

Inżynieria obrazów

Metoda śledzenia promieni

Marcin Lasak 272886

Spis treści

1. Wstęp teoretyczny	3
1.1 Oświetlenie globalne	3
1.2 Oświetlenie lokalne	3
1.3 Modele empiryczne i analityczne	3
1.4 Metoda śledzenia promieni	3
1.5 Model oświetlenia Phong'a	4
2. Zadania	6
2.1 Kule	6
2.2 Promienie wtórne	7
2.3 Cienie	9
2.4 Przezroczystość	11
2.5 Trójkąt	14
3. Ogólna struktura projektu	17
Źródła	18

1. Wstęp teoretyczny

1.1 Oświetlenie globalne

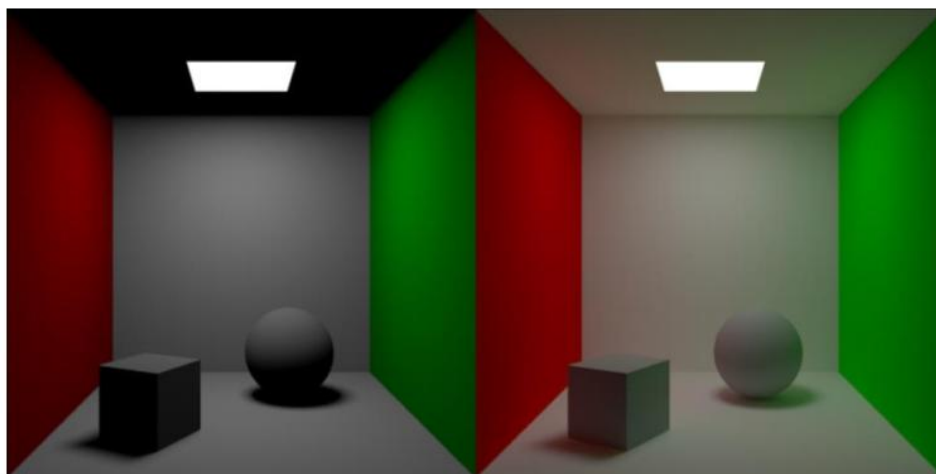
Jest to model oświetlenia w grafice komputerowej, który uwzględnia właściwe źródło światła oraz interakcje między oświetlanymi obiektami. Zawiera propagację światła przez całą scenę oraz to jak obiekty oddziałują między sobą i otoczeniem. W praktyce oznacza to śledzenie światła poprzez scenę i uwzględnianie wszelkich zjawisk optycznych z nim związanych. Realnie nie jest całkowicie możliwe do odwzorowania i jest jednym z nadal badanych zagadnień grafiki komputerowej.

Podstawowe metody realizacji oświetlenia globalnego to metoda energetyczna i metoda śledzenia promieni.

1.2 Oświetlenie lokalne

Jest to model oświetlenia uwzględniający jedynie źródło światła. Każdy obiekt jest rozpatrywany niezależnie.

Jednym z podstawowych modeli oświetlenia lokalnego jest model Phong'a.



Rysunek 1 Po lewej oświetlenie lokalne, po prawej – oświetlenie globalne

1.3 Modele empiryczne i analityczne

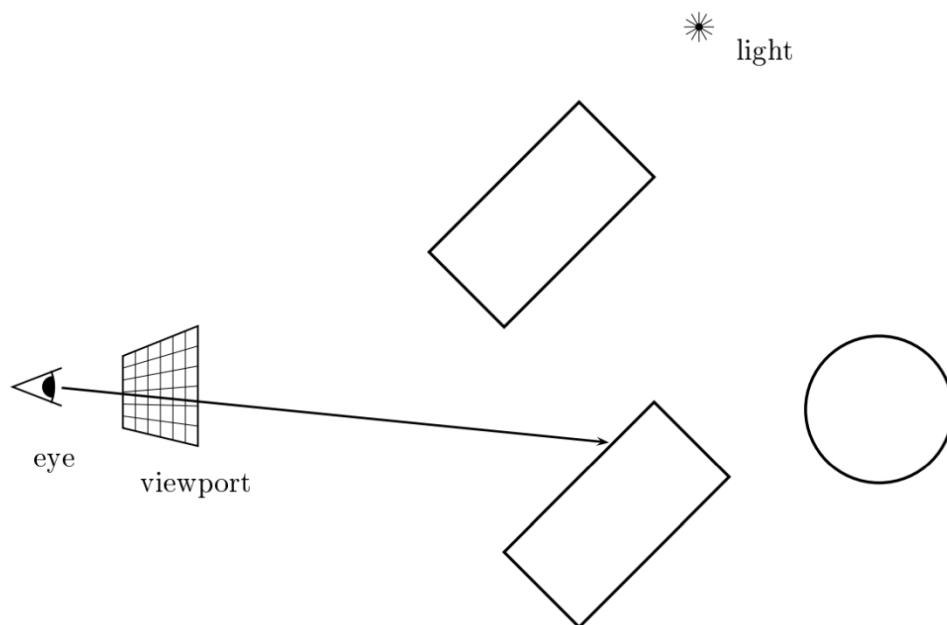
Modele empiryczne oświetlenia bazują na przybliżeniach obserwacji zjawisk optycznych. Uwzględniają jedynie pierwotne źródło światła. Cechuje je duża szybkość.

Modele analityczne natomiast korzystają ze znanych nam praw fizyki, by zobrazować realnie występujące zjawiska optyczne. Przypisują dużą wagę do zasady zachowania energii. Są dużo bardziej skomplikowane obliczeniowo od modeli empirycznych.

Modele hybrydowe łączą cechy obu tych grup modeli.

1.4 Metoda śledzenia promieni

Jest to technika generowania fotorealistycznych obrazów scen trójwymiarowych polegająca na śledzeniu promieni światła padających na obserwatora. Nazywana czasem też metodą odwrotnego śledzenia promieni, ponieważ śledzone są tylko te promienie, które trafiają do obserwatora, więc droga promienia zaczyna się w oku obserwatora, a kończy we właściwym źródle światła. Uwzględnia ona całą gamę realnych zjawisk optycznych jak odbicie czy załamanie. W miejscu zderzenia się promienia z obiektem powstają promienie wtórne, które biegną dalej zgodnie z zachowaniem praw odbicia i załamania w zależności od właściwości



Rysunek 2 Schemat działania metody (odwrotnego) śledzenia promieni

obiekty. Proces generowania promieni wtórnych trwa, dopóki wszystkie promienie nie wyjdą poza scenę lub algorytm nie osiągnie odpowiedniego poziomu rekurencji. Metoda ta, polegająca na tworzeniu i śledzeniu promieni wtórnych, jest nazywana rekurencyjną metodą śledzenia promieni. Dzięki temu uzyskujemy efekty wzajemnych, wielokrotnych odbić obiektów. Im więcej odbić promieni rozpatrujemy, tym obraz jest wierniejszy, jednakże jest też bardziej kosztowny obliczeniowo.

1.5 Model oświetlenia Phong

Jest to najczęściej używany model oświetlenia w grafice komputerowej. W tym modelu światło rozbite jest na trzy komponenty:

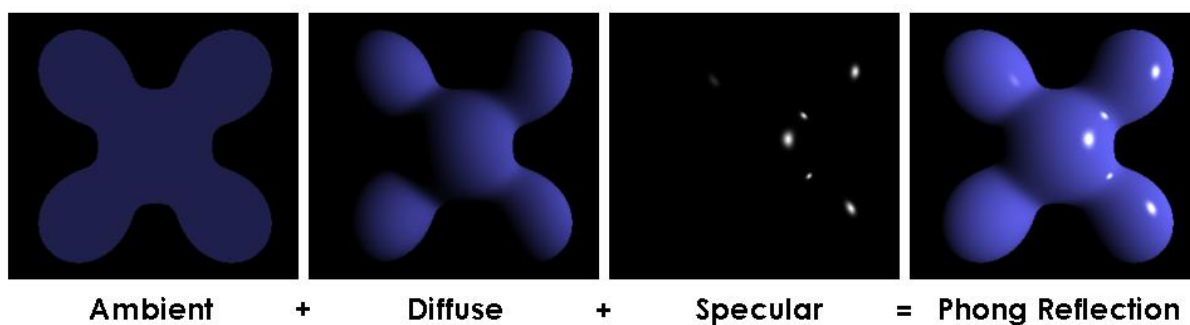
- Ambient – symulacja światła globalnego,
- Diffuse – oświetlenie bezpośrednie,
- Specular – odbicie światła.

Komponenty te określamy dla każdego ze źródeł światła oraz dla materiału, którego powierzchni obliczamy kolor. Kolor powierzchni to suma komponentów dla każdego ze źródeł.

$$I = \sum_{i=1}^N I_a + I_d + I_s$$

Rysunek 3

Należy zauważyć, że składowa ambient jest bardzo dużym przybliżeniem realnego światła globalnego. Składowa diffuse uwzględnia dodatkowo kąt padania światła. Składowa specular symuluje odbicie światła.



Rysunek 4 Komponenty oświetlenia w modelu Phong

2. Zadania

2.1 Kule

W pierwszym zadaniu należało zmodyfikować podany kod w języku Python, aby wyświetlał dodatkową kulę.

Przykładowy program zawiera klasy:

- Promienia,
- Światła,
- Obiektu,
- Sfery (dziedziczy po obiekcie),
- Kamery,
- Sceny,
- Śledzenia promieni.

Klasy te i ich funkcje umożliwiają wyrenderowanie trójwymiarowej sceny zawierającą kulę, z wykorzystaniem modelu Phong'a oraz metody śledzenia promieni.

Zadanie polegało na dodaniu dodatkowej kuli do już istniejącej sceny. W tym celu do pliku lab5.py wklejono kod ze strony datko.pl. Kod zmodyfikowano, aby zamiast biblioteki matplotlib używać Pillow, ze względu na niekompatybilność matplotlib z narzędziami tworzącymi pliki wykonywalne na podstawie skryptów w języku Python. Do obliczeń wykorzystano bibliotekę numpy.

Zmodyfikowana scena jest tworzona i wyświetlana za pomocą funkcji zad1() (rysunek 5).



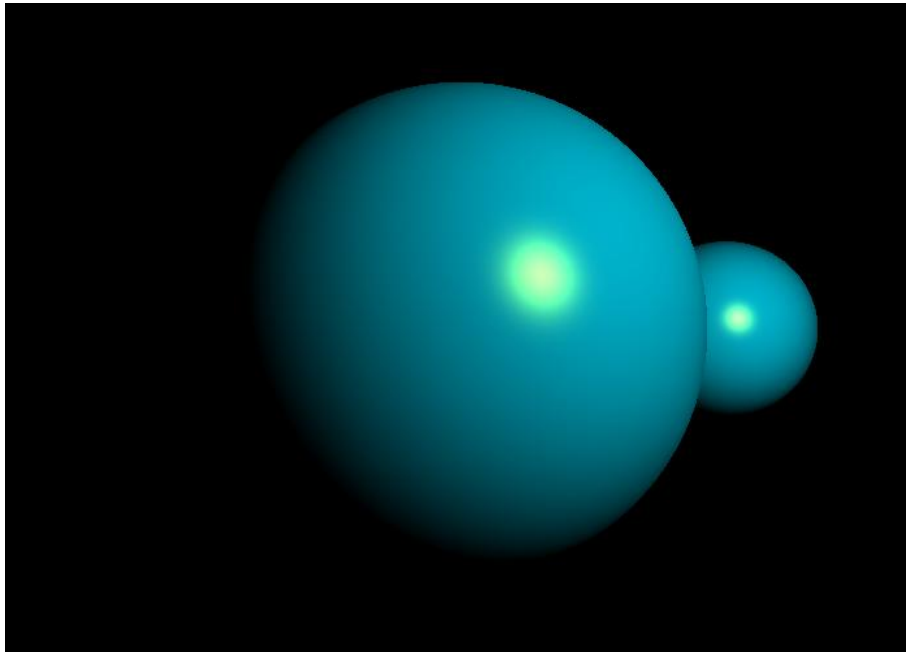
```
def zad1():
    scene = Scene(
        objects=[
            Sphere(position=np.array([0, 0, 0]), radius=1.5),
            Sphere(position=np.array([3, 0, -4]), radius=1)
        ],
        light=Light(position=np.array([3, 2, 5])),
        camera=Camera(position=np.array([0, 0, 5]))
    )

    rt = RayTracer(scene)
    image = np.clip(rt.generate_image(), 0, 1)
    image_uint8 = (image * 255).astype(np.uint8)
    img = Image.fromarray(image_uint8)
    img.save("render1.png")
    img.show()
```

Rysunek 5

Dodano mniejszą kulę po prawej stronie, bardziej z tyłu sceny. Wynik działania tej funkcji dany jest rysunkiem nr 6. Jest on zapisany w pliku render1.png.

Analogiczny efekt osiągniemy, po wybraniu opcji nr 1 w programie main.exe.



Rysunek 6

2.2 Promienie wtórne

Kolejne zadanie polegało na modyfikacji śledzenia promieni, aby uwzględnić promienie wtórne. W tym celu zaimplementowano klasę `MyRayTracer` dziedziczącą po `RayTracer`. Kluczowe zmiany wprowadzono w funkcji `_get_pixel_color()` (rysunek 7).

```
lab5.py
class MyRayTracer(RayTracer):
    def _get_pixel_color(self, ray, depth=3):
        obj, distance, cross_point = self._get_closest_object(ray)
        if not obj:
            return self.scene.background
        if depth == 0:
            return obj.get_color(cross_point, ray.direction, self.scene)
        new_ray = Ray(cross_point, reflect(ray.direction, normalize(obj.get_normal(cross_point))))
        return (0.75 * obj.get_color(cross_point, ray.direction, self.scene) + 0.25 *
                self._get_pixel_color(new_ray, depth=depth - 1))
```

Rysunek 7

Dodano dodatkowy parametr `depth`, określający maksymalną ilość odbić. Wprowadzono zmienną `new_ray`, która odpowiada nowemu promieniowi. Zaczyna się on w punkcie przecięcia poprzedniego promienia z obiektem (`cross_point`), a następnie leci w kierunku odbicia od powierzchni obiektu. Funkcja `obj.get_normal(cross_point)` zwraca wektor normalny do powierzchni obiektu w miejscu trafienia, `normalize()` przekształca normalny wektor do jednostkowej długości (bo odbicie wymaga znormalizowanych wektorów), a `reflect()` oblicza kierunek odbitego promienia na podstawie wektora padającego (`ray.direction`) oraz wektora normalnego (funkcja z programu wzorcowego). Funkcja zwraca kolor ważony, uwzględniając kolejne rekurencyjne wywołanie `_get_pixel_color()`. 75% wpływu ma kolor lokalny, a 25% to światło odbite.

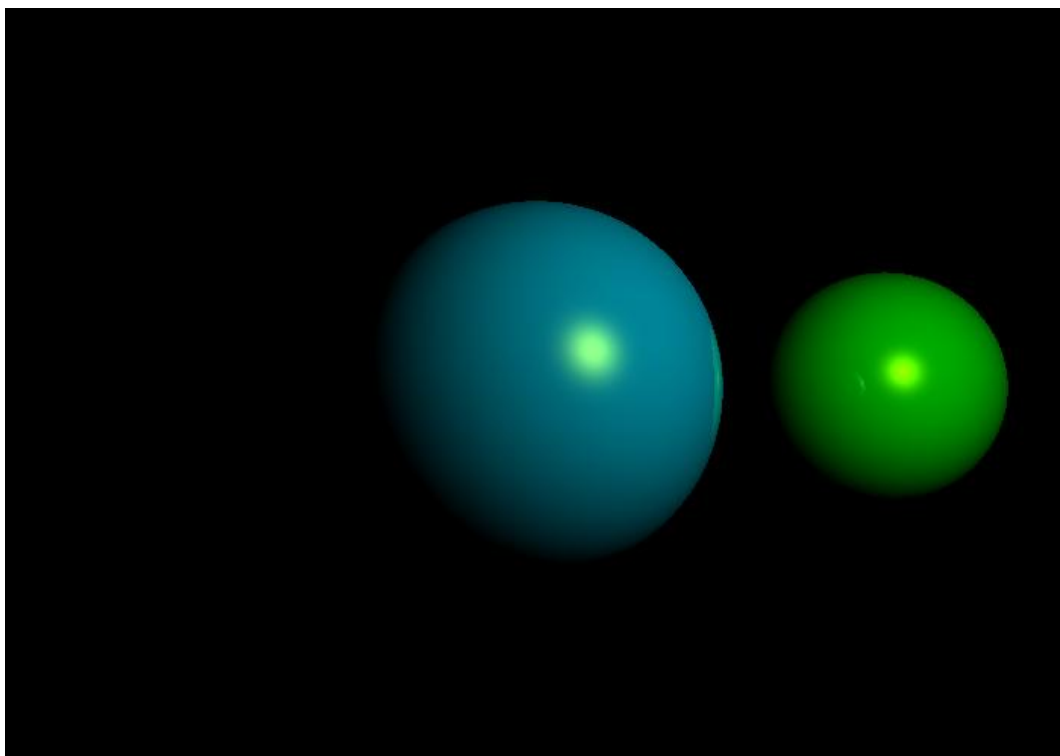
Następnie zaimplementowano funkcję tworzącą scenę analogicznie jak w zadaniu 1, lecz z wykorzystaniem MyRayTracer oraz z położeniem sfer, lepiej uwypuklające działanie odbić. Zmieniono także kolor jednej z kul (składowa diffuse). Funkcja zad2() dana jest rysunkiem nr 8. Wybranie opcji nr 2 w main.exe skutkuje wywołaniem tej funkcji. Wygenerowany obraz zapisany jest jako render2.png. Wynik działania funkcji zad2() dany jest rysunkiem nr 9. Głębokość (maksymalną liczbę odbić) ustawiono na 3.

```
lab5.py

def zad2():
    scene = Scene(
        objects=[
            Sphere(position=np.array([0, 0, 0]), radius=1),
            Sphere(position=np.array([3, 0, -3]), radius=1, diffuse=np.array([0.2, 0.9, 0])),
        ],
        light=Light(position=np.array([3, 2, 5])),
        camera=Camera(position=np.array([0, 0, 5]))
    )

    rt = MyRayTracer(scene)
    image = np.clip(rt.generate_image(), 0, 1)
    image_uint8 = (image * 255).astype(np.uint8)
    img = Image.fromarray(image_uint8)
    img.save("render2.png")
    img.show()
```

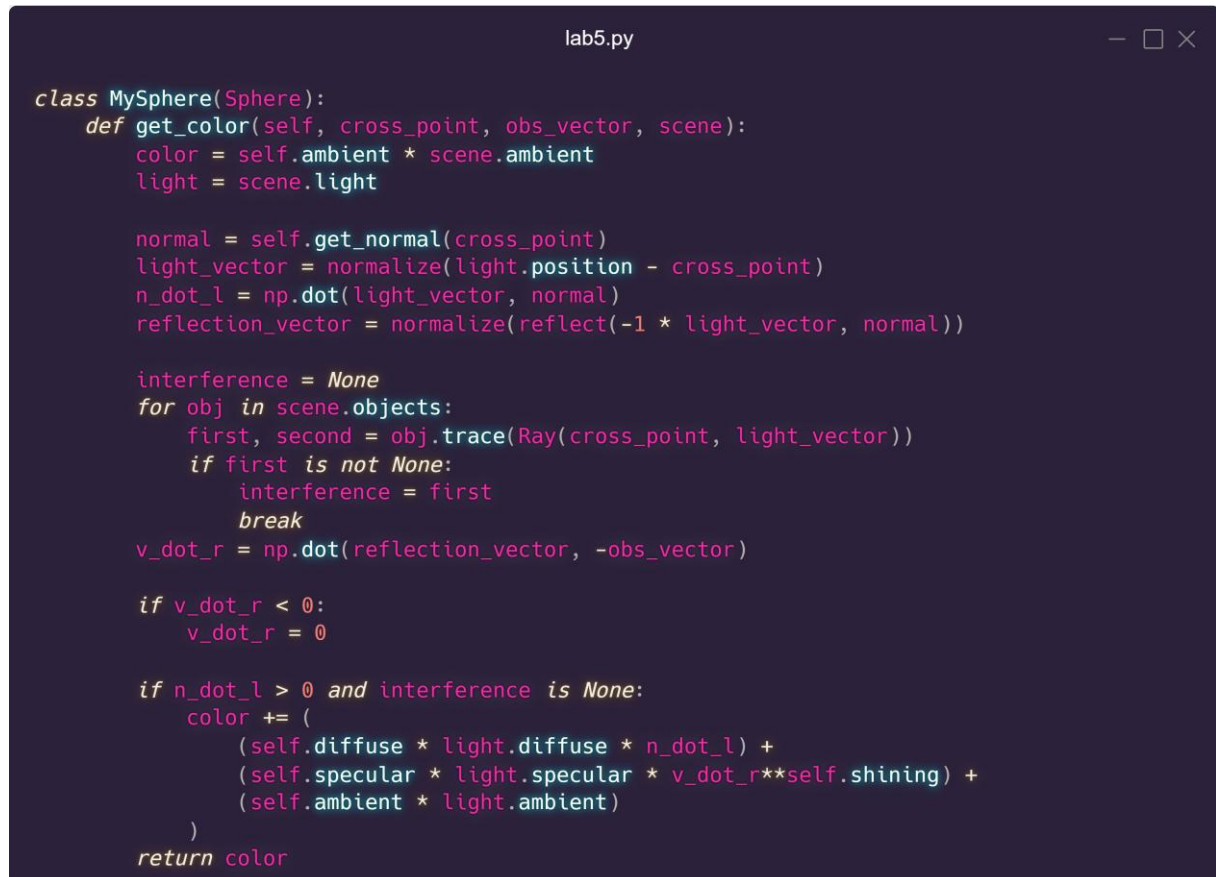
Rysunek 8



Rysunek 9

2.3 Cienie

W tym zadaniu należało zaimplementować prostą obsługę cieni. Utworzono więc klasę `MySphere`, dziedziczącą po klasie `Sphere`, która to dziedziczy po klasie `SceneObject`. Następnie zmodyfikowano funkcję `get_color()`, aby wprowadzić cienie (rysunek 10). Dodano pętlę, w której sprawdzana jest kolizja z innymi obiektami. `MySphere` wysyła promień od punktu do światła i sprawdza, czy coś go przecina. Jeśli na drodze promienia coś się znajduje, parametr `interference` przestaje być ustawiony na `None`, tylko na punkt przecięcia, światło zostaje zablokowane. Dodano również warunek do obliczania koloru, gdy promień nie natrafi na przeszkodę.



```
lab5.py

class MySphere(Sphere):
    def get_color(self, cross_point, obs_vector, scene):
        color = self.ambient * scene.ambient
        light = scene.light

        normal = self.get_normal(cross_point)
        light_vector = normalize(light.position - cross_point)
        n_dot_l = np.dot(light_vector, normal)
        reflection_vector = normalize(reflect(-1 * light_vector, normal))

        interference = None
        for obj in scene.objects:
            first, second = obj.trace(Ray(cross_point, light_vector))
            if first is not None:
                interference = first
                break
        v_dot_r = np.dot(reflection_vector, -obs_vector)

        if v_dot_r < 0:
            v_dot_r = 0

        if n_dot_l > 0 and interference is None:
            color += (
                (self.diffuse * light.diffuse * n_dot_l) +
                (self.specular * light.specular * v_dot_r**self.shining) +
                (self.ambient * light.ambient)
            )
        return color
```

Rysunek 10

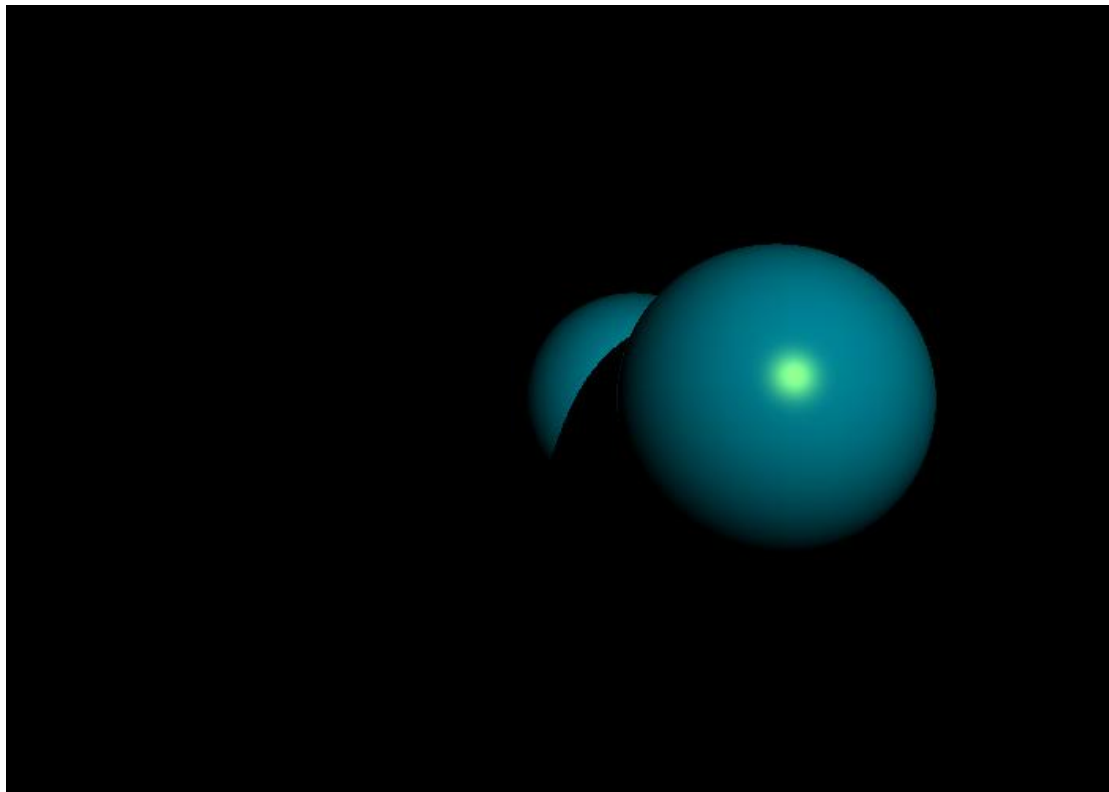
Następnie utworzono scenę z obiektami `MySphere`, w takim ułożeniu, aby obecny był cień rzucany przez jedną kulę na drugą. Scena ta tworzona jest w funkcji `zad3()` daną rysunkiem nr 11. Wybranie opcji nr 3 w programie `main.exe` skutkuje wywołaniem tej funkcji. Obraz wygenerowany przez tę funkcję zapisywany jest jako `render3.png`. Wynik działania programu dany jest rysunkiem nr 12.

```
lab5.py

def zad3():
    scene = Scene(
        objects=[
            MySphere(position=np.array([-1, 0, -3]), radius=1),
            MySphere(position=np.array([1, 0, -1]), radius=1)
        ],
        light=Light(position=np.array([5, 2, 5])),
        camera=Camera(position=np.array([3, 0, 5]))
    )

    rt = MyRayTracer(scene)
    image = np.clip(rt.generate_image(), 0, 1)
    image_uint8 = (image * 255).astype(np.uint8)
    img = Image.fromarray(image_uint8)
    img.save("render3.png")
    img.show()
```

Rysunek 11



Rysunek 12

2.4 Przezroczystość

W kolejnym zadaniu należało zaimplementować obsługę obiektów z zadaną przezroczystością. W tym celu utworzono dwie nowe klasy: `MySphere_2` oraz `MyRayTracer2`.

Klasa `MySphere_2` dziedziczy po klasie `Sphere`. Zawiera jednak nowe pola: `clarity` – odpowiadające za przezroczystość oraz `refraction` – odpowiadające za załamanie światła. Następnie zmodyfikowano funkcję `trace()`, aby zawsze obliczane były dwa punkty przecięcia. Tworzy ona pustą listę `valid`, która będzie zawierać tylko poprawne (fizycznie możliwe) przecięcia promienia z kulą. Reszta funkcji jest analogiczna do pierwowzoru z klasy `Sphere`. Dodatkowo klasa `MySphere_2` posiada jeszcze jedną funkcję śledzenia – `trace_refraction()`. Oblicza ona dwa punkty przecięcia promienia z przezroczystą kulą i zwraca oba. Obliczane są współczynniki równania kwadratowego wynikającego z podstawienia równania promienia do równania kuli. Jeżeli wyróżnik kwadratowy jest mniejszy od zera, nie ma przecięcia — zwraca brak trafienia. Następnie wyliczane są punkty przecięcia, `r1` – bliższy, `r2` – dalszy. Jeśli oba przecięcia są za blisko, zwracany jest brak trafienia. Klasa `MySphere_2` dana jest rysunkami 13 i 14.

Następnie zaimplementowano klasę `MyTracer2`, dziedziczącą po `RayTracer`. Zmodyfikowana została funkcja `_get_pixel_color`.

A screenshot of a code editor window titled 'lab5.py'. The code defines a class 'MySphere_2' that inherits from 'Sphere'. The '__init__' method initializes attributes: 'position', 'radius', 'clarity' (0.0), 'refraction' (0.0), 'ambient' (a numpy array [0, 0, 0]), 'diffuse' (a numpy array [0.6, 0.7, 0.8]), 'specular' (a numpy array [0.8, 0.8, 0.8]), and 'shining' (25). It then calls the parent class's '__init__' method with the same arguments except for 'clarity' and 'refraction', which are assigned to 'self.clarity' and 'self.refraction' respectively.

```
class MySphere_2(Sphere):
    def __init__(
        self,
        position,
        radius,
        clarity=0.0,
        refraction=0.0,
        ambient=np.array([0, 0, 0]),
        diffuse=np.array([0.6, 0.7, 0.8]),
        specular=np.array([0.8, 0.8, 0.8]),
        shining=25
    ):
        super(MySphere_2, self).__init__(
            position=position,
            radius=radius,
            ambient=ambient,
            diffuse=diffuse,
            specular=specular,
            shining=shining
        )
        self.clarity = clarity
        self.refraction = refraction
```

Rysunek 13

```

def trace(self, ray):
    distance = ray.starting_point - self.position
    a = np.dot(ray.direction, ray.direction)
    b = 2 * np.dot(ray.direction, distance)
    c = np.dot(distance, distance) - self.radius**2
    d = b**2 - 4*a*c

    if d < 0:
        return (None, None)

    sqrt_d = d**(0.5)
    denominator = 1 / (2 * a)
    r1 = (-b - sqrt_d) * denominator
    r2 = (-b + sqrt_d) * denominator

    valid = []
    if r1 > EPSILON:
        valid.append(r1)
    if r2 > EPSILON:
        valid.append(r2)

    if not valid:
        return (None, None)

    r = min(valid)
    cross_point = ray.starting_point + r * ray.direction
    return (cross_point, r)

def trace_refraction(self, ray):
    distance = ray.starting_point - self.position
    a = np.dot(ray.direction, ray.direction)
    b = 2 * np.dot(ray.direction, distance)
    c = np.dot(distance, distance) - self.radius**2
    d = b**2 - 4*a*c

    if d < 0:
        return None, None

    sqrt_d = np.sqrt(d)
    denominator = 1 / (2 * a)
    r1 = (-b - sqrt_d) * denominator
    r2 = (-b + sqrt_d) * denominator

    if r1 > r2:
        r1, r2 = r2, r1
    if r2 < EPSILON:
        return None, None

    return r1, r2

```

Rysunek 14

MyTracer2 (rysunek 15) uwzględniając odbicia i załamanie światła. Działa rekurencyjnie: najpierw znajduje najbliższy obiekt przecięty przez promień, a następnie oblicza jego lokalny kolor przy oświetleniu. Jeśli obiekt ma właściwości przezroczyste i załamujące światło, funkcja wyznacza promień refrakcyjny wychodzący z obiektu i uwzględnia jego wpływ na końcowy

kolor. W przeciwnym wypadku łączy kolor lokalny z odbitym. W rezultacie uzyskujemy realistyczne efekty materiałów takich jak szkło, woda czy błyszczące powierzchnie. Do obliczania promienia załamanego wykorzystywany jest uproszczony wzór zadany prawem Snella.

```
lab5.py

class MyRayTracer2(RayTracer):
    def _get_pixel_color(self, ray, depth=3):
        if depth <= 0:
            return np.array([0, 0, 0])

        obj, distance, cross_point = self._get_closest_object(ray)

        if not obj:
            return self.scene.background

        normal = obj.get_normal(cross_point)
        reflected_dir = reflect(ray.direction, normal)
        reflected_ray = Ray(cross_point + normal * 1e-5, reflected_dir)

        local_obj_color = obj.get_color(cross_point, ray.direction, self.scene)
        reflected_pixel_color = self._get_pixel_color(reflected_ray, depth=depth - 1)

        if obj.clarity > 0 and obj.refraction > 0:
            r1, r2 = obj.trace_refraction(ray)
            if r1 is None or r2 is None:
                return 0.85 * local_obj_color + 0.15 * reflected_pixel_color

            exit_point = ray.starting_point + r2 * ray.direction
            exit_normal = obj.get_normal(exit_point)

            cos_theta = np.dot(ray.direction, exit_normal)
            if cos_theta < 0:

                eta = 1.0 / obj.refraction
            else:

                eta = obj.refraction
            exit_normal = -exit_normal

            cosi = -np.dot(exit_normal, normalize(ray.direction))
            k = 1 - eta**2 * (1 - cosi**2)

            if k < 0:
                return local_obj_color * 0.4 + reflected_pixel_color * 0.6

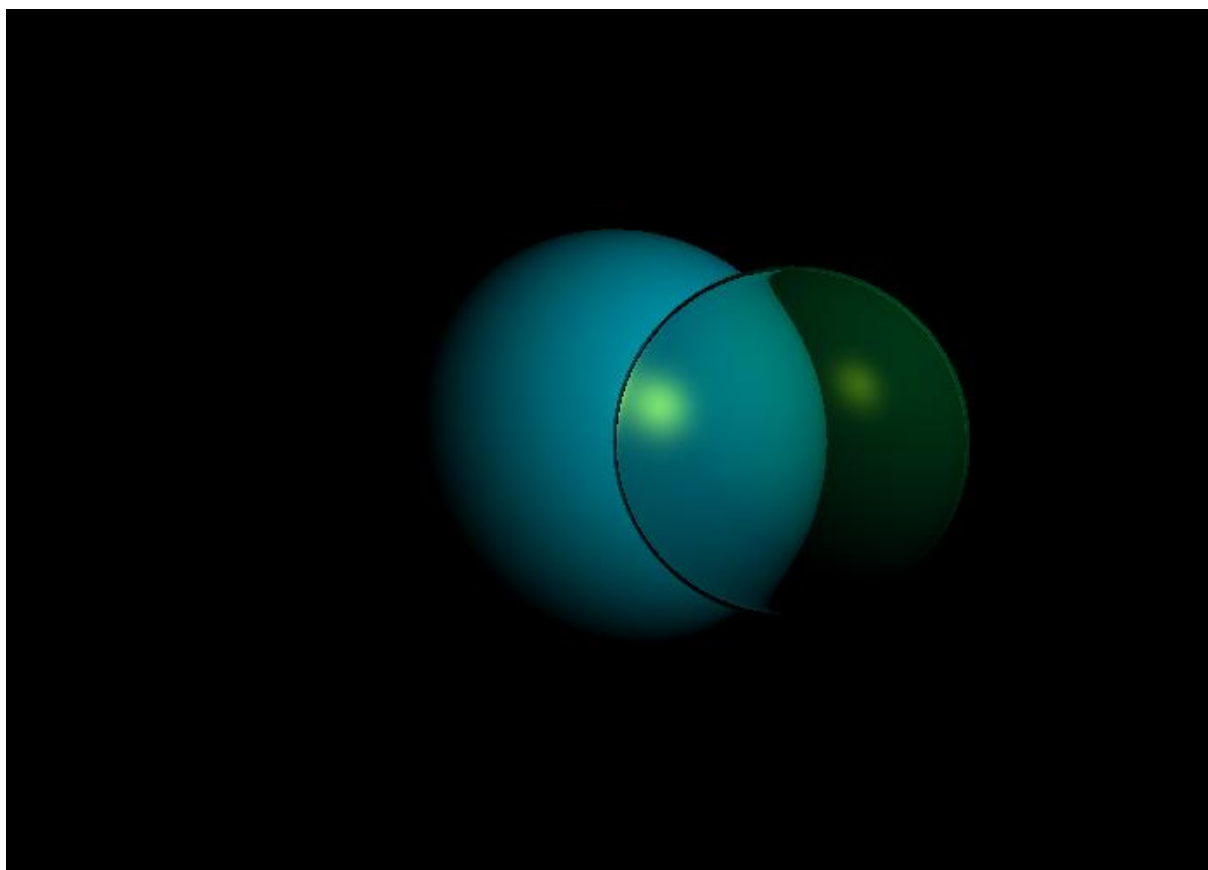
            refracted_dir = eta * normalize(ray.direction) + (eta * cosi - k**(0.5)) * exit_normal
            refract_start = exit_point + exit_normal * 1e-5
            refracted_ray = Ray(refract_start, normalize(refracted_dir))
            refracted_color = self._get_pixel_color(refracted_ray, depth - 1)

            return (
                (1 - obj.clarity) * local_obj_color +
                0.3 * reflected_pixel_color +
                obj.clarity * refracted_color
            )
        else:
            return 0.85 * local_obj_color + 0.15 * reflected_pixel_color
```

Rysunek 15

Składnik k jest obliczany właśnie z prawa Snelliusa. Jeśli jest mniejszy od 0, zachodzi całkowite wewnętrzne odbicie. W przeciwnym wypadku zwracany jest kolor powstały z odpowiednio zważonych kolorów składowych (lokalny, odbicie i refrakcja).

Następnie zaimplementowano funkcję `zad4()` (rysunek 16), generującą scenę zawierającą kulę nieprzezroczystą i przezroczystą. Funkcja ta zostanie wywołana po wyborze opcji nr 4 w programie `main.exe`. Zapisuje ona wygenerowany obraz jako `render4.png`. Wynik działania programu dany jest rysunkiem nr 16. Współczynnik załamania ustawiono na 1.02.



Rysunek 16

2.5 Trójkąt

Ostatnie zadanie polegało na dodaniu dodatkowej figury. Wybrano trójkąt. Klasa `Triangle` dziedziczy po `SceneObject` i bazuje swoją budową na klasie `Sphere`. W przeciwieństwie jednak do tej klasy, przyjmuje jako argumenty współrzędne wierzchołków, a nie pozycję środka kuli. Jest to obiekt płaski, więc prosta normalna do niego jest stała, a nie jak w kuli, gdzie zależy od punktu przecięcia. Funkcja `trace` zawiera w sobie odpowiednik funkcji `is_point_in_triangle()` z laboratorium nr 4. Działa jednak w przestrzeni trójwymiarowej, wykorzystując iloczyn wektorowy i skalarny. Najpierw obliczany jest iloczyn wektorowy krawędzi oraz wektora od wierzchołka trójkąta, do punktu przecięcia. Następnie obliczany jest iloczyn skalarny z normalną, sprawdzający, czy iloczyn wektorowy i normalna mają ten sam zwrot. Dokładniejszy matematyczny opis wykorzystanego tu mechanizmu opisany jest w sprawozdaniu do laboratorium nr 4.

Klasa `triangle` dana jest rysunkiem nr 17.


```

lab5.py

class Triangle(SceneObject):
    def __init__(self, a, b, c, **kwargs):
        super().__init__(**kwargs)
        self.a = a
        self.b = b
        self.c = c
        self.normal = self._calculate_normal()

    def _calculate_normal(self):
        v1 = self.b - self.a
        v2 = self.c - self.a
        return normalize(np.cross(v1, v2))

    def get_normal(self, point):
        return self.normal

    def trace(self, ray):

        denominator = np.dot(self.normal, ray.direction)
        if abs(denominator) < EPSILON: return (None, None)

        t = np.dot(self.a - ray.starting_point, self.normal)/denominator
        if t < EPSILON: return (None, None)

        point = ray.starting_point + ray.direction*t

        edge1 = self.b - self.a
        edge2 = self.c - self.b
        edge3 = self.a - self.c
        vp1 = point - self.a
        vp2 = point - self.b
        vp3 = point - self.c

        c1 = np.dot(np.cross(edge1, vp1), self.normal)
        c2 = np.dot(np.cross(edge2, vp2), self.normal)
        c3 = np.dot(np.cross(edge3, vp3), self.normal)

        if (c1 > 0 and c2 > 0 and c3 > 0) or (c1 < 0 and c2 < 0 and c3 < 0):
            return (point, t)
        return (None, None)

```

Rysunek 17

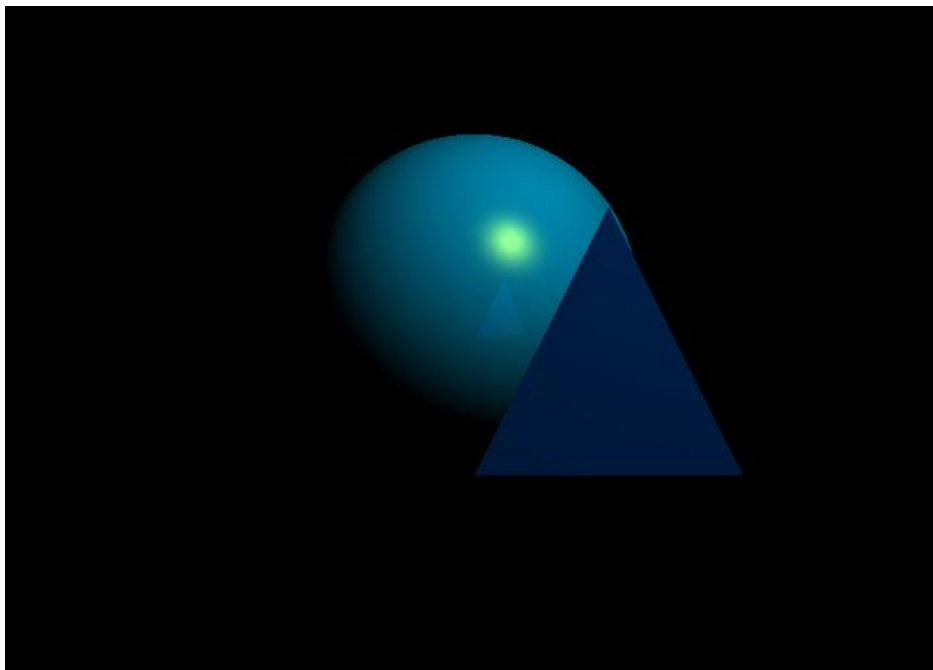
Następnie zaimplementowano funkcję zad5(), daną rysunkiem nr 18. Generowana jest w niej scena zawierająca kulę oraz nowo zaimplementowany trójkąt. Wygenerowana scena zapisywana jest jako render5.png. Wybranie opcji nr 5 w programie main.exe skutkuje wywołaniem tej funkcji. Wynik działania tej funkcji dany jest rysunkiem nr 19.

```
lab5.py

def zad5():
    scene = Scene(
        objects=[
            MySphere_2(
                position=np.array([0, 0.5, -2]),
                radius=2,
                diffuse=np.array([0.3, 0.7, 0.9])
            ),
            Triangle(
                a=np.array([0, -1, 2]),
                b=np.array([2, -1, 2]),
                c=np.array([1, 1, 2]),
                diffuse=np.array([0.5, 0.3, 0.7])
            )
        ],
        light=Light(position=np.array([3, 5, 5])),
        camera=Camera(
            position=np.array([0, 0, 8]),
            look_at=np.array([0, 0, 0])
        )
    )

    rt = MyRayTracer(scene)
    image = np.clip(rt.generate_image(), 0, 1)
    img = Image.fromarray((image*255).astype(np.uint8))
    img.save("render5.png")
    img.show()
```

Rysunek 18



Rysunek 19

3. Ogólna struktura projektu

Realizacje zadań znajduje się w pliku lab5.py. Interfejs tekstowy do wywoływania funkcji z lab4.py dany jest plikiem main.py. Za pomocą narzędzia PyInstaller utworzono plik wykonywalny main.exe, odpowiadający plikowi main.py. Znajduje się on w folderze dist.

Źródła

https://pl.wikipedia.org/wiki/O%C5%9Bwietlenie_globalne

https://sites.cc.gatech.edu/classes/AY2003/cs4451a_fall/global-illumination.pdf

<https://www.dgp.toronto.edu/~karan/courses/csc418/lectures/l15.pdf>

<https://personal.math.ubc.ca/~cass/courses/m309-03a/m309-projects/olafson/illumination.htm>

<http://mchwesiuk.pl/wp-content/uploads/2019/04/Grafika-Komputerowa-W5.pdf>

https://hd.fizyka.umk.pl/~jacek/dydaktyka/3d/grafika3d_animacja/2013L_OpenGL_ES_PhysX/referaty/2013-06-10_MMoczadlo_Model_Oswietlenia_W_Grafice_3D.pdf

<https://sound.eti.pg.gda.pl/student/tpm/rendering.pdf>

https://edirlei.com/aulas/cg-2020/CG_Lecture_08_Global_Illumination_2020.html

<https://users.pja.edu.pl/~denisjuk/gk/wyklady/g14-rayTracing.pdf>

https://en.wikipedia.org/wiki/Phong_reflection_model#/media/File:Phong_components_version_4.png

https://www.researchgate.net/publication/370763969_Accelerating_Java_Ray_Tracing_Applications_on_Heterogeneous_Hardware

<https://datko.pl/IOb/lab7.zip>