Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Алгоритмы компьютерной графики

Отчет по лабораторной работе №5

Выполнил:
Козлов Василий Сергеевич
P3315

Проверил:
Меженин Александр Владимирович

Санкт-Петербург, 20 декабря 2025

Содержание

1 Задание

Цель работы — освоить методы вычисления и визуализации распределения яркости на сферических поверхностях с учётом:

- Ламбертовской модели освещения;
- цветных и нецветных точечных источников света;
- модели Блинна–Фонга (диффузная и зеркальная компоненты);
- самозатенения и взаимного затенения сфер.

Необходимо разработать приложение на Python, выполняющее трассировку лучей от камеры через экран с вычислением освещённости в точках пересечения лучей со сферами. Приложение должно визуализировать результат и сохранять изображение.

1.1 Исходные данные

Система координат, точечные источники света с Ламбертовской диаграммой излучения, прямоугольный экран, координаты наблюдателя, координаты центров сфер и их радиус, свойства поверхности сферы (согласно модели Блинн-Фонга).

Рекомендуемые пределы значений параметров для расчета:

- Размер прямоугольного экрана по высоте (H) и ширине (W) варьируются в диапазоне от 100 до 10000 миллиметров.
- Разрешение изображения по высоте (Hres) и ширине (Wres) варьируются в диапазоне от 200 до 800 пикселей. Разрешение должно обеспечивать квадратные пиксели.
- Координаты источников света (xLi, yLi, zLi) [мм] по осям X и Y X$\pm$10000, по оси Z от 100 до 10000.
- Цвет источников света: (RLi, GLi, BLi) в диапазоне от 0 до 1.
- Координаты наблюдателя: (0, 0, zO) [мм].
- Координаты центров сфер (xCi, yCi, zCi) [мм] по осям X и Y $\pm$10000, по оси Z от 100 до 10000. Сферы должны целиком помещаться в область видимости и располагаться так, чтобы одна из них отбрасывала тень на другую.
- Цвет поверхности сфер: (RCi, GCi, BCi) в диапазоне от 0 до 1.
- Сила излучения I0 варьируется от 0.01 до 10000 Вт/ср.
- Параметры модели Блинн-Фонга. Выбираются так, чтобы на изображении были видны области тени и полутени.

2 Описание алгоритма

Алгоритм реализует упрощённый рэйкастинг с проверкой пересечения лучей со сферами и вычислением освещения. Расчет распределения яркости на сфере в пределах заданной области экрана происходит следующим образом:

Для каждого пикселя экрана формируется луч от точки наблюдателя (камеры) через центр пикселя. Затем проверяется пересечение этого луча со всеми сферами в сцене. Если пересечение найдено, вычисляется ближайшая точка пересечения, нормаль в этой точке, и освещенность с учетом модели Блинна-Фонга, вклада всех источников света, а также затенения (самозатенения и взаимного затенения сфер). Если луч не пересекает ни одну сферу, пиксель окрашивается в фоновый цвет (черный).

Пользователь может изменять:

- параметры камеры;
- параметры освещения (цвет, интенсивность, координаты, радиус для площадных источников);
- параметры сфер (координаты, радиус, цвет);
- параметры модели Блинна–Фонга;
- разрешение изображения.

2.1 Формирование лучей

Пусть камера расположена в точке:

$$O = (x_O, y_O, z_O)$$

Каждому пикселю соответствует точка на экране:

$$P_s = (x_s, y_s, z_s)$$

Вектор направления луча:

$$d = P_s - O$$

Нормированное направление:

$$\hat{d} = \frac{d}{\|d\|}$$

Расчет точки на экране P_s для пикселя с индексами (i, j) (где i — строка, j — столбец) зависит от вида проекции. В программе используются ортогональные проекции для четырех видов: фронтальный, боковой (x+), сверху, боковой (x-).

2.2 Пересечение луча со сферой

Сфера:

$$\|P - C\|^2 = R^2$$

Луч:

$$P(t) = O + t\hat{d}, \quad t > 0$$

Подставляем:

$$\|O + t\hat{d} - C\|^2 = R^2$$

Получаем квадратное уравнение:

$$at^2 + bt + c = 0$$

где:

$$a = \hat{d} \cdot \hat{d} = 1,$$

$$b = 2\hat{d} \cdot (O - C),$$

$$c = \|O - C\|^2 - R^2$$

Дискриминант:

$$D = b^2 - 4ac$$

Если $D < 0$, пересечения нет. Иначе:

$$t_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$$

Берём минимальный положительный корень t. Точка пересечения:

$$P = O + t\hat{d}$$

Для нескольких сфер выбирается ближайшая точка пересечения (минимальное положительное t).

2.3 Нормаль и векторы освещения

Нормаль:

$$N = \frac{P - C}{\|P - C\|}$$

Вектор на камеру (направление обзора):

$$V = \frac{O - P}{\|O - P\|}$$

Вектор на источник света:

$$L_d = \frac{L - P}{\|L - P\|}$$

2.4 Модель Блинна–Фонга

Диффузное освещение:

$$I_d = k_d \cdot \max(0, N \cdot L_d)$$

Полувектор:

$$H = \frac{L_d + V}{\|L_d + V\|}$$

Зеркальная компонента:

$$I_s = k_s \cdot (\max(0, N \cdot H))^n$$

Затухание:

$$A = \frac{I_0}{r^2 + 1}$$

где $r = \|L - P\|$ — расстояние до источника.

Итоговый вклад источника света:

$$I = A(I_d \cdot C_{surf} \cdot C_{light} + I_s \cdot C_{light})$$

где C_{surf} — цвет поверхности, C_{light} — цвет источника.

Для нескольких источников суммируются вклады. Цвет вычисляется отдельно для R, G, B каналов.

2.5 Затенение

Пускается теневой луч от точки $P + \epsilon N$ в направлении L_d. Если теневой луч пересекает другую сферу до достижения источника, visibility = 0 (тень), иначе 1.

2.6 Нормировка изображения

После расчета всех пикселей, значения освещенности нормируются:

$$image = \left(\frac{image}{\max(image)} \cdot 255 \right)$$

и преобразуются в uint8 для отображения и сохранения в PNG.

3 Код программы

```
import tkinter as tk
from tkinter import ttk

class ScreenParamsComponent:
    def __init__(self, parent_frame):
        self.screen_params = {}
        self.create_inputs(parent_frame)

    def create_inputs(self, parent_frame):
        row = 0
        ttk.Label(parent_frame, text="Ширина экрана (мм):").grid(row=row, colu
        self.screen_params['w_mm'] = ttk.Entry(parent_frame, width=10)
        self.screen_params['w_mm'].grid(row=row, column=1)
        self.screen_params['w_mm'].insert(0, "2000")
        row += 1

        ttk.Label(parent_frame, text="Высота экрана (мм):").grid(row=row, colu
        self.screen_params['h_mm'] = ttk.Entry(parent_frame, width=10)
        self.screen_params['h_mm'].grid(row=row, column=1)
        self.screen_params['h_mm'].insert(0, "2000")
        row += 1

        ttk.Label(parent_frame, text="Разрешение по ширине:").grid(row=row, c
        self.screen_params['w_res'] = ttk.Entry(parent_frame, width=10)
        self.screen_params['w_res'].grid(row=row, column=1)
        self.screen_params['w_res'].insert(0, "600")
        row += 1

        ttk.Label(parent_frame, text="Разрешение по высоте:").grid(row=row, c
        self.screen_params['h_res'] = ttk.Entry(parent_frame, width=10)
        self.screen_params['h_res'].grid(row=row, column=1)
        self.screen_params['h_res'].insert(0, "600")
        row += 1

        ttk.Label(parent_frame, text="Расстояние наблюдателя (мм):").grid(row
        self.screen_params['zo'] = ttk.Entry(parent_frame, width=10)
        self.screen_params['zo'].grid(row=row, column=1)
        self.screen_params['zo'].insert(0, "1800")

    def get_params(self):
        return {key: float(entry.get()) for key, entry in self.screen_params.


import os
import numpy as np
from PIL import Image, ImageTk
from multiprocessing import Pool
```

```
class RendererComponent:
    def render(self, screen_p, spheres, lights):
        w_mm = screen_p['w_mm']
        h_mm = screen_p['h_mm']
        w_res = int(screen_p['w_res'])
        h_res = int(screen_p['h_res'])
        zo = screen_p['zo']

        if spheres:
            scene_center = np.mean([s['center'] for s in spheres], axis=0)
        else:
            scene_center = np.array([0, 0, 0])

        views = {
            'front': {'eye': scene_center + np.array([0, 0, zo]), 'sx': np.ar
            'side1': {'eye': scene_center + np.array([zo, 0, 0]), 'sx': np.ar
            'top': {'eye': scene_center + np.array([0, zo, 0]), 'sx': np.arra
            'side2': {'eye': scene_center + np.array([-zo, 0, 0]), 'sx': np.a
        }

        view_images = {}
        images = {}

        for view_name, view in views.items():
            image = self.render_view(w_mm, h_mm, w_res, h_res, view, spheres,
            pil_image = Image.fromarray(image)
            images[view_name] = pil_image
            view_images[view_name] = pil_image

        return images, view_images

    def render_view(self, w_mm, h_mm, w_res, h_res, view, spheres, lights):
        image = np.zeros((h_res, w_res, 3), dtype=np.float32)
        eye = view['eye']
        sx = view['sx']
        sy = view['sy']
        sc = view['sc']

        pixel_w = w_mm / w_res
        pixel_h = h_mm / h_res

        args = [(i, j, eye, sx, sy, sc, pixel_w, pixel_h, w_res, h_res, spher

        num_processes = os.cpu_count()
        chunksize = max(1, len(args) // num_processes)

        with Pool(processes=num_processes) as pool:
            results = pool.starmap(self.compute_pixel, args, chunksize=chunks
```

```
        idx = 0
        for i in range(h_res):
            for j in range(w_res):
                image[i, j] = results[idx]
                idx += 1

        image = self.normalize_image(image)
        return image

    def compute_pixel(self, i, j, eye, sx, sy, sc, pixel_w, pixel_h, w_res, h
        ray_dir = self.compute_ray_direction(i, j, w_res, h_res, pixel_w, pix

        if ray_dir is None:
            return np.zeros(3)

        hit_sphere_idx, min_t = self.find_closest_intersection(eye, ray_dir,

        if hit_sphere_idx is not None:
            hit_point = eye + min_t * ray_dir
            sphere = spheres[hit_sphere_idx]
            normal = (hit_point - sphere['center']) / sphere['radius']
            view_dir = -ray_dir
            return self.compute_lighting(hit_point, normal, view_dir, sphere,

        return np.zeros(3)

    def compute_ray_direction(self, i, j, w_res, h_res, pixel_w, pixel_h, sc,
        px = (j - w_res / 2 + 0.5) * pixel_w
        py = -(i - h_res / 2 + 0.5) * pixel_h
        pixel_pos = sc + px * sx + py * sy
        ray_dir = pixel_pos - eye
        ray_norm = np.linalg.norm(ray_dir)

        if ray_norm == 0:
            return None

        return ray_dir / ray_norm

    def find_closest_intersection(self, eye, ray_dir, spheres):
        min_t = np.inf
        hit_sphere_idx = None

        for idx, sphere in enumerate(spheres):
            t = self.ray_sphere_intersect(eye, ray_dir, sphere['center'], sph
            if 0 < t < min_t:
                min_t = t
                hit_sphere_idx = idx

        return hit_sphere_idx, min_t
```

```
def compute_lighting(self, hit_point, normal, view_dir, sphere, lights, s
    color = np.zeros(3)

    for light in lights:
        visibility, light_dir, dist_sample = self.compute_visibility_and_

        if visibility == 0:
            continue

        diffuse = sphere['kd'] * max(0, np.dot(normal, light_dir))
        h = light_dir + view_dir
        h_norm = np.linalg.norm(h)

        if h_norm > 0:
            h /= h_norm

        specular = sphere['ks'] * max(0, np.dot(normal, h)) ** sphere['sh
        atten = light['i0'] / (dist_sample ** 2 + 1)
        # atten = light['i0']
        light_contrib = light['color'] * atten * visibility
        color += sphere['color'] * light_contrib * diffuse + light_contri

    return color

def compute_visibility_and_direction(self, hit_point, normal, light, sphe
    return self.compute_point_light_visibility(hit_point, normal, light,

def compute_point_light_visibility(self, hit_point, normal, light, sphere
    light_dir = light['pos'] - hit_point
    dist = np.linalg.norm(light_dir)

    if dist == 0:
        return 0, None, 0

    light_dir /= dist
    epsilon = 0.001
    shadow_origin = hit_point + normal * epsilon

    shadowed = any(
        0 < self.ray_sphere_intersect(shadow_origin, light_dir, spheres[o
        for other_idx in range(len(spheres)) if other_idx != hit_sphere_i
    )

    visibility = 0 if shadowed else 1

    return visibility, light_dir, dist

def normalize_image(self, image):
    max_val = np.max(image)
```

```
        if max_val > 0:
            image = (image / max_val * 255).astype(np.uint8)
        else:
            image = image.astype(np.uint8)

        return image

    def ray_sphere_intersect(self, origin, dir, center, radius):
        oc = origin - center
        a = np.dot(dir, dir)
        b = 2 * np.dot(oc, dir)
        c = np.dot(oc, oc) - radius ** 2
        disc = b ** 2 - 4 * a * c

        if disc < 0:
            return np.inf

        sqrt_disc = np.sqrt(disc)
        t1 = (-b - sqrt_disc) / (2 * a)
        t2 = (-b + sqrt_disc) / (2 * a)

        if t1 > 0 and t2 > 0:
            return min(t1, t2)
        elif max(t1, t2) > 0:
            return max(t1, t2)
        else:
            return np.inf

import numpy as np
import tkinter as tk
from tkinter import ttk
from tkinter.colorchooser import askcolor

class ObjectManagerComponent:
    def __init__(self, parent_frame, regrid_callback):
        self.spheres = []
        self.lights = []
        self.parent_frame = parent_frame
        self.regrid_callback = regrid_callback

    def add_sphere(self):
        sphere_id = len(self.spheres) + 1
        sphere_params = {}
        frame = ttk.LabelFrame(self.parent_frame, text=f"Сфера {sphere_id}")

        row = 0
        ttk.Label(frame, text="Центр (x,y,z мм):").grid(row=row, column=0)
        sphere_params['x'] = ttk.Entry(frame, width=5)
        sphere_params['x'].grid(row=row, column=1)
        sphere_params['x'].insert(0, "-200" if sphere_id == 1 else "200" if s
```

```
        sphere_params['y'] = ttk.Entry(frame, width=5)
        sphere_params['y'].grid(row=row, column=2)
        sphere_params['y'].insert(0, "-500" if sphere_id == 1 else "500" if s
        sphere_params['z'] = ttk.Entry(frame, width=5)
        sphere_params['z'].grid(row=row, column=3)
        sphere_params['z'].insert(0, "-200" if sphere_id == 1 else "200" if s
        row += 1

        ttk.Label(frame, text="Радиус (мм):").grid(row=row, column=0)
        sphere_params['r'] = ttk.Entry(frame, width=10)
        sphere_params['r'].grid(row=row, column=1)
        sphere_params['r'].insert(0, "400" if sphere_id == 1 else "250" if sp
        row += 1

        ttk.Label(frame, text="Цвет (R,G,B 0-1):").grid(row=row, column=0)
        sphere_params['cr'] = ttk.Entry(frame, width=5)
        sphere_params['cr'].grid(row=row, column=1)
        sphere_params['cr'].insert(0, "1" if sphere_id == 1 else "0" if spher
        sphere_params['cg'] = ttk.Entry(frame, width=5)
        sphere_params['cg'].grid(row=row, column=2)
        sphere_params['cg'].insert(0, "0" if sphere_id == 1 else "1" if spher
        sphere_params['cb'] = ttk.Entry(frame, width=5)
        sphere_params['cb'].grid(row=row, column=3)
        sphere_params['cb'].insert(0, "0" if sphere_id == 1 else "0" if spher
        row += 1

        ttk.Label(frame, text="kd, ks, shininess:").grid(row=row, column=0)
        sphere_params['kd'] = ttk.Entry(frame, width=5)
        sphere_params['kd'].grid(row=row, column=1)
        sphere_params['kd'].insert(0, "0.8")
        sphere_params['ks'] = ttk.Entry(frame, width=5)
        sphere_params['ks'].grid(row=row, column=2)
        sphere_params['ks'].insert(0, "0.5")
        sphere_params['shin'] = ttk.Entry(frame, width=5)
        sphere_params['shin'].grid(row=row, column=3)
        sphere_params['shin'].insert(0, "32")
        row += 1

        ttk.Button(frame, text="Удалить", command=lambda: self.delete_sphere(
        ttk.Button(frame, text="Выбрать", command=lambda: self.pick_color(sph

        self.spheres.append(sphere_params)
        self.regrid_callback()

        return sphere_params

    def delete_sphere(self, params):
        frame = params['x'].master
        index = self.spheres.index(params)
        frame.grid_forget()
```

```
        frame.destroy()
        del self.spheres[index]
        self.regrid_callback()

    def add_light(self):
        light_id = len(self.lights) + 1
        light_params = {}
        frame = ttk.LabelFrame(self.parent_frame, text=f"Источник {light_id}"

        row = 0
        ttk.Label(frame, text="Положение (x,y,z мм):").grid(row=row, column=0
        light_params['x'] = ttk.Entry(frame, width=5)
        light_params['x'].grid(row=row, column=1)
        light_params['x'].insert(0, "2000" if light_id == 1 else "-2000" if l
        light_params['y'] = ttk.Entry(frame, width=5)
        light_params['y'].grid(row=row, column=2)
        light_params['y'].insert(0, "2000" if light_id == 1 else "2000" if li
        light_params['z'] = ttk.Entry(frame, width=5)
        light_params['z'].grid(row=row, column=3)
        light_params['z'].insert(0, "2000" if light_id == 1 else "0" if light
        row += 1

        ttk.Label(frame, text="Цвет (R,G,B 0-1):").grid(row=row, column=0)
        light_params['cr'] = ttk.Entry(frame, width=5)
        light_params['cr'].grid(row=row, column=1)
        light_params['cr'].insert(0, "1")
        light_params['cg'] = ttk.Entry(frame, width=5)
        light_params['cg'].grid(row=row, column=2)
        light_params['cg'].insert(0, "1")
        light_params['cb'] = ttk.Entry(frame, width=5)
        light_params['cb'].grid(row=row, column=3)
        light_params['cb'].insert(0, "1")
        row += 1

        ttk.Label(frame, text="I0 (Вт/ср):").grid(row=row, column=0)
        light_params['i0'] = ttk.Entry(frame, width=10)
        light_params['i0'].grid(row=row, column=1)
        light_params['i0'].insert(0, "10000")
        row += 1

        ttk.Button(frame, text="Удалить", command=lambda: self.delete_light(l
        ttk.Button(frame, text="Выбрать", command=lambda: self.pick_color(lig

        self.lights.append(light_params)
        self.regrid_callback()

        return light_params

    def delete_light(self, params):
        frame = params['x'].master
```

```
        index = self.lights.index(params)
        frame.grid_forget()
        frame.destroy()
        del self.lights[index]
        self.regrid_callback()

    def pick_color(self, params):
        color = askcolor(title="Выберите цвет")
        if color[1] is not None:
            r, g, b = color[0]
            params['cr'].delete(0, tk.END)
            params['cr'].insert(0, str(r / 255.0))
            params['cg'].delete(0, tk.END)
            params['cg'].insert(0, str(g / 255.0))
            params['cb'].delete(0, tk.END)
            params['cb'].insert(0, str(b / 255.0))

    def get_spheres(self):
        spheres_list = []
        for sphere in self.spheres:
            s = {
                'center': np.array([float(sphere['x'].get()), float(sphere['y
                'radius': float(sphere['r'].get()),
                'color': np.array([float(sphere['cr'].get()), float(sphere['c
                'kd': float(sphere['kd'].get()),
                'ks': float(sphere['ks'].get()),
                'shin': float(sphere['shin'].get())
            }
            spheres_list.append(s)
        return spheres_list

    def get_lights(self):
        lights_list = []
        for light in self.lights:
            l = {
                'pos': np.array([float(light['x'].get()), float(light['y'].ge
                'color': np.array([float(light['cr'].get()), float(light['cg'
                'i0': float(light['i0'].get())
            }
            lights_list.append(l)
        return lights_list


import tkinter as tk
from tkinter import ttk
from PIL import Image, ImageTk

class ImageDisplayComponent:
    def __init__(self, parent_frame, resize_callback):
        self.labels = self.create_labels(parent_frame)
```

```
        self.images = {}
        self.view_images = {}
        self.prev_width = 0
        self.prev_height = 0
        self.resize_callback = resize_callback

    def create_labels(self, parent_frame):
        ttk.Label(parent_frame, text="Вид спереди:").grid(row=0, column=0, st
        label_front = ttk.Label(parent_frame)
        label_front.grid(row=1, column=0, sticky='nsew')

        ttk.Label(parent_frame, text="Вид сбоку (x+):").grid(row=0, column=1,
        label_side1 = ttk.Label(parent_frame)
        label_side1.grid(row=1, column=1, sticky='nsew')

        ttk.Label(parent_frame, text="Вид сверху:").grid(row=2, column=0, sti
        label_top = ttk.Label(parent_frame)
        label_top.grid(row=3, column=0, sticky='nsew')

        ttk.Label(parent_frame, text="Вид сбоку (x-):").grid(row=2, column=1,
        label_side2 = ttk.Label(parent_frame)
        label_side2.grid(row=3, column=1, sticky='nsew')

        return {
            'front': label_front,
            'side1': label_side1,
            'top': label_top,
            'side2': label_side2
        }

    def update_images(self, images, view_images):
        self.images = images
        self.view_images = view_images
        self.prev_width = 0
        self.prev_height = 0
        self.resize_images()

    def resize_images(self):
        if not self.view_images:
            return

        front_width = self.labels['front'].winfo_width()
        front_height = self.labels['front'].winfo_height()
        if front_width <= 1 or front_height <= 1:
            return

        if front_width == self.prev_width and front_height == self.prev_heigh
            return

        self.prev_width = front_width
```

```
        self.prev_height = front_height

        side_length = min(front_width, front_height)

        for view_name, pil_image in self.view_images.items():
            resized = pil_image.resize((side_length, side_length), Image.LANC
            photo = ImageTk.PhotoImage(resized)
            self.labels[view_name].config(image=photo)
            self.labels[view_name].image = photo

    def save_images(self):
        for view_name, img in self.images.items():
            img.save(f"{view_name}.png")

import tkinter as tk
from tkinter import ttk
from screen_params import ScreenParamsComponent
from object_manager import ObjectManagerComponent
from renderer import RendererComponent
from image_display import ImageDisplayComponent

class App:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Лабораторная работа 5: Визуализация распределения яр

        self.input_frame = ttk.Frame(self.root)
        self.input_frame.grid(row=0, column=0, rowspan=30, sticky='ns')

        self.screen_params_comp = ScreenParamsComponent(self.input_frame)

        ttk.Button(self.input_frame, text="Добавить сферу", command=self.add_
        ttk.Button(self.input_frame, text="Добавить источник", command=self.a

        ttk.Button(self.input_frame, text="Рендерить", command=self.render).g
        ttk.Button(self.input_frame, text="Сохранить изображения", command=se

        self.canvas = tk.Canvas(self.input_frame)
        self.canvas.grid(row=14, column=0, columnspan=4, sticky='nsew')
        self.scrollbar = ttk.Scrollbar(self.input_frame, orient='vertical', c
        self.scrollbar.grid(row=14, column=4, sticky='ns')
        self.canvas.configure(yscrollcommand=self.scrollbar.set)
        self.input_frame.rowconfigure(14, weight=1)
        self.input_frame.columnconfigure(0, weight=1)

        self.inner_frame = ttk.Frame(self.canvas)
        self.canvas.create_window((0, 0), window=self.inner_frame, anchor='nw
        self.inner_frame.bind("<Configure>", lambda e: self.canvas.configure(

        self.object_manager = ObjectManagerComponent(self.inner_frame, self.r
```

```
        self.image_frame = ttk.Frame(self.root)
        self.image_frame.grid(row=0, column=1, sticky='nsew')

        self.image_display = ImageDisplayComponent(self.image_frame, self.res

        self.root.columnconfigure(1, weight=1)
        self.root.rowconfigure(0, weight=1)

        self.image_frame.columnconfigure(0, weight=1)
        self.image_frame.columnconfigure(1, weight=1)
        self.image_frame.rowconfigure(1, weight=1)
        self.image_frame.rowconfigure(3, weight=1)

        self.add_sphere()
        self.add_sphere()
        self.add_light()
        self.add_light()

        self.root.bind("<Configure>", self.on_resize)

        self.root.geometry("1200x800")

        self.renderer = RendererComponent()

        self.root.mainloop()

    def add_sphere(self):
        self.object_manager.add_sphere()

    def add_light(self):
        self.object_manager.add_light()

    def regrid_all(self):
        row = 0
        for params in self.object_manager.spheres:
            frame = params['x'].master
            frame.grid(row=row, column=0, columnspan=4, sticky='ew')
            row += 1
        for params in self.object_manager.lights:
            frame = params['x'].master
            frame.grid(row=row, column=0, columnspan=4, sticky='ew')
            row += 1
        self.canvas.configure(scrollregion=self.canvas.bbox("all"))

    def get_params(self):
        try:
            screen_p = self.screen_params_comp.get_params()
            spheres = self.object_manager.get_spheres()
            lights = self.object_manager.get_lights()
```

```
            return screen_p, spheres, lights
        except ValueError as e:
            tk.messagebox.showerror("Ошибка", f"Некорректное значение: {e}")
            raise

    def render(self):
        screen_p, spheres, lights = self.get_params()
        images, view_images = self.renderer.render(screen_p, spheres, lights)
        self.image_display.update_images(images, view_images)

    def resize_images(self):
        self.image_display.resize_images()

    def on_resize(self, event):
        self.resize_images()

    def save_images(self):
        self.image_display.save_images()


if __name__ == "__main__":
    App()
```

4 Результаты работы

В ходе выполнения лабораторной работы было разработано приложение, позволяющее интерактивно визуализировать несколько сфер с цветными источниками света, корректной диффузной и зеркальной моделью освещения и затенением.

Ниже приведены примеры изображений сцены:

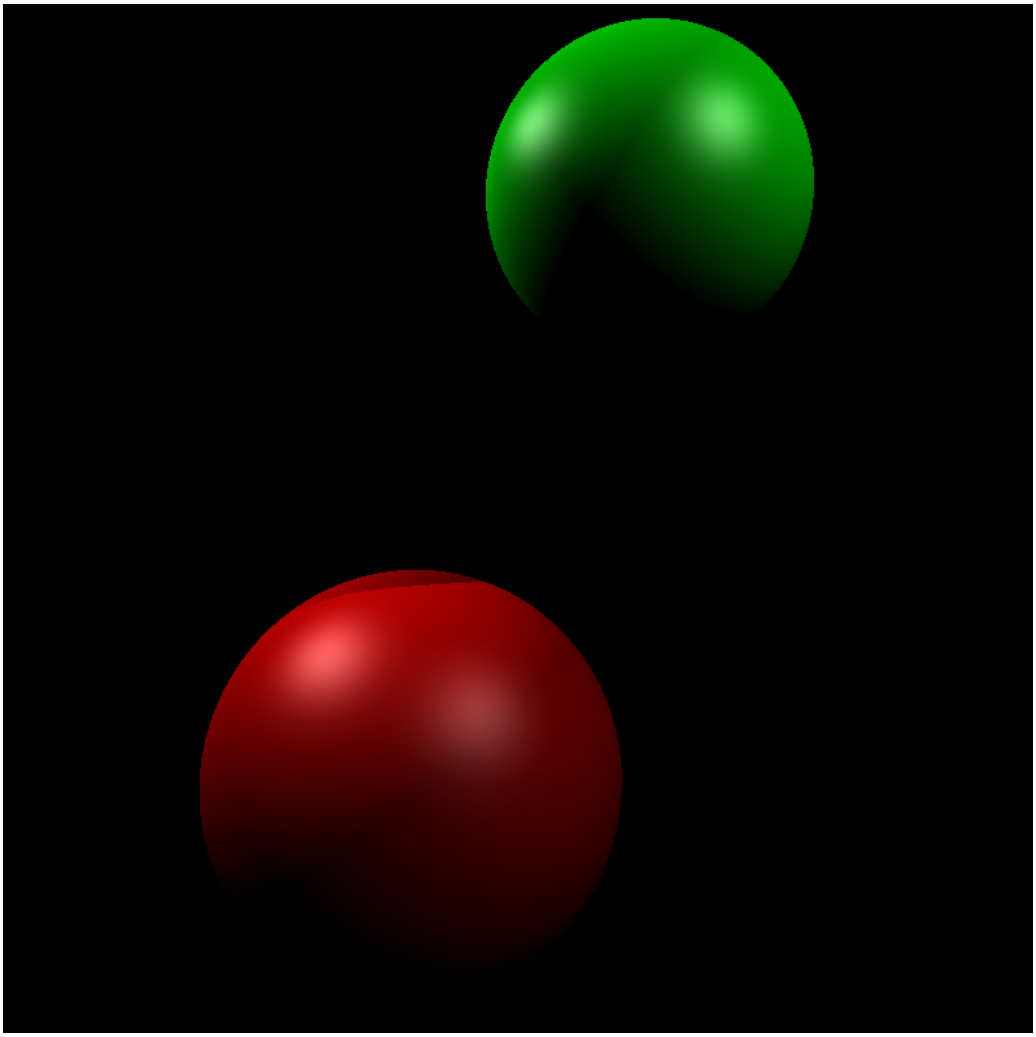

Рис. 1: Вид спереди

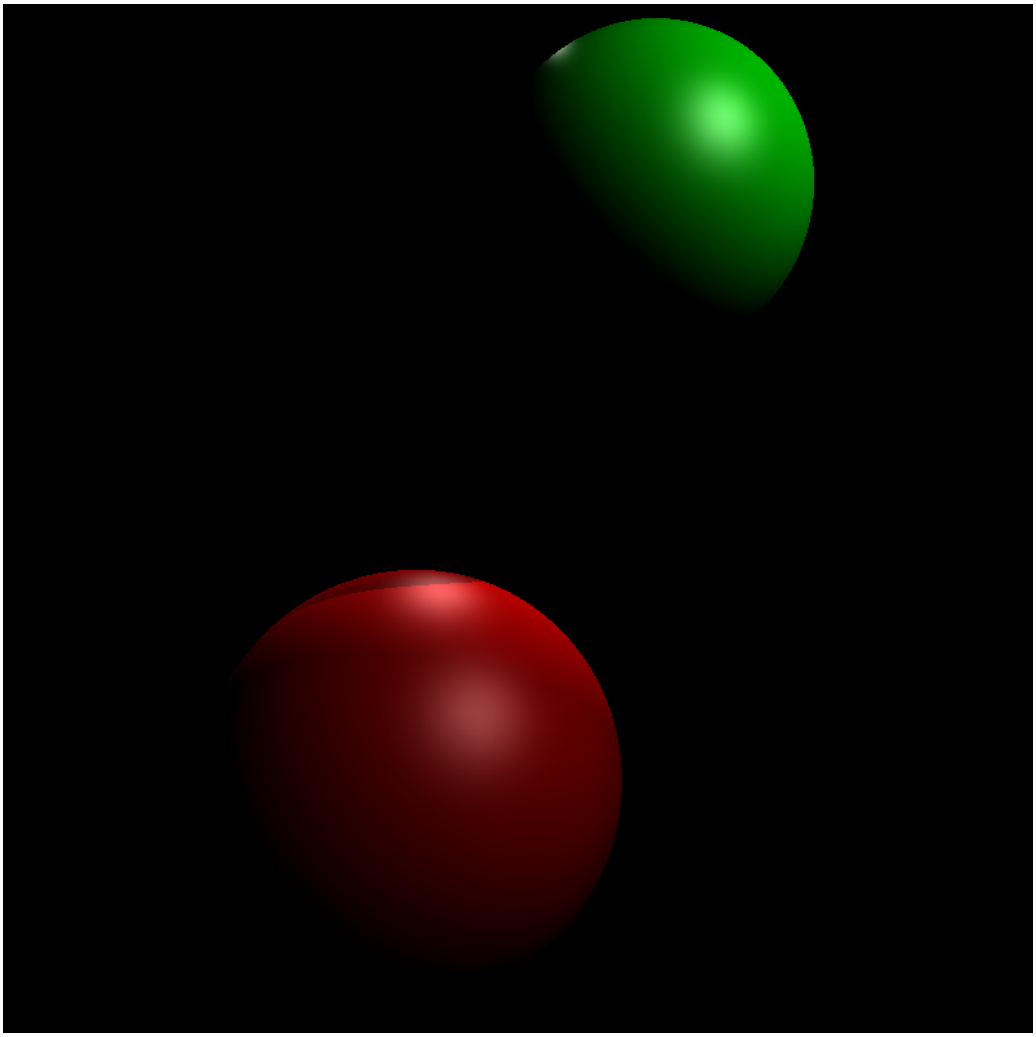

Рис. 2: Вид сбоку (x+)

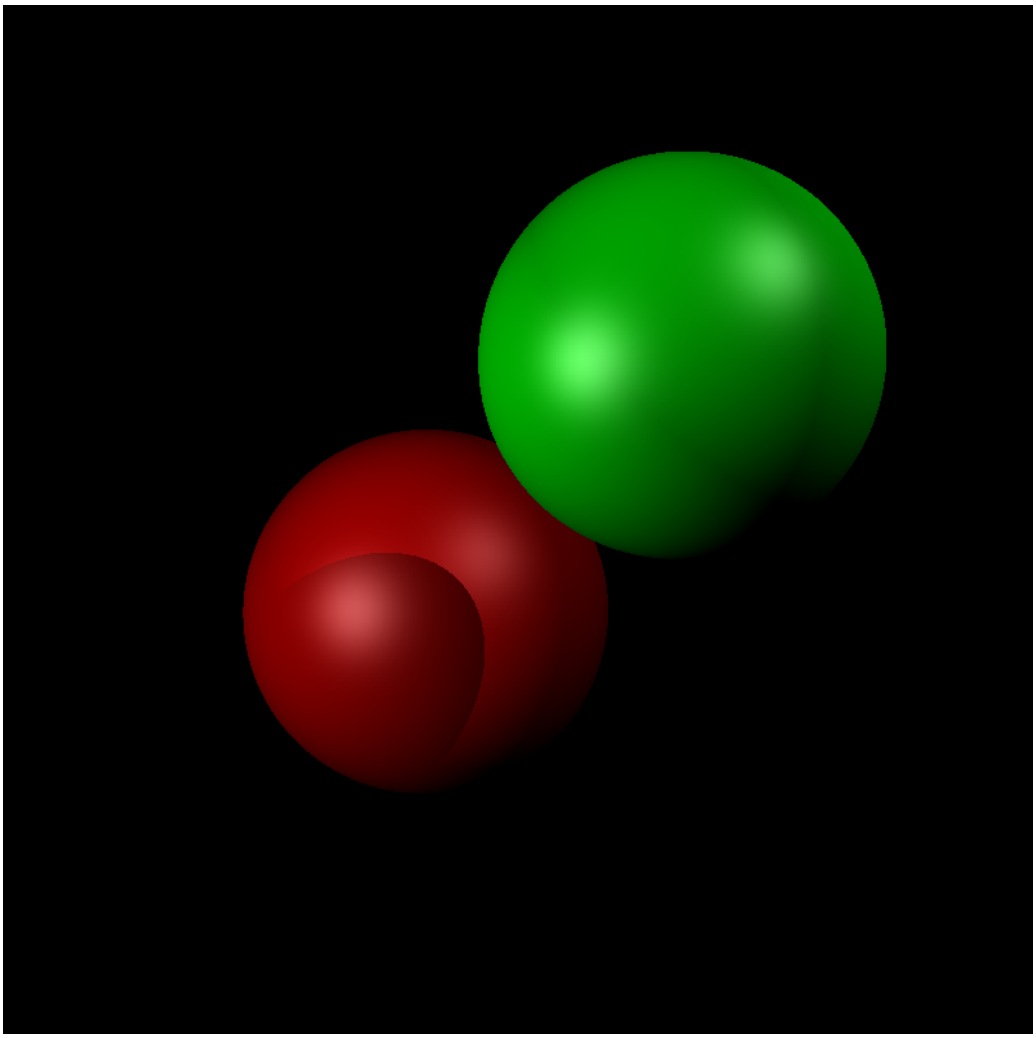

Рис. 3: Вид сверху

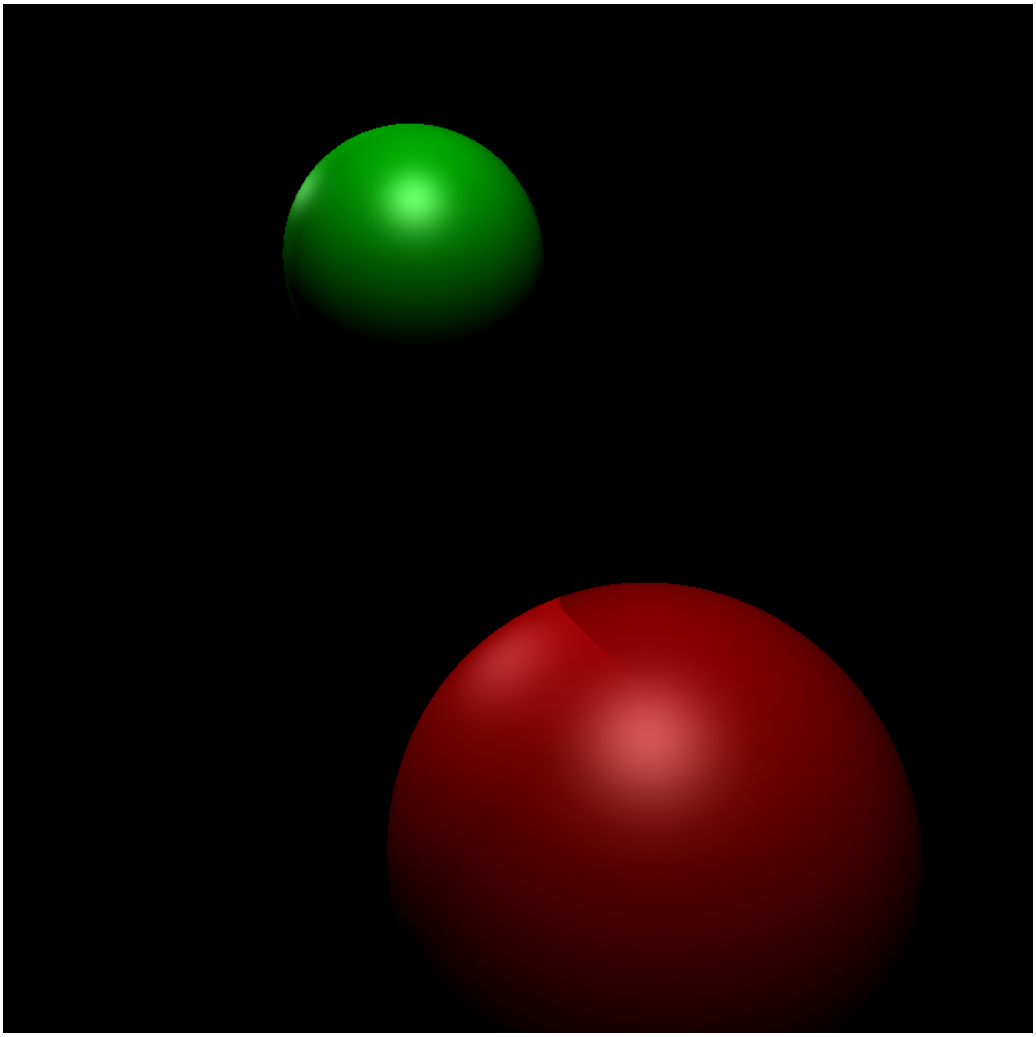

Рис. 4: Вид сбоку (x-)

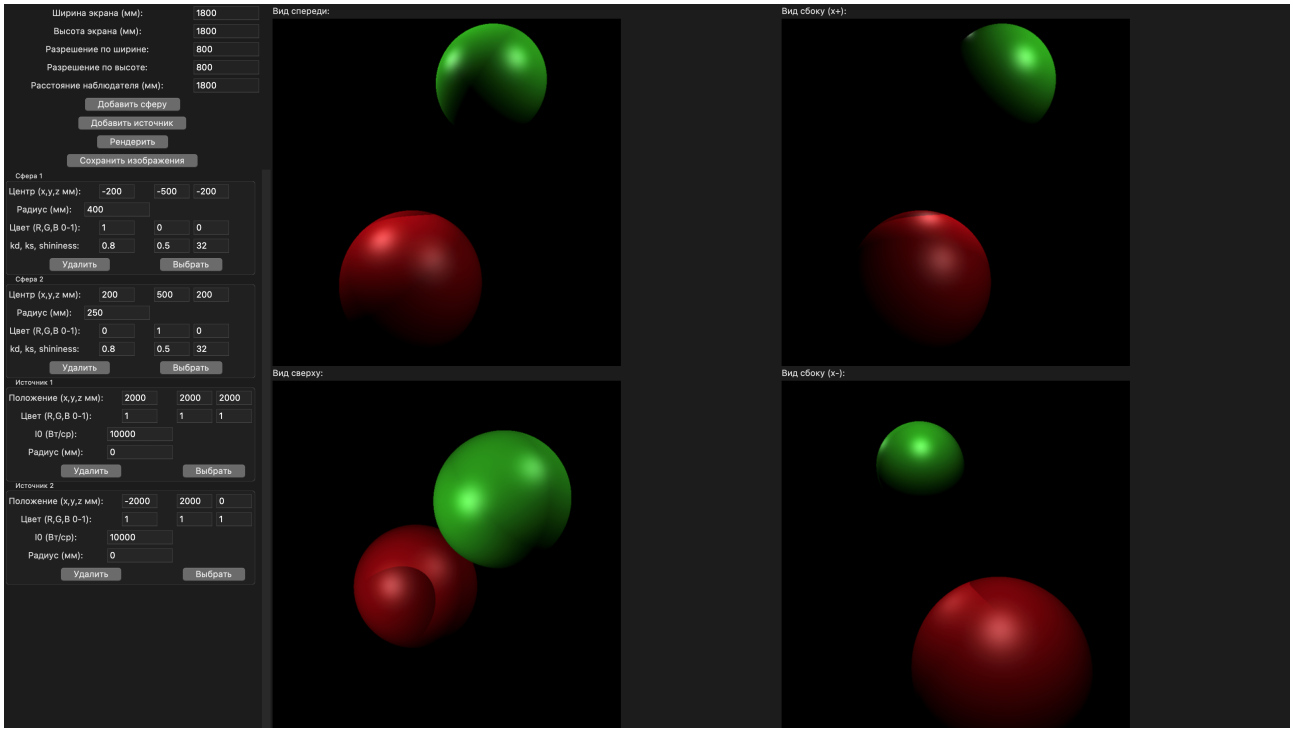

Рис. 5: Пример работы прогрммы

5 Выводы

В результате выполнения лабораторной работы было:

- изучено вычисление освещения по модели Блинна-Фонга;
- реализовано затенение объектов на основе трассировки теневых лучей;
- создано приложение для визуализации сфер с цветными источниками света.

Работа продемонстрировала принципы физически корректной визуализации и методологию построения рендерера на основе рэйкастинга.