**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ  
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра систем автоматизированного проектирования**

**ОТЧЁТ**

**по лабораторной работе №6  
по дисциплине «Компьютерная графика»**

**Тема: . Формирования реалистических изображений с использованием**

**простых моделей освещения одним или двумя точечными источниками**

|  |  |
| --- | --- |
| Студенты \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Матвеева И.В. |

Санкт-Петербург

2024

**Задание на лабораторную работу**

Вариант 5: Реализовать алгоритм трассировки лучей при формировании сложной сцены.

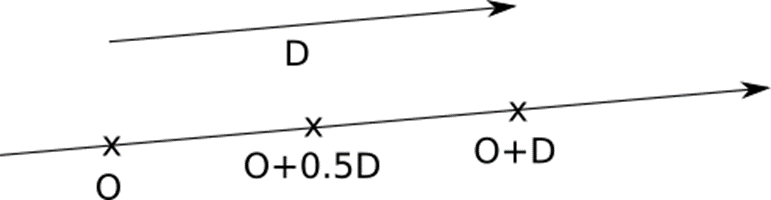
**Теоретические положения**

Трассировка лучей — один из методов геометрической оптики — исследование оптических систем путем отслеживания взаимодействия отдельных лучей с поверхностями.

Наилучшим способом представления лучей для нашей цели будет использование параметрического уравнения. Мы можем выразить любую точку луча как – , где – произвольное действительное число > 1, – положение камеры, – точка просмотра.

Обозначим , то есть направление луча, как ; тогда уравнение примет простой вид

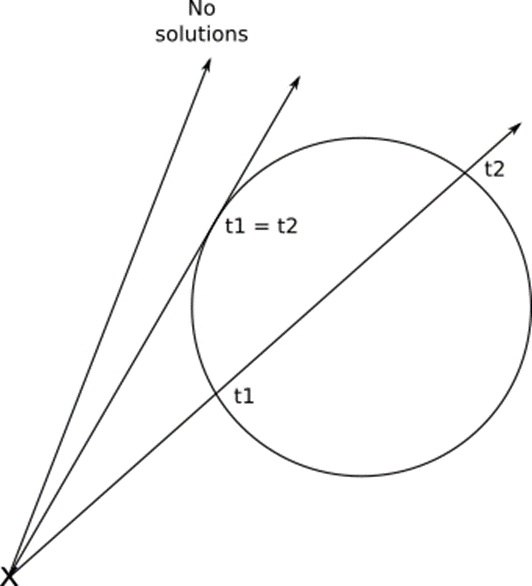
Если мы начнём из начальной точки и продвинемся на какое-нибудь кратное направления луча, то всегда будем двигаться вдоль луча:



Сферу можно представить следующим уравнением:

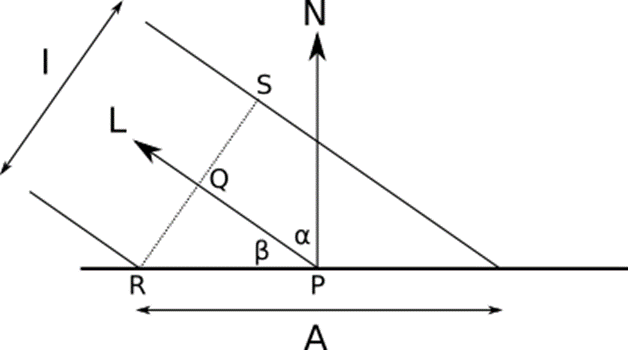
Точки пересечения со сферой можно найти следующим образом:

,



Сформулируем уравнение диффузного отражения. Оно используется для вычисления полного количества света, полученного точкой с нормалью в сцене с окружающим освещением яркостью и точечных и направленных источников света с яркостью и световыми векторами или известными (для направленных источников), или вычисленными для (для точечных источников):

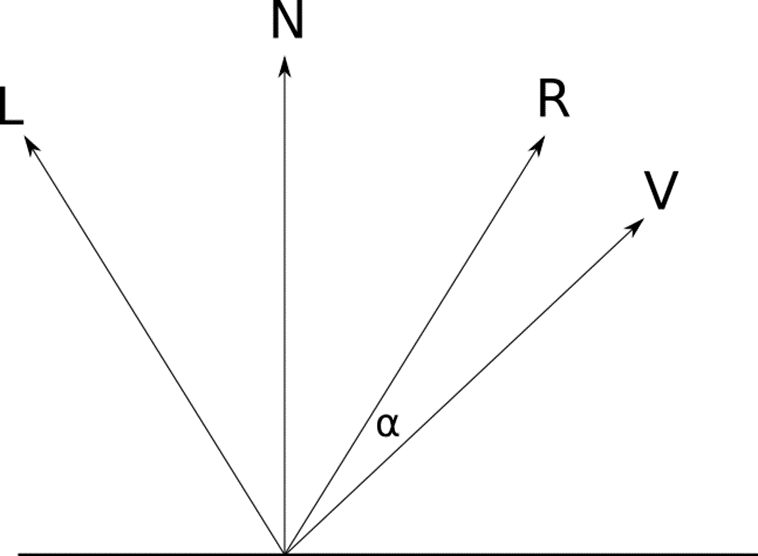
Исключением является < 0, так как освещённость не может быть отрицательной.



Нормаль сферы можно найти с помощью уравнения:

Уравнение зеркального отражения:

– коэффициент падающего луча; – вектор обзора; – нормаль; – падающий луч света; – отражённый свет; – отражённый свет; – коэффициент отраженного света; s – показатель отражения.



Проверка теней осуществляется следующим способом: «Если между источником света и точкой есть другой объект, то освещение от этого источника учитывать не надо». Проверка осуществляется на подобии испускания луча света из источника к сфере, только в другом направлении. Следовательно, можно представить луч следующим уравнением:

Только один случай, когда . При данном параметре получается так, что объект отбрасывает тень на самого себя. Чтобы этого избежать примем .

Отражение работает также как тени, с одним исключением, что направление луча, которое было становится . Для того, чтобы не произошел эффект «бесконечного коридора» нужно ввести предел рекурсии r.

**Ход работы**

Функция нахождения пересечения луча со сферой – ClosestIntersection(O, D, t\_min, t\_max).

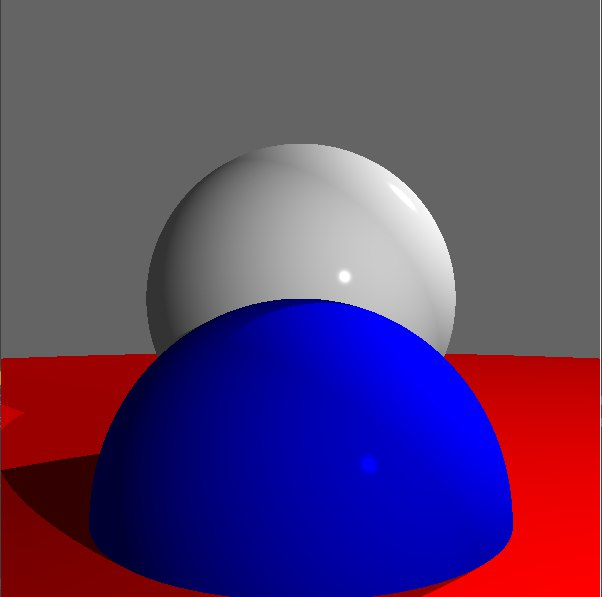
Функция нахождения отраженного луча – ReflectRay(R, N).

Функция решения квадратного уравнения – IntersectRaySphere(O, D, DD, sphere).

Функция подсчёта освещённости – ComputeLighing(P, N, V, s).

Нами были реализованы два класса Sphere и Light, соответственно с параметрами сферы и источника света. Мы создаем 3 сферы разного цвета и 3 источника света с разных координат и с разной интенсивностью. Также одну из сфер благодаря ее размерам мы используем как поверхность. Кроме того, у одной из сфер параметр отражения имеет значение 0.2, у остальных параметр 0.

**Результаты лабораторной работы**



**Вывод**

В данной работе был реализован алгоритм трассировки лучей при формировании сложной сцены.

**Приложение 1: исходный код программы**

from tqdm import tqdm

import sys

import numpy as np

from PyQt5.QtWidgets import QWidget, QApplication

from PyQt5.QtGui import QPainter, QColor, QPen

from PyQt5.QtCore import Qt

pixels = []

Cw = 0

Ch = 0

def set\_window(x, y):

global Cw

global Ch

Ch = y

Cw = x

class DrawQt(QWidget):

global Cw

global Ch

global pixels

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

self.setGeometry(50, 50, Cw, Ch)

self.setWindowTitle('Ray Tracing')

self.show()

def paintEvent(self, e):

qp = QPainter()

qp.begin(self)

self.drawPoints(qp)

qp.end()

def drawPoints(self, qp):

qp.setPen(Qt.red)

size = self.size()

for pixel in tqdm(pixels, desc='drawing'):

qp.setPen(QColor(pixel[2][0], pixel[2][1], pixel[2][2]))

qp.drawPoint(pixel[0], pixel[1])

def draw\_qt\_points(hash\_map):

global pixels

pixels = hash\_map.copy()

app = QApplication(sys.argv)

ex = DrawQt()

sys.exit(app.exec\_())

import numpy as np

class Sphere:

def \_\_init\_\_(self, \_center=[0, 0, 0], \_radius=1, \_color=[0, 0, 0],

\_specular=500, \_reflective=0, \_camera=[0, 0, 0]):

self.center = \_center

self.radius = \_radius

self.color = \_color

self.specular = \_specular

self.reflective = \_reflective

self.OC = []

self.OCOC = []

self.rr = 0

self.count\_oc(\_camera)

self.count\_r()

def get\_elements(self):

return {'center': self.center, 'radius': self.radius, 'color': self.color, 'specular': self.specular,

'reflective': self.reflective}

def get\_oc(self):

return self.OC

def get\_ococ(self):

return self.OCOC

def get\_rr(self):

return self.rr

def count\_r(self):

self.rr = self.radius\*self.radius

def count\_oc(self, \_camera):

self.OC = (\_camera - self.center).copy()

self.OCOC = np.dot(self.OC, self.OC)

Lights\_type = ['ambient', 'point', 'directional']

class Light:

global Lights\_type

def \_\_init\_\_(self, \_type, \_intensity, \_position=None, \_direction=None):

self.type = Lights\_type[\_type]

self.intensity = \_intensity

self.position = \_position

self.direction = \_direction

def get\_elements(self):

return {'type': self.type, 'intensity': self.intensity, 'position': self.position, 'direction': self.direction}

import numpy as np

from figures import Sphere

from figures import Light

import drawing

from tqdm import tqdm, trange

from multiprocessing import Pool

thread\_count = 0

Cw = 600

Ch = 600

Vw = 1

Vh = 1

d = 1

center\_window = np.array([0, 0, 0])

recursion\_depth = 2

spheres = []

lights = []

BACKGROUND\_COLOR = [100, 100, 100]

spheres.append(Sphere([0, -1, 3], 1, [0, 0, 255], \_specular=500, \_reflective=0.2, \_camera=center\_window.copy()))

spheres.append(Sphere([0, 0, 4], 1, [255, 255, 255], \_specular=500, \_reflective=0, \_camera=center\_window.copy()))

spheres.append(Sphere([0, -251, 0], 250, [255, 0, 0], \_specular=1000, \_reflective=0, \_camera=center\_window.copy()))

lights.append(Light(0, 0.2))

lights.append(Light(1, 0.6, \_position=[2, 1, 0]))

lights.append(Light(2, 0.2, \_direction=[1, 4, 4]))

def CanvasToViewport(x, y):

global Cw, Ch, Vw, Vh, d

return np.array([x\*Vw/Cw, y\*Vh/Ch, d])

def ClosestIntersection(O, D, t\_min, t\_max):

closest\_t = np.inf

closest\_sphere = None

global spheres

DD = np.dot(D, D)

for sphere in spheres:

t1, t2 = IntersectRaySphere(O, D, DD, sphere)

if t1 > t\_min and t1 < t\_max and t1 < closest\_t:

closest\_t = t1

closest\_sphere = sphere

if t2 > t\_min and t2 < t\_max and t2 < closest\_t:

closest\_t = t2

closest\_sphere = sphere

return closest\_sphere, closest\_t

def ClosestIntersection\_P(O, D, t\_min, t\_max):

closest\_t = np.inf

closest\_sphere = None

global spheres

DD = np.dot(D, D)

for sphere in spheres:

t1, t2 = IntersectRaySphere\_P(O, D, DD, sphere)

if t1 > t\_min and t1 < t\_max and t1 < closest\_t:

closest\_t = t1

closest\_sphere = sphere

if t2 > t\_min and t2 < t\_max and t2 < closest\_t:

closest\_t = t2

closest\_sphere = sphere

return closest\_sphere, closest\_t

def ReflectRay(R, N):

return 2\*N\*np.dot(N, R) - R

def TraceRay(O, D, t\_min, t\_max, depth):

global BACKGROUND\_COLOR

closest\_sphere, closest\_t = ClosestIntersection(O, D, t\_min, t\_max)

if closest\_sphere is None:

return BACKGROUND\_COLOR

P = np.array(O + closest\_t\*D)

N = P - closest\_sphere.get\_elements()['center']

N = N / np.linalg.norm(N)

local\_color = np.array(closest\_sphere.get\_elements()['color']) \* ComputeLighing(P, N, -D, closest\_sphere.get\_elements()['specular'])

reflective = closest\_sphere.get\_elements()['reflective']

if depth <= 0 or reflective <= 0:

return local\_color

R = ReflectRay(-D, N)

reflected\_color = TraceRay\_P(P, R, 0.001, np.inf, depth - 1)

return np.array(np.dot(local\_color, (1 - reflective)) + np.dot(reflected\_color, reflective))

def TraceRay\_P(O, D, t\_min, t\_max, depth):

global BACKGROUND\_COLOR

closest\_sphere, closest\_t = ClosestIntersection\_P(O, D, t\_min, t\_max)

if closest\_sphere is None:

return BACKGROUND\_COLOR

P = np.array(O + closest\_t\*D)

N = P - closest\_sphere.get\_elements()['center']

N = N / np.linalg.norm(N)

local\_color = np.array(closest\_sphere.get\_elements()['color']) \* ComputeLighing(P, N, -D, closest\_sphere.get\_elements()['specular'])

reflective = closest\_sphere.get\_elements()['reflective']

if depth <= 0 or reflective <= 0:

return local\_color

R = ReflectRay(-D, N)

reflected\_color = TraceRay\_P(P, R, 0.001, np.inf, depth - 1)

return np.array(np.dot(local\_color, (1 - reflective)) + np.dot(reflected\_color, reflective))

def IntersectRaySphere(O, D, DD, sphere):

global center\_window

rr = sphere.get\_rr()

OC = sphere.get\_oc()

k3 = sphere.get\_ococ() - rr

k1 = DD

k2 = 2\*np.dot(OC, D)

discriminant = k2\*k2 - 4\*k1\*k3

if discriminant < 0:

return np.inf, np.inf

t1 = (-k2 + np.sqrt(discriminant)) / (2\*k1)

t2 = (-k2 - np.sqrt(discriminant)) / (2\*k1)

return t1, t2

def IntersectRaySphere\_P(O, D, DD, sphere):

global center\_window

rr = sphere.get\_rr()

OC = O - sphere.get\_elements()['center']

k3 = np.dot(OC, OC) - rr

k1 = DD

k2 = 2\*np.dot(OC, D)

discriminant = k2\*k2 - 4\*k1\*k3

if discriminant < 0:

return np.inf, np.inf

t1 = (-k2 + np.sqrt(discriminant)) / (2\*k1)

t2 = (-k2 - np.sqrt(discriminant)) / (2\*k1)

return t1, t2

def ComputeLighing(P, N, V, s):

global lights

i = 0.0

for light in lights:

if light.get\_elements()['type'] == 'ambient':

i += light.get\_elements()['intensity']

else:

L = []

if light.get\_elements()['type'] == 'point':

L = (light.get\_elements()['position'] - P).copy()

t\_max = 1

else:

L = light.get\_elements()['direction'].copy()

t\_max = np.inf

shadow\_sphere, shadow\_t = ClosestIntersection\_P(P, L, 0.001, t\_max)

if shadow\_sphere != None:

continue

NL = np.dot(N, L)

if NL > 0:

i += light.get\_elements()['intensity']\*NL/(np.linalg.norm(N)\*np.linalg.norm(L))

if s != -1:

R = 2\*N\*np.dot(N, L) - L

RV = np.dot(R, V)

if RV > 0:

i += light.get\_elements()['intensity']\*np.power(RV/(np.linalg.norm(R)\*np.linalg.norm(V)), s)

return i

def processing(i, array):

global Ch

global center\_window

global recursion\_depth

hash\_mapp = []

text = 'progressbar #{position}'.format(position=i)

bar = tqdm(array, position=0, desc=text, leave=True)

for x in bar:

for y in range(-Ch//2, Ch//2, 1):

D = CanvasToViewport(x, y)

color = TraceRay(center\_window.copy(), D, 1, np.inf, recursion\_depth)

x1 = Cw / 2 + x

y1 = Ch / 2 - y - 1

if x1 < 0 or x1 > Cw or y1 < 0 or y1 > Ch:

return

else:

\_color = [min(255, color[0]), min(255, color[1]), min(255, color[2])]

hash\_mapp.append([x1, y1, \_color])

return hash\_mapp

def main():

global Cw

global Ch

drawing.set\_window(Cw, Ch)

global thread\_count

thread\_count = 6

mas = np.array(np.linspace(-Cw/2, Cw/2, Cw))

part = Cw // thread\_count

pool = Pool(processes=thread\_count)

output = [pool.apply\_async(processing, args=(i, mas[i\*part:(i + 1)\*part])) for i in range(thread\_count)]

result\_buf = [p.get() for p in output]

result = []

for row in result\_buf:

for j in row:

result.append(j)

drawing.draw\_qt\_points(result)

if \_\_name\_\_ == "\_\_main\_\_":

main()