**FULL NAME: PHAN THI MY DUNG**

**STUDDENT ID: 127343**

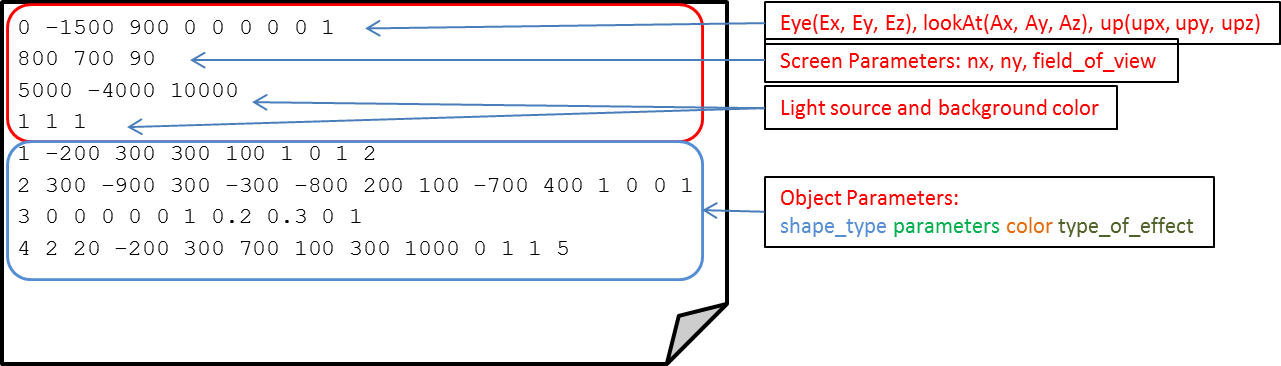
**CLASS: IMAGE SYNTHESIS THEORY**

**LECTURER: PROF. KIM JAY JONG**

**FINAL PROJECT: RAY CASTING ALGORITHM**

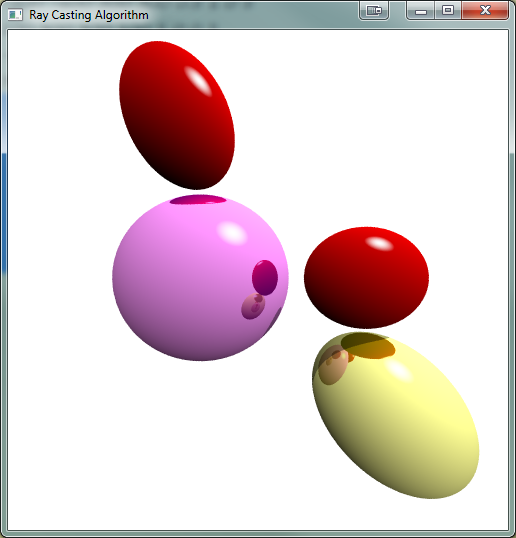
1. **Problem**

* Using ray casting algorithm to render objects including ball, plane, triangle and implicit object with some effects such as diffusion, highlight, shadow, reflection, refraction.
* Render stereo pair image.
* Apply anti-aliasing to improve image quality.

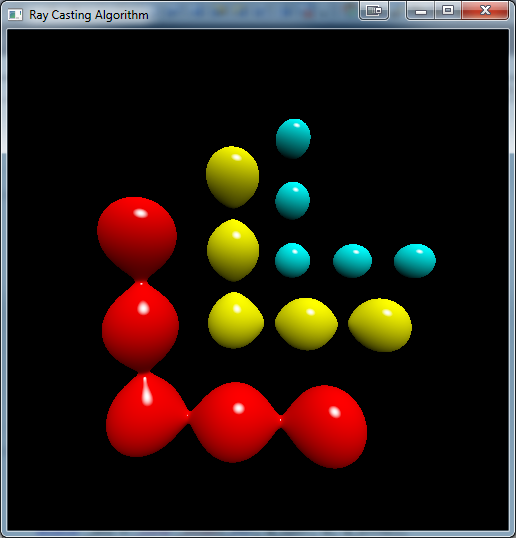
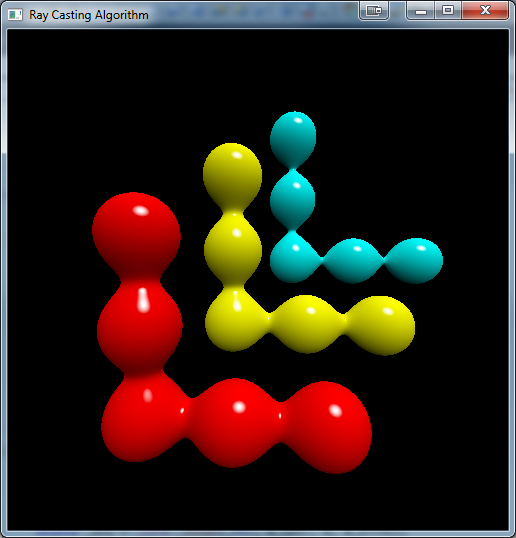
1. **Data format **

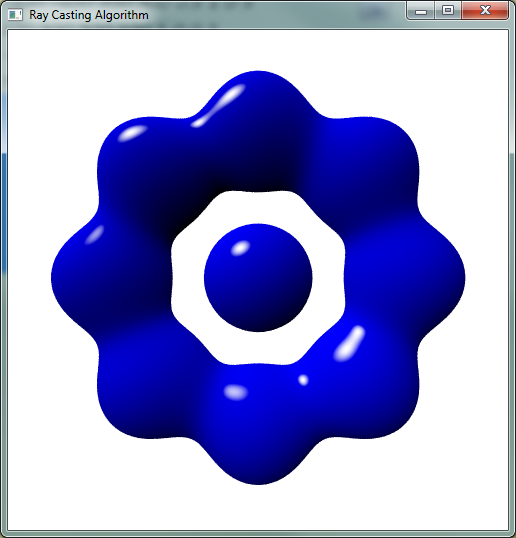
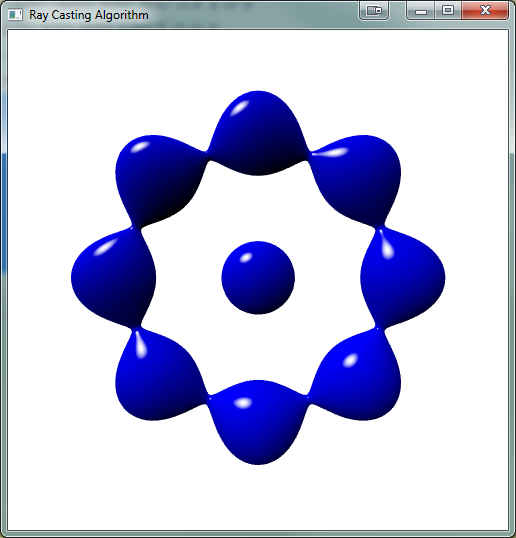
|  |  |
| --- | --- |
| **Object** | **Parameters** |
| Ball | * Center point * Radius |
| Triangle | * 3 vertexes |
| Plane | * A point * Normal vector |
| Implicit surface | * Number of energy source * Surface Energy * Energy Source Coordinate |

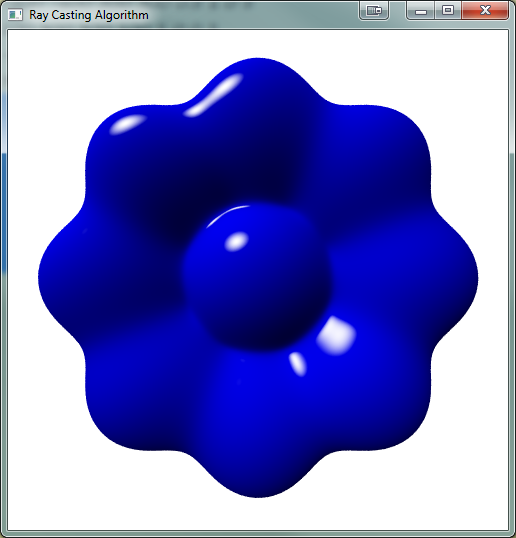
1. **Results**
   1. **Ball with diffuse, highlight, reflection effect**



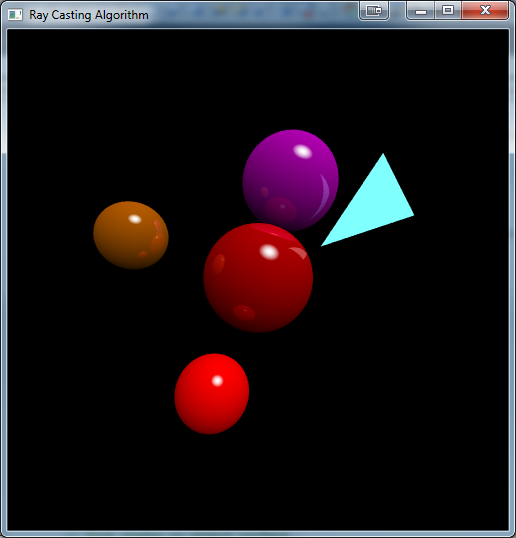
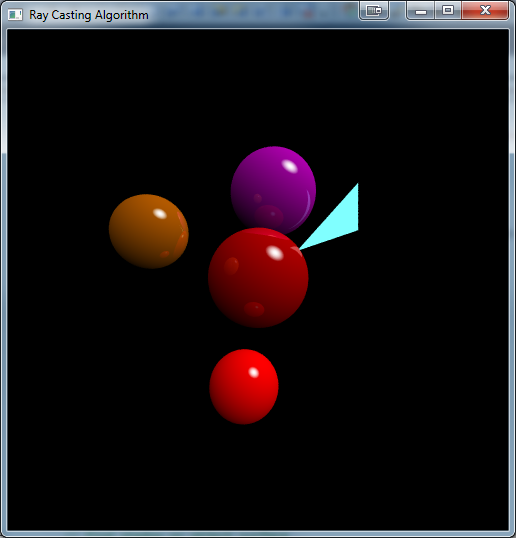
* 1. **Implicit surfaces**
* Generate 3 implicit objects, 5 energy sources for each implicit object.

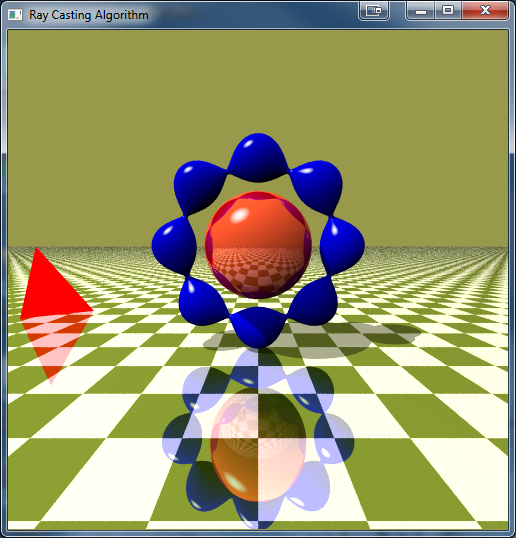
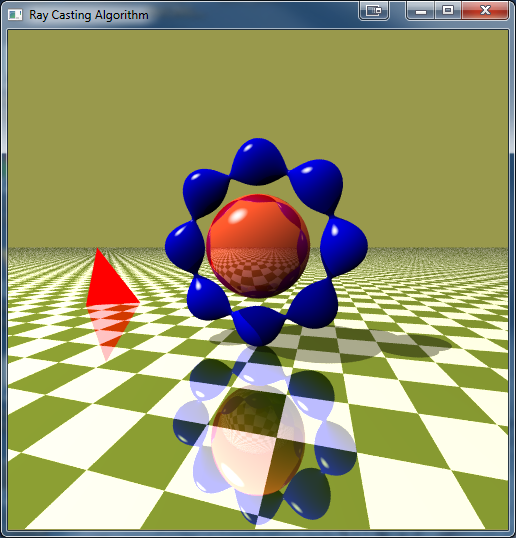
 

* Generate an implicit object with 9 energy sources with many value of energy surface.

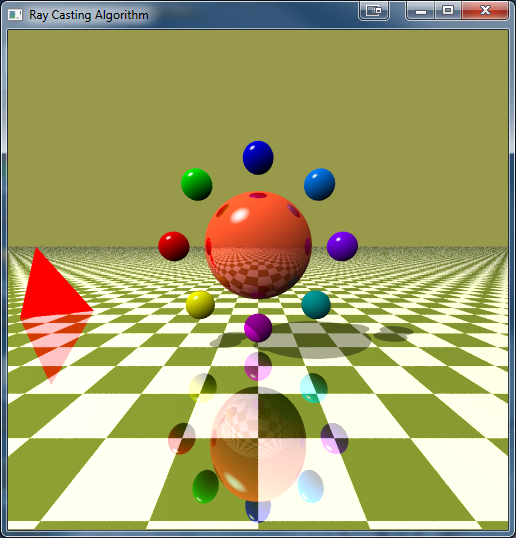


* 1. **Stereo pair Image**

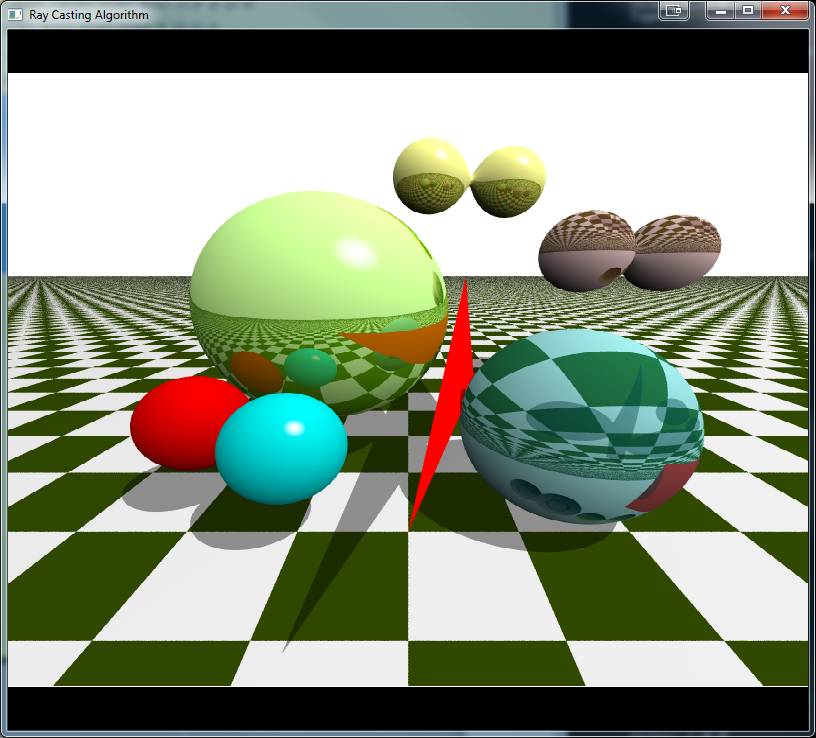
 



* Plane with reflection effect



* 1. **All objects**



1. **Source code**
   1. **Reading data file**

int readFromFile(const char\* filename, myViewPoint &vp, myScreen &screen, vector<myObject \*> &listObj){

ifstream myReadFile;

myReadFile.open(filename);

if(!myReadFile.is\_open()){

cout<<"Reading file is failed.";

return -1;

}

int i=0;

while(!myReadFile.eof()){

i++;

string line;

getline(myReadFile,line);

vector<double> listStr(20);

listStr = splitString(line);

// blank line

if(listStr.size()<1) continue;

int flag = 0;

if(i<5){

// view point

switch(i){

case 1:

// view point

vp.E = myVector(listStr[0],listStr[1],listStr[2]);

vp.A = myVector(listStr[3],listStr[4],listStr[5]);

vp.up = myVector(listStr[6],listStr[7],listStr[8]);

break;

case 2:

// screen

screen.nx = listStr[0];

screen.ny = listStr[1];

screen.theta = listStr[2];

break;

case 3:

// light source

lightSource = myVector(listStr[0],listStr[1],listStr[2]);

break;

case 4:

// background color

bgcolor = myVector(listStr[0],listStr[1],listStr[2]);

break;

}

}else{

myBall \*ball = new myBall();

myTriangle \*triangle = new myTriangle();

myPlane \*plane = new myPlane();

myImplicitSurface \*implicitObj = new myImplicitSurface();

int ncount = 0;;

switch(int(listStr[0])){

case 1:

ball->shape\_type = 1;

ball->Center = myVector(listStr[1],listStr[2],listStr[3]);

ball->radius = listStr[4]; ball->color = myVector(listStr[5],listStr[6],listStr[7]); ball->setType(listStr[8]);

listObj.push\_back(ball);

break;

case 2:

// triangle

triangle->shape\_type = 2;

triangle->P0 = myVector(listStr[1],listStr[2],listStr[3]);

triangle->P1 = myVector(listStr[4],listStr[5],listStr[6]);

triangle->P2 = myVector(listStr[7],listStr[8],listStr[9]);

triangle->color = myVector(listStr[10],listStr[11],listStr[12]);

triangle->setupParameter();

listObj.push\_back(triangle);

break;

case 3:

plane->shape\_type = 3;

plane->P\_zero = myVector(listStr[1],listStr[2],listStr[3]);

plane->normal = myVector(listStr[4],listStr[5],listStr[6]);

plane->color = myVector(listStr[7],listStr[8],listStr[9]); plane->setType(listStr[10]);

listObj.push\_back(plane);

break;

case 4:

implicitObj->shape\_type = 4;

implicitObj->n = listStr[1];

implicitObj->fbound = listStr[2];

ncount = 3;

for(int j=0; j<implicitObj->n; j++){

implicitObj->source[j] = myVector(listStr[ncount], listStr[ncount+1], listStr[ncount+2]);

ncount+=3;

}

implicitObj->color = myVector(listStr[ncount], listStr[ncount+1], listStr[ncount+2]);

ncount += 3;

implicitObj->setType(listStr[ncount++]);

implicitObj->setupParameter();

listObj.push\_back(implicitObj);

break;

default:

break;

}

}

}

myReadFile.close();

return 0;

}

* 1. **Display function**

void display()

{

glClear( GL\_COLOR\_BUFFER\_BIT );

glBegin( GL\_POINTS );

// create transformation matrix from Eye Coordinate System (ECS) to World Coordinate System (WCS)

createTransformMatrixECS2WCS(vp);

double d = screen.ny/(2\*tan(screen.theta\*PI/360));

// send a ray

for( int i=-screen.nx/2; i<=screen.nx/2; i++){

for( int j=-screen.ny/2; j<=screen.ny/2; j++)

{

glVertex2i(i,j);

//Step 2: Find the first hit object return to global variable

myVector color(0,0,0);

double ni = 1, nt = 1.8;

#ifdef ANTI\_ALIAS\_DECMP

// ANTI-ALIASING TO INCREASE RESOLUTION

int num\_sub = ANTI\_ALIAS\_DECMP;

double randx=0, randy=0, sub\_pixel\_x=0, sub\_pixel\_y=0;

// do anti aliasing

for(int xx = 0; xx<num\_sub; xx++){

for(int yy = 0; yy<num\_sub; yy++){

// random a number from 0 to 1

randx = (rand()%100)/100.0;

// random a number form (i-1-nx\*1/3) to (i-1-nx\*1/3+1/3)

sub\_pixel\_x = ((1.0\*i-1-xx\*1.0/num\_sub) + 1.0/num\_sub\*randx);

randy = (rand()%100)/100.0;

sub\_pixel\_y = ((1.0\*j-1-yy\*1.0/num\_sub) + 1.0/num\_sub\*randy);

//Transform pixel(i,j) on the screen to point(x,y,z) in ECS

myVector pe(sub\_pixel\_x+0.5, sub\_pixel\_y+0.5, -d);

//Transform point(x,y,z) in ECS to point(x,y,z) in WCS

myVector pw = transformESC2WCS(pe);

//Calculate ray = pw - E = t\*v (v: unit vector)

myVector ray = (myVector)(pw-vp.E);

myVector vray = ray.unit();

//Find the first hit object and get color, return to a global variable

color = color + rayTracing(vp.E,vray,1,ni,nt);

}

}

color = color/(num\_sub\*num\_sub);

#else

//Transform pixel(i,j) on the screen to point(x,y,z) in ECS

myVector pe(i+0.5, j+0.5, -d);

//Transform point(x,y,z) in ECS to point(x,y,z) in WCS

myVector pw = transformESC2WCS(pe);

//Calculate ray = pw - E = t\*v (v: unit vector)

myVector ray = (myVector)(pw-vp.E);

myVector vray = ray.unit();

color = rayTracing(vp.E,vray,1,ni,nt);

#endif

// Set color of a pixel on the screen corresponding to a point on the hit object

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

}

}

glEnd();

glFlush();

}

* 1. **Ray tracing**

myVector rayTracing(myVector E, myVector shineRay,int depth,double ni, double nt){

double t = -1;

double t\_firstHit = MAXVAL;

int firstHitObjID = -1;

myVector color = bgcolor;

for(int i=0;i<listObj.size();i++){

t = listObj[i]->isHit(E, shineRay);

if(t > EPS && t<t\_firstHit){

//if(t>EPS && t<t\_firstHit){

t\_firstHit = t;

firstHitObjID = i;

}

}

if(firstHitObjID == -1) return color;

// hit object

myObject \*hitObj = listObj[firstHitObjID];

// get hit point and normal vector at hit point

myVector hitPoint = E + shineRay\*t\_firstHit;

myVector vnormal = (hitObj->getNormalAtPoint(hitPoint)).unit();

// get color at hit point

color = hitObj->getPointColor(hitPoint);

// recursive ray tracing if object is reflected

// COMPUTE REFLECTION RAY & REFRACTION RAY

myVector reflectionRay, refractionRay;

double NI = shineRay\*vnormal;

reflectionRay = computeReflectionRay(shineRay,vnormal);

refractionRay = computeRefractionRay(shineRay,vnormal,ni,nt);

double RN = refractionRay\*vnormal;

myVector reflectionColor(0,0,0);

myVector refractionColor(0,0,0);

if(hitObj->reflection && depth < MAX\_RAY\_DEPTH){

//REFLECTION

reflectionColor = rayTracing(hitPoint,reflectionRay.unit(),depth+1,nt, ni);

color = color\*0.3 + reflectionColor\*0.7;

}

if(hitObj->refraction && depth < MAX\_RAY\_DEPTH ){

// REFRACTION

refractionColor = rayTracing(hitPoint-shineRay\*EPS,refractionRay.unit(), depth+1,nt, ni);

color = color\*0.3 + refractionColor\*0.7;

}

// GET SHADDOW

bool shaddow\_area = false;

if(firstHitObjID > -1){

// find shadow on object surface

// find hit object when send shadow ray back to eye

//myVector hitPoint = vp.E + vray\*t\_firstHit;

myVector shadow\_ray = lightSource - hitPoint;

vector<myObject \*> listObj\_shadow = listObj;

getFirstHitObjectForSRay(hitPoint,shadow\_ray,firstHitObjID, t\_firstHit, firstHitObjID); //update global var firstHitObj

if(firstHitObjID > -1){

color = color\*0.6;//\*(0.5\*t\_firstHit);

shaddow\_area = true;

}

}

// GET DIFFUSION

if(hitObj->diffuse){

myVector PL = lightSource-hitPoint;

double cos\_theta = PL.unit()\*vnormal.unit();

cos\_theta = (cos\_theta+1)/2;

color = color\*cos\_theta;

}

// GET HIGHLIGHT

if(!shaddow\_area && hitObj->specular){

myVector incidentRay = hitPoint - lightSource;

myVector reflectionRay\_ = computeReflectionRay(incidentRay,vnormal);

double specular = reflectionRay\_.unit()\*((E-hitPoint).unit());

specular = pow(specular,3);

if(specular > HIGHLIGHT\_CRITICAL){

myVector light\_source\_cl(1,1,1);

myVector HC = (color\*(1-specular) + light\_source\_cl\*(specular-HIGHLIGHT\_CRITICAL))/(1-HIGHLIGHT\_CRITICAL);

color = HC;

}

}

return color;

}

* 1. **Utilities functions for ray tracing**

// for sending shadow ray

void getFirstHitObjectForSRay(myVector E, myVector v,int ignoreID, double &t\_firstHit, int &firstHitObjID){

double t = -1;

t\_firstHit = MAXVAL;

firstHitObjID = -1;

for(int i=0;i<listObj.size();i++){

// ignore object itself in case of sending shadow ray. In case of sending shine ray, ignoreID=-1

if(i==ignoreID) continue;

t = listObj[i]->isHit(E,v);

if(t>0 && t<t\_firstHit){

t\_firstHit = t;

firstHitObjID = i;

}

}

}

vector<double> splitString(string line){

stringstream ss(line);

string buf;

vector<double> tokens;

while(ss >> buf){

tokens.push\_back(stof(buf));

}

return tokens;

}

void createTransformMatrixECS2WCS(myViewPoint vp){

myVector u,v,w;

w = (myVector)(vp.E-vp.A);

u = vp.up&w;

v = w&u;

myVector u\_hat, v\_hat, w\_hat;

u\_hat = u.unit();

v\_hat = v.unit();

w\_hat = w.unit();

M[0][0] = u\_hat.x; M[1][0] = u\_hat.y; M[2][0] = u\_hat.z;

M[0][1] = v\_hat.x; M[1][1] = v\_hat.y; M[2][1] = v\_hat.z;

M[0][2] = w\_hat.x; M[1][2] = w\_hat.y; M[2][2] = w\_hat.z;

M[0][3] = vp.E.x; M[1][3] = vp.E.y; M[2][3] = vp.E.z;

}

myVector transformESC2WCS(myVector pe){

myVector vpe = (myVector)pe;

myVector pW;

pW.x = M[0][0]\*pe.x + M[0][1]\*pe.y + M[0][2]\*pe.z + M[0][3];

pW.y = M[1][0]\*pe.x + M[1][1]\*pe.y + M[1][2]\*pe.z + M[1][3];

pW.z = M[2][0]\*pe.x + M[2][1]\*pe.y + M[2][2]\*pe.z + M[2][3];

return myVector(pW.x,pW.y,pW.z);

}

myVector computeReflectionRay(myVector shineRay,myVector vnormal){

double R = shineRay\*vnormal/vnormal.norm();

return (shineRay-vnormal\*2\*R).unit();

}

myVector computeRefractionRay(myVector shineRay,myVector vnormal, double ni, double nt){

myVector NN = vnormal/vnormal.norm();

myVector K = NN\*((shineRay\*(-1))\*NN);

double cos\_theta\_i = NN\*shineRay\*(-1)/(shineRay.norm()\*NN.norm());

double sin\_theta\_t = (ni/nt)\*(sqrt(1-pow(cos\_theta\_i,2)));

double cos\_theta\_t = sqrt(1-pow(sin\_theta\_t,2));

myVector M = K.unit()\*shineRay.norm()\*sin\_theta\_t;

myVector N\_ = NN\*(-1) \* shineRay.norm() \* cos\_theta\_t;

return M + N\_;

}

* 1. **Object classes**
     1. **Ball**

#define MYBALL\_H

#include "myVector.h"

#include "myObject.h"

#include "define.h"

class myBall:public myObject{

public:

//data

myVector Center;

double radius;

myBall(){shape\_type = 1;};

myBall(myVector iC, double iradius, myVector icolor, int itype);

double isHit(myVector iE, myVector iray);

void printInfo();

myVector getPointColor(myVector hitPoint);

myVector getNormalAtPoint(myVector hitPoint);

~myBall(void);

};

#endif

#pragma once

#include "myBall.h"

#include <iostream>

using namespace std;

myBall::~myBall(void)

{

}

myBall::myBall(myVector iC, double iradius, myVector icolor, int itype):myObject(){

kind = 1;

Center = iC;

radius = iradius;

color = icolor;

type = itype;

}

double myBall::isHit(myVector iE, myVector ray){

// coefficient of equation

double a=0, b=0, c=0;

double delta = 0;

myVector vCO;

vCO.x = iE.x-Center.x;

vCO.y = iE.y-Center.y;

vCO.z = iE.z-Center.z;

a = ray\*ray;

b = ray\*vCO;

c = vCO\*vCO - radius\*radius;

delta = b\*b - a\*c;

if( delta > EPS){

double t1 = (-b-sqrt(delta))/a;

double t2 = (-b+sqrt(delta))/a;

/\*if(t1<t2) return t1;

if(t1>t2) return t2;\*/

if(t1<EPS && t2<EPS) return -1;

if (t1<t2)

{

if(t1>EPS) return t1;

if(t2>EPS) return t2;

return -1;

}

else if (t2<t1)

{

if(t2>EPS) return t2;

if(t1>EPS) return t1;

return -1;

}

return -1;

}

}

void myBall::printInfo(){

cout<<"Ball Parameters"<< endl;

cout<<"\tCenter:\("<<Center.x<<","<<Center.y<<","<<Center.z<<"\)"<< endl;

cout<<"\tRadius: "<<radius<< endl;

cout<<"\tColor: "<<color.x<<","<<color.y<<","<<color.z<<endl;

cout<<"\tType: "<<type<<endl;

}

myVector myBall::getPointColor(myVector hitPoint){

return color;

}

myVector myBall::getNormalAtPoint(myVector hitPoint){

return (hitPoint - Center).unit();

}

* + 1. **Plane**

#pragma once

#ifndef MYPLANE\_H

#define MYPLANE\_H

#include "myVector.h"

#include "myObject.h"

class myPlane:public myObject

{

public:

myVector P\_zero;

myVector normal;

myPlane(void){};

myPlane(myVector P\_zero1, myVector normal1)

{

P\_zero = P\_zero1;

normal = normal1;

}

~myPlane(void){};

double isHit(myVector iE, myVector iray){

return ((P\_zero-iE)\*normal)/(iray\*normal);

}

myVector myPlane::getPointColor(myVector hitPoint){

myVector white(1,1,1);

myVector n\_color = color;

// chessboard

int square = (int)(floor(hitPoint.x/200) + floor(hitPoint.y/200));

if(square % 2 == 0){

n\_color = white;

}

return n\_color;

}

myVector myPlane::getNormalAtPoint(myVector hitPoint){

return normal;

}

};

#endif

* + 1. **Triangle**

#pragma once

#ifndef MYTRIANGLE\_H

#define MYTRIANGLE\_H

#include "myObject.h"

#include "myVector.h"

#include "myPlane.h"

#include "define.h"

class myTriangle:public myObject{

public:

//data

myVector P0;

myVector P1;

myVector P2;

myVector normal;

myVector u\_hat;

myVector v\_hat;

myPlane obj\_plane;

void setupParameter();

myTriangle(){shape\_type = 2;};

myTriangle(myVector P0, myVector P1, myVector P2);

void printInfo();

myVector getPointColor(myVector hitPoint);

double isHit(myVector iE, myVector iray);

~myTriangle(void);

};

#endif

#pragma once

#include "myTriangle.h"

#include <iostream>

using namespace std;

myTriangle::~myTriangle(void)

{

}

void myTriangle::setupParameter(){

myVector P0P1 = (P1-P0);

myVector P0P2 = (P2-P0);

normal = (P0P1 & P0P2).unit();

u\_hat = (P0P2 & normal)/((P0P1 & P0P2)\*normal);

v\_hat = (P0P1 & normal)/((P0P1 & P0P2)\*normal);

obj\_plane = myPlane(P0,normal);

}

double myTriangle::isHit(myVector iE, myVector ray){

shape\_type = 2;

// check if ray hit plane

double t = obj\_plane.isHit(iE, ray);

if(t > EPS){

// check hit point is in triangle

myVector hitPoint = iE + ray\*t;

double u = (hitPoint-P0)\*u\_hat;

double v = (hitPoint-P0)\*v\_hat;

if( u>EPS && v>EPS && (u+v)<1 ){

return t;

}

return -1; // hit point is not in triangle

}

return -1; // hit point does not hit triangle plane

}

myVector myTriangle::getPointColor(myVector hitPoint){

return color;

}

void myTriangle::printInfo(){

cout<<"Triangle Parameters"<< endl;

cout<<"\tPoint A:\("<<P0.x<<","<<P0.y<<","<<P0.z<<"\)"<< endl;

cout<<"\tPoint B:\("<<P1.x<<","<<P1.y<<","<<P1.z<<"\)"<< endl;

cout<<"\tPoint C:\("<<P2.x<<","<<P2.y<<","<<P2.z<<"\)"<< endl;

cout<<"\tColor: "<<color.x<<","<<color.y<<","<<color.z<<endl;

cout<<"\tType: "<<type<<endl;

}

* + 1. **Implicit Surface**

#pragma once

#ifndef MYIMPLICITSURFACE\_H

#define MYIMPLICITSURFACE\_H

#include "myObject.h"

#include "myVector.h"

#include "stdio.h"

#include <math.h>

#include "define.h"

class myImplicitSurface:public myObject{

public:

//data

myVector source[MAX\_NUM\_SOURCE];

double E[MAX\_NUM\_SOURCE];

double b[MAX\_NUM\_SOURCE];

double r[MAX\_NUM\_SOURCE];

// num of sources

int n;

// bounding box of object to get the first interval when finding root of object function

myVector vMin, vMax;

// bounding energy

double fbound;

myImplicitSurface(){

type = 4;

color = myVector(1,0,0);

n=MAX\_NUM\_SOURCE;

for(int i=0;i<n;i++){

source[i] = myVector(0,0,0);

E[i] = 130;

b[i] = 0.00005;

r[i] = 0;

}

vMin = myVector(0,0,0);

vMax = myVector(100,100,100);

fbound = 50;// T value

};

void setupParameter(){

calcRadius();

calcBoundary();

};

int initFirstInterval(myVector iE, myVector ray, myVector vMin, myVector vMax, double &tmin, double &tmax);

int isContainZero(myVector iE, myVector ray, double tmin, double tmax);

int findRoot(myVector iE, myVector ray, double tmin, double tmax, double &root);

double isHit(myVector iE, myVector ray);

void findQuadMinMax(double a, double b, double c, double tmin, double tmax, double &minval, double &maxval);

myVector getPointColor(myVector hitPoint);

void calcBoundary();

void calcRadius();

double energyAtPoint(myVector p);

myVector getNormalAtPoint(myVector hitPoint);

void printInfo();

~myImplicitSurface(void){};

};

#endif

#pragma once

#include "myImplicitSurface.h"

#include <iostream>

using namespace std;

int myImplicitSurface:: initFirstInterval(myVector iE, myVector ray, myVector vMin, myVector vMax, double &tmin, double &tmax){

double tminX, tminY, tminZ;

double tmaxX, tmaxY, tmaxZ;

if (ray.x==0) tminX = MINVAL;

else{

tminX = (vMin.x - iE.x)/ray.x;tmaxX = (vMax.x - iE.x)/ray.x; if (ray.x<0) swap(tminX,tmaxX);

}

if (ray.y==0) tminY = MINVAL;

else{

tminY = (vMin.y - iE.y)/ray.y;tmaxY = (vMax.y - iE.y)/ray.y; if (ray.y<0) swap(tminY,tmaxY);

}

if (ray.z==0) tminZ = MINVAL;

else{

tminZ = (vMin.z - iE.z)/ray.z;tmaxZ = (vMax.z - iE.z)/ray.z; if (ray.z<0) swap(tminZ,tmaxZ);

}

tmin = fmin3(tminX, tminY, tminZ);

tmax = fmax3(tmaxX, tmaxY, tmaxZ);

return (tmin<tmax);

}

int myImplicitSurface::isContainZero(myVector iE, myVector ray, double tmin, double tmax){

double fmin, fmax;

// calculate fmin, fmax

fmin = -fbound;

fmax = -fbound;

double r2max, r2min;

double a0, b0, c0;

for(int i=0; i<n;i++){

a0 = ray\*ray;

b0 = (iE-source[i])\*ray\*2;

c0 = (iE-source[i])\*(iE-source[i]);

findQuadMinMax(a0, b0, c0, tmin, tmax, r2min, r2max);

double tmp1 = E[i]\*exp(-b[i]\*r2max);

double tmp2 = E[i]\*exp(-b[i]\*r2min);

fmin += E[i]\*exp(-b[i]\*r2max);

fmax += E[i]\*exp(-b[i]\*r2min);

}

return (fmin<=0 && fmax >=0);

}

int myImplicitSurface::findRoot(myVector iE, myVector ray, double tmin, double tmax, double &root){

double m = (tmin + tmax)/2;

if(!isContainZero(iE, ray, tmin, tmax)) {

return 0; // no\_root

}

if(abs(tmax-tmin) < EPS){

root = m;

return 1; // has root

}

if(!findRoot(iE, ray, tmin, m, root))

return findRoot(iE, ray, m, tmax, root);

else

return 1;

}

double myImplicitSurface::isHit(myVector iE, myVector ray){

// initialize first interval [t1,t2] to find root t

double t1, t2;

double t = -1;

if(initFirstInterval(iE, ray, vMin, vMax, t1, t2)){

// find hit point using interval method

if(!findRoot(iE, ray, t1, t2, t)){

if(t < EPS) t = -1;

}

}

return t;

}

myVector myImplicitSurface::getPointColor(myVector hitPoint){

return color;

}

void myImplicitSurface::findQuadMinMax(double a, double b, double c, double tmin, double tmax, double &minval, double &maxval){

if (abs(a)<EPS) {

minval = b\*tmin + c;

maxval = b\*tmax + c;

if (b<0) swap(minval, maxval);

return;

}

double v\_at\_right = a\*tmin\*tmin + b\*tmin +c;

double v\_at\_left = a\*tmax\*tmax + b\*tmax + c;

double extrema = c-b\*b/4/a;

if( -b/2/a >= tmin && -b/2/a <= tmax ){

minval = fmin3(extrema,v\_at\_left,v\_at\_right);

maxval = fmax3(extrema,v\_at\_left,v\_at\_right);

}else{

minval = min(v\_at\_left, v\_at\_right);

maxval = max(v\_at\_left, v\_at\_right);

}

}

void myImplicitSurface::calcRadius(){

if (n==0) return;

for (int i = 0; i<n; i++){

//external energy

double extE = 0;

for (int j = 0; j<n; j++){

if (j!=i){

extE = extE + E[j]\*exp((source[i]-source[j])\*(source[i]-source[j])\*(-b[j]));

}

}

//sum of external energy and internal energy

r[i] = 2\*sqrt(log((E[i]+extE)/fbound)/b[i]);

}

}

//get normal vector at a point on the surface

myVector myImplicitSurface::getNormalAtPoint(myVector p){

myVector grad;

double ep = energyAtPoint(p);

grad.x = energyAtPoint(p + myVector(DELTAXYZ,0,0)) - energyAtPoint(p - myVector(DELTAXYZ,0,0));

grad.y = energyAtPoint(p + myVector(0,DELTAXYZ,0)) - energyAtPoint(p - myVector(0,DELTAXYZ,0));

grad.z = energyAtPoint(p + myVector(0,0,DELTAXYZ)) - energyAtPoint(p - myVector(0,0,DELTAXYZ));

grad.unit();

return grad;

}

//calculating bounding box represented by min and max

void myImplicitSurface::calcBoundary(){

if (n == 0) return;

vMin.x = vMin.y = vMin.z = MAXVAL;

vMax.x = vMax.y = vMax.z = MINVAL;

for (int i = 0; i<n; i++){

if ((source[i].x - r[i])<vMin.x){

vMin.x = source[i].x - r[i];

}

if ((source[i].y - r[i])<vMin.y){

vMin.y = source[i].y - r[i];

}

if ((source[i].z - r[i])<vMin.z){

vMin.z = source[i].z - r[i];

}

if ((source[i].x + r[i])>vMax.x){

vMax.x = source[i].x + r[i];

}

if ((source[i].y + r[i])>vMax.y){

vMax.y = source[i].y + r[i];

}

if ((source[i].z + r[i])>vMax.z){

vMax.z = source[i].z + r[i];

}

}

}

//energy at a point p

double myImplicitSurface::energyAtPoint(myVector p){

double energy = 0;

double distsq;

for (int i = 0; i<n ; i++){

distsq = (p - source[i])\*(p - source[i]);

energy += E[i]\*exp(-b[i]\*distsq);

}

return -energy;

}

void myImplicitSurface::printInfo(){

cout << "Implicit Surface" << endl;

cout <<"Num obj: " << n <<endl;

cout <<"Energy: " << fbound << endl;

cout << "Color: " << color.x << "," << color.y << "," << color.z << endl;

for(int i=0; i<n; i++){

cout << source[i].x << "," << source[i].y << "," << source[i].z << endl;

cout <<"E="<<E[i] << ", b=" << b[i] << endl;

}

}

* + 1. **Define.h**

#include <math.h>

#define MAX\_NUM\_SOURCE 10

#define MAXVAL 1.e12

#define MINVAL -MAXVAL

#define fmin3(x,y,z) min(min(x,y),z)

#define fmax3(x,y,z) max(max(x,y),z)

#define DELTAXYZ 0.0000000001

#define ANTI\_ALIAS\_DECMP 3

#define HIGHLIGHT\_CRITICAL 0.85

#define double double

#define PI 3.141592

#define MAX\_RAY\_DEPTH 6

#define EPS 0.00003

#define EPSILON\_IMPLICITSF 0.0003

#define MAX2(a,b) ((a)>(b)?a:b)

**Thank you**