの値を返す*

}

else

{ *// 交わらない*

return -1.0;

}

}

};

*// 板。xz平面に平行な面とする*

**class** Board

{

**public:**

**double** y; *// y座標値*

Board(**double** \_y)

{

y = \_y;

}

*// 点pを通り、v方向のRayとの交わりを判定する。*

*// 交点が p+tv として表せる場合の t の値を返す。交わらない場合は負の値を返す*

*//(viewPosition, ray)*

**double** getIntersec(Vector3d **&**p, Vector3d **&**v)

{

if (fabs(v.y) < 1.0e-10)

{

return -1; *// 水平なRayは交わらない*

}

**double** t = -1;

*// ★ここで t の値を計算する*

*// ★ただしz座標が-3000より小さいなら交わらないものとする*

Vector3d o(0, y, 0);

Vector3d n(0, -y, 0);

**double** sn = (p - o) \* n;

**double** dn = (v - p) \* n;

if (-(sn) / dn > 0)

{

t = -(sn) / (dn);

}

if (v.z <= -3000)

{

t = -1;

}

return t;

}

*// x と z の値から床の色を返す（格子模様になるように）*

Vector3d

getColorVec(**double** x, **double** z)

{

*// ★ x, z の値によって(1.0, 1.0, 0.7)または(0.6, 0.6, 0.6)のどちらかの色を返すようにする*

if (**int**(x / 100) % 2 == 0)

{

if (**int**(z / 100) % 2 == 0)

{

return Vector3d(1.0, 1.0, 0.7);

}

else

{

return Vector3d(0.6, 0.6, 0.6);

}

}

else

{

if (**int**(z / 100) % 2 == 0)

{

return Vector3d(0.6, 0.6, 0.6);

}

else

{

return Vector3d(1.0, 1.0, 0.7);

}

}

}

};

**int** halfWidth; *// 描画領域の横幅/2*

**int** halfHeight; *// 描画領域の縦幅/2*

**double** screen\_z = -1000; *// 投影面のz座標*

**double** Kd = 0.8; *// 拡散反射定数*

**double** Ks = 0.8; *// 鏡面反射定数*

**double** Iin = 0.5; *// 入射光の強さ*

**double** Ia = 0.5; *// 環境光*

Vector3d viewPosition(0, 0, 0); *// 原点=視点*

Vector3d lightDirection(-2, -4, -2); *// 入射光の進行方向*

*// レンダリングする球体*

Sphere sphere(0.0, 0.0, -1500, *// 中心座標*

150.0, *// 半径*

0.1, 0.7, 0.7); *// RGB値*

*// 球体の置かれている床*

Board board(-150); *// y座標値を -150 にする。（球と接するようにする）*

*// 2つのベクトルの成す角のcos値を計算する*

**double** getCos(Vector3d **&**v0, Vector3d **&**v1)

{

return (v0.x \* v1.x + v0.y \* v1.y + v0.z \* v1.z) / (v0.length() \* v1.length());

}

*// x, y で指定されたスクリーン座標での色(RGB)を colorVec のxyzの値に格納する*

**void** getPixelColor(**double** x, **double** y, Vector3d **&**colorVec)

{

*// 原点からスクリーン上のピクセルへ飛ばすレイの方向*

Vector3d ray(x - viewPosition.x, y - viewPosition.y, screen\_z - viewPosition.z);

ray.normalize(); *// レイの長さの正規化*

*// レイを飛ばして球と交差するか求める*

**double** t\_sphere = sphere.getIntersec(viewPosition, ray);

if (t\_sphere > 0)

{ *// 球との交点がある*

*// ★前回の課題を参考に、球体の表面の色を計算で求め、colorVecに設定する*

**double** t\_Is = 0; *// 鏡面反射光*

**double** t\_Id = 0; *// 拡散反射光*

*// ★ここで t\_Is および t\_Id の値を計算する*

Vector3d n((t\_sphere \* ray).x - sphere.center.x, (t\_sphere \* ray).y - sphere.center.y, (t\_sphere \* ray).z - sphere.center.z);

n.normalize();

if (-lightDirection \* n < 0)

{

t\_Id = 0;

}

else

{

t\_Id = Iin \* Kd \* (-lightDirection \* n);

}

**double** a = -lightDirection \* n;

Vector3d R(lightDirection + 2 \* a \* n);

t\_Is = Iin \* Ks \* (R \* -(ray - viewPosition));

**double** I = t\_Id + t\_Is + Ia;

**double** r = std::min(I \* sphere.cR, 1.0); *// 1.0 を超えないようにする*

**double** g = std::min(I \* sphere.cG, 1.0); *// 1.0 を超えないようにする*

**double** b = std::min(I \* sphere.cB, 1.0); *// 1.0 を超えないようにする*

colorVec.set(r, g, b);

return;

}

*// レイを飛ばして床と交差するか求める*

**double** t\_board = board.getIntersec(viewPosition, ray);

if (t\_board > 0)

{ *// 床との交点がある*

*// ★床の表面の色を設定する*

*// ★球の影になる場合は、RGBの値をそれぞれ0.5倍する*

**double** Is = 0; *// 鏡面反射光*

**double** Id = 0; *// 拡散反射光*

*// ★ここで Is および Id の値を計算する*

Vector3d n((t\_board \* ray).x, (t\_board \* ray).y - board.y, (t\_board \* ray).z);

n.normalize();

if (-lightDirection \* n < 0)

{

Id = 0;

}

else

{

Id = Iin \* Kd \* (-lightDirection \* n);

}

**double** a = -lightDirection \* n;

Vector3d R(lightDirection + 2 \* a \* n);

Is = Iin \* Ks \* (R \* -(ray - viewPosition));

**double** I = Id + Is + Ia;

Vector3d rgb = board.getColorVec((viewPosition + t\_board \* ray).x, (viewPosition + t\_board \* ray).z);

**double** r = std::min(I \* rgb.x, 1.0); *// 1.0 を超えないようにする*

**double** g = std::min(I \* rgb.y, 1.0); *// 1.0 を超えないようにする*

**double** b = std::min(I \* rgb.z, 1.0); *// 1.0 を超えないようにする*

Vector3d shadePosition = viewPosition + t\_board \* ray;

lightDirection.normalize();

Vector3d ray\_shade = -lightDirection;

**double** t\_shade = sphere.getIntersec(shadePosition, ray\_shade);

if (t\_shade > 0)

{

r \*= 0.5;

g \*= 0.5;

b \*= 0.5;

colorVec.set(r, g, b);

}

else

{

colorVec.set(r, g, b);

}

return;

}

*// 何とも交差しない*

colorVec.set(0, 0, 0); *// 背景色（黒）を設定する*

}

*// 描画を行う*

**void** display(**void**)

{

glClear(GL\_COLOR\_BUFFER\_BIT); *// 描画内容のクリア*

*// ピクセル単位で描画色を決定するループ処理*

for (**int** y = (-halfHeight); y <= halfHeight; y++)

{

for (**int** x = (-halfWidth); x <= halfWidth; x++)

{

Vector3d colorVec;

*// x, y 座標の色を取得する*

getPixelColor(x, y, colorVec);

*//取得した色で、描画色を設定する*

glColor3d(colorVec.x, colorVec.y, colorVec.z);

*// (x, y) の画素を描画*

glBegin(GL\_POINTS);

glVertex2i(x, y);

glEnd();

}

}

glFlush();

}

**void** resizeWindow(**int** w, **int** h)

{

h = (h == 0) ? 1 : h;

glViewport(0, 0, w, h);

halfWidth = w / 2;

halfHeight = h / 2;

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

*// ウィンドウ内の座標系設定*

glOrtho(-halfWidth, halfWidth, -halfHeight, halfHeight, 0.0, 1.0);

glMatrixMode(GL\_MODELVIEW);

}

**void** keyboard(**unsigned** **char** key, **int** x, **int** y)

{

switch (key)

{

case 27:

exit(0); */\* ESC code \*/*

}

glutPostRedisplay();

}

**int** main(**int** argc, **char** **\*\***argv)

{

lightDirection.normalize();

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(400, 400);

glutInitWindowPosition(180, 10);

glutCreateWindow(argv[0]);

glClearColor(1.0, 1.0, 1.0, 1.0);

glShadeModel(GL\_FLAT);

glutDisplayFunc(display);

glutReshapeFunc(resizeWindow);

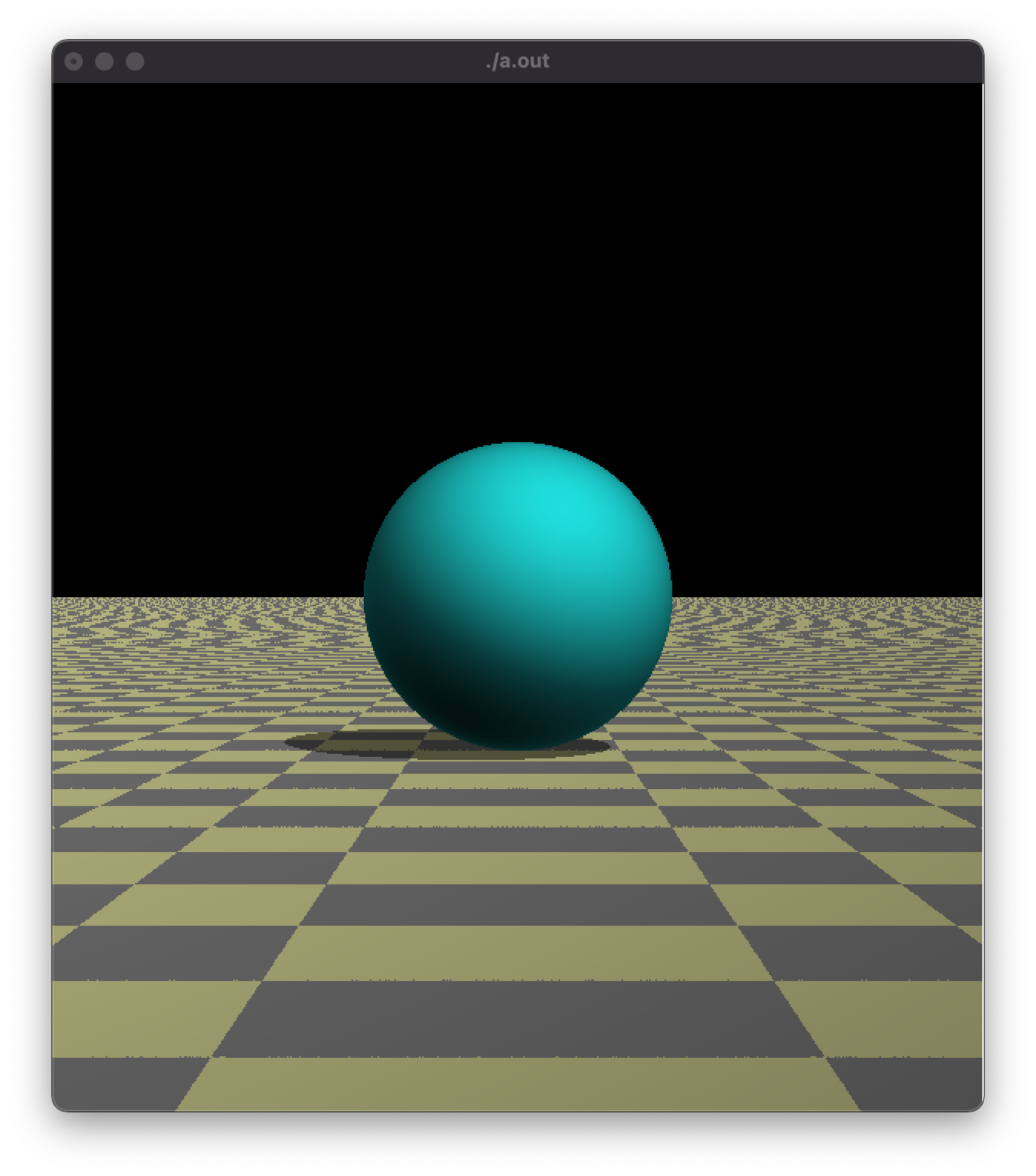
glutKeyboardFunc(keyboard);

glutMainLoop();

return 0;

}

## 実行結果



# 課題4

## プログラム

#define NOMINMAX *// Windows 上で min/max のマクロを無効化するためのおまじない*

#include <stdio.h>

#include <stdlib.h>

#include <GLUT/glut.h>

#include <math.h>

#include <algorithm>

**class** Vector3d

{

**public:**

**double** x, y, z;

Vector3d() { x = y = z = 0; }

Vector3d(**double** \_x, **double** \_y, **double** \_z)

{

x = \_x;

y = \_y;

z = \_z;

}

**void** set(**double** \_x, **double** \_y, **double** \_z)

{

x = \_x;

y = \_y;

z = \_z;

}

*// 長さを1に正規化する*

**void** normalize()

{

**double** len = length();

x /= len;

y /= len;

z /= len;

}

*// 長さを返す*

**double** length() { return sqrt(x \* x + y \* y + z \* z); }

*// s倍する*

**void** scale(**const** **double** s)

{

x \*= s;

y \*= s;

z \*= s;

}

*// 加算の定義*

Vector3d operator+(Vector3d v) { return Vector3d(x + v.x, y + v.y, z + v.z); }

*// 減算の定義*

Vector3d operator-(Vector3d v) { return Vector3d(x - v.x, y - v.y, z - v.z); }

*// 内積の定義*

**double** operator\*(Vector3d v) { return x \* v.x + y \* v.y + z \* v.z; }

*// 外積の定義*

Vector3d operator%(Vector3d v) { return Vector3d(y \* v.z - z \* v.y, z \* v.x - x \* v.z, x \* v.y - y \* v.x); }

*// 代入演算の定義*

Vector3d **&**operator=(**const** Vector3d **&**v)

{

x = v.x;

y = v.y;

z = v.z;

return (\*this);

}

*// 加算代入の定義*

Vector3d **&**operator+=(**const** Vector3d **&**v)

{

x += v.x;

y += v.y;

z += v.z;

return (\*this);

}

*// 減算代入の定義*

Vector3d **&**operator-=(**const** Vector3d **&**v)

{

x -= v.x;

y -= v.y;

z -= v.z;

return (\*this);

}

*// 値を出力する*

**void** print() { printf("Vector3d(%f %f %f)\n", x, y, z); }

};

*// マイナスの符号の付いたベクトルを扱えるようにするための定義 例：b=(-a); のように記述できる*

Vector3d operator-(**const** Vector3d **&**v) { return (Vector3d(-v.x, -v.y, -v.z)); }

*// ベクトルと実数の積を扱えるようにするための定義 例： c=5\*a+2\*b; c=b\*3; のように記述できる*

Vector3d operator\*(**const** **double** **&**k, **const** Vector3d **&**v) { return (Vector3d(k \* v.x, k \* v.y, k \* v.z)); }

Vector3d operator\*(**const** Vector3d **&**v, **const** **double** **&**k) { return (Vector3d(v.x \* k, v.y \* k, v.z \* k)); }

*// ベクトルを実数で割る操作を扱えるようにするための定義 例： c=a/2.3; のように記述できる*

Vector3d operator/(**const** Vector3d **&**v, **const** **double** **&**k) { return (Vector3d(v.x / k, v.y / k, v.z / k)); }

*// 球体*

**class** Sphere

{

**public:**

Vector3d center; *// 中心座標*

**double** radius; *// 半径*

**double** cR, cG, cB; *// Red, Green, Blue 値 0.0～1.0*

Sphere(**double** x, **double** y, **double** z, **double** r,

**double** cr, **double** cg, **double** cb)

{

center.x = x;

center.y = y;

center.z = z;

radius = r;

cR = cr;

cG = cg;

cB = cb;

}

*// 点pを通り、v方向のRayとの交わりを判定する。*

*// 交点が p+tv として表せる場合の t の値を返す。交わらない場合は-1を返す*

**double** getIntersec(Vector3d **&**p, Vector3d **&**v)

{

*// A\*t^2 + B\*t + C = 0 の形で表す*

**double** A = v.x \* v.x + v.y \* v.y + v.z \* v.z;

**double** B = 2.0 \* (p.x \* v.x - v.x \* center.x +

p.y \* v.y - v.y \* center.y +

p.z \* v.z - v.z \* center.z);

**double** C = p.x \* p.x - 2 \* p.x \* center.x + center.x \* center.x +

p.y \* p.y - 2 \* p.y \* center.y + center.y \* center.y +

p.z \* p.z - 2 \* p.z \* center.z + center.z \* center.z -

radius \* radius;

**double** D = B \* B - 4 \* A \* C; *// 判別式*

if (D >= 0)

{ *// 交わる*

**double** t1 = (-B - sqrt(D)) / (2.0 \* A);

**double** t2 = (-B + sqrt(D)) / (2.0 \* A);

return t1 < t2 ? t1 : t2; *// 小さいほうのtの値を返す*

}

else

{ *// 交わらない*

return -1.0;

}

}

};

*// 板。xz平面に平行な面とする*

**class** Board

{

**public:**

**double** y; *// y座標値*

Board(**double** \_y)

{

y = \_y;

}

*// 点pを通り、v方向のRayとの交わりを判定する。*

*// 交点が p+tv として表せる場合の t の値を返す。交わらない場合は負の値を返す*

*//(viewPosition, ray)*

**double** getIntersec(Vector3d **&**p, Vector3d **&**v)

{

if (fabs(v.y) < 1.0e-10)

{

return -1; *// 水平なRayは交わらない*

}

**double** t = -1;

*// ★ここで t の値を計算する*

*// ★ただしz座標が-3000より小さいなら交わらないものとする*

Vector3d o(0, y, 0);

Vector3d n(0, -y, 0);

**double** sn = (p - o) \* n;

**double** dn = (v - p) \* n;

if (-(sn) / dn > 0)

{

t = -(sn) / (dn);

}

if (v.z <= -3000)

{

t = -1;

}

return t;

}

*// x と z の値から床の色を返す（格子模様になるように）*

Vector3d

getColorVec(**double** x, **double** z)

{

*// ★ x, z の値によって(1.0, 1.0, 0.7)または(0.6, 0.6, 0.6)のどちらかの色を返すようにする*

if (x > 0)

{

if (**int**(x / 100) % 2 == 0)

{

if (**int**(z / 100) % 2 == 0)

{

return Vector3d(1.0, 1.0, 0.7);

}

else

{

return Vector3d(0.6, 0.6, 0.6);

}

}

else

{

if (**int**(z / 100) % 2 == 0)

{

return Vector3d(0.6, 0.6, 0.6);

}

else

{

return Vector3d(1.0, 1.0, 0.7);

}

}

}

else

{

if (**int**(x / 100) % 2 == 0)

{

if (**int**(z / 100) % 2 == 0)

{

return Vector3d(0.6, 0.6, 0.6);

}

else

{

return Vector3d(1.0, 1.0, 0.7);

}

}

else

{

if (**int**(z / 100) % 2 == 0)

{

return Vector3d(1.0, 1.0, 0.7);

}

else

{

return Vector3d(0.6, 0.6, 0.6);

}

}

}

}

};

**int** halfWidth; *// 描画領域の横幅/2*

**int** halfHeight; *// 描画領域の縦幅/2*

**double** screen\_z = -1000; *// 投影面のz座標*

**double** Kd = 0.8; *// 拡散反射定数*

**double** Ks = 0.8; *// 鏡面反射定数*

**double** Iin = 0.5; *// 入射光の強さ*

**double** Ia = 0.5; *// 環境光*

Vector3d viewPosition(0, 0, 0); *// 原点=視点*

Vector3d lightDirection(-2, -4, -2); *// 入射光の進行方向*

*// レンダリングする球体*

Sphere sphere(0.0, 0.0, -1500, *// 中心座標*

150.0, *// 半径*

0.1, 0.7, 0.7); *// RGB値*

*// 球体の置かれている床*

Board board(-150); *// y座標値を -150 にする。（球と接するようにする）*

*// 2つのベクトルの成す角のcos値を計算する*

**double** getCos(Vector3d **&**v0, Vector3d **&**v1)

{

return (v0.x \* v1.x + v0.y \* v1.y + v0.z \* v1.z) / (v0.length() \* v1.length());

}

*// x, y で指定されたスクリーン座標での色(RGB)を colorVec のxyzの値に格納する*

**void** getPixelColor(**double** x, **double** y, Vector3d **&**colorVec)

{

*// 原点からスクリーン上のピクセルへ飛ばすレイの方向*

Vector3d ray(x - viewPosition.x, y - viewPosition.y, screen\_z - viewPosition.z);

ray.normalize(); *// レイの長さの正規化*

*// レイを飛ばして球と交差するか求める*

**double** t\_sphere = sphere.getIntersec(viewPosition, ray);

if (t\_sphere > 0)

{ *// 球との交点がある*

*// ★前回の課題を参考に、球体の表面の色を計算で求め、colorVecに設定する*

**double** t\_Is = 0; *// 鏡面反射光*

**double** t\_Id = 0; *// 拡散反射光*

*// ★ここで t\_Is および t\_Id の値を計算する*

Vector3d n((t\_sphere \* ray).x - sphere.center.x, (t\_sphere \* ray).y - sphere.center.y, (t\_sphere \* ray).z - sphere.center.z);

n.normalize();

if (-lightDirection \* n < 0)

{

t\_Id = 0;

}

else

{

t\_Id = Iin \* Kd \* (-lightDirection \* n);

}

**double** a = -lightDirection \* n;

Vector3d R(lightDirection + 2 \* a \* n);

t\_Is = Iin \* Ks \* (R \* -(ray - viewPosition));

**double** I = t\_Id + t\_Is + Ia;

**double** r = std::min(I \* sphere.cR, 1.0); *// 1.0 を超えないようにする*

**double** g = std::min(I \* sphere.cG, 1.0); *// 1.0 を超えないようにする*

**double** b = std::min(I \* sphere.cB, 1.0); *// 1.0 を超えないようにする*

colorVec.set(r, g, b);

return;

}

*// レイを飛ばして床と交差するか求める*

**double** t\_board = board.getIntersec(viewPosition, ray);

if (t\_board > 0)

{ *// 床との交点がある*

*// ★床の表面の色を設定する*

*// ★球の影になる場合は、RGBの値をそれぞれ0.5倍する*

**double** Is = 0; *// 鏡面反射光*

**double** Id = 0; *// 拡散反射光*

*// ★ここで Is および Id の値を計算する*

Vector3d n((t\_board \* ray).x, (t\_board \* ray).y - board.y, (t\_board \* ray).z);

n.normalize();

if (-lightDirection \* n < 0)

{

Id = 0;

}

else

{

Id = Iin \* Kd \* (-lightDirection \* n);

}

**double** a = -lightDirection \* n;

Vector3d R(lightDirection + 2 \* a \* n);

Is = Iin \* Ks \* (R \* -(ray - viewPosition));

**double** I = Id + Is + Ia;

Vector3d rgb = board.getColorVec((viewPosition + t\_board \* ray).x, (viewPosition + t\_board \* ray).z);

**double** r = std::min(I \* rgb.x, 1.0); *// 1.0 を超えないようにする*

**double** g = std::min(I \* rgb.y, 1.0); *// 1.0 を超えないようにする*

**double** b = std::min(I \* rgb.z, 1.0); *// 1.0 を超えないようにする*

Vector3d shadePosition = viewPosition + t\_board \* ray;

lightDirection.normalize();

Vector3d ray\_shade = -lightDirection;

**double** t\_shade = sphere.getIntersec(shadePosition, ray\_shade);

if (t\_shade > 0)

{

r \*= 0.5;

g \*= 0.5;

b \*= 0.5;

colorVec.set(r, g, b);

}

else

{

colorVec.set(r, g, b);

}

return;

}

*// 何とも交差しない*

colorVec.set(0, 0, 0); *// 背景色（黒）を設定する*

}

*// 描画を行う*

**void** display(**void**)

{

glClear(GL\_COLOR\_BUFFER\_BIT); *// 描画内容のクリア*

*// ピクセル単位で描画色を決定するループ処理*

for (**int** y = (-halfHeight); y <= halfHeight; y++)

{

for (**int** x = (-halfWidth); x <= halfWidth; x++)

{

Vector3d colorVec;

**double** x\_slide;

**double** y\_slide;

**double** r = 0;

**double** g = 0;

**double** b = 0;

for (y\_slide = y - 0.5; y\_slide <= y + 0.5; y\_slide += 0.5)

{

for (x\_slide = x - 0.5; x\_slide <= x + 0.5; x\_slide += 0.5)

{

*// x, y 座標の色を取得する*

getPixelColor(x, y, colorVec);

r += colorVec.x;

g += colorVec.y;

b += colorVec.z;

}

}

r /= 9;

g /= 9;

b /= 9;

*//取得した色で、描画色を設定する*

glColor3d(r, g, b);

*// (x, y) の画素を描画*

glBegin(GL\_POINTS);

glVertex2i(x, y);

glEnd();

}

}

glFlush();

}

**void** resizeWindow(**int** w, **int** h)

{

h = (h == 0) ? 1 : h;

glViewport(0, 0, w, h);

halfWidth = w / 2;

halfHeight = h / 2;

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

*// ウィンドウ内の座標系設定*

glOrtho(-halfWidth, halfWidth, -halfHeight, halfHeight, 0.0, 1.0);

glMatrixMode(GL\_MODELVIEW);

}

**void** keyboard(**unsigned** **char** key, **int** x, **int** y)

{

switch (key)

{

case 27:

exit(0); */\* ESC code \*/*

}

glutPostRedisplay();

}

**int** main(**int** argc, **char** **\*\***argv)

{

lightDirection.normalize();

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(800, 800);

glutInitWindowPosition(180, 10);

glutCreateWindow(argv[0]);

glClearColor(1.0, 1.0, 1.0, 1.0);

glShadeModel(GL\_FLAT);

glutDisplayFunc(display);

glutReshapeFunc(resizeWindow);

glutKeyboardFunc(keyboard);

glutMainLoop();

return 0;

}

## 実行結果

バブル チャート が含まれている画像

自動的に生成された説明