Image Synthesis Theory-Final Project

Nguyen Trung Quy – 128399

Problems: Implement ray tracing algorithm of some objects:

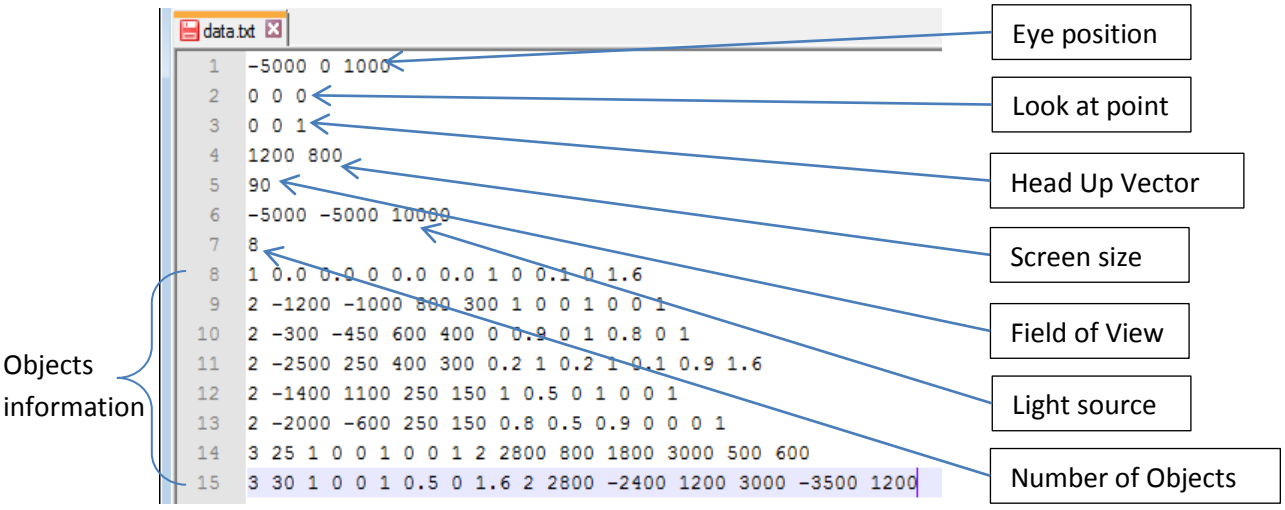
- Ball
- Plane (floor surface)
- Implicit surfaces
- Triangle

Supported effects:

- Diffuse
- Specular (Highlight)
- Shadow
- Reflection
- Refraction
- Anti-Aliasing
- Stereo pair Images

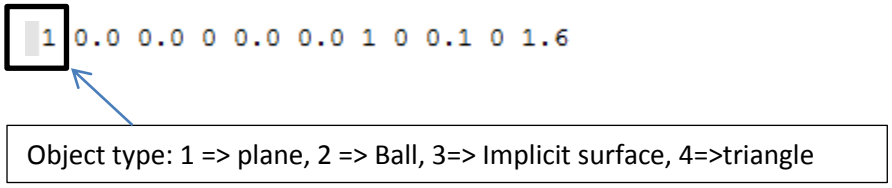
Data input come from text file

1. Input file format

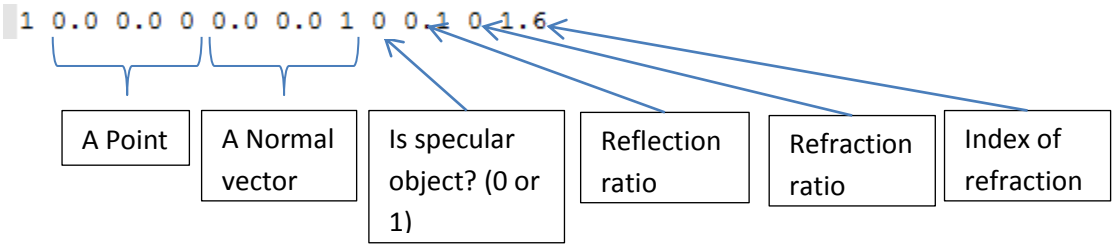


Information of each object is in one line:

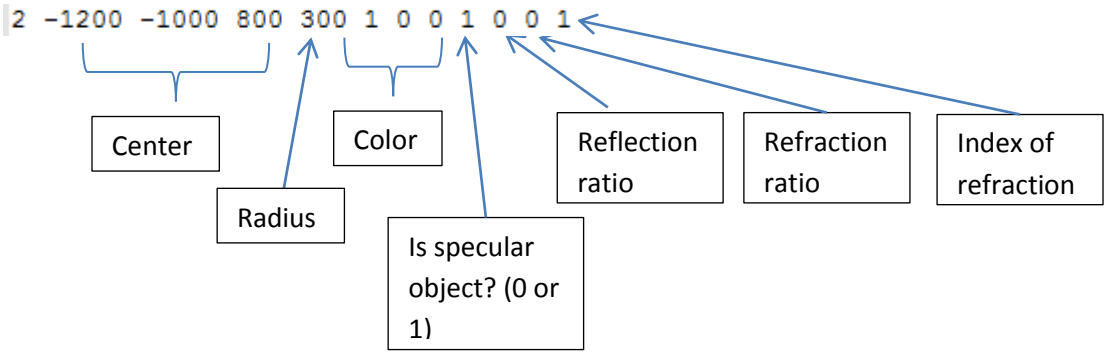
- 1st number is object type:
  - 1 => object is plane
  - 2 => object is ball
  - 3 => object is implicit surface
  - 4 => object is triangle



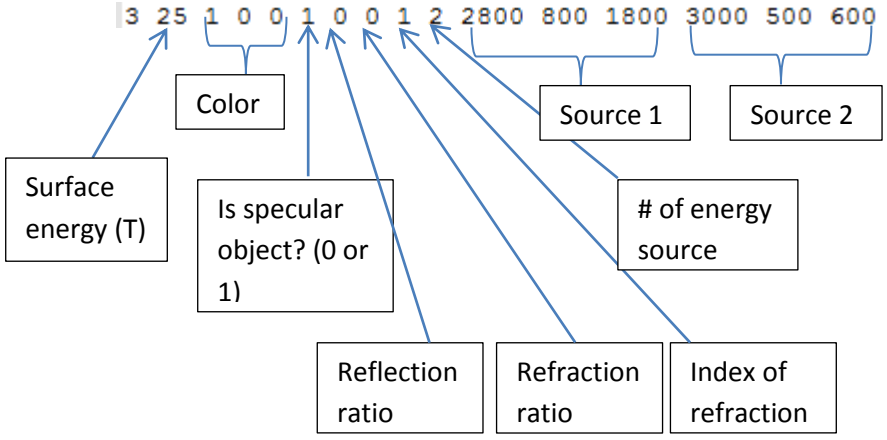
- The rest of line is object information:
  - Plane:



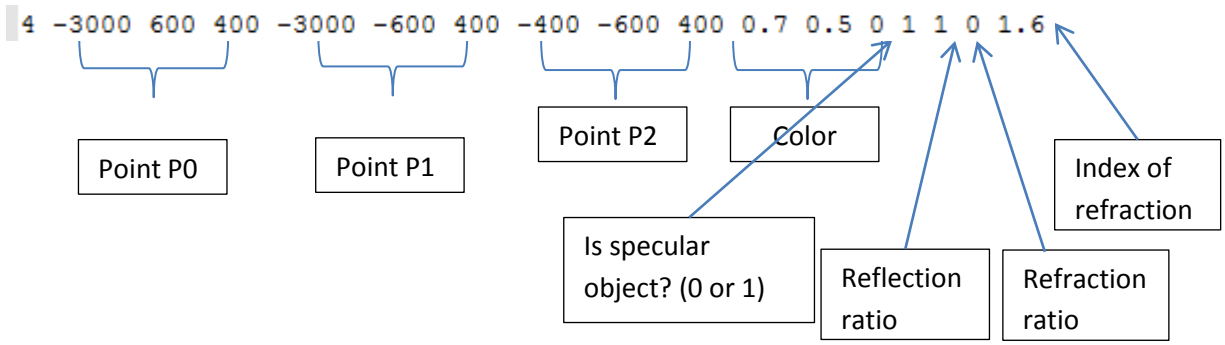
  - Ball:



  - Implicit surface:



  - Triangle:



Note:

- The reflection and refraction ratio is from 0 to 1.
- Total value of the reflection and refraction ratio should be less than or equal 1.
- The reflection ratio is 1 => object reflect 100% coming light => object is mirror.
- The refraction ratio is 1 => object refract 100% coming light => object is glass.
- Highlight effect applies for all objects that “is_specular” set to 1.

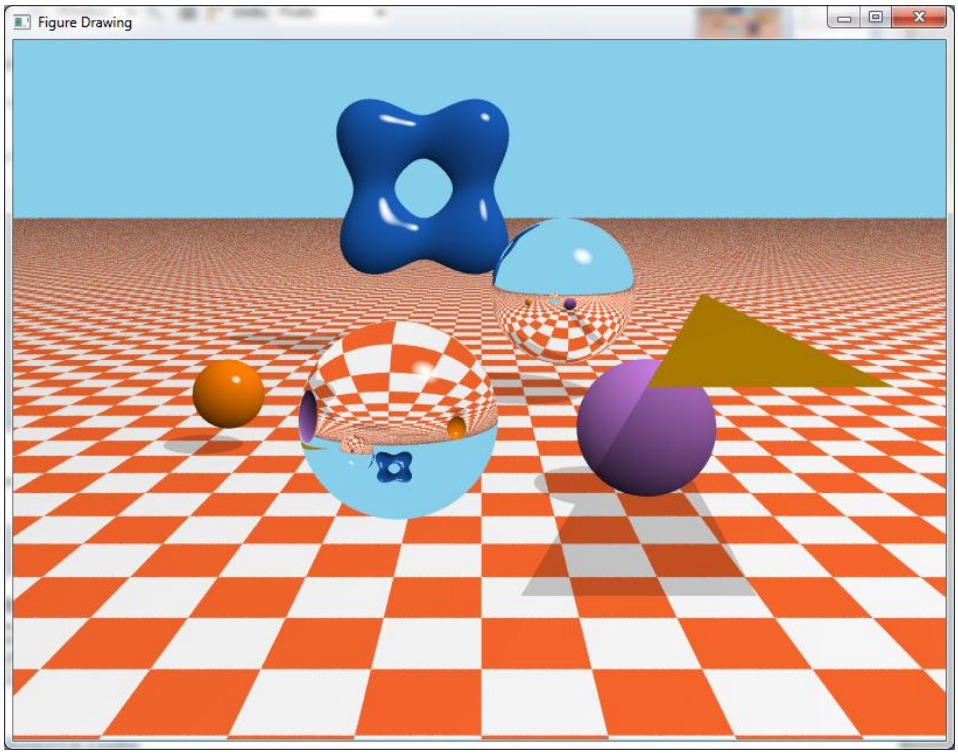
2. Coloring Model:

(Object color + Reflection color + refraction color) => Shadow effect => Highlight effect => Final color

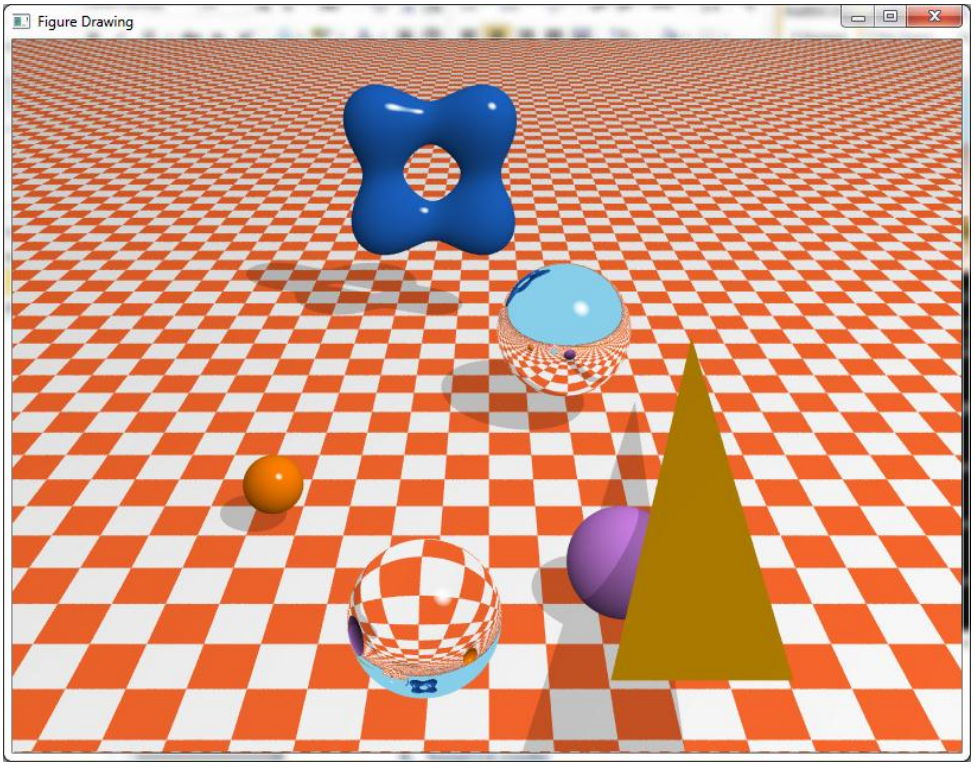
Notices:

- I used anti-aliasing with jittered sampling method (9 samples for one pixel).
- Check-board floor is created by using plane object.
- Ignore shadow effect for glass objects (refraction ratio is 1).
- Have no highlight effect in shadow area.

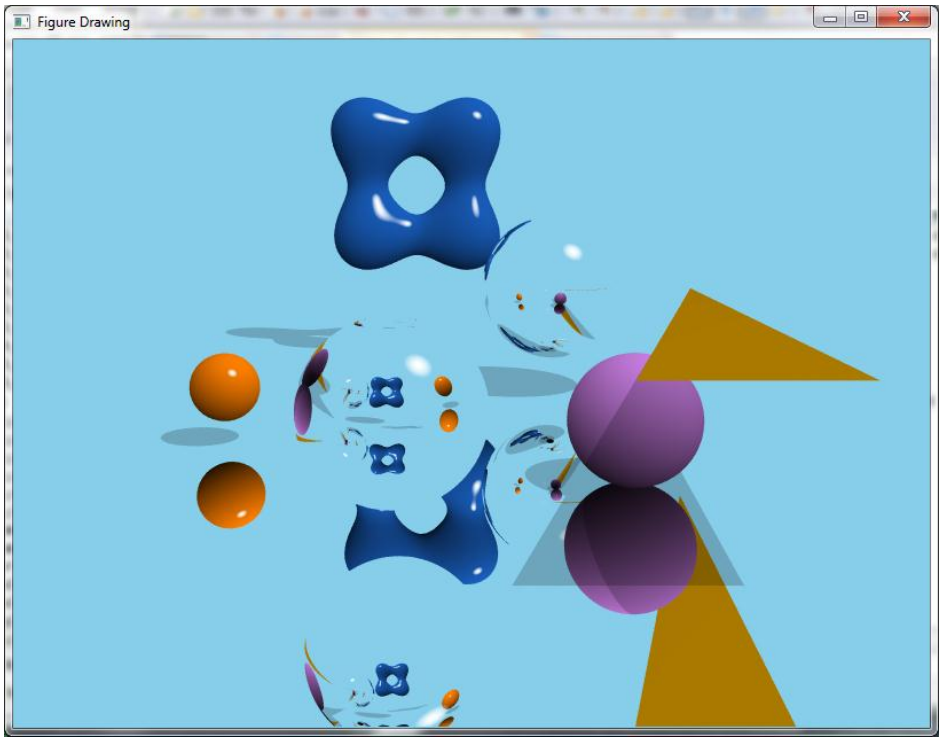
3. Results



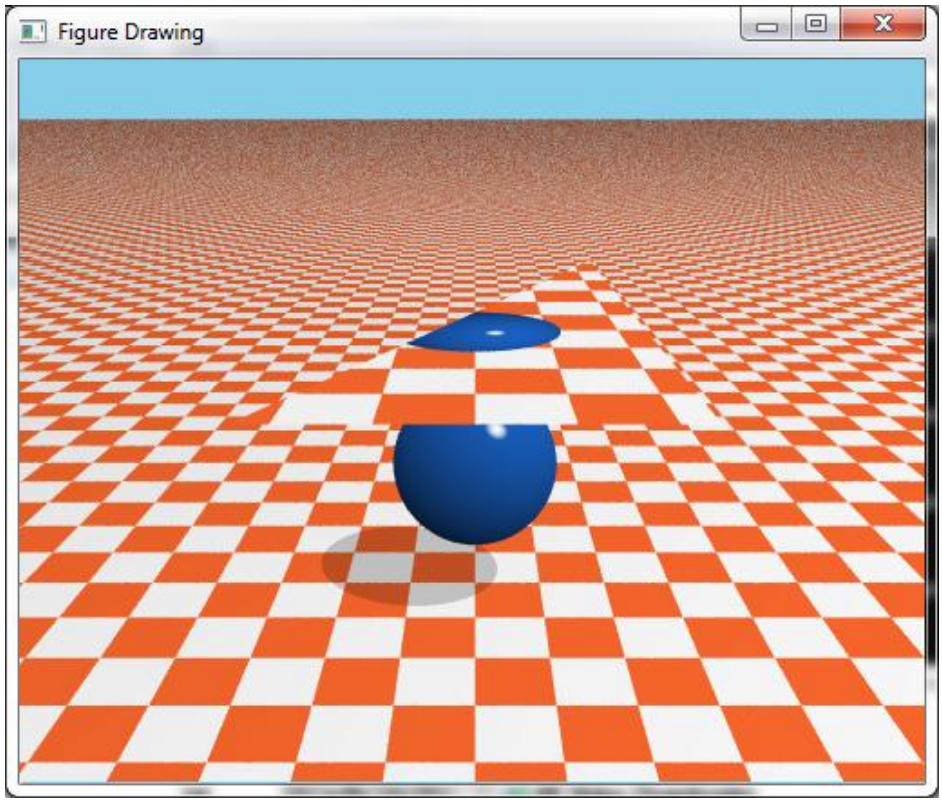
All supported objects and effects (reflection, refraction, shadow, highlight, diffuse)



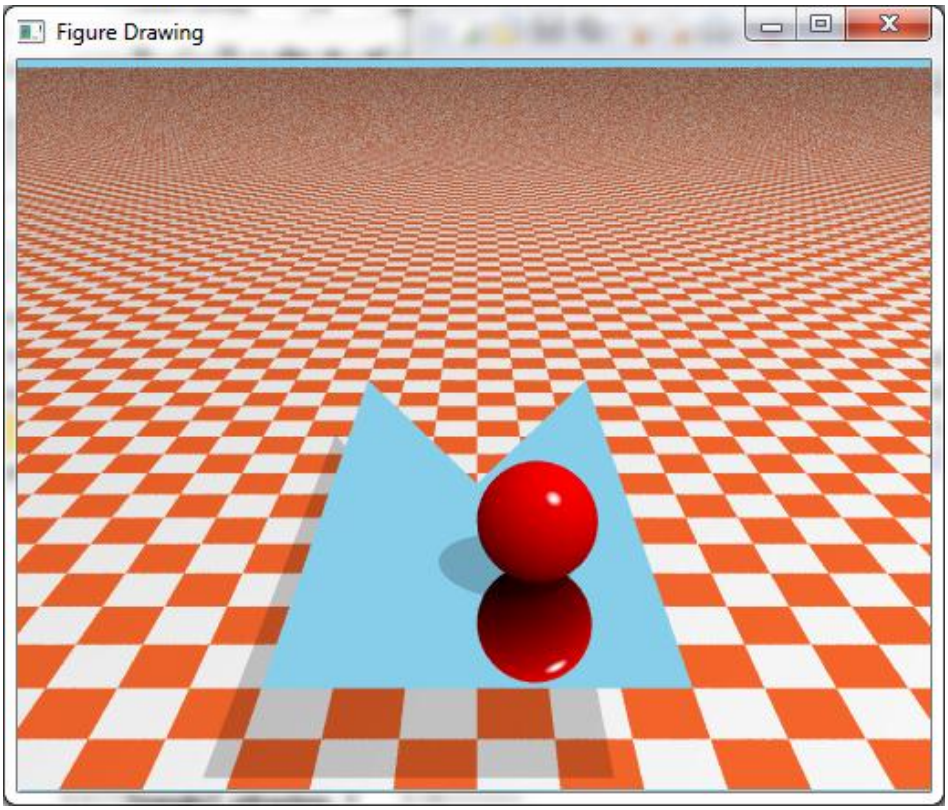
High view of above scene



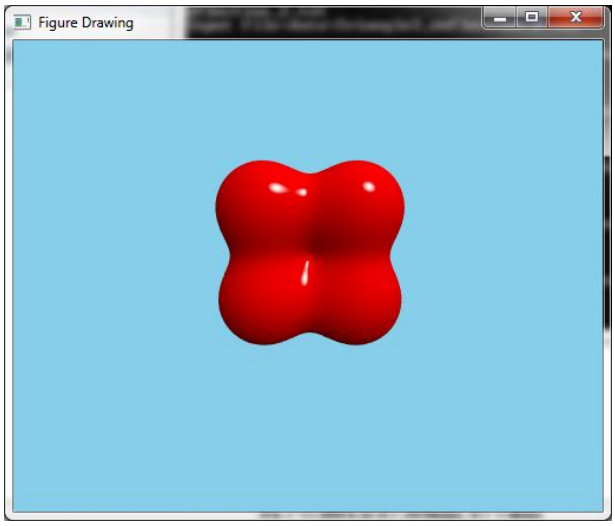
Floor with total reflection (reflection ratio = 1)



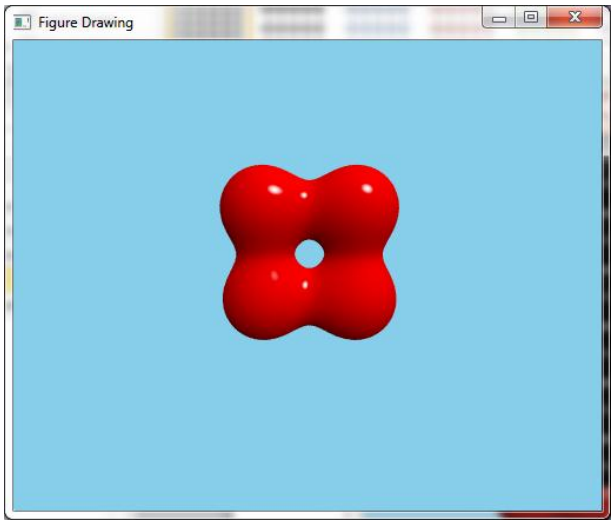
Triangle with refraction effect



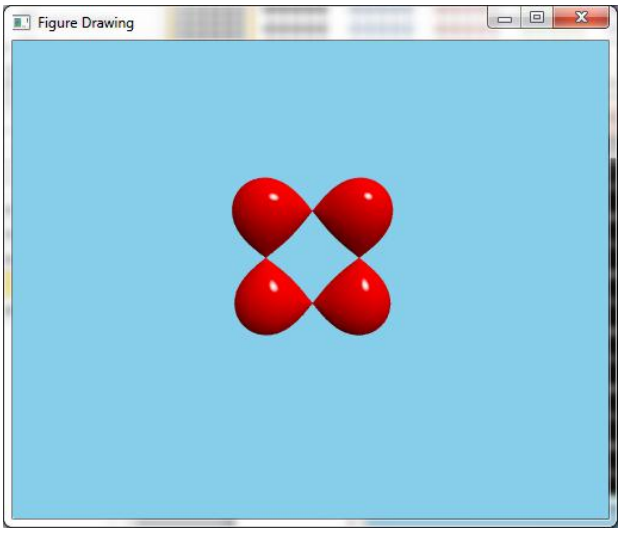
Two triangles with reflection effect



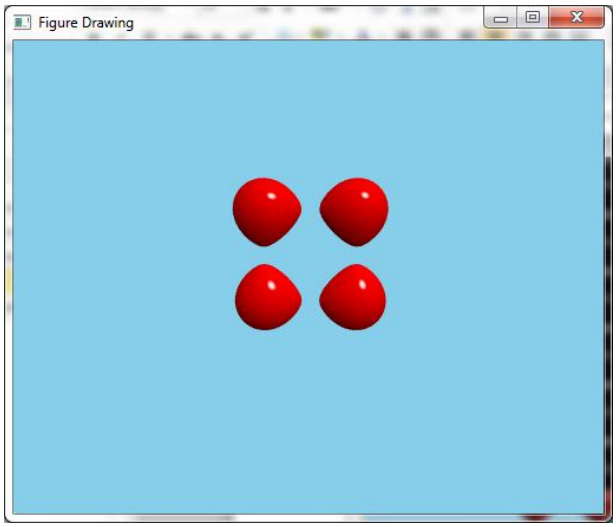
T = 40



T = 50

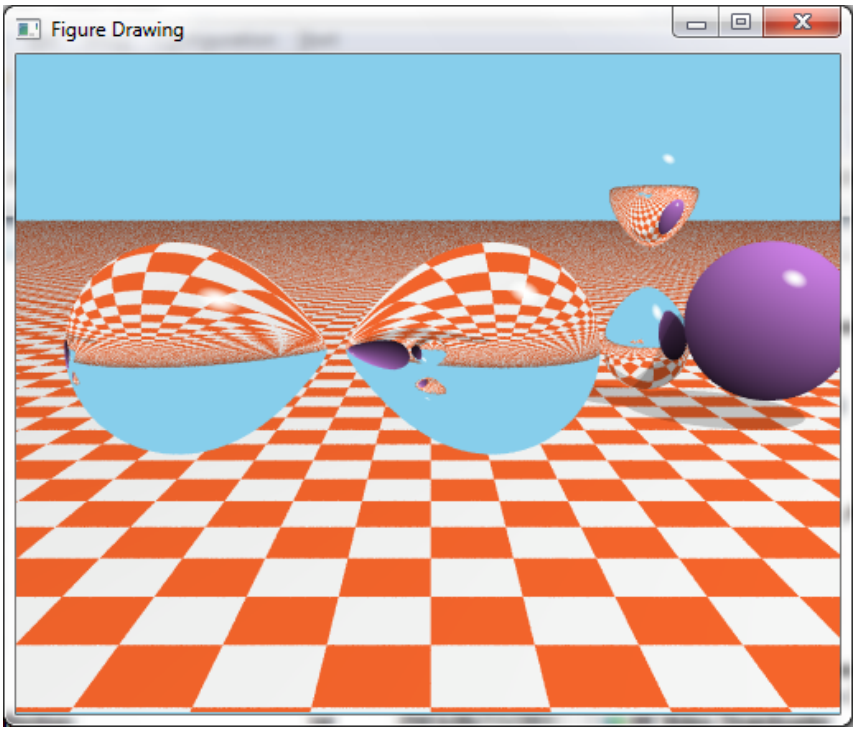


T = 75

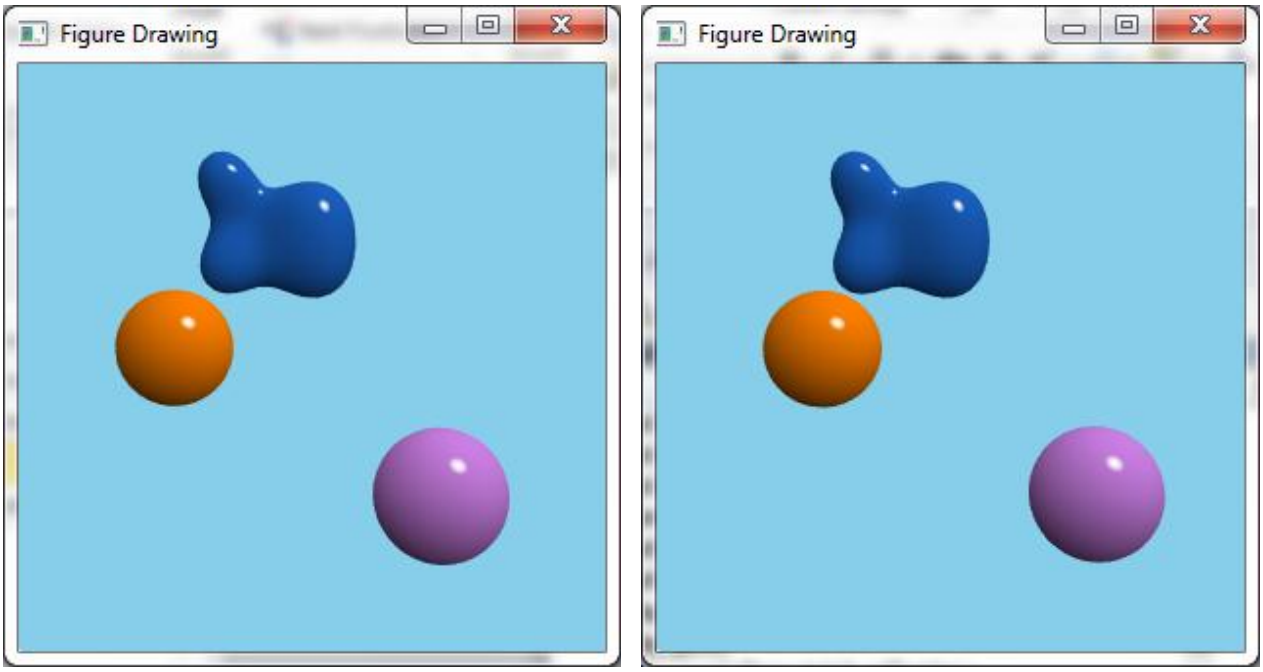


T = 80

Implicit surface object with difference T values

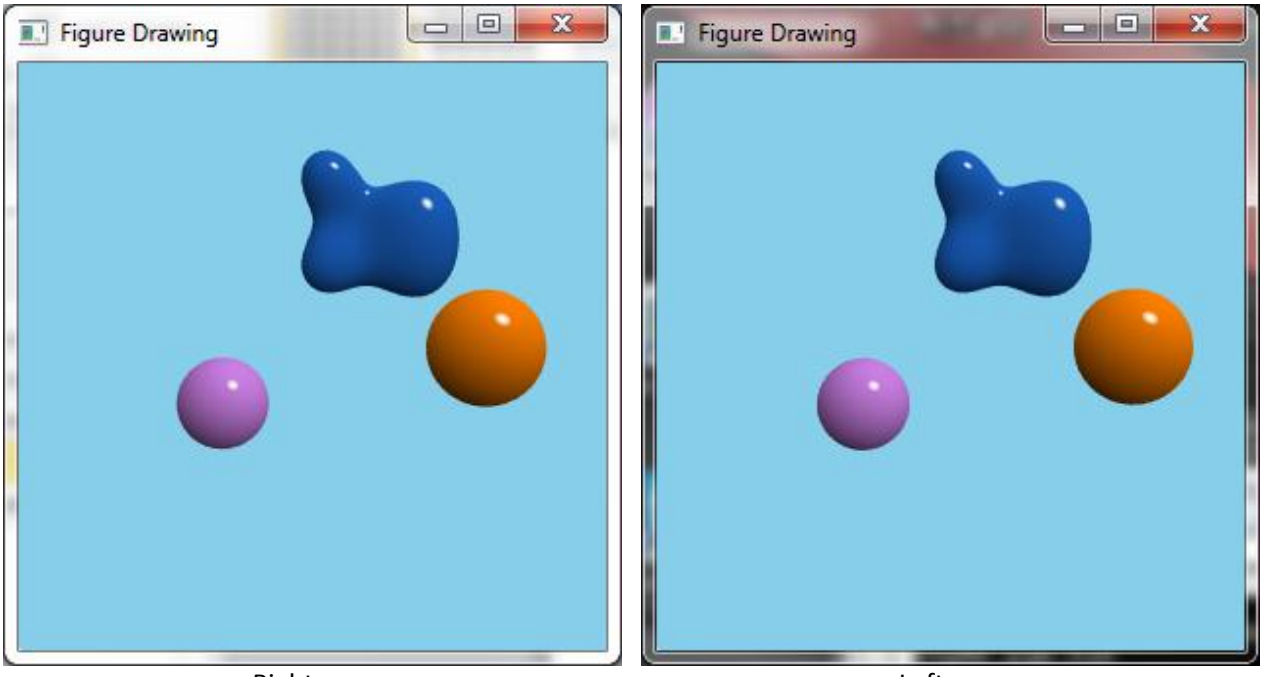


Implicit surface objects with reflection and refraction effects



Right eye Left eye

Stereo pair images – Example 1



Right eye Left eye

Stereo pair images – Example 2

4. Source code

main.cpp

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <complex>
#include <string>
#include <math.h>
#include <sstream>
#include <algorithm>
#include <iterator>
#include <fstream>

#include "define.h"
#include "QVector.h"
#include "QObject.h"
#include "QPlane.h"
#include "QBall.h"
#include "QTriangle.h"

#include "QImplicitSurface.h"
#include "ViewingParams.h"
#include "ScreenParams.h"
#include "GL/glut.h"

using namespace std;
//int _count = 0;
bool debug = false;

QVector eye;
QVector lookAtPt;
QVector direction;
QVector headUp;
QVector lightSrc;
ViewingParams g_view;
ScreenParams screen;
std::vector<QObject*> listObjs;

char *data = "data.txt";

QVector backgroundColor = QVector::parseColor(COLOR_SKY);
int nx2;
int ny2;
void readInputFile(bool readObjOnly = false){
        listObjs.clear();
        FILE *f;
        QObject obj;
        f = fopen(data, "r");
        if( f<=0 )
        {
```

```
                cout<<"File Opening error"<<endl;
                exit(0);
        }
        double ex,ey,ez;
        double atx,aty,atz;
        double upx,upy,upz;
        fscanf(f,"%lf %lf %lf ", &ex,  &ey,  &ez);
        fscanf(f,"%lf %lf %lf ", &atx, &aty, &atz);
        fscanf(f,"%lf %lf %lf ", &upx, &upy, &upz);

        int nx,ny;
        double fov;
        fscanf(f,"%d %d ", &nx, &ny);
        fscanf(f,"%lf ", &fov );
        double lx,ly,lz;
        fscanf(f,"%lf %lf %lf", &lx,  &ly,  &lz );

        if(!readObjOnly){
                eye = QVector(ex,ey,ez);
                lookAtPt = QVector(atx,aty,atz);
                direction = lookAtPt-eye;
                headUp = QVector(upx,upy,upz);
                screen = ScreenParams(nx,ny,fov*M_PI/360);
                nx2 = (int)screen.get_nx() / 2;
                ny2 = (int)screen.get_ny() / 2;
                lightSrc = QVector(lx,ly,lz);
                g_view = ViewingParams(eye, direction, headUp);
        }
        int nObj;
        fscanf(f,"%d ", &nObj);
        char line [1000];
        int k = 0;
        while(k < nObj && fgets(line,sizeof line,f)!= NULL) /* read a line from a file */
        {
                std::string str(line);
                vector<string> tokens;
                vector<double> values;
                istringstream iss(line);
                copy(istream_iterator<string>(iss),
                        istream_iterator<string>(),
                        back_inserter<vector<string> >(tokens));
                for (int i = 0;i<tokens.size();i++)
                {
                        values.push_back(stof(tokens[i]));
                }
                QPlane *pl;
                QBall *bl;
                QImplicitSurface *imsur;
                QTriangle *tr;
                bool is_sqecular;
                std::vector<QVector> centers;
                switch (int(values[0]))
```

```
		{
		case OBJ_TYPE_PLANE:
			//plane
			if (abs(values[7]) > EPSILON)
				is_sqecular = true;
			else
				is_sqecular = false;
			pl = new QPlane(QVector(values[1],values[2],values[3]),
QVector(values[4],values[5],values[6]),is_sqecular,values[8],values[9],values[10]);
			listObjs.push_back(pl);
			break;
		case OBJ_TYPE_BALL:
			//ball
			if (abs(values[8]) > EPSILON)
				is_sqecular = true;
			else
				is_sqecular = false;
			bl = new QBall(QVector(values[1],values[2],values[3]), values[4],
QVector(values[5],values[6],values[7]),is_sqecular,values[9],values[10],values[11]);
			listObjs.push_back(bl);
			break;
		case OBJ_TYPE_IMPLICIT:{
			// implicit surface object
			if (abs(values[5]) > EPSILON)
				is_sqecular = true;
			else
				is_sqecular = false;
			double n_center = values[9];
			for (int i =0;i<n_center;i++){
				int pos = 9 + 3*i;

	centers.push_back(QVector(values[pos+1],values[pos+2],values[pos+3]));
			}
			imsur = new
QImplicitSurface(centers,values[1],QVector(values[2],values[3],values[4]),is_sqecular,values[6],va
lues[7],values[8]);
			listObjs.push_back(imsur);
			break;
			   }//case 3 block
		case OBJ_TYPE_TRIANGLE:
			//Triangle
			if (abs(values[13]) > EPSILON)
				is_sqecular = true;
			else
				is_sqecular = false;
			tr = new QTriangle(QVector(values[1],values[2],values[3]),
QVector(values[4],values[5],values[6]),
QVector(values[7],values[8],values[9]),QVector(values[10],values[11],values[12]),is_sqecular,value
s[14],values[15],values[16]);
			listObjs.push_back(tr);
			break;
		}
```

```
                k++;
        }
        fclose(f);
}
void printData(){
        //Eye
        cout<<"Eye:"<<eye.toString()<<endl;
        cout<<"LookAtPoint:"<<lookAtPt.toString()<<endl;
        cout<<"HeadUp:"<<headUp.toString()<<endl;
        cout<<"Screen Size:["<<screen.get_nx()<<","<<screen.get_ny()<<"]"<<endl;
        cout<<"Field Of View:"<<screen.get_theta()*360/M_PI<<endl;
        cout<<"Number Of Object:"<<listObjs.size()<<endl;
        for (int i=0;i<listObjs.size();i++)
        {
                cout<<"Object "<<i<<":"<<endl;
                listObjs[i]->printInfo();
        }
}
bool find1stHit(const QVector& rayOrig, const QVector& rayDir,int *hit_id, double *tmin){
        *tmin = INFINITY;
        *hit_id = -1;
        for (int i=0;i<listObjs.size();i++){
                double t = -1;
                if(listObjs[i]->intersect(rayOrig, rayDir, &t)){
                        if(t < *tmin){
                                *tmin = t;
                                *hit_id = i;
                        }
                }
        }
        if(*hit_id == -1 || abs(*tmin - INFINITY) < EPSILON)
                return false;
        return true;
}

bool isInShadowArea(const QVector& rayOrig, const QVector& rayDir, int obj_id){
        for (int i=0;i<listObjs.size();i++){
                if(i == obj_id)
                        continue;
                double t = -1;
                if(listObjs[i]->intersect(rayOrig, rayDir, &t)){
                        if(t > 0 && t < 1 && abs(listObjs[i]->getRefraction() - 1) > EPSILON
)//ignore total transparent obj
                                return true;
                }
        }
        return false;
}
/*return color*/
QVector trace(const QVector& rayOrig, const QVector& rayDir, double eta_I, double eta_T, int
depth_level){
        double tmin = -1;
```

```
	int hit_id = -1;
	QVector pColor;
	if(find1stHit(rayOrig,rayDir,&hit_id,&tmin)){
		QObject *hitObj = listObjs[hit_id];
		QVector phit = rayOrig + rayDir*tmin;
		QVector N = hitObj->getNormalAt(phit);
		double cosTheta = QVector::cos(lightSrc - phit,N);
		double alpha = ((1-MIN_DARK)*cosTheta + MIN_DARK + 1)/2;
		QVector hitColor = hitObj->getColor()*alpha;//color of hitting point

		QVector reflecColor;
		QVector refracColor;
		QVector shadowColor;

		if(depth_level<DEPTH_MAX && (hitObj->getReflection() > 0 || hitObj->getRefraction()
> 0 )){
			// reflection
			QVector R = QVector::reflect(rayDir,N);
			//recurse trace
			if(hitObj->getReflection() > 0){
				reflecColor = trace(phit, R ,eta_I, eta_T, depth_level+1);
			}
			QVector T;
			if(hitObj->getRefraction() > 0){
				// refraction
				T = QVector::refract(rayDir,N,eta_I,hitObj->getEta());
				if (hitObj->getObjType() != OBJ_TYPE_TRIANGLE){
					refracColor = trace(phit+N*EPSILON, T,hitObj->getEta(),
eta_I, depth_level+1);
				} else {
					refracColor = trace(phit+N*EPSILON, T, eta_I,eta_I,
depth_level+1);
				}
			}
		}
		//combine colors
		pColor = hitColor*(1-hitObj->getReflection()-hitObj->getRefraction()) +
reflecColor*hitObj->getReflection() + refracColor*hitObj->getRefraction();
		//shadow effect
		QVector shadowRay = lightSrc - phit;
		bool isInShadow = false;
		if(isInShadowArea(phit,shadowRay, hit_id)){
			pColor = pColor*SHADOW;
			isInShadow = true;
		}
		if(!isInShadow && hitObj->isSqecular()){// Don't have highlight effect in shadow
area
			QVector R = QVector::reflect(phit-lightSrc,N);
			double costheta2 = pow(QVector::cos(R,rayOrig-phit),N_HL);
			//if(depth_level == 1 && costheta2 > HIGHLIGHT_MIN){
			if(costheta2 > HIGHLIGHT_MIN){
				QVector w = QVector::parseColor(COLOR_WHITE);
```

```
                                QVector HC = (pColor*(1-costheta2) + w*(costheta2-
HIGHLIGHT_MIN))/(1-HIGHLIGHT_MIN);
                                pColor = HC;
                        }
                }

        }else{
                pColor = backgroundColor;
        }
        return pColor;
}
void display(){
        glClear( GL_COLOR_BUFFER_BIT);
        //readInputFile(true);
        glBegin( GL_POINTS );
        for( int i=-nx2; i<nx2; i++)
                for( int j=-ny2; j<ny2; j++){
                        glVertex2i( i,j);
                        QVector pColor;
                        if (!ANTI_ALIASING){
                                QVector p = QVector(i, j, -(screen.get_distance2eye()),1);
                                p = g_view.calPosInWorldCoord(p);
                                QVector ray = p-eye;
                                pColor = trace(eye,ray,AIR_ETA,INFINITY,1);
                        }else{
                                //Jittered anti-aliasing
                                for(int xx = 0; xx<NUM_SUB_ANTI_ALIASING; xx++){
                                        for(int yy = 0; yy<NUM_SUB_ANTI_ALIASING; yy++){
                                                // random a number from 0 to 1
                                                double randx = (rand()%100)/100.0;

                                                double sub_x = ((i-1-xx*1.0/NUM_SUB_ANTI_ALIASING) +
1.0/NUM_SUB_ANTI_ALIASING*randx);
                                                double randy = (rand()%100)/100.0;
                                                double sub_y = ((j-1-yy*1.0/NUM_SUB_ANTI_ALIASING) +
1.0/NUM_SUB_ANTI_ALIASING*randy);
                                                QVector p = QVector(sub_x, sub_y, -
(screen.get_distance2eye()),1);
                                                p = g_view.calPosInWorldCoord(p);
                                                QVector ray = p-eye;
                                                pColor = pColor + trace(eye,ray,AIR_ETA,INFINITY,1);
                                        }
                                }
                                pColor = pColor/(NUM_SUB_ANTI_ALIASING*NUM_SUB_ANTI_ALIASING);
                        }
                        glColor3d( pColor.getX(), pColor.getY(), pColor.getZ());
                }
        glEnd();
        glFlush();
}

int main( int argc, char **argv){
```

```
        glutInit( &argc, argv );
        if (argc>=2){
                data = argv[1];
        }
        cout<<"Input file:"<<data<<endl;
        readInputFile();
        glutInitDisplayMode( GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH );
        glutInitWindowPosition( 100, 100);
        glutInitWindowSize( screen.get_nx(),screen.get_ny() );
        glutCreateWindow( "Figure Drawing" );
        glClearColor( backgroundColor.getX(), backgroundColor.getY(), backgroundColor.getZ(),0 );
        glMatrixMode( GL_PROJECTION );
        glLoadIdentity();
        glOrtho( -nx2, nx2, -ny2, ny2, -1.0, 1.0 );
        glutDisplayFunc( display );
        glutMainLoop();
        getchar();
        return 0;
}
```

define.h

```
#ifndef DEFINE_H
#define DEFINE_H

#define M_PI 3.141592653589793
#define INFINITY 1e8
#define MIN_DARK 0.1
#define N_HL 3
#define HIGHLIGHT_MIN 0.9
#define EPSILON pow(10.0,-10)
#define SHADOW 0.8
#define DEPTH_MAX 7
#define AIR_ETA 1
#define ANTI_ALIASING 1
#define NUM_SUB_ANTI_ALIASING 3

// Colors
#define COLOR_WHITE           0xffffff
#define COLOR_BLACK           0x000000
#define COLOR_CHECKBORAD1 0xFF692D
#define COLOR_SKY             0x87CEEB

#define OBJ_TYPE_OBJ 0
#define OBJ_TYPE_PLANE 1
#define OBJ_TYPE_BALL 2
#define OBJ_TYPE_IMPLICIT 3
#define OBJ_TYPE_TRIANGLE 4
#endif
```

QVector.h

```
#ifndef QVECTOR_H
#define QVECTOR_H

#include <string>
#include "define.h"
using namespace std;

class QVector
{
private:
	double x;
	double y;
	double z;
	double t; // Point = 1, vector = 0 or and arbitrary values for 4-D vector
public:
	QVector();
	QVector(double _x, double _y, double _z):x(_x), y(_y), z(_z) {t=0;};
	QVector(double _x, double _y, double _z, double _t):x(_x), y(_y), z(_z), t(_t) {};
	QVector(const QVector& _other);
	void setX(double _x);
	void setY(double _y);
	void setZ(double _z);
	void setT(double _t);
	string toString();
	double getX() const;
	double getY() const;
	double getZ() const;
	double getT() const;

	QVector& operator=(const QVector& _other);
	bool operator==(const QVector& _other);

	QVector operator+(const QVector& _other) const;
	QVector operator-(const QVector& _other) const;
	QVector operator-() const;
	QVector operator*(double c) const; //scalar multiplication
	double operator*(const QVector& _other) const;// dot product
	QVector operator/(double c) const; //scalar multiplication
	QVector operator&(const QVector& _other) const;// cross product
	friend std::ostream & operator << (std::ostream &os, const QVector &v);

	static double dotProduct(const QVector& v1, const QVector& v2);
	static QVector crossProduct(const QVector& v1, const QVector& v2);
	QVector& normalize();
	double norm() const;
	// Distance to a point
	static double distance(const QVector& v, const QVector& p);
	double distance(const QVector& p);
	// Dot product with four elements
	static double dot4(const QVector& p, const QVector& q);
```

```
        static QVector reflect(QVector I, QVector N);
        static QVector refract(const QVector& I,const QVector& N, double eta_I, double eta_T);
        static double cos(QVector V1, QVector V2);
        static QVector parseColor(int c);
};
#endif
```

QVector.cpp

```
#include "QVector.h"

QVector::QVector(){
        x = 0;
        y = 0;
        z = 0;
        t = 0;
}
QVector::QVector(const QVector& _other){
        QVector();
        x = _other.getX();
        y = _other.getY();
        z = _other.getZ();
        t = _other.getT();
}
void QVector::setX(double _x){x = _x;}
void QVector::setY(double _y){y = _y;}
void QVector::setZ(double _z){z = _z;}
void QVector::setT(double _t){t = _t;}
double QVector::getX() const{ return x;}
double QVector::getY() const{ return y;}
double QVector::getZ() const{ return z;}
double QVector::getT() const{ return t;}
string QVector::toString(){
        string str = "("+ std::to_string((long double)x) + "," + std::to_string((long double)y) +
"," + std::to_string((long double)z) + ")";
        return str;
}

QVector& QVector::operator=(const QVector& _other){
        if(this == &_other)
                return *this;
        x = _other.getX();
        y = _other.getY();
        z = _other.getZ();
        t = _other.getT();
        return *this;
}
bool QVector::operator==(const QVector& _other){
        return abs(x-_other.getX())<EPSILON && abs(y-_other.getY())<EPSILON && abs(z-
_other.getZ())<EPSILON;
}
```

```
QVector QVector::operator+(const QVector& _other) const{
	double _x = x + _other.getX();
	double _y = y + _other.getY();
	double _z = z + _other.getZ();
	return QVector(_x,_y,_z);
}
QVector QVector::operator-(const QVector& _other) const{
	double _x = x - _other.getX();
	double _y = y - _other.getY();
	double _z = z - _other.getZ();
	return QVector(_x,_y,_z);
}
QVector QVector::operator-() const{
	return QVector(-x,-y,-z,t);
}
//scalar multiplication
QVector QVector::operator*(double c) const{
	return QVector(x*c, y*c, z*c,t);
}
//scalar division
QVector QVector::operator/(double c) const{
	return QVector(x/c, y/c, z/c,t);
}
//dot product
double QVector::operator*(const QVector& _other) const{
	return dotProduct(*this, _other);
}
QVector QVector::operator&(const QVector& _other) const{// cross product
	return crossProduct(*this, _other);
}

std::ostream & operator << (std::ostream &os, const QVector &v)
{
	os << "[" << v.x << " " << v.y << " " << v.z << "]";
	return os;
}


double QVector::dotProduct(const QVector& v1, const QVector& v2){
	return v1.getX()*v2.getX()+v1.getY()*v2.getY()+v1.getZ()*v2.getZ();
}
QVector QVector::crossProduct(const QVector& v1, const QVector& v2){
	double _x = v1.getY() * v2.getZ() - v2.getY() * v1.getZ();
	double _y = -(v1.getX() * v2.getZ() - v2.getX() * v1.getZ());
	double _z = v1.getX() * v2.getY() - v2.getX() * v1.getY();
	return QVector(_x,_y,_z);
}

double QVector::norm() const{
	return sqrt(x*x+y*y+z*z);
}
QVector& QVector::normalize(){
```

```
        double n = norm();
        x /=n; y/=n; z/=n;
        return *this;
}
// Distance to a point
double QVector::distance(const QVector& v, const QVector& p){
        return (v-p).norm();
}

double QVector::distance(const QVector& p){
        return (*this-p).norm();
}
double QVector::dot4(const QVector& p, const QVector& q){
        return p.getX()*q.getX()+p.getY()*q.getY()+p.getZ()*q.getZ()+p.getT()*q.getT();
}

QVector QVector::reflect(QVector I, QVector N){
        return I - N*((I*N)/(N*N))*2;
}

QVector QVector::refract(const QVector& I,const QVector& N, double eta_I, double eta_T){
        QVector NN = N/(N.norm());
        QVector K = NN*((-I)*NN);
        double sin_phiT = (eta_I/eta_T)*(sqrt(1-pow(QVector::cos(-I,NN),2)));
        double cos_phiT = sqrt(1-pow(sin_phiT,2));
        QVector M = K.normalize()*I.norm()*sin_phiT;
        QVector N_ = -NN * I.norm() * cos_phiT;
        return M + N_;
}
double QVector::cos(QVector V1, QVector V2){
        return (V1*V2)/(V1.norm()*V2.norm());
}

QVector QVector::parseColor(int c){
        double r = ((c & 0xff0000) / 0xffff) / 255.0;
        double g = ((c & 0x00ff00) / 0xff) / 255.0;
        double b = ((c & 0x0000ff)) / 255.0;
        return QVector(r,g,b);
}
```

ScreenParams.h

```
#ifndef SCREEN_PARAMS
#define SCREEN_PARAMS

#include "QVector.h"
class ScreenParams {
private:
        double nx;
        double ny;
        double theta;// field of view
```

```
private:
	double d;// distance from eye to screen
public:
	ScreenParams(){d=-1;};
	ScreenParams(double _nx, double _ny, double _theta):nx(_nx), ny(_ny), theta(_theta){

		update_distance2eye();
	};
	double get_nx(){return nx;};
	double get_ny(){return ny;};
	double get_theta(){return theta;};
	double update_distance2eye(){d=ny/(2*tan(theta/2)); return d;};
	double get_distance2eye(){
		return d;
	}
};
#endif
```

ViewingParams.h

```
#ifndef VIEWINGPARAMS_H
#define VIEWINGPARAMS_H

#include "QVector.h"
#include <vector>

class ViewingParams{
private:
	//input parameters
	QVector eye;
	QVector dir; //direction vector
	QVector headUp; //head-up vector

	//calculated values
	std::vector<QVector> eyeCoord;
	std::vector<QVector> transMatrix;
public:
	ViewingParams();
	ViewingParams(const QVector& _eye, const QVector& _dir, const QVector& _headUp );
	QVector getEye() const {return eye;};
	QVector getDirection() const {return dir;};
	QVector getHeadUp() const {return headUp;};
	std::vector<QVector> updateEyeCoordinate();
	std::vector<QVector> getEyeCoordinate();
	std::vector<QVector> updateVewingTranformationMatrix();
	std::vector<QVector> getVewingTranformationMatrix();
	//convert a point in eye coordinate to world coordinate using viewing transformation
matrix
	QVector calPosInWorldCoord(const QVector& p );
};
#endif
```

ViewingParams.cpp

```
#include "ViewingParams.h"

ViewingParams::ViewingParams(){
}
ViewingParams::ViewingParams(const QVector& _eye, const QVector& _dir, const QVector& _headUp ){
	eye = _eye;
	dir	= _dir;
	headUp = _headUp;
	this->updateEyeCoordinate();
	this->updateVewingTranformationMatrix();
}
std::vector<QVector> ViewingParams::updateEyeCoordinate(){
	QVector w = -dir;
	QVector u = dir & headUp;//cross product
	QVector v = w & u;
	w.normalize(); u.normalize(); v.normalize();
	eyeCoord.clear();
	eyeCoord.push_back(u);// append one element to a vector
	eyeCoord.push_back(v);
	eyeCoord.push_back(w);
	return eyeCoord;
}
std::vector<QVector> ViewingParams::getEyeCoordinate(){
	return eyeCoord;
}
std::vector<QVector> ViewingParams::updateVewingTranformationMatrix(){
	QVector row1(eyeCoord[0].getX(),eyeCoord[1].getX(),eyeCoord[2].getX(),eye.getX());
	QVector row2(eyeCoord[0].getY(),eyeCoord[1].getY(),eyeCoord[2].getY(),eye.getY());
	QVector row3(eyeCoord[0].getZ(),eyeCoord[1].getZ(),eyeCoord[2].getZ(),eye.getZ());
	QVector row4(0,0,0,1);
	transMatrix.clear();
	transMatrix.push_back(row1);
	transMatrix.push_back(row2);
	transMatrix.push_back(row3);
	transMatrix.push_back(row4);
	return transMatrix;
}
std::vector<QVector> ViewingParams::getVewingTranformationMatrix(){
	return transMatrix;
}
//transform a point from eye coordinate to world coordinate using viewing transformation matrix
QVector ViewingParams::calPosInWorldCoord(const QVector& p ){
	//P_w = M . P_e
	double _x = QVector::dot4(transMatrix[0],p);
	double _y = QVector::dot4(transMatrix[1],p);
	double _z = QVector::dot4(transMatrix[2],p);
	return QVector(_x, _y, _z,1);
}
```

QBall.h

```
#ifndef QBALL_H
#define QBALL_H

#include "QObject.h"
#include "QVector.h"

class QBall:public QObject {
private:
	QVector center;
	double radius; //radius
	QVector color;
private:
	QVector phit; // temporary hitting point
public:
	QBall();
	QBall(const QVector& _center, double _r, const QVector& _color);
	QBall(const QVector& _center, double _r, const QVector& _color,bool _is_sqecular, double
_reflection, double _refraction, double _eta);
	QBall(const QBall& _other);
	QVector getCenter() const;
	void setCenter(const QVector& _center) ;
	double getRadius() const;
	void setRadius(double _r);
	QVector getColor() const {return color;};
	QVector getNormalAt(QVector phit);
	bool intersect(const QVector& rayOrig, const QVector& rayDir, double* hit);
	void printInfo();
	int getObjType(){ return OBJ_TYPE_BALL;}
};

#endif /*QBALL_H*/
```

QBall.cpp

```
#include "QBall.h"

QBall::QBall(){
	center = QVector();
}
QBall::QBall(const QVector& _center, double _r, const QVector& _color){
	QObject();
	center = _center;
	radius = _r;
	color = _color;
}
QBall::QBall(const QVector& _center, double _r, const QVector& _color,bool _is_sqecular, double
_reflection, double _refraction, double _eta){
	center = _center;
	radius = _r;
```

```
        color = _color;
        is_sqecular = _is_sqecular;
        reflection = _reflection;
        refraction = _refraction;
        eta = _eta;
}
QBall::QBall(const QBall& _other){
        center = _other.getCenter();
        radius = _other.getRadius();
}
QVector QBall::getCenter() const{return center;}
void QBall::setCenter(const QVector& _center){center = _center;}
double QBall::getRadius() const{return radius;}
void QBall::setRadius(double _r){radius = _r;}

QVector QBall::getNormalAt(QVector phit){
        return (phit - center).normalize();
}
bool QBall::intersect(const QVector& rayOrig, const QVector& rayDir, double* t){
        double a = rayDir*rayDir;
        double b = rayDir*(rayOrig-center);
        double c = (rayOrig-center)*(rayOrig-center) - radius*radius;
        double d = b*b - a*c;
        if (d < EPSILON) // d<=0
                return false;
        double t1 = (-b-sqrt(d))/a;
        double t2 = (-b+sqrt(d))/a;
        if (t1 < 0 && t2 < 0)
                return false;
        // choose smaller positive t for 1st hitting point
        *t = t1 < t2 ? (t1>0 ? t1 : t2) : (t2>0 ? t2 : t1);
        if(abs(*t) < EPSILON)
                return false;
        phit = rayOrig + rayDir*(*t);
        return true;
}

void QBall::printInfo(){
        cout<<"WtType:  Ball"<<endl;
        cout<<"WtData:"<<endl;
        cout<<"WtWtCenter:"<<center.toString()<<endl;
        cout<<"WtWtRadius:"<<radius<<endl;
        cout<<"WtWt[reflection, refraction, eta] = "<<getObjectInfo()<<endl;
}
```

QPlane.h

```
#ifndef QPLANE_H
#define QPLANE_H

#include "QObject.h"
#include "QVector.h"
```

```
class QPlane: public QObject{
private:
	QVector point;  //a point
	QVector normal; //a normal vector
private:
	QVector phit; // temporary hitting point
public:
	QPlane();
	QPlane(const QVector& _point, const QVector& _normal);
	QPlane(const QVector& _point, const QVector& _normal,bool _is_sqecular, double
_reflection, double _refraction, double _eta);
	QVector getPoint() const;
	QVector getNormal() const;
	QVector getNormalAt(QVector phit);
	QVector getColor() const;
	QVector getHittingPoint() {return phit;};
	bool intersect(const QVector& rayOrig, const QVector& rayDir, double* t);
	void printInfo();
	int getObjType(){ return OBJ_TYPE_PLANE;}
};
#endif
```

QPlane.cpp

```
#include "QPlane.h"

QPlane::QPlane(){
	point = QVector();
	normal = QVector();
}
QPlane::QPlane(const QVector& _point, const QVector& _normal){
	QObject();
	point = _point;
	normal = _normal;
	normal.normalize();
}
QPlane::QPlane(const QVector& _point, const QVector& _normal,bool _is_sqecular, double
_reflection, double _refraction, double _eta){
	point = _point;
	normal = _normal;
	normal.normalize();
	is_sqecular = _is_sqecular;
	reflection = _reflection;
	refraction = _refraction;
	eta = _eta;
}
QVector QPlane::getPoint() const {return point;}
QVector QPlane::getNormal() const{return normal;}
QVector QPlane::getNormalAt(QVector phit){return normal.normalize();}
QVector QPlane::getColor() const{
```

```
        int sq = (int)floor(phit.getX()/250) + (int)floor(phit.getY()/250);
        if(sq % 2 == 0){
                return QVector::parseColor(COLOR_CHECKBORAD1);
        }else{
                return QVector::parseColor(COLOR_WHITE);
        }
}
bool QPlane::intersect(const QVector& rayOrig, const QVector& rayDir, double* t){
        double k = rayDir*normal;
        if( abs(k) < EPSILON){//k == 0
                return false;
        }
        *t = (-(rayOrig - point)*normal)/k;
        if(*t<EPSILON)
                return false;
        phit = rayOrig + rayDir*(*t);
        return true;
}
void QPlane::printInfo(){
        cout<<"₩tType:  Plane"<<endl;
        cout<<"₩tData:"<<endl;
        cout<<"₩t₩tPoint:"<<point.toString()<<endl;
        cout<<"₩t₩tNormal Vector:"<<normal.toString()<<endl;
        cout<<"₩t₩t[reflection, refraction, eta] = "<<getObjectInfo()<<endl;
}
```

QTriangle.h

```
#ifndef QTRIANGLE_H
#define QTRIANGLE_H
#include "QObject.h"
#include "QVector.h"

class QTriangle:public QObject {
private:
        QVector p0;
        QVector p1;
        QVector p2;
        QVector color;
private:
        //pre-computed values
        QVector u_hat;
        QVector v_hat;
        QVector normal;
public:
        QTriangle();
        QTriangle(const QVector& _p0,const QVector& _p1,const QVector& _p2, const QVector&
_color,bool _is_sqecular, double _reflection, double _refraction, double _eta);
        QVector getColor() const {return color;};
        QVector getNormalAt(QVector phit){ return normal;};
        QVector getP0() const {return p0;}
        QVector getP1() const {return p1;}
```

```
        QVector getP2() const {return p2;}
        bool intersect(const QVector& rayOrig, const QVector& rayDir, double* t);
        void printInfo();
        int getObjType(){ return OBJ_TYPE_TRIANGLE;}
};
#endif
```

QTriangle.cpp

```
#include "QTriangle.h"

QTriangle::QTriangle(){
        p0 = QVector();
        p1 = QVector();
        p2 = QVector();
        color = QVector();
}
QTriangle::QTriangle(const QVector& _p0,const QVector& _p1,const QVector& _p2, const QVector&
_color,bool _is_sqecular, double _reflection, double _refraction, double _eta){
        p0 = _p0;
        p1 = _p1;
        p2 = _p2;
        color = _color;
        is_sqecular = _is_sqecular;
        reflection = _reflection;
        refraction = _refraction;
        eta = _eta;

        QVector p0p2 = p2-p0;
        QVector p0p1 = p1-p0;
        normal =  (p0p1 & p0p2).normalize(); //cross product
        u_hat = (p0p2 & normal) / ((p0p1 & p0p2) * normal);
        v_hat = (normal & p0p1) / ((p0p1 & p0p2) * normal);
}
bool QTriangle::intersect(const QVector& rayOrig, const QVector& rayDir, double* t){
        double k = rayDir*normal;
        if( abs(k) < EPSILON){//k == 0
                return false;
        }
        *t = (-(rayOrig - p0)*normal)/k;
        if(*t<EPSILON)
                return false;
        QVector phit = rayOrig + rayDir*(*t);
        double u = (phit - p0)*u_hat;
        double v = (phit - p0)*v_hat;
        if( u > 0 && v > 0 && u+v < 1)
                return true;
        return false;
}
void QTriangle::printInfo(){}
```

QImplicitSurface.h

```
#ifndef QIMPLICITSURFACE_H
#define QIMPLICITSURFACE_H

#include <vector>
#include "QObject.h"
#include "QVector.h"

class QImplicitSurface: public QObject{
private:
	std::vector<QVector> centers;
	double T; // iso value T
	QVector color;
	double a;
	double b;
private:
	QVector Bmin;
	QVector Bmax;
private:
	double calEnergy(const QVector& c,const QVector& p);
	double calTotalEnergy(const QVector& p);
	double calTotalEnergy(const QVector& rayOrig, const QVector& rayDir,double t);
	bool intervalApproximate(double t1, double t2, double *t,const QVector& rayOrig, const
QVector& rayDir);
	void calBoundary();
	bool calInitialRange(const QVector& rayOrig, const QVector& rayDir, double *t1, double
*t2);
	void calQuadRange(double coff_a, double coff_b, double coff_c, double t1, double t2,
double *r1, double *r2);
	void calEnergyRange(double t1, double t2,const QVector& rayOrig, const QVector& rayDir,
double *f_min, double *f_max);
	void swap(double *t1,double *t2){double t=*t1;*t1=*t2;*t2=t;};

public:
	QImplicitSurface();
	QImplicitSurface(const std::vector<QVector>& _centers, double _T, const QVector& _color);
	QImplicitSurface(const std::vector<QVector>& _centers, double _T, const QVector&
_color,bool _is_sqecular, double _reflection, double _refraction, double _eta);
	std::vector<QVector> getCenters() const;
	void setCenters(const std::vector<QVector>& _center);
	double getT() const;
	void setT(double _T);
	QVector getColor() const {return color;};
	QVector getNormalAt(QVector phit);
	bool intersect(const QVector& rayOrig, const QVector& rayDir, double* t);
	void printInfo();
	int getObjType(){ return OBJ_TYPE_IMPLICIT;}

};
#endif
```

QImplicitSurface.cpp

```
#include "QImplicitSurface.h"

QImplicitSurface::QImplicitSurface(){
	QObject();
}
QImplicitSurface::QImplicitSurface(const std::vector<QVector>& _centers, double _T, const QVector&
_color){
	QObject();
	centers = _centers;
	T = _T;
	color = _color;
	calBoundary();
}
QImplicitSurface::QImplicitSurface(const std::vector<QVector>& _centers, double _T, const QVector&
_color,bool _is_sqecular, double _reflection, double _refraction, double _eta){
	QObject();
	centers = _centers;
	T = _T;
	color = _color;
	is_sqecular = _is_sqecular;
	reflection = _reflection;
	refraction = _refraction;
	eta = _eta;
	a = 130;
	b = 0.000005;
	//b = 0.001;
	calBoundary();
}
std::vector<QVector> QImplicitSurface::getCenters() const{return centers;}
void QImplicitSurface::setCenters(const std::vector<QVector>& _center){centers = _center;}
double QImplicitSurface::getT() const{return T;}
void QImplicitSurface::setT(double _T){T = _T;}
QVector QImplicitSurface::getNormalAt(QVector phit){
	QVector dx(EPSILON,0,0);
	QVector dy(0,EPSILON,0);
	QVector dz(0,0,EPSILON);
	double dfx = calTotalEnergy(phit+dx)-calTotalEnergy(phit-dx);
	double dfy = calTotalEnergy(phit+dy)-calTotalEnergy(phit-dy);
	double dfz = calTotalEnergy(phit+dz)-calTotalEnergy(phit-dz);
	QVector normal(dfx,dfy,dfz);
	return -(normal);
}
bool QImplicitSurface::intervalApproximate(double t1, double t2, double *t,const QVector& rayOrig,
const QVector& rayDir){
	double m = (t1+t2)/2;
	// Calculate interval of f([t1, t2])
	double f_min=0, f_max=0;
	calEnergyRange(t1,t2,rayOrig, rayDir,&f_min,&f_max);

	if(!(f_min<=EPSILON && f_max>=EPSILON))
```

```
                return false;
        if(abs(t2-t1) < EPSILON){
                *t = m;
                return true;
        }
        if(intervalApproximate(t1,m,t,rayOrig,rayDir))
                return true;
        return intervalApproximate(m,t2,t,rayOrig,rayDir);
}
bool QImplicitSurface::intersect(const QVector& rayOrig, const QVector& rayDir, double* t){
        //calculate initial interval
        // check whether ray hit the boundary volume (Bmin, Bmax) or not
        double t1, t2;
        if (callnitialRange(rayOrig,rayDir,&t1,&t2)){
                QVector p1 = rayOrig + rayDir*t1;
                QVector p2 = rayOrig + rayDir*t2;
                return intervalApproximate(t1,t2,t,rayOrig,rayDir) && *t>EPSILON;
        }
        return false;
}
bool QImplicitSurface::callnitialRange(const QVector& rayOrig, const QVector& rayDir, double *t1,
double *t2){
        double tmin_x, tmin_y, tmin_z;
        double tmax_x, tmax_y, tmax_z;
        if (rayDir.getX() == 0){
                tmin_x = -INFINITY;
                tmax_x = INFINITY;
        } else {
                tmin_x = (Bmin.getX()-rayOrig.getX())/rayDir.getX();
                tmax_x = (Bmax.getX()-rayOrig.getX())/rayDir.getX();
                if (rayDir.getX() < 0){
                        swap(&tmin_x,&tmax_x);
                }
        }
        if (rayDir.getY() == 0){
                tmin_y = -INFINITY;
                tmax_y = INFINITY;
        } else {
                tmin_y = (Bmin.getY()-rayOrig.getY())/rayDir.getY();
                tmax_y = (Bmax.getY()-rayOrig.getY())/rayDir.getY();
                if (rayDir.getY() < 0){
                        swap(&tmin_y,&tmax_y);
                }
        }
        if (rayDir.getZ() == 0){
                tmin_z = -INFINITY;
                tmax_z = INFINITY;
        } else {
                tmin_z = (Bmin.getZ()-rayOrig.getZ())/rayDir.getZ();
                tmax_z = (Bmax.getZ()-rayOrig.getZ())/rayDir.getZ();
                if (rayDir.getZ() < 0){
                        swap(&tmin_z,&tmax_z);
```

```
                }
        }
        *t1 = std::max(std::max(tmin_x,tmin_y),tmin_z);
        *t2 = std::min(std::min(tmax_x,tmax_y),tmax_z);
        return *t1<*t2;
}

void QImplicitSurface::calQuadRange(double coff_a, double coff_b, double coff_c, double t1, double
t2, double *r1, double *r2){
        if (abs(coff_a)<EPSILON) {
                *r1 = coff_b*t1 + coff_c;
                *r2 = coff_b*t2 + coff_c;
                if (coff_b<0) swap(r1, r2);
                return;
        }
        double extp = -coff_b/2/coff_a;
        double v1 = coff_a*t1*t1 + coff_b*t1 +coff_c;
        double v2 = coff_a*t2*t2 + coff_b*t2 + coff_c;
        double extrema = coff_c-coff_b*coff_b/4/coff_a;
        if (extp > t1 && extp < t2){
                *r1 = std::min(std::min(v1,v2),extrema);
                *r2 = std::max(std::max(v1,v2),extrema);
        }else{
                *r1 = std::min(v1,v2);
                *r2 = std::max(v1,v2);
        }
}

void QImplicitSurface::calEnergyRange(double t1, double t2,const QVector& rayOrig, const QVector&
rayDir, double *f_min, double *f_max){
        *f_min=-T, *f_max=-T;
        double r_min, r_max;
        for (int i=0;i<centers.size();i++){
                double coff_a = rayDir*rayDir;
                double coff_b = (rayOrig-centers[i])*rayDir*2;
                double coff_c = (rayOrig-centers[i])*(rayOrig-centers[i]);
                calQuadRange(coff_a,coff_b,coff_c,t1,t2,&r_min,&r_max);
                *f_min += a*exp(-b*r_max);
                *f_max += a*exp(-b*r_min);
        }
}
double QImplicitSurface::calEnergy(const QVector& c,const QVector& p){
        return a*exp(-b*pow(QVector::distance(p,c),2));
}
double QImplicitSurface::calTotalEnergy(const QVector& p){
        double ret = 0;
        for (int i = 0; i<centers.size();i++){
                ret += calEnergy(centers[i],p);
        }
        return ret;
}
double QImplicitSurface::calTotalEnergy(const QVector& rayOrig, const QVector& rayDir,double t){
```

```
        QVector p = rayOrig + rayDir*t;
        return calTotalEnergy(p);
}
void QImplicitSurface::calBoundary(){
        Bmin = QVector(INFINITY, INFINITY, INFINITY);
        Bmax = QVector(-INFINITY, -INFINITY, -INFINITY);
        for (int i=0;i<centers.size();i++){
                // calculate total energy at i-th center
                double e = calTotalEnergy(centers[i]);
                double r_max = 2*sqrt(log(e/T)/b);//multiply by 2 to make sure it contain objects
                //r_max = 300;
                QVector p1 = centers[i]-QVector(r_max,r_max,r_max);
                QVector p2 = centers[i]+QVector(r_max,r_max,r_max);
                if(Bmin.getX() > p1.getX()){Bmin.setX(p1.getX());}
                if(Bmax.getX() < p2.getX()){Bmax.setX(p2.getX());}
                if(Bmin.getY() > p1.getY()){Bmin.setY(p1.getY());}
                if(Bmax.getY() < p2.getY()){Bmax.setY(p2.getY());}
                if(Bmin.getZ() > p1.getZ()){Bmin.setZ(p1.getZ());}
                if(Bmax.getZ() < p2.getZ()){Bmax.setZ(p2.getZ());}
        }
}
void QImplicitSurface::printInfo(){}
```