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

#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> *// 小さい方の値を返す std::min 関数を使うため*

**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;

}

}

};

**int** halfWidth; *// 描画領域の横幅/2*

**int** halfHeight; *// 描画領域の縦幅/2*

*// 各種定数*

**double** d = 1000; *// 視点と投影面との距離*

**double** Kd = 0.8; *// 拡散反射定数*

**double** Ks = 0.8; *// 鏡面反射定数*

**double** Iin = 1.0; *// 入射光の強さ*

**double** Ia = 0.2; *// 環境光*

Vector3d viewPosition(0, 0, 0); *// 視点位置*

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*

0.2, 0.9, 0.9); *// RGB値*

*// 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 ray(x - viewPosition.x, y - viewPosition.y, -d - viewPosition.z); *// 原点からスクリーン上のピクセルへ飛ばすレイの方向*

ray.normalize(); *// レイの長さの正規化*

*// レイを飛ばして球との交点を求める*

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

if (t > 0)

{ *// 交点がある*

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

if (-lightDirection \* n < 0)

{

Id = 0;

}

else

{

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

}

**double** I = Id + 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*

*// 描画色の設定*

glColor3d(r, g, b);

}

else

{ *// 交点が無い*

*// 描画色を黒にする*

glColor3f(0.0f, 0.0f, 0.0f);

}

*// (x, y) の画素を描画*

glBegin(GL\_POINTS);

glVertex2i(x, y);

glEnd();

}

}

glFlush();

}

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

{

if (h < 1)

return;

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);

glutCreateWindow(argv[0]);

glClearColor(1.0, 1.0, 1.0, 1.0);

glShadeModel(GL\_FLAT);

glutDisplayFunc(display);

glutReshapeFunc(resize);

glutKeyboardFunc(keyboard);

glutMainLoop();

return 0;

}

# 実行結果

コンピューターに表示されたアイコン

中程度の精度で自動的に生成された説明

# 課題2

#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> *// 小さい方の値を返す std::min 関数を使うため*

**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;

}

}

};

**int** halfWidth; *// 描画領域の横幅/2*

**int** halfHeight; *// 描画領域の縦幅/2*

*// 各種定数*

**double** d = 1000; *// 視点と投影面との距離*

**double** Kd = 0.8; *// 拡散反射定数*

**double** Ks = 0.8; *// 鏡面反射定数*

**double** Iin = 1.0; *// 入射光の強さ*

**double** Ia = 0.2; *// 環境光*

Vector3d viewPosition(0, 0, 0); *// 視点位置*

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*

0.2, 0.9, 0.9); *// RGB値*

*// 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 ray(x - viewPosition.x, y - viewPosition.y, -d - viewPosition.z); *// 原点からスクリーン上のピクセルへ飛ばすレイの方向*

ray.normalize(); *// レイの長さの正規化*

*// レイを飛ばして球との交点を求める*

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

if (t > 0)

{ *// 交点がある*

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

**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*

*// 描画色の設定*

glColor3d(r, g, b);

}

else

{ *// 交点が無い*

*// 描画色を黒にする*

glColor3f(0.0f, 0.0f, 0.0f);

}

*// (x, y) の画素を描画*

glBegin(GL\_POINTS);

glVertex2i(x, y);

glEnd();

}

}

glFlush();

}

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

{

if (h < 1)

return;

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);

glutCreateWindow(argv[0]);

glClearColor(1.0, 1.0, 1.0, 1.0);

glShadeModel(GL\_FLAT);

glutDisplayFunc(display);

glutReshapeFunc(resize);

glutKeyboardFunc(keyboard);

glutMainLoop();

return 0;

}

# 実行結果

コンピューターの画面に表示されたアイコン

中程度の精度で自動的に生成された説明

# 課題3

入射光の進行方向を変化させると：光が当たる位置が変化する。

環境光の強さを変化させると：全体的な明るさが変化する。比例関係。

入射光の強さを変化させると：反射する部分の全体的な明るさが変化する。比例関係。

鏡面反射定数を変化させると：反射する部分の明るさが変化する。比例関係。視点に依存する。

拡散反射定数を変化させると：反射する部分の明るさが変化する。比例関係。視点に依存しない。