Skrypt do zajęć laboratoryjnych z przedmiotu Grafika i wizualizacja

Radosław Mantiuk

Luty 2023

Spis treści

1 Ray Tracing **5**
- 1.1 Środowisko programistyczne . . . 5
  - 1.1.1 Biblioteka wspomagająca obliczenia wektorowo-macierzowe (GLM) . . . 6
  - 1.1.2 Podstawowe przekształcenia geometryczne . . . 6
  - 1.1.3 Struktura oprogramowania ray tracera . . . 8
  - 1.1.4 Klasa `CImage` . . . 9
- 1.2 Kamera . . . 10
  - 1.2.1 Model kamery perspektywicznej . . . 10
  - 1.2.2 Obliczanie parametrów macierzy rzutowania . . . 11
  - 1.2.3 Obliczanie kierunku promienia pierwotnego . . . 12
- 1.3 Widoczność obiektów . . . 13
  - 1.3.1 Reprezentacja kuli . . . 14
  - 1.3.2 Reprezentacja trójkąta . . . 14
  - 1.3.3 Lista obiektów . . . 15
  - 1.3.4 Obliczanie przecięcia promienia z kulą . . . 15
  - 1.3.5 Obliczanie widoczności obiektów . . . 16
  - 1.3.6 Obliczanie przecięcia promienia z trójkątem . . . 17
- 1.4 Model oświetlenia lokalnego . . . 18
  - 1.4.1 Materiał obiektu . . . 18
  - 1.4.2 Źródła światła . . . 19
  - 1.4.3 Wektor normalny . . . 19
  - 1.4.4 Równanie oświetlenia . . . 20
  - 1.4.5 Teksturowanie . . . 22
- 1.5 Elementy modelu oświetlenia globalnego . . . 24
  - 1.5.1 Globalny model oświetlenia . . . 25
  - 1.5.2 Obliczanie cieni . . . 25
  - 1.5.3 Odbicia zwierciadlane . . . 26
  - 1.5.4 Rekurencyjne śledzenie promieni . . . 27

2 Wizualizacja **31**
- 2.1 Kwantyzacja kolorów . . . 31
- 2.2 Korekcja gamma . . . 31

A Materiały pomocnicze **35**
- A.1 Repozytorium GIT . . . 35
  - A.1.1 Tworzenie repozytorium . . . 35
  - A.1.2 Podstawowe komendy GIT . . . 35
- A.2 Transformacje geometryczne . . . 36

Rozdział 1

Ray Tracing

Omówienie podstawowych algorytmów grafiki komputerowej na przykładzie implementacji metody śledzenia promieni (ang.*ray tracing*). Tematyka rozdziału obejmuje:

- Pojęcie kamery i związane z nim rzutowanie z przestrzeni trójwymiarowej to dwuwymiarowej płaszczyzny obrazy (Podroz.1.2).
- Reprezentację obiektów geometrycznych na trójwymiarowej scenie oraz zagadnienie widoczności obiektów z punktu widzenia położenia kamery (Podroz.1.3).
- Model oświetlenia wykorzystywany do obliczanie koloru obiektów, reprezentację materiałów obiektów oraz sposób definiowania źródeł światła (Podroz.1.4).
- Pojęcie globalnego modelu oświetlenia wykorzystywanego do renderingu cieni oraz obiektów o powierzchniach zwierciadlanych (Podroz.1.5).

1.1 Środowisko programistyczne

Cel zajęć: Zapoznanie się ze strukturą oprogramowania do syntezy obrazów metodą śledzenia promieni oraz z narzędziami używanymi do jego implementacji.

Kompilacja i uruchamianie oprogramowania ray tracera przeprowadzane będzie w środowisku programistycznym **JetBrains CLion**. Oprogramowanie IDE CLion udostępniane jest bezpłatnie do celów edukacyjnych. Uzyskanie licencji akademickiej wymaga założenia konta z wykorzystaniem adresu w domenie `@zut.edu.pl`. Aktywację przeprowadza się po zainstalowaniu i uruchomieniu CLion poprzez zalogowanie na wcześniej założone konto.

Dokumentacja do środowiska IDE CLion dostępna jest stronach JetBrains (adres strony).

1. Ćwiczenie - Przygotowanie środowiska programistycznego

Wykonać poniższe czynności (pomocnicze materiały znajdują się w Dodatku A.1):

1. Na stronie JetBrains złożyć wniosek o licencję oraz utworzyć konto akademickie (link do strony).
2. Pobrać ze strony przedmiotu i rozpakować archiwum z szablonem klas ray tracera (ray_tracer_template.zip).
3. Przekopiować pliki z archiwum z podkatalogu `[zip]/ray_tracer_templete/` do podkatalogu `[git]/ray_tracer/` w lokalnym repozytorium GIT. Pliki powinny zostać skopiowane bezpośrednio do `[git]/ray_tracer/` bez dodatkowego podkatalogu.
4. Przesłać dane z lokalnego repozytorium na serwer GIT wykonując sekwencję komend git *add* -> *commit* -> *push* dla wszystkich plików z archiwum.
5. Uruchomić i aktywować środowisko IDE *CLion* poprzez zalogowanie się na konto akademickie.
6. Utworzyć nowy projekt poprzez naciśnięcie klawisza *[Open]* i wskazanie katalogu `[git]/ray_tracer/`.
7. Zatwierdzić kolejne okna dialogowe bez wprowadzania w nich zmian.

8. Skompilować i uruchomić projekt.
9. Wprowadzić zmiany w jednym z plików źródłowych, np. dodając komentarz.
10. Zatwierdzić zmiany w lokalnym repozytorium (*git commit -a -m "opis zmian"*).
11. Przesłać zmiany na serwer GITa.
12. Sprawdzić poprawność działań logując się do repozytorium w przeglądarce i sprawdzając zmiany wprowadzone w wybranym pliku.

W czasie realizacji projektów wymagane jest korzystanie z repozytorium GIT, w którym na bieżąco trzeba archiwizować pliki tworzone i edytowane podczas zajęć laboratoryjnych. Przesłanie danych z lokalnego kopii do repozytorium na serwerze trzeba zawsze wykonać na zakończenie zajęć.

1.1.1 Biblioteka wspomagająca obliczenia wektorowo-macierzowe (GLM)

GLM (ang. Graphics Library Mathematics) to biblioteka nagłówkowa (patrz `[git]/ray_tracer/3rd/glm/`), w której zaimplementowano klasy reprezentujące wektory i macierze oraz operacje matematyczne na ich strukturach danych.

Najważniejszymi, z punktu widzenia implementacji ray tracera, klasami i metodami biblioteki GLM są:

- `glm::vec3` - trójwymiarowy wektor opisujący np. położenie (x, y, z) bądź wartości koloru (r, g, b).
- `glm::mat3` - macierz 3x3 elementowa (3 wiersze i 3 kolumny).
- `glm::cross()` - metoda obliczająca iloczyn wektorowy.
- `glm::dot()` - metoda obliczająca iloczyn skalarny.

1.1.2 Podstawowe przekształcenia geometryczne

Do podstawowych przekształceń geometrycznych umożliwiających zmianę kształtu oraz położeni obiektów należą **translacja**, **skalowanie** i **obrót**. Sposobem wykonywania przekształceń jest zmiana położenia wierzchołka $\boldsymbol{p}$ obiektu poprzez wymnożenie jego położenia przez macierz przekształcenia:

$$\boldsymbol{p'} = M \cdot \boldsymbol{p}, \tag{1.1}$$

gdzie $\boldsymbol{p'}$ to położenie wierzchołka po przekształceniu, M to macierz przekształcenia.

Składanie przekształceń polega na wymnażaniu macierzy odpowiadających poszczególnym przekształceniom. Takie rozwiązanie jest efektywne pod względem obliczeniowym, ponieważ wierzchołek można poddać złożonemu przekształceniu geometrycznemu tylko jednokrotnie pobierając i zapisując jego dane do pamięci. Najczęściej wszystkie wierzchołki obiektu poddawane są jednemu przekształceniu złożonemu, dlatego jego macierz obliczana jest tylko raz dla całego obiektu. Aby możliwe było składanie trzech przekształceń podstawowych, macierz przekształcenia musi mieć o jeden wymiar więcej niż wektor położenia wierzchołka. Dla dwuwymiarowego przypadku obiektów na płaszczyźnie oznacza to stosowanie macierzy 3x3:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \tag{1.2}$$

gdzie (x, y) i (x', y') to położenie wierzchołka odpowiednio przed i po przekształceniu, a_{0-8} to parametry decydujące o rodzaju i wielkości przekształcenia.

Macierz translacji o wektor (t_x, t_y) ma postać:

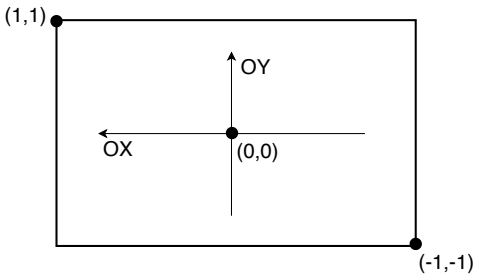
$$M_t = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{1.3}$$

Macierz skalowania o współczynniki (s_x, s_y) ma postać:

$$M_s = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{1.4}$$

Macierz obrotu wokół punktu $(0,0)$ o kąt θ ma postać:

$$M_r = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{1.5}$$



Rysunek 1.1: Orientacja oraz zakres wartości dwuwymiarowego układu współrzędnych.

2. Ćwiczenie - Przekształcenia geometryczne z wykorzystaniem biblioteki GLM

Wykorzystując bibliotekę GLM oraz macierze przekształceń geometrycznych narysować na ekranie kształt przypominający robota (parz Rys.1.2 (po lewej)). Kod rysujący robota zaimplementować w funkcji `draw_robot()` wywoływanej na początku funkcji `main()`.

Sposób rysowania trójkąta oraz wykonania translacji tego trójkąta prezentowany jest w funkcji `draw_triangle()` w pliku `main.cpp`. Rysowanie trójkąta polega na zdefiniowaniu jego wierzchołków, a następnie wywołaniu metody `CImage::drawLine()` rysującej linię pomiędzy wierzchołkami:

```
// triangle vertices
glm::vec3 pp0(0.4, 0.3, 1);
glm::vec3 pp1(-0.4, 0.3, 1);
glm::vec3 pp2(0.4, -0.3, 1);
// draws triangle in 2D
img.drawLine(pp0, pp1, color1);
img.drawLine(pp1, pp2, color1);
img.drawLine(pp2, pp0, color1);
```

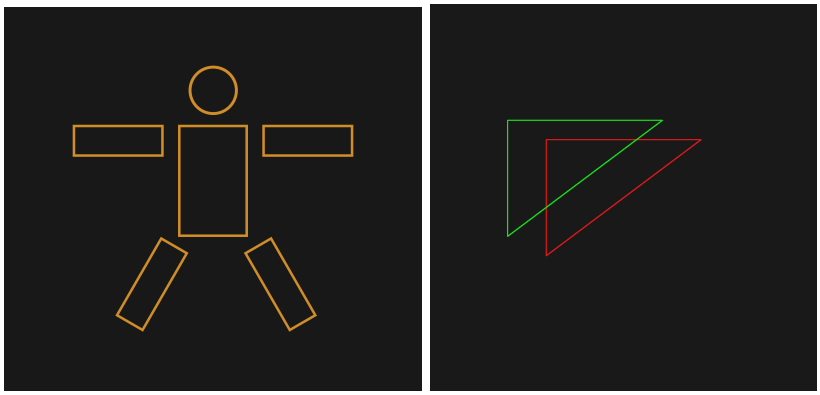
Na potrzeby projektu do rysowania dwumiarowych kształtów wektorowych wykorzystany został układ współrzędnych prezentowany na Rys.1.1. Zakłada on położenie punktu $(0,0)$ na środku obrazu ze współrzędnymi zmieniającymi się w zakresie $< -1, 1 >$ pomiędzy krawędziami obrazu.

Translacja wierzchołków trójkąta wymaga utworzenia macierzy translacji, w której podane są wartości wektora translacji, w następnie wymnożenia tej macierzy razy położenie wierzchołków:

```
// translation
float tX = 0.2f; // OX translation
float tY = 0.1f; // OY translation
glm::mat3x3 mTrans {{1,0,0}, {0,1,0}, {tX,tY,1}}; // translation matrix
// translation of vertices
pp0 = mTrans * pp0;
pp1 = mTrans * pp1;
pp2 = mTrans * pp2;
// draws triangle after translation
img.drawLine(pp0, pp1, color2);
img.drawLine(pp1, pp2, color2);
img.drawLine(pp2, pp0, color2);
```

Rezultat działania kodu prezentowany jest na Rys.1.2 (po prawej). Zwrócić uwagę na to, że mnożenie macierzy nie jest przemienne, dlatego obiekt po transformacji będzie miał inny kształt po zmianie kolejności przekształceń wykonywanych na tym obiekcie. Obrót odbywa się zawsze wokół punktu $(0, 0)$.

Dobrą praktyką jest rysowanie kolejnych części robota poprzez utworzenie kwadratu o środku w punkcie $(0, 0)$, a następnie przekształcanie tego kwadratu do pożądanego kształtu i położenia kolejno wykonując skalowanie, obrót, a na końcu translację. Wymienione transformacje należy wykonywać mnożąc macierz przekształcenia razy wierzchołki kwadratu. Przekształcenie złożone uzyskujemy poprzez wymnożenie macierzy poszczególnych przekształceń.



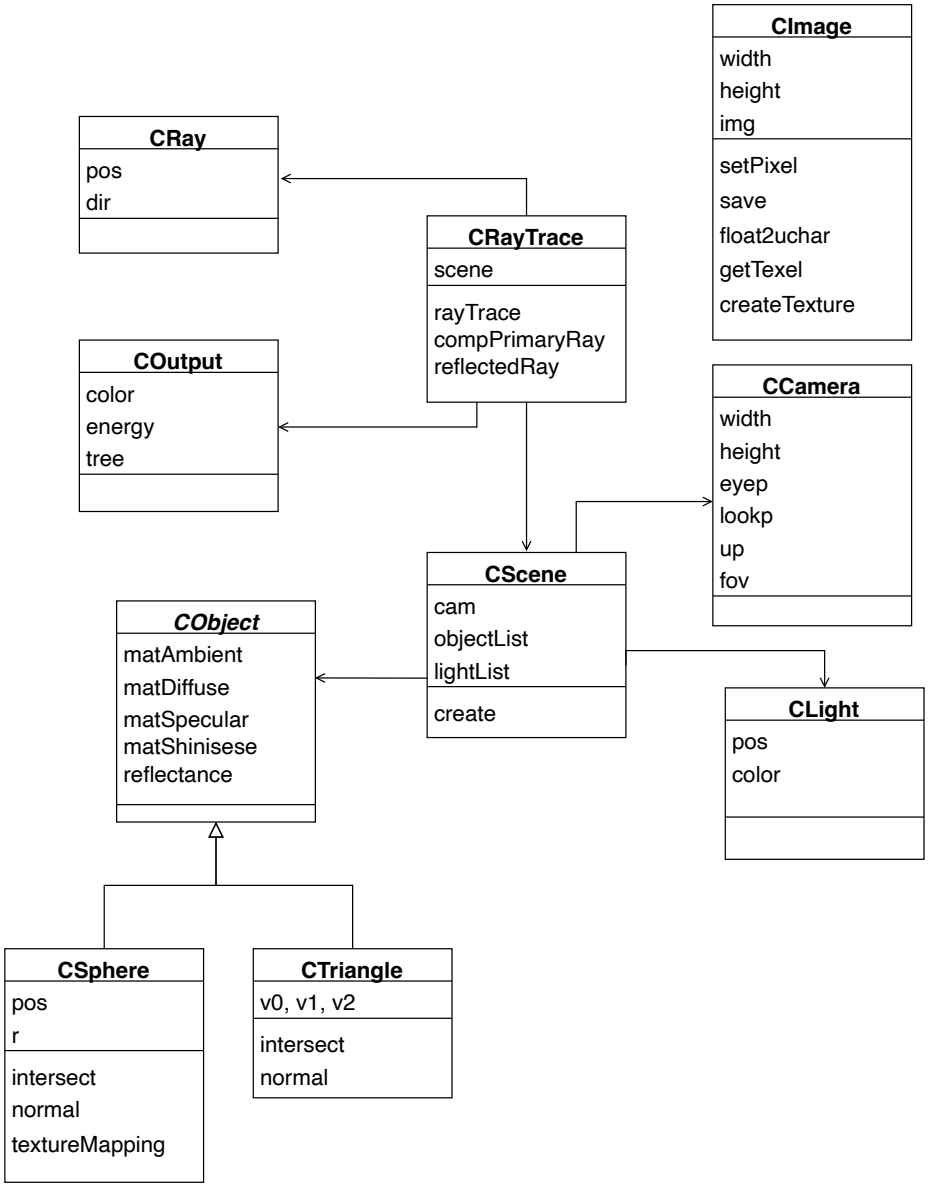
Rysunek 1.2: Przykłady dwuwymiarowej grafiki wektorowej.

1.1.3 Struktura oprogramowania ray tracera

Ray tracer należy zaimplementować w języku C++ w oparciu o dostarczony szablon klas. Struktura klas została zaprojektowana pod kątem czytelności kodu ray tracera. Schemat klas prezentowany jest na Rys.1.3.

Algorytm ray tracingu realizowany jest w następujących etapach:

1. Utworzenie sceny (`CScene::create()`) poprzez umieszczenie na niej kamery (`CCamera`), źródeł światła (`CLight`) oraz obiektów (`CSphere` lub `CTriangle`) (Podroz.1.3).
2. Obliczenie macierzy projekcji definiującej sposób obliczania rzutowania perspektywicznego (Podroz.1.2.2).
3. Dla każdego piksela obrazu:
   (a) Obliczenie parametrów promienia pierwotnego (`CRayTrace::compPrimaryRay()`) oraz przekazanie śledzenia promienia do metody `CRayTrace` (Podroz.1.2.3).
   (b) Znalezienie punktu przecięcia promienia z obiektem (`CSphere::intersect()`, `CTriangle::intersect()`) (Podroz.1.3.4, Podroz.1.3.6).
   (c) Obliczenie wektora normalnego w punkcie przecięcia (`CSphere::normal()`, `CTriangle::normal()`) (Podroz.1.4.3).
   (d) Sprawdzenie czy punkt przecięcia leży w cieniu (Podroz.1.5.2).
   (e) Oblicznie koloru dla punktu przecięcia (Podroz.1.4.4).
   (f) Uwzględnienie koloru tektury w kolorze obiektu (Podroz.1.4.5).
   (g) Dla powierzchni zwierciadlanych obliczenie promienia odbitego od powierzchni (`CRayTrace::rayReflected()`) (Podroz.1.5.3).
   (h) Rekurencyjne wywołanie śledzenia promienia odbitego (Podroz.1.5.4).
   (i) Zapisanie koloru piksela w danych obrazu (`CImage::setPixel()`).
4. Wykonanie kwantyzacji oraz korekcji gamma danych obrazu (Podroz.2.1, Podroz.2.2).
5. Zapisanie obrazu do pliku lub wyświetlenie go na ekranie monitora (`CImage::save()`).



Rysunek 1.3: Struktura klas w implementacji ray tracera.

1.1.4 Klasa `CImage`

Wynikiem działania metody śledzenia promieni są obliczone kolory pikseli wyjściowego obrazu rastrowego. W szablonie oprogramowania ray tracera udostępniona została klasa `CImage` służąca do przechowywania danych obrazu oraz zawierająca zestaw funkcji do przetwarzania obrazu oraz zapisywania go na dysku w plikach graficznych.

W klasie `CImage` udostępnione zostały metody:

- `CImage::save()` - służąca do zapisywania danych obrazu w pliku graficznym oraz wykonująca korekcję gamma obrazu.
- `CImage::setPixel()` - zapisująca kolor piksela o wskazanych współrzędnych.
- `CImage::drawLine()` - rysująca linię w obrazie rastrowym.
- `CImage::drawCircle()` - rysująca okrąg w obrazie rastrowym.
- `CImage::createTexture()` - służąca do tworzenia obrazu rastrowego wykorzystywanego jako tekstura.
- `CImage::getTexel()` - zwracająca kolor wskazanego teksela tekstury.

- `CImage::plotCalibChart()` - rysująca wzorzec służący do pomiaru krzywej współczynnika gamma wyświetlacza.

Klasa `CImage` wykorzystuje funkcje z biblioteki OpenCV.

1.2 Kamera

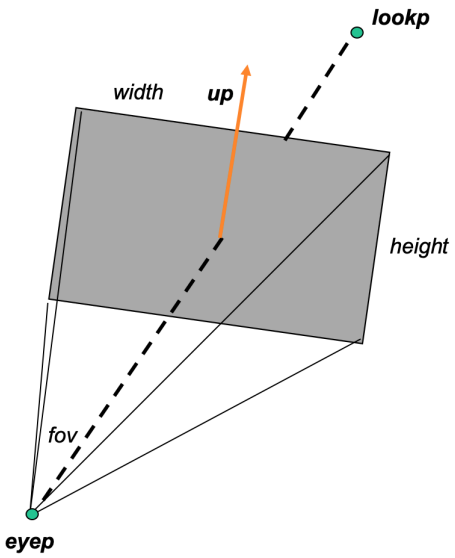
Cel: Zapoznanie się ze sposobami reprezentacji kamery w systemach graficznych w tym z parametryzacją modelu kamery w ray tracingu. Implementacja obliczania kierunku promieni pierwotnych dla kamery perspektywicznej odpowiedzialnego za sposób rzutowania obiektów z trójwymiarowej przestrzeni sceny do dwuwymiarowej płaszczyzny obrazu.

1.2.1 Model kamery perspektywicznej

Model kamery zdefiniowany jest za pomocą parametrów zilustrowanymi na Rys. 1.4.

W implementacji ray tracera parametry kamery reprezentowane są przez klasę `CCamera`:

```
class CCamera {
public:
    int width; // number of pixels in horizontal direction
    int height; // number of pixels in vertical direction
    float fov; // horizontal field-of-view
    glm::vec3 eyep; // camera position
    glm::vec3 lookp; // camera "looks" at this position
    glm::vec3 up; // image orientation
};
```



Rysunek 1.4: Model kamery.

3. Ćwiczenie - Definicja parametrów kamery

W metodzie `CScene:create()` nadać odpowiednim polom obiektu `CSene::cam` poniższe wartości:

$$\boldsymbol{eyep} = [0, 0, 10], \boldsymbol{lookp} = [0, 0, 0], \boldsymbol{up} = [0, 1, 0], fov = 50, width = 500, height = 400. \tag{1.6}$$

1.2.2 Obliczanie parametrów macierzy rzutowania

Macierz rzutowania jest reprezentacją kamery służącą do obliczania kierunku promieni pierwotnych z uwzględnieniem projekcji perspektywicznej sceny. Macierz przyjmuje formę:

$$P = \begin{bmatrix} u_x & v_x & o_x \\ u_y & v_y & o_y \\ u_z & v_z & o_z \end{bmatrix} \tag{1.7}$$

Parametry wektorów $\boldsymbol{u}(x,y,z), \boldsymbol{v}(x,y,z)$ i $\boldsymbol{o}(x,y,z)$ (patrz Rys. 1.5) obliczane są ze wzorów:

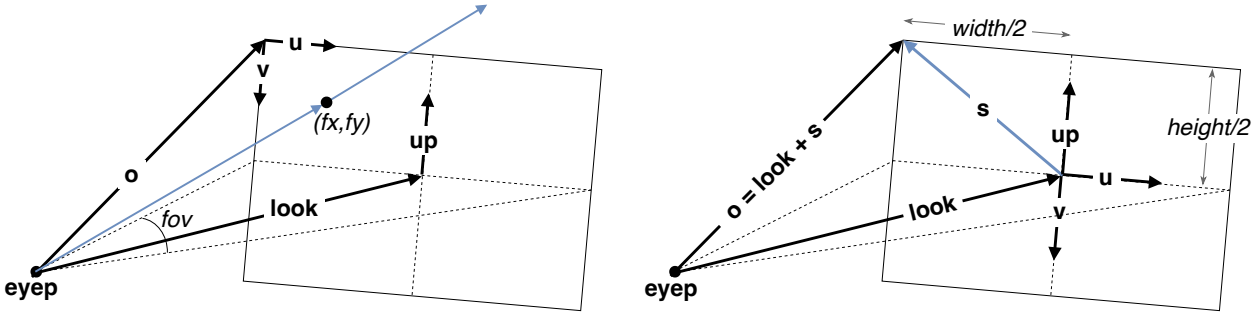
$$\boldsymbol{u} = \frac{\boldsymbol{up} \times \boldsymbol{look}}{|\boldsymbol{up} \times \boldsymbol{look}|}, \tag{1.8}$$

$$\boldsymbol{v} = \frac{\boldsymbol{u} \times \boldsymbol{look}}{|\boldsymbol{u} \times \boldsymbol{look}|}, \tag{1.9}$$

$$\boldsymbol{o} = \frac{\boldsymbol{look}}{|\boldsymbol{look}|} \cdot \frac{width}{2 \cdot tan(fov/2)} - (\frac{width}{2} \cdot \boldsymbol{u} + \frac{height}{2} \cdot \boldsymbol{v}), \tag{1.10}$$

$$\boldsymbol{look} = \boldsymbol{lookp} - \boldsymbol{eyep}, \tag{1.11}$$

gdzie symbol $\times$ reprezentuje iloczyn wektorowy (`glm::cross()`), natomiast symbol $|...|$ długość wektora (`glm::length()`).



Rysunek 1.5: Parametry macierzy rzutowania.

4. Ćwiczenie - Macierz rzutowania

Zaimplementować metodę obliczającą macierz rzutowania (`CRayTrace::compPrimaryRayMatrix()`). Przetestować działanie metody dla parametrów kamery:

$$\boldsymbol{eyep} = [0,0,10], \boldsymbol{lookp} = [0,0,0], \boldsymbol{up} = [0,1,0], fov = 50, width = 500, height = 400: \tag{1.12}$$

$$M = \begin{bmatrix} -1 & 0 & 250 \\ 0 & -1 & 200 \\ 0 & 0 & -536.436 \end{bmatrix} \tag{1.13}$$

$$\boldsymbol{eyep} = [2,5,10], \boldsymbol{lookp} = [-2,1,0], \boldsymbol{up} = [0,1,0], fov = 50, width = 500, height = 400: \tag{1.14}$$

$$M = \begin{bmatrix} -0.928477 & 0.129302 & 19.4956 \\ 0 & -0.937437 & 0.724167 \\ 0.371391 & 0.323254 & -624.406 \end{bmatrix} \tag{1.15}$$

Uwaga: Przykładowe wyniki obliczone zostały dla konwersji stopni do radianów (wykorzystywanym do obliczenia tangensa kąta) za pomocą równania `radiany = stopnie * 3.14f / 180.0f`. Zastosowanie dokładniejszej wartości liczby π może nieznacznie zmienić końcowe rezultaty.

1.2.3 Obliczanie kierunku promienia pierwotnego

Promień pierwotny to wektor o początku w punkcie położenia kamery i kierunku wyznaczonym przez położenie punktu na obrazie.

W implementacji ray tracera parametry promienia reprezentowane są przez klasę `CRay`:

```
class CRay{
public:
    glm::vec3 pos; // starting position of the ray
    glm::vec3 dir; // direction of the ray
};
```

Punkt początkowy promienia znajduje się w punkcie położenia kamery, tzn. $\boldsymbol{ray.pos} = \boldsymbol{cam.eyep}$. Obliczenie kierunku promienia pierwotnego polega na wykonaniu operacji mnożenia macierzy rzutowania przez położenie punktu na płaszczyźnie obrazu:

$$\boldsymbol{ray.dir} = normalize(\begin{bmatrix} u_x & v_x & o_x \\ u_y & v_y & o_y \\ u_z & v_z & o_z \end{bmatrix} \cdot \begin{bmatrix} fx \\ fy \\ 1 \end{bmatrix}), \tag{1.16}$$

gdzie $normalize(...)$ oznacza normalizację wektora, tzn. podzielenie parametrów wektora przez jego długość L2. W rezultacie normalizacji wektor $\boldsymbol{ray.dir}$ będzie miał długość jednostkową.

Parametry (fx, fy) nie muszą przyjmować wartości całkowitych co jest równoznaczne z traktowaniem piksela jako obszaru na płaszczyźnie obrazu. Promień może przechodzić w dowolnym miejscu tego obszaru. Na potrzeby tego projektu można przyjąć, że promień przechodzi przez środek piksela, tzn. że parametry $fx = i + 0.5$ i $fy = j + 0.5$, gdzie (i, j) to położenie piksela w rastrze obrazu.

5. Ćwiczenie - Promień pierwotny

Zaimplementować w *main()* obliczanie kierunku promienia pierwotnego dla przykładowych parametrów kamery oraz położenia piksela. Zaobserwować zmianę kierunku w trójwymiarowym układzie współrzędnych.

Parametry kamery:

$$\boldsymbol{eyep} = [0, 0, 10], \boldsymbol{lookp} = [0, 0, 0], \boldsymbol{up} = [0, 1, 0], fov = 50, width = 500, height = 400 \tag{1.17}$$

Wartości parametrów kierunku promieni pierwotnych przechodzących przez testowe punkty na płaszczyźnie obrazu:

- $\boldsymbol{dir}$ = `[0.0,0.0,-1.0]` dla `(fx,fy)=[width/2-1+0.5),(height/2-1+0.5)]`,
- $\boldsymbol{dir} \simeq [0.40, 0.32, -0.86]$ dla $(fx, fy) = [0.5, 0.5)]$,
- $\boldsymbol{dir} \simeq [0, 0.35, -0.94]$ dla $(fx, fy) = [(width/2 - 1 + 0.5), 0.5]$,
- $\boldsymbol{dir} \simeq [0.42, 0, -0.91]$ dla $(fx, fy) = [0.5, (height/2 - 1 + 0.5)]$,
- $\boldsymbol{dir} \simeq [-0.4, -0.32, -0.86]$ dla $(fx, fy) = [(width - 1 + 0.5), (height - 1 + 0.5)]$.

6. Ćwiczenie - Główna pętla ray tracera

Zaimplementować w *main()* obliczanie kierunku promienia pierwotnego dla promieni przechodzących przez wszystkie piksele obrazu:

```
int main() {
  ...
  for(int j = 0; j < scene.cam.height; j++) {
    for(int i = 0; i < scene.cam.width; i++) {

            // position of the image point
            float fx = (float)i + 0.5f;
            float fy = (float)j + 0.5f;
            ...
            // computes ray.pos and ray.dir
            ...
```

```
    }
  }
  ...
};
```

Dla parametrów kamery:

$$\boldsymbol{eyep} = [0,0,10], \boldsymbol{lookp} = [0,0,0], \boldsymbol{up} = [0,1,0], fov = 50, width = 500, height = 400. \tag{1.18}$$

obliczyć kierunki wektorów pierwotnych. Obliczone wartości kierunku promieni przedstawić w formie trzech obrazów graficznych niezależnie dla współrzędnych x, y i z. Dane obrazów zapisać do trzech plików dyskowych stosując konwersję kierunku promienia do wartości kanału koloru:

- dla $dir.x$:

```
glm::vec3 rgb(0.0f,0.0f,0.0f);
rgb.x = (ray.dir.x + 1.0f)/2.0f; // conversion from <-1,1> to <0,1>
image.setPixel(i, j, rgb);
```

- dla $dir.y$:

```
glm::vec3 rgb(0.0f,0.0f,0.0f);
rgb.y = (ray.dir.y + 1.0f)/2.0f;
image.setPixel(i, j, rgb);
```

- dla $dir.z$:

```
glm::vec3 rgb(0.0f,0.0f,0.0f);
rgb.z = (ray.dir.z + 1.0f)/2.0f;
image.setPixel(i, j, rgb);
```

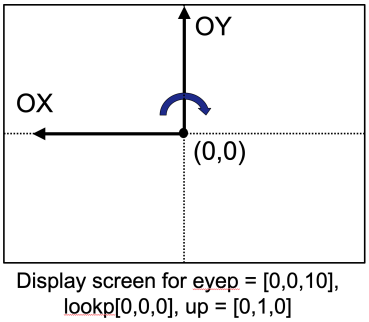
Obrazy obliczone z wykorzystaniem konwersji kierunku promienia pierwotnego do wartości współrzędnej koloru prezentowane są na Rys.1.6. Rezultatem ćwiczenia powinno być uzyskanie takich samym obrazów. Z gradacji odcienia koloru można odczytać orientację układu współrzędnych prezentowaną na Rys./reffig:coordinates-screen.

OX OY OZ

Rysunek 1.6: Graficzna reprezentacja kierunków promieni pierwotnych dla współrzędnych x, y, i z. Zmiana nasycenia kolorów odpowiada zmianie kierunku promieni. Współrzędna z przyjmuje niewielkie i takie same wartości dla wszystkich pikseli obrazu.

1.3 Widoczność obiektów

Cel zajęć: Zapoznanie się algorytmem obliczającym, jakie obiekty na scenie znajdują się w polu widzenia kamery oraz jakie jest wzajemne przesłanianie się obiektów. Wprowadzenie reprezentacji obiektu o kształcie kuli oraz sposobu umieszczania obiektów na scenie.



Rysunek 1.7: Wartości współrzędnych na ekranie dla bazowych parametrów kamery. Oś OZ skierowana jest prostopadle do ekranu w stronę obserwatora.

1.3.1 Reprezentacja kuli

W ray tracingu stosuje się analityczne reprezentacje obiektów geometrycznych, tzn. obiekty mogą być definiowane za pomocą równań matematycznych, dla których podaje się parametry określające rozmiary obiektu. W przypadku **kuli** parametrami będą położenie jej środka oraz promień:

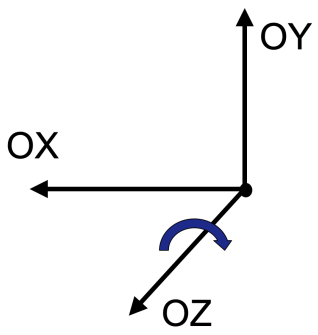
```
class CSphere : public CObject {
public:
    float r; // sphere radius
    glm::vec3 pos; // position of the sphere center
    ...
};
```

1.3.2 Reprezentacja trójkąta

Parametrami **trójkąta** określającymi jego kształt i położenie jest położenie trzech wierzchołków trójkąta:

```
class CTriangle : public CObject {
public:
    glm::vec3 v0;
    glm::vec3 v1;
    glm::vec3 v2;
    ...
};
```

Ważnym elementem jest kolejność definiowania wierzchołków. Zależy ona od przyjętej orientacji układu współrzędnych. W tej implementacji wykorzystywany jest **lewoskrętny układ współrzędnych** prezentowany na Rys. 1.8. W takim układzie kolejność wierzchołków powinna być podawana zgodnie z kierunkiem wskazówek zegara.



Rysunek 1.8: Orientacja osi w układzie współrzędnych.

1.3.3 Lista obiektów

W czasie renderowania obrazu algorytm ray tracingu przegląda obiekty znajdujące się na scenie, dlatego powinna zostać utworzona lista tych obiektów. W przykładowym ray tracerze utworzony został wektor ze wskaźnikami do obiektów dodanych do sceny:

```
std::vector<CObject*> objectList; // list of objects defined in the scene
```

Wykorzystano mechanizm polimorfizmu, który umożliwia definiowanie różnych klas obiektów (np. `CSphere`, `CTriangle`) i dodawanie ich wskaźników do `CScene::objectList`. Każda z tych klas dziedziczy klasę `CObject`. Opisany mechanizm zapewnia, że po odczytaniu wskaźnika z listy automatycznie wywołana będzie metoda klasy konkretnego obiektu, a nie klasy `CObject`. Przykład utworzenia obiektu i dodania go do listy:

```
CSphere* sphere = new CSphere({-2.5,1.3,-3}, 1.0);
objectList.push_back(sphere);
```

7. Ćwiczenie - Tworzenie obiektu o kształcie kuli

W metodzie `CScene:create()` utworzyć dwie kule o poniższych parametrach:

$$\boldsymbol{pos} = [-2.5, 1.3, -3], r = 1, \tag{1.19}$$

$$\boldsymbol{pos} = [0, 0, 0], r = 1.6. \tag{1.20}$$

Kule dodać do listy obiektów (`CScene::objectList`).

1.3.4 Obliczanie przecięcia promienia z kulą

Problem znalezienia trafienia promienia w kulę można sprowadzić do znalezienia punktów wspólnych promienia i kuli. Takich punktów może być dwa, jeden lub promień może nie trafić w kulę.

Promień jest wektorem o początku w $\boldsymbol{ray.pos}$ i o kierunku $\boldsymbol{ray.dir}$. Równanie parametryczne wektora zapisywane jest w postaci:

$$x = ray.pos.x + t \cdot ray.dir.x \tag{1.21}$$

$$y = ray.pos.y + t \cdot ray.dir.y \tag{1.22}$$

$$z = ray.pos.z + t \cdot ray.dir.z \tag{1.23}$$

$$lub \tag{1.24}$$

$$\boldsymbol{w} = \boldsymbol{ray.pos} + t \cdot \boldsymbol{ray.dir}, \tag{1.25}$$

gdzie $\boldsymbol{w} = x, y, z$.

Równanie kuli zapisujemy w postaci uwikłanej:

$$||\boldsymbol{w} - \boldsymbol{obj.pos}||^2 = obj.r^2. \tag{1.26}$$

Zastępując w tym równaniu $\boldsymbol{w}$ wyrażeniem $\boldsymbol{ray.pos} + t \cdot \boldsymbol{ray.dir}$ otrzymujemy równanie kwadratowe:

$$A \cdot t^2 + B \cdot t + C = 0, \tag{1.27}$$

gdzie

$$A = \boldsymbol{ray.dir}^2 = \boldsymbol{ray.dir} \bullet \boldsymbol{ray.dir}, \tag{1.28}$$

$$B = (2 \cdot \boldsymbol{v} \bullet \boldsymbol{ray.dir}), \tag{1.29}$$

$$C = \boldsymbol{v}^2 - obj.r^2 = \boldsymbol{v} \bullet \boldsymbol{v} - obj.r^2, \tag{1.30}$$

gdzie

$$\boldsymbol{v} = \boldsymbol{ray.pos} - \boldsymbol{obj.pos}. \tag{1.31}$$

Równanie rozwiązuje się szukając **najmniejszej ale większej od zera** wartości t:

$$\Delta = B^2 - 4 \cdot A \cdot C, \tag{1.32}$$

$$t = \frac{-B \pm \sqrt{\Delta}}{2 \cdot A}. \tag{1.33}$$

8. Ćwiczenie - Przecięcie promienia z kulą

Zaimplementować obliczanie najmniejszego i większego od zera parametru t w metodzie `CSphere::intersect()`. W przypadku braku przecięcia metoda powinna zwracać ujemną wartość t.

Poprawność działania metody `CSphere::intersect()` przetestować dla przykładu z kulą położoną w punkcie $(0, 0, 0)$ o promieniu $r = 5$, dla promienia w początku w punkcie $(0, 0, 10)$ i kierunku $(0.3, 0.3, -1)$. Dla wymienionych danych wejściowych wartość $t \simeq 5.6$.

1.3.5 Obliczanie widoczności obiektów

W algorytmie ray tracingu przez każdy punkt obrazu wysyłany jest promień pierwotny o kierunku definiowanym przez parametry kamery. Dla każdego z tych promieni trzeba sprawdzić czy trafia on w któryś z obiektów znajdujących się na scenie. W przypadku trafienia punkt obrazu przyjmie kolor, na którego wpływ będzie miał kolor trafionego obiektu. W przypadku braku trafienia kolor punktu będzie równy kolorowi tła.

Dla każdego promienia trzeba wykonać test trafienia tego promienia w każdy z obiektów na scenie, a następnie wybrać ten obiekt, który znajduje się najbliżej położenia kamery. Taki algorytm zapewnia prawidłowe obliczanie **wzajemnego przesłaniania się obiektów**, tzn. fragmenty obiektów znajdujące się bliżej kamery powinny przesłaniać fragmenty bardziej oddalone.

9. Ćwiczenie - Obliczanie widoczności obiektów

W metodzie `CRayTrace::rayTrace()` zaimplementować algorytm, który będzie wywoływał metodę `CSphere::intersect()` dla każdego obiektu z listy `CScene::objectList`. Następnie wyznaczy najmniejszą ale większą od zera wartość parametru t. W przypadku braku trafienia metoda `CRayTrace::rayTrace()` ma zwrócić kolor tła (np. kolor $[0, 0, 0]$). W przypadku trafienia w któryś z obiektów kolor $[0, 0.5, 0]$. Metodę `CRayTrace::rayTrace()` wywołać dla każdego piksela obrazu. Wynik zapisać do pliku z danymi obrazu.

Kod ray tracera w funkcji `main()` powinien przyjąć postać:

```
int main() {
  ...
  CImage image(scene.cam.width, scene.cam.height);

  // computes projection matrix in CRayTrace::compPrimaryRayMatrix()
  ...
  for(int j = 0; j < scene.cam.height; j++) {
    for(int i = 0; i < scene.cam.width; i++) {

            // position of the image point
            float fx = (float)i + 0.5f;
            float fy = (float)j + 0.5f;

            // computes ray.pos and normalized ray.dir using projection matrix
            ...

            // sets background color
            results.col = {0,0,0};

            // ray tracing
            rt.rayTrace(scene, ray, results);

            // handles pixel over-saturation
            if(results.col.x > 1 || results.col.y > 1 || results.col.z > 1) {
                results.col = {1,1,1};
            }

            // writes pixel to output image
            image.setPixel(i, j, results.col);
    }
  }
  // writes image to disk file
  image.save("output_image.png", false);
}
```

Pseudokod obliczania najbliższego przecięcia z obiektem w metodzie `CRayTrace::rayTrace()`:

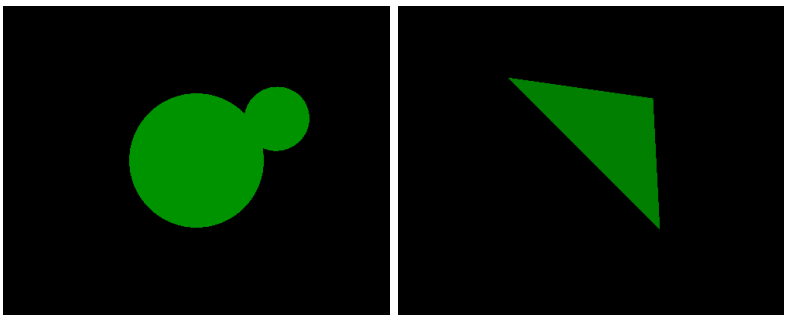
```
bool CRayTrace::rayTrace(CScene scene, CRay &ray, COutput& res) {

    CObject* obj;
    tmin = MAX_FLOAT
    EPS = 0.0001
    is_intersection = false
    // for each object in CScene::objectList {
  t = obj->intersect(ray)
  if(t > EPS && t < tmin) {
      tmin = t
            is_intersection = true
        }
    }
    if(is_intersection == true)
  res.col = [0,0.5,0]
    else
  res.col = [0,0,0]
}
```

W wyniku działania zaimplementowanego moduły ray tracera powinien zostać wygenerowany obraz prezentowany jest na Rys. 1.9 (po lewej). Obraz został wyrenderowany dla sceny składającej się z dwóch kul o parametrach prezentowanych w Tab.1.1.

Tabela 1.1: Parametry sceny (sphere-intersection)

Camera	$width = 500$	$height = 400$	$\boldsymbol{eyep} = [0, 0, 10]$	$\boldsymbol{lookp} = [0, 0, 0]$	$\boldsymbol{up} = [0, 1, 0]$	$fov = 50$
Sphere	$\boldsymbol{pos} = [-2.5, 1.3, -3]$	r=1				
Sphere	$\boldsymbol{pos} = [0, 0, 0]$	r=1.6				



Rysunek 1.9: Kule widoczne w polu widzenia kamery po wykonaniu testów przecięcia promieni pierwotnych z kulami.

1.3.6 Obliczanie przecięcia promienia z trójkątem

Problem znalezienia trafienia promienia w trójkąt można sprowadzić do znalezienia punktu wspólnego wektora oraz płaszczyzny, na której leży trójkąt. Następnie sprawdzane jest czy punkt wspólny leży wewnątrz lub na brzegu trójkąta. Liczenie przecięć z trójkątami jest wielokrotnie powtarzaną funkcją, której szybkość działania ma znaczący wpływ na czas renderingu. Dlatego stosowane są do tego celu zoptymalizowane algorytmy wykorzystujące układ barycentryczny.

10. Ćwiczenie - Przecięcie promienia z trójkątem

Zaimplementować obliczanie punktu przecięcia promienia z trójkątem w metodzie `CTriangle::intersect()`. W przypadku braku przecięcia metoda powinna zwracać ujemną wartość t. Wykorzystać funkcje z biblioteki GLM:

- **glm::intersectRayTriangle()** do sprawdzenia czy promień przecina się z trójkątem,

- **glm::intersectRayPlane()** do obliczenia odległości t od położenia kamery do punktu przecięcia.

Parametrem jednej z ww. funkcji jest wektor normalnego do powierzchni trójkąta. Sposób obliczania tego wektora opisany jest w Podroz.1.4.3. W wyniku działania zaimplementowanego moduły ray tracera powinien zostać wygenerowany obraz prezentowany jest na Rys. 1.9 (po prawej). Obraz został wyrenderowany dla sceny składającej się z jednego trójkąta o parametrach prezentowanych w Tab.1.2.

Alternatywnie można wykorzystać położenie punktu przecięcia obliczane w układzie barycentrycznym przez funkcję glm::intersectRayTriangle() . Trzeba to położenie wyrazić w układzie kartezjańskim, a następnie obliczyć odległości od położenia kamery do punktu przecięcia.

Tabela 1.2: Parametry sceny (triangle-intersection)

Camera	$width = 500$	$height = 400$	$\boldsymbol{eyep} = [0, 0, 10]$	$\boldsymbol{lookp} = [0, 0, 0]$	$\boldsymbol{up} = [0, 1, 0]$	$fov = 50$
Triangle	$\boldsymbol{v0} = [3, 3, -5]$	$\boldsymbol{v1} = [-3, 3, -10]$	$\boldsymbol{v2} = [-3, -3, -8]$			

1.4 Model oświetlenia lokalnego

Cel zajęć: Zapoznanie się sposobem obliczania kolorów obiektów opartym na modelu oświetlenia Blinna-Phona. Definicje źródeł światła oraz materiałów obiektów. Pojęcie tekstury oraz mapowania tekstury.

1.4.1 Materiał obiektu

Wygląd powierzchni obiektu zależy od tego, z jakiego materiału zrobiona jest ta powierzchnia. Algorytmy syntezy obrazów wykorzystują modele materiałów, a konkretniej **modele określające w jaki sposób światło odbija się od powierzchni**. Dany model można parametryzować, tzn. przypisując mu określone wartości parametrów, zmienia się np. to czy powierzchnia mam być metaliczna czy ma rozpraszać światło tak jak np. plastik.

W ray tracingu stosuje się model powierzchni oparty na definiowaniu parametrów:

- **matDiffuse** rozpraszania światła,
- **matSpecular i matShininess** kierunkowego rozpraszania światła,
- **matAmbient** określającego jak powierzchnia reaguje ze światłem pochodzącym nie bezpośrednio od źródeł światła, ale przychodzącym od innych obiektów, które odbijają światło,
- **isTexture** oraz **texture** wskazujący odpowiednio na fakt mapowania tekstury na powierzchnięobiketu oraz na dane tej tekstury.
- **reflectance** określającego czy i w jakim stopniu powierzchnia odbija światło w sposób kierunkowy,
- **transmitance** definiującego stopień przezroczystości powierzchni.

Parametry materiału zdefiniowane są w klasie `CObject`, którą dziedziczą pozostałe klasy definiujące różne rodzaje obiektów:

```
class CObject {
public:
    glm::vec3 matAmbient;
    glm::vec3 matDiffuse;
    glm::vec3 matSpecular;
    float matShininess;
    bool isTexture;
    CImage texture;
    float reflectance;
    float transmitance;
...
};
```

Parametry `matAmbient`, `matDiffuse` i `matSpecular` przyjmują wartości w zakresie <0,1>. `matShininess` może przyjmować wartości większe bądź równe jedności. Parametry `reflectance` oraz `transmitance` zostaną omówione w dalszej części opracowania.

1.4.2 Źródła światła

Źródła światła to obiekty na scenie, które emitują światło. Nie mają one swojej fizycznej reprezentacji na obrazie, tzn. że źródło światła nie będzie widoczne na renderingu. Będzie natomiast emitowało energię, która po odbiciu od powierzchni obiektów trafia do kamery skutkiem czego na obrazie widoczne są obiekty.

Na potrzeby tego opracowania wykorzystany zostanie najprostszy model źródła światła, tzw. **punktowe źródło światła**. Tego typu światło ma swoje położenie na scenie $\boldsymbol{L_{pos}}$ i emituje energię we wszystkich kierunkach z jednakowym natężeniem, tzn. świeci światłem o kolorze $\boldsymbol{L_{col}}$.

Renderowanie kolorowych obrazów wymaga uwzględnienia kolorów w definicji źródła światła, dlatego punktowym światło będzie przyporządkowany kolor $\boldsymbol{RGB}$, który decyduje zarówno o odcieniu i nasyceniu, jak i natężeniu emitowanego światła.

Parametry źródła światła zdefiniowane są w klasie `CLight`:

```
class CLight {
public:
    glm::vec3 pos; // light position
    glm::vec3 color; // light color
...
};
```

Do sceny musi zostać dodane co najmniej jedno źródło światła, aby na obrazie widoczne były obiekty. Światło dodajemy definiując w metodzie `CScene::create()` obiekt klasy `CLight`, nadając parametry temu obiektowi, a następnie dodając ten obiekt do listy `CScene::lightList`, np.:

```
    CLight light1(glm::vec3(-3,-2,8));
    light1.color = {0.6,0.6,0.6};
    lightList.push_back(light1);
```

1.4.3 Wektor normalny

Wektor normalny to wektor prostopadły do powierzchni obiektu. W przypadku kuli kierunek wektora normalnego wyznaczany jest przez znormalizowaną różnicę położenia punktu na powierzchni kuli oraz położenia środka kuli:

$$\boldsymbol{n} = normalize(\boldsymbol{p} - \boldsymbol{sphere.pos}). \tag{1.34}$$

(1.35)

Punkt $\boldsymbol{p}$ to punkt przecięcia promienia z powierzchnią obiektu położoną najbliżej w stosunku do kamery (tzn. o najmniejszym ale większym od zera t). Położenie tego puntu $\boldsymbol{p}(x, y, z)$ można obliczyć za pomocą równań:

$$x = ray.pos.x + t_{min} \cdot ray.dir.x \tag{1.36}$$
$$y = ray.pos.y + t_{min} \cdot ray.dir.y \tag{1.37}$$
$$z = ray.pos.z + t_{min} \cdot ray.dir.z. \tag{1.38}$$

W przypadku trójkąta wektor prostopadły do jego powierzchni obliczamy ze wzorów:

$$\boldsymbol{n} = normalize(\boldsymbol{u} \times \boldsymbol{v}), \tag{1.39}$$
$$\boldsymbol{u} = \boldsymbol{v_1} - \boldsymbol{v_0}, \tag{1.40}$$
$$\boldsymbol{v} = \boldsymbol{v_2} - \boldsymbol{v_0}, \tag{1.41}$$

gdzie $\boldsymbol{n}$ to wektor normalny, $\boldsymbol{v_0}$, $\boldsymbol{v_1}$ i $\boldsymbol{v_2}$ to położenie wierzchołków trójkąta.

Wektory normalne obliczane w punkcie przecięcia promienia z powierzchnią obiektu $\boldsymbol{p}$ wykorzystane są do obliczania oświetlenia.

11. Ćwiczenie - Wektor normalny

Zaimplementować obliczanie wektorów normalnych dla kuli i trójkąta odpowiednio w metodach `CSphere::normal()` oraz `CTriangle::normal()`. Parametrem wejściowym do tych metod jest położenie punktu przecięcia na powierzchni kuli i trójkąta. W drugim przypadku wspomniany parametr nie jest wymagany, ponieważ kierunek wektora normalnego będzie taki sam dla każdego punktu na trójkącie. Stosuje się go w celu zachowania takiej samej struktury kodu.

1.4.4 Równanie oświetlenia

Każdemu promieniowi trafiającemu w powierzchnię obiektu przyporządkowywana jest **energia świetlna**, która pochodzi ze źródła światła i odbija się w kierunku biegu promienia, tzn. w kierunku kamery. Wspomniana energia wyrażana jest w postaci koloru o trzech składowych R, G i B. Nie każdy obiekt jest lustrem odbijającym światło w sposób kierunkowy, dlatego w kierunku kamery zostanie wysłana część energii ze źródła światła, która została w złożony sposób odbita od powierzchni, podlegając również częściowej absorpcji. O sposobie obliczania tej energii decyduje przyjęty **model oświetlenia**, tzn. model, który w analityczny sposób opisuje rozchodzenie się światła na scenie.

W ray tracerach powszechnie wykorzystuje się model oświetlenia Blinna-Phonga, w którym sumowane są ze sobą komponenty oświetlenia:

- **diffuse** - światło rozpraszane ($\boldsymbol{m_{dif}} = [r, g, b]$),
- **specular** - światło rozpraszane kierunkowo ($\boldsymbol{m_{spec}} = [r, g, b]$),
- **ambient** - światło otoczenia pochodzące z wielokrotnego odbijania się promieni światła od obiektów ($\boldsymbol{m_{amb}} = [r, g, b]$).

Każdy z komponentów jest wyrażony wektorem koloru obliczanym w równaniu oświetlenia:

$$\boldsymbol{I} = \sum_{k=1}^{N} (\boldsymbol{I_{amb}}^k + \boldsymbol{I_{dif}}^k + \boldsymbol{I_{spec}}^k), \tag{1.42}$$

$$\boldsymbol{I_{amb}} = \boldsymbol{m_{amb}} \cdot \boldsymbol{L_{col}}, \tag{1.43}$$

$$\boldsymbol{I_{dif}} = \boldsymbol{m_{dif}} \cdot \boldsymbol{L_{col}} \cdot (\boldsymbol{L} \bullet \boldsymbol{n}), \tag{1.44}$$

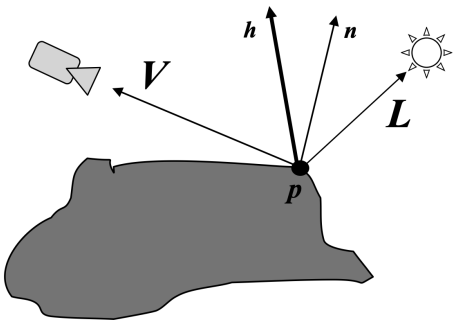
$$\boldsymbol{I_{spec}} = \boldsymbol{m_{spec}} \cdot \boldsymbol{L_{col}} \cdot (\boldsymbol{n} \bullet \boldsymbol{h})^{m_{shin}}, \tag{1.45}$$

$$\boldsymbol{h} = \frac{\boldsymbol{L} + \boldsymbol{V}}{|\boldsymbol{L} + \boldsymbol{V}|}, \tag{1.46}$$

$$\boldsymbol{L} = normalize(\boldsymbol{L_{pos}} - \boldsymbol{p}), \tag{1.47}$$

$$\boldsymbol{V} = -\boldsymbol{ray.dir}. \tag{1.48}$$

$\boldsymbol{I}$ to kolor obliczany niezależnie dla składowych r, g i b. $\boldsymbol{L_{col}} = [r, g, b]$ wyraża kolor źródła światła (patrz Rozdz. 1.4.2), natomiast $\boldsymbol{m_{amb}}$, $\boldsymbol{m_{dif}}$, $\boldsymbol{m_{spec}}$ i $\boldsymbol{m_{shin}}$ to parametry materiału powierzchni (patrz Rozdz. 1.4.1). N wyraża liczbę źródeł światła umieszczonych na scenie. Znaczenie wektorów zawartych w równaniach ilustruje Rys. 1.10.



Rysunek 1.10: Parametry równania oświetlenia. p to punkt przecięcia promienia z powierzchnią obiektu.

12. Ćwiczenie - Równanie oświetlenia

Dodać do sceny w metodzie `CScene.create()` obiekt źródła światła zgodnie przykładem podanym w Podroz.1.4.2. Do obliczania kolorów wykorzystana zostanie również metoda obliczająca wektor normalny z Podroz.1.4.3.

Zaimplementować równanie oświetlania w metodzie `CRayTrace::rayTrace()`. Wyrenderować obrazy, których wzorcowe przykłady prezentowane są na Rys. 1.11:

- Obraz z wykorzystaniem wyłącznie komponentu **ambient** (parametry sceny znajdują się w Tab. 1.3).
- Obraz z wykorzystaniem wyłącznie komponentu **diffuse** (parametry sceny znajdują się w Tab. 1.4).
- Obraz z wykorzystaniem wyłącznie komponentu **specular** (parametry sceny znajdują się w Tab. 1.5).
- Obraz z wykorzystaniem pełnego modelu Blinna-Phonga (parametry sceny znajdują się w Tab. 1.6). Sprawdzić jaki wpływ na wygląd powierzchni ma zmiana wartości parametrów m_{shi} (patrz Rys. 1.11, parametry sceny znajdują się w Tab. 1.6). W parametrach z tabeli trzeba zmienić wartości m_{shi} na 10 oraz 50.

Poprawna struktura obliczanie poszczególnych komponentów prezentowana jest na poniższym pseudo-kodzie:

```
// main.cpp
...
out.col = color_background
rayTrace(scene, ray, out)
...

// CRayTrace.cpp
CRayTrace::rayTrace(CScene scene, CRay& ray, COutput& out) {
  ...
  // compute the closest intersection, store distance (t_min) and reference to
       // closest object (hit_obj)

  if no intersection then return

  // compute position of the intersection point p using the ray equation
  p = ray.pos + t_min * ray.dir

  for each light source {

    // compute and add ambient color component
    col = col + light.color * hit_obj.m_ambient

    // compute normal vector at p
    n = hit_obj.normal(p)

    // compute vector from p to light position
    L = normalize(light.pos - p)

    // compute angle between n and vector to light
    cos_angle = dot(n, L)

    if cos_angle > 0.001 {
        // compute and add diffuse color component
        col = col + light.color * hit_obj.m_diffuse * cos_angle

        // compute half angle vector h
        h = normalize(L + V), V = -ray.dir

        // compute angle between n and h
        cos_beta = dot(n, h)

        if(cos_beta > 0.001 {
            // compute and add specular color component
            col = col + light.color * hit_obj.m_spoecular * cos_beta^hit_obj.shininesse
        }
    }
  }
}
```

Tabela 1.3: Parametry sceny (illumination-ambient)

Camera	$width = 500$	$height = 400$	$\boldsymbol{eyep} = [0, 0, 10]$	$\boldsymbol{lookp} = [0, 0, 0]$	$\boldsymbol{up} = [0, 1, 0]$	$fov = 50$
Light source	$\boldsymbol{pos} = [-3, -2, 8]$	$\boldsymbol{color} = [0.6, 0.6, 0.6]$				
Sphere	$\boldsymbol{pos} = [0, 0, 0]$ $\boldsymbol{m_{amb}} = [0.5, 0, 0]$	r=1.6 $\boldsymbol{m_{dif}} = [0, 0, 0]$	$\boldsymbol{m_{spec}} = [0, 0, 0]$	shininese=30	reflectance=0	isTexture=0

Tabela 1.4: Parametry sceny (illumination-diffuse)

Camera	$width = 500$	$height = 400$	$\boldsymbol{eyep} = [0, 0, 10]$	$\boldsymbol{lookp} = [0, 0, 0]$	$\boldsymbol{up} = [0, 1, 0]$	$fov = 50$
Light source	$\boldsymbol{pos} = [-3, -2, 8]$	$\boldsymbol{color} = [0.6, 0.6, 0.6]$				
Sphere	$\boldsymbol{pos} = [0, 0, 0]$ $\boldsymbol{m_{amb}} = [0, 0, 0]$	r=1.6 $\boldsymbol{m_{dif}} = [0.7, 0, 0]$	$\boldsymbol{m_{spec}} = [0, 0, 0]$	shininese=0	reflectance=0	isTexture=0

Tabela 1.5: Parametry sceny (illumination-specular)

Camera	$width = 500$	$height = 400$	$\boldsymbol{eyep} = [0, 0, 10]$	$\boldsymbol{lookp} = [0, 0, 0]$	$\boldsymbol{up} = [0, 1, 0]$	$fov = 50$
Light source	$\boldsymbol{pos} = [-3, -2, 8]$	$\boldsymbol{color} = [0.6, 0.6, 0.6]$				
Sphere	$\boldsymbol{pos} = [0, 0, 0]$ $\boldsymbol{m_{amb}} = [0, 0, 0]$	r=1.6 $\boldsymbol{m_{dif}} = [0, 0, 0]$	$\boldsymbol{m_{spec}} = [0.7, 0.7, 0.7]$	shininese=30	reflectance=0	

1.4.5 Teksturowanie

Teksturowanie w grafice komputerowej to proces przypisywania tekstury (czyli obrazu lub wzoru) na powierzchnię trójwymiarowego modelu, który ma symulować właściwości powierzchni danego obiektu. Tekstury wykorzystywane są, aby nadać obiektom w grafice komputerowej bardziej realistyczny i szczegółowy wygląd, uwzględniający różne rodzaje faktur, kolorów, cieni i odcieni.

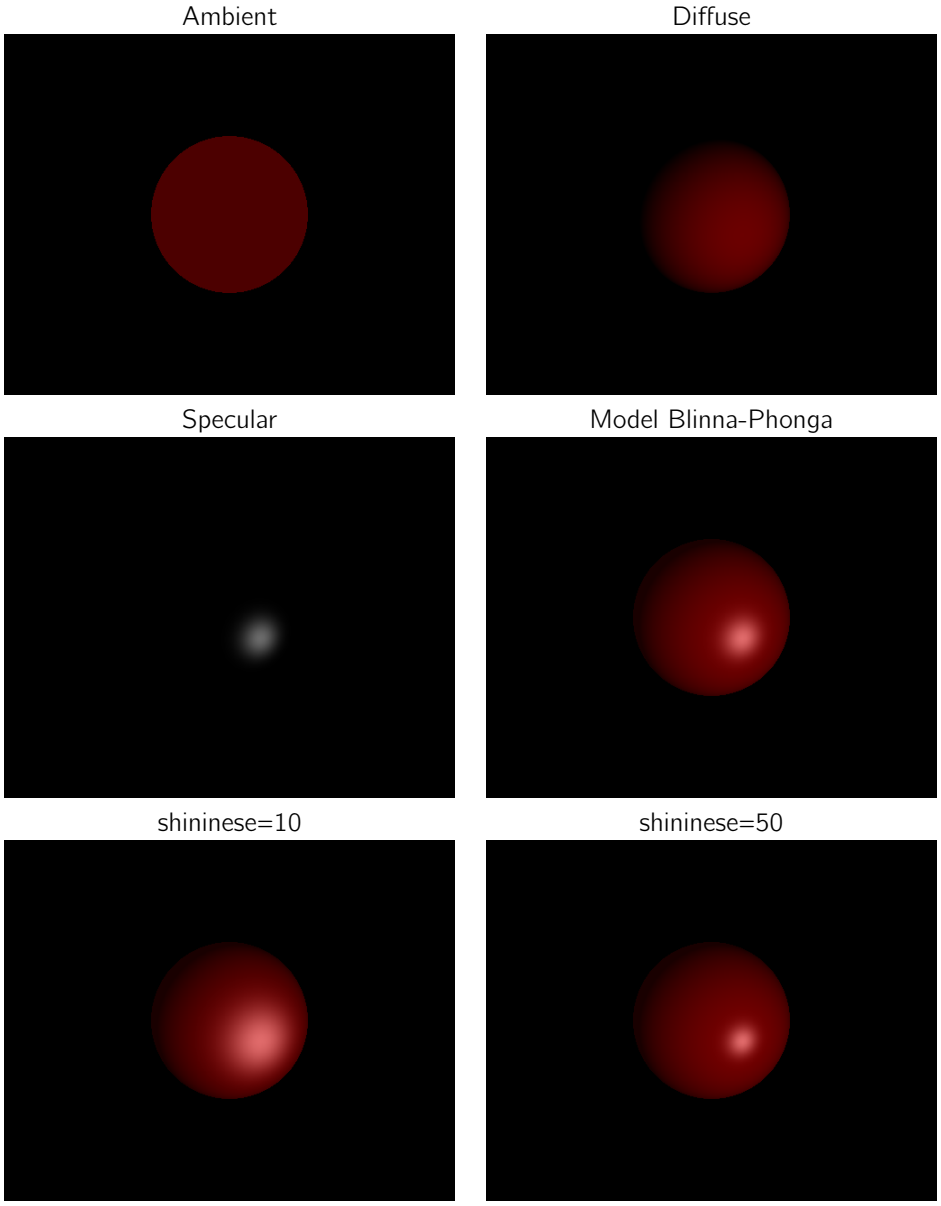
Proces nakładania tekstury na powierzchnię obiektu można podzielić na etapy:

1. Zdefiniowanie tektury w postaci obrazu rastrowego i przypisanie tej tektury do obiektu.
2. Po obliczeniu koloru powierzchni w punkcie przecięcia:
   (a) Obliczenie współrzędnych (u, v) punktu przecięcia w układzie współrzędnych związanych z teksturą, tzn. obliczenie **mapowania tekstury**.
   (b) Pobranie koloru teksela o współrzędnych (u, v).
   (c) Zmodyfikowanie koloru powierzchni poprzez wymnożenie koloru obliczonego w równaniu oświetlenia i koloru teksela.

Na potrzeby projektu zaimplementowane zostanie **sferyczne mapowanie tekstury** (ang. spherical texture mapping) polegające na przypisaniu tekstury na powierzchnię sfery, która następnie zostaje "rozwinięta" i "rzucona" na trójwymiarowy model obiektu, w taki sposób, aby tekstura zachowała odpowiedni kształt i proporcje na powierzchni obiektu (patrz Rys.1.12). Sferyczne mapowanie tekstury jest szczególnie przydatne dla obiektów, które posiadają kulisty lub niemal kulisty kształt, takich jak planety, kule, a także np. postacie w grach komputerowych. Dzięki takiemu mapowaniu tekstury można uzyskać płynne i realistyczne przejścia między różnymi częściami powierzchni obiektu, co przyczynia się do uzyskania bardziej naturalnego i estetycznego wyglądu wizualizacji 3D.

Tabela 1.6: Parametry sceny (illumination)

Camera	$width = 500$	$height = 400$	$\boldsymbol{eyep} = [0, 0, 10]$	$\boldsymbol{lookp} = [0, 0, 0]$	$\boldsymbol{up} = [0, 1, 0]$	$fov = 50$
Light source	$\boldsymbol{pos} = [-3, -2, 8]$	$\boldsymbol{color} = [0.6, 0.6, 0.6]$				
Sphere	$\boldsymbol{pos} = [0, 0, 0]$ $\boldsymbol{m_{amb}} = [0.1, 0, 0]$	r=1.6 $\boldsymbol{m_{dif}} = [0.6, 0, 0]$	$\boldsymbol{m_{spec}} = [0.7, 0.7, 0.7]$	shininese=30	reflectance=0	



Rysunek 1.11: Rendering obrazów z różnymi komponentami modelu oświetlania Blinna-Phonga.

Do obliczenia współrzędnych (u, v) tekstury dla mapowania sferycznego można wykorzystać równania:

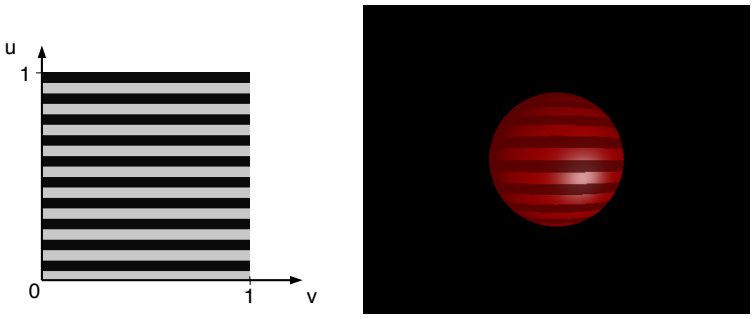
$$u = 0.5 + arcsin(n_x)/\pi, \tag{1.49}$$

$$v = 0.5 - arcsin(n_y)/\pi, \tag{1.50}$$

gdzie n_x i n_y to współrzędne kierunku znormalizowanego wektora normalnego ($\boldsymbol{n} = [n_x, n_y, n_z]$) do powierzchni kuli w punkcie przecięcia.

13. Ćwiczenie - Teksturowanie

W celu nałożenia tekstury na powierzchnię kuli:



Rysunek 1.12: Po lewej tekstura wygenerowana za pomocą metody `CImage::createTexture()`. Po prawej tekstura nałożona na czerwoną kulę.

1. Zdefiniować scenę wykorzystując parametry z Tab.1.7. Dodatkowo definiując kulę należy utworzyć teksturę bitmapową za pomocą metody `CImage::createTexture`, a następnie przypisać ją do pola `CSphere::texture` tworzonego obiektu `CSphere`:

```
CSphere sphere;
...
sphere->isTexture = true;
sphere->texture = CImage::createTexture(400,400);
objectList.push_back(sphere)
```

2. W metodzie `CSpehere::textureMapping` zaimplementować technikę sferycznego mapowanie tekstury.

3. Wykorzystać `CSpehere::textureMapping` w metodzie `CRayTrace::rayTrace` do obliczenia współrzędnych (u, v) teksela tekstury odpowiadającego położeniu punktu przecięcia na powierzchni kuli.

4. Wartość koloru teksela pobraną za pomocą metody `CImage::getTexel` użyć do zmodyfikowania koloru promienia obliczonego w poprzednim kroku na podstawie równania oświetlenia:

```
...
if(hit_obj->isTexture) {
    glm::vec2 uv = hit_obj->textureMapping(n);
    glm::vec3 tex_col = CImage::getTexel(hit_obj->texture, uv.x, uv.y);
    out.col = out.col * tex_col;
}
...
```

5. Wyrenderować obraz dla parametrów sceny z Tab.1.7 (patrz Rys.1.12).

Tabela 1.7: Parametry sceny (texture)

Camera	$width = 500$	$height = 400$	$\boldsymbol{eyep} = [0, 0, 10]$	$\boldsymbol{lookp} = [0, 0, 0]$	$\boldsymbol{up} = [0, 1, 0]$	$fov = 50$
Light source	$\boldsymbol{pos} = [-3, -2, 8]$	$\boldsymbol{color} = [0.6, 0.6, 0.6]$				
Sphere	$\boldsymbol{pos} = [0, 0, 0]$	r=1.6				
	$\boldsymbol{m_{amb}} = [0.1, 0, 0]$	$\boldsymbol{m_{dif}} = [0.6, 0, 0]$	$\boldsymbol{m_{spec}} = [0.7, 0.7, 0.7]$	shininese=30	reflectance=0	isTexture=1

1.5 Elementy modelu oświetlenia globalnego

Cel zajęć: Zapoznanie się z pojęciem oświetlenia globalnego. Implementacja generowania cieni oraz materiałów o powierzchnia zwierciadlanych. Dodanie procedury rekurencyjnej do ray tracera.

1.5.1 Globalny model oświetlenia

W opisanym w Rozdz. 1.4 **modelu oświetlenia lokalnego** w obliczeniach kolorów obiektów uwzględniane jest wyłącznie światło emitowane ze źródeł światła i bezpośrednio docierające do obiektów. W przypadku **modelu oświetlenia globalnego** dodatkowo na kolor obiektów ma wpływ światło, które wcześniej zostało odbite od innych obiektów znajdujących się na scenie. Innymi słowy obiekty, same nie emitując światła, wzajemnie się oświetlają odbijając światło pochodzące ze źródeł światła.

Model globalny jest bardziej złożony pod względem obliczeniowym, dlatego w ray tracingu wykorzystywane są tylko jego elementy. Przykładem jest obliczanie cieni, które powstają w wyniku oddziaływania obiektu rzucającego cień na inny obiekt znajdujący się w cieniu. Kolejnym przykładem są powierzchnie zwierciadlane, w których odbijają się inne obiekty, tzn. pozostałe obiekty wpływają na wygląd obiektu o powierzchni zwierciadlanej.

1.5.2 Obliczanie cieni

Po znalezieniu punktu przecięcia obiektu z promieniem pierwotnym (oznaczamy ten punkt symbolem $\boldsymbol{p}$) obliczany jest kolor tego punktu. Jednak wcześniej trzeba sprawdzić czy do wspomnianego punktu bezpośrednio dociera światło ze źródeł światła. Jeżeli na drodze promienia wychodzącego z punktu przecięcia i kończącego się w punkcie położenia źródła światła znajdują się inne obiekty, to punkt przecięcia będzie znajdował się w cieniu z punktu widzenia tego źródła światła. W takim przypadku nie trzeba obliczać koloru, ponieważ dane źródło światła nie będzie miało wpływu na kolor punktu $\boldsymbol{p}$.

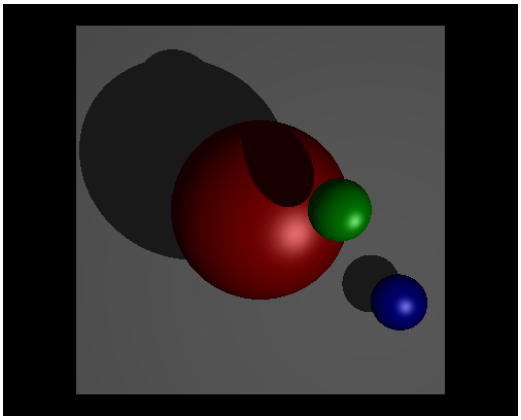
Wyjątkiem jest składowa **ambient** koloru, którą dodaje się do wartości koloru wyjściowego bez względu na to czy punkt przecięcia jest w cieniu. Takie rozwiązanie sprawia, że widoczne są kontury obiektów w zacienionych obszarach sceny.

Aby sprawdzić czy punkt przecięcia znajduje się w cieniu, trzeba dla każdego źródła światła wyznaczyć promień $\boldsymbol{ray_s}$ wychodzący z punktu przecięcia w kierunku źródła światła:

$$\boldsymbol{ray_s.pos} = \boldsymbol{p}, \tag{1.51}$$

$$\boldsymbol{ray_s.dir} = normalize(\boldsymbol{L_{pos}} - \boldsymbol{p}). \tag{1.52}$$

Następnie należy sprawdzić czy promień $\boldsymbol{ray_s}$ na swojej drodze z $\boldsymbol{p}$ do $\boldsymbol{L_{pos}}$ trafia w obiekty na scenie. Implementacja tego algorytmu jest podobna do szukania przecięcia promienia pierwotnego z obiektami na scenie. Różnice polegają na zastąpieniu promienia pierwotnego promieniem $\boldsymbol{ray_s}$ zwanym **shadow ray**. Nie zawsze trzeba również przeszukiwać całej listy obiektów `CScene::objectList`, ponieważ już pierwsze trafienie w obiekt będzie oznaczało, że punkt przecięcia jest w cieniu.



Rysunek 1.13: Przykład obrazu z wyrenderowanymi cieniami.

14. Ćwiczenie - Renderowanie cieni

Zaimplementować obliczanie cieni w metodzie `CRayTrace::rayTrace()`. Przetestować poprawność implementacji renderując obraz sceny o parametrach wymienionych w Tab.1.8. Generowany obraz powinien być taki sam jak obraz prezentowany na Rys.1.13. Struktura metody `CRayTrace::rayTrace()` uwzględniająca obliczanie cieni prezentowana jest na poniższym listingu:

Tabela 1.8: Parametry sceny (shadows)

Camera	$width = 500$	$height = 400$	$\boldsymbol{eyep} = [0, 0, 10]$	$\boldsymbol{lookp} = [0, 0, 0]$	$\boldsymbol{up} = [0, 1, 0]$	$fov = 50$
Light source	$\boldsymbol{pos} = [-3, -2, 8]$	$\boldsymbol{color} = [0.6, 0.6, 0.6]$				
Sphere	$\boldsymbol{pos} = [-1, 0, 3]$ $\boldsymbol{m_{amb}} = [0, 0.1, 0]$	r=0.4 $\boldsymbol{m_{dif}} = [0, 0.6, 0]$	$\boldsymbol{m_{spec}} = [0.7, 0.7, 0.7]$	shininese=30	reflectance=0	
Sphere	$\boldsymbol{pos} = [0, 0, 0]$ $\boldsymbol{m_{amb}} = [0.1, 0, 0]$	r=1.6 $\boldsymbol{m_{dif}} = [0.6, 0, 0]$	$\boldsymbol{m_{spec}} = [0.7, 0.7, 0.7]$	shininese=30	reflectance=0	
Sphere	$\boldsymbol{pos} = [-3, -2, -2]$ $\boldsymbol{m_{amb}} = [0, 0, 0.1]$	r=0.6 $\boldsymbol{m_{dif}} = [0, 0, 0.6]$	$\boldsymbol{m_{spec}} = [0.7, 0.7, 0.7]$	shininese=30	reflectance=0	
Triangle	$\boldsymbol{v0} = [5, 5, -5]$ $\boldsymbol{m_{amb}} = [0.1, 0.1, 0.1]$	$\boldsymbol{v1} = [-5, 5, -5]$ $\boldsymbol{m_{dif}} = [0.4, 0.4, 0.4]$	$\boldsymbol{v2} = [-5, -5, -5]$ $\boldsymbol{m_{spec}} = [0, 0, 0]$	shininese=0	reflectance=0	
Triangle	$\boldsymbol{v0} = [5, 5, -5]$ $\boldsymbol{m_{amb}} = [0.1, 0.1, 0.1]$	$\boldsymbol{v1} = [-5, -5, -5]$ $\boldsymbol{m_{dif}} = [0.4, 0.4, 0.4]$	$\boldsymbol{v2} = [5, -5, -5]$ $\boldsymbol{m_{spec}} = [0, 0, 0]$	shininese=0	reflectance=0	

```
// CRayTrace.cpp
CRayTrace::rayTrace(CScene scene, CRay& ray, COutput& out) {
  ...
  for each light source {

    // compute ambient color component
    col = col + light.color * hit_obj.m_ambient

    // compute normalized shadow ray from the intersection point towards the light source

    // for each object in the scene search for intersection with the shadow ray
        if intersection exists
            break
    if intersection exists
        stop and continue to the next light source

    // compute diffuse and spreculer color components based on illumination equation
    col = col + (diffuse color component)
    col = col + (specular color component)
    ...
  }
}
```

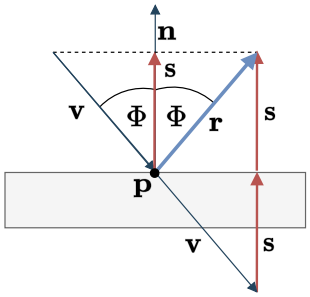
1.5.3 Odbicia zwierciadlane

Na scenie mogą znajdować się obiekty o powierzchniach zwierciadlanych, tzn. takie, w których odbijają się pozostałe elementy sceny. O własnościach odbijania danego obiektu decyduje parametr `CObject::reflectance` przyjmujący wartości 0 - brak odbicia lub 1 oznaczające powierzchnię zwierciadlaną. Odbicia zwierciadlanego nie należy mylić z komponentem **specular** w równaniu oświetlenia, który określa kierunkowe rozproszenie światła.

W ray tracingu powierzchnie zwierciadlane renderuje się poprzez śledzenie **promieni wtórnych** powstających w skutek odbicia promienia pierwotnego od powierzchni obiektu. Przyjmowany jest podstawowy model odbicia, w którym kąt padania promienia równy jest kątowi odbicia. Parametry promienia wtórnego $\boldsymbol{ray_r}$ obliczane są za pomocą równań (patrz Rys.1.14):

$$\begin{aligned} \boldsymbol{ray_r.pos} &= \boldsymbol{p}, \\ \boldsymbol{ray_r.dir} &= normalize(\boldsymbol{r}), \\ \boldsymbol{r} = \boldsymbol{v} + 2 \cdot \boldsymbol{s} = \boldsymbol{v} + 2 \cdot \boldsymbol{n} \cdot \cos\Phi = \boldsymbol{v} + 2 \cdot \boldsymbol{n} \cdot (-\boldsymbol{v} \bullet \boldsymbol{n}) &= \boldsymbol{v} - (2 \cdot \boldsymbol{v} \bullet \boldsymbol{n}) \cdot \boldsymbol{n}, \\ \boldsymbol{v} &= \boldsymbol{ray.dir}, \end{aligned}$$

gdzie $\boldsymbol{p}$ to punkt przecięcia promienia pierwotnego z obiektem, $\boldsymbol{n}$ to wektor normalny do powierzchni obiektu w punkcie przecięcia, a $\boldsymbol{ray.dir}$ to kierunek wektora pierwotnego.



Rysunek 1.14: Obliczanie promienia odbitego od powierzchni.

15. Ćwiczenie - Promień odbity od powierzchni

W klasie `CRayTrace` zaimplementować metodę `CRayTrace::reflectedRay()` obliczającą promień odbity od powierzchni obiektu. Przetestować poprawność implementacji dla poniższych danych:

```
ray.pos = {0,0,0}; // punkt poczatkowy promienia pierwotnego
ray.dir = {0.5f,0.5f,0.0f}; // kierunek promienia pierwotnego
glm::vec3 n(0,1.0,0); // kierunek wektora normalnego
glm::vec3 pos = {0,0,0}; // polozenie punktu przeciecia
CRay ray_reflected = rt.reflectedRay(ray, n, pos);
```

Wynikowy promień odbity powinien mieć parametry w przybliżeniu równe: `ray_reflected.pos = [0,0,0]`, `ray_reflected.dir = [0.71,-0.71,0.0]`.

1.5.4 Rekurencyjne śledzenie promieni

Ray tracing ograniczony do śledzenia promieni pierwotnych nazywany jest określeniem **ray casting**. We właściwym ray tracingu w przypadku trafienia w obiekt o powierzchni odbijającej, tworzony jest **promień wtórny** (patrz Rozdz.1.5.3), do którego śledzenia wykorzystywana jest ta sama metoda co w przypadku promieni pierwotnych. Metoda `CRayTrace::rayTrace()` wywoływana jest **rekurencyjnie**, tzn. w linii znajdującej się wewnątrz tej metody. W przypadku tego wywołania do metody przekazuje się parametry promienia wtórnego.

Należy również we właściwy sposób obliczać wartości kolorów, tzn. wartość koloru jest zerowana przed wywołaniem `CRayTrace::rayTrace()` dla promienia pierwotnego. Następnie wartość ta jest inkrementowana z wykorzystaniem współczynnika energii promienia. Opisany algorytm ilustruje poniższy pseudokod:

```
// main.cpp
for each pixel {
  compute primary_ray
  first_out.col = {0,0,0}; // sets color to backgroud color
  first_out.energy = 1.0f; // sets energy to 1
  first_out.tree = 0; // sets tree counter to 0
  rayTrace(scene, primary_ray, first_out); // calls CRayTrace::rayTrace() for primary ray
}
// CRayTrace.cpp
CRayTrace::rayTrace(CScene scene, CRay& ray, COutput& out) {
  ...
  // compute ambient color component
  col = col + out.energy * light.color * hit_obj.m_ambient

  // check shadow conditions

  // compute diffuse and specular color components based on illumination equation
  //    and using out.energy coefficient
  col = col + out.energy * (diffuse color)
  col = col + out.energy * (specular color)

  if(object_reflectance > 0 and out.tree < MAX_RAY_TREE and out.energy > MIN_RAY_ENERGY) {
    // increments ray tree counter
  out.tree++;
```

```
  // decreases the energy
  out.energy = out.energy * object_reflectance;
  // computes secondary ray
  secondary_ray = reflectedRay(...)
  // calls rayTrace() for secondary ray
  rayTrace(scene, secondary_ray, out)
  }
}
```

W algorytmie wykorzystana jest zmienna `out.tree`, która pełni funkcję licznika liczby odbić. Kolejne odbicie nie będzie generowane, jeżeli licznik przekroczy maksymalną zadaną liczbę odbić `MAX_RAY_TREE` (wartość tej stałej można ustawić na 1). Tworzenie promieni wtórnych jest też blokowane w przypadku, gdy energia promienia spadnie poniżej określonego minimum `MIN_RAY_ENERGY` (wartość tej stałej może wynosić 0.01). Wymienione mechanizmy mają na celu zapobieganie "zapętlaniu się" ray tracera w przypadkach ciągłego trafiania promieni w powierzchnie zwierciadlane.

16. Ćwiczenie - Rekurencja

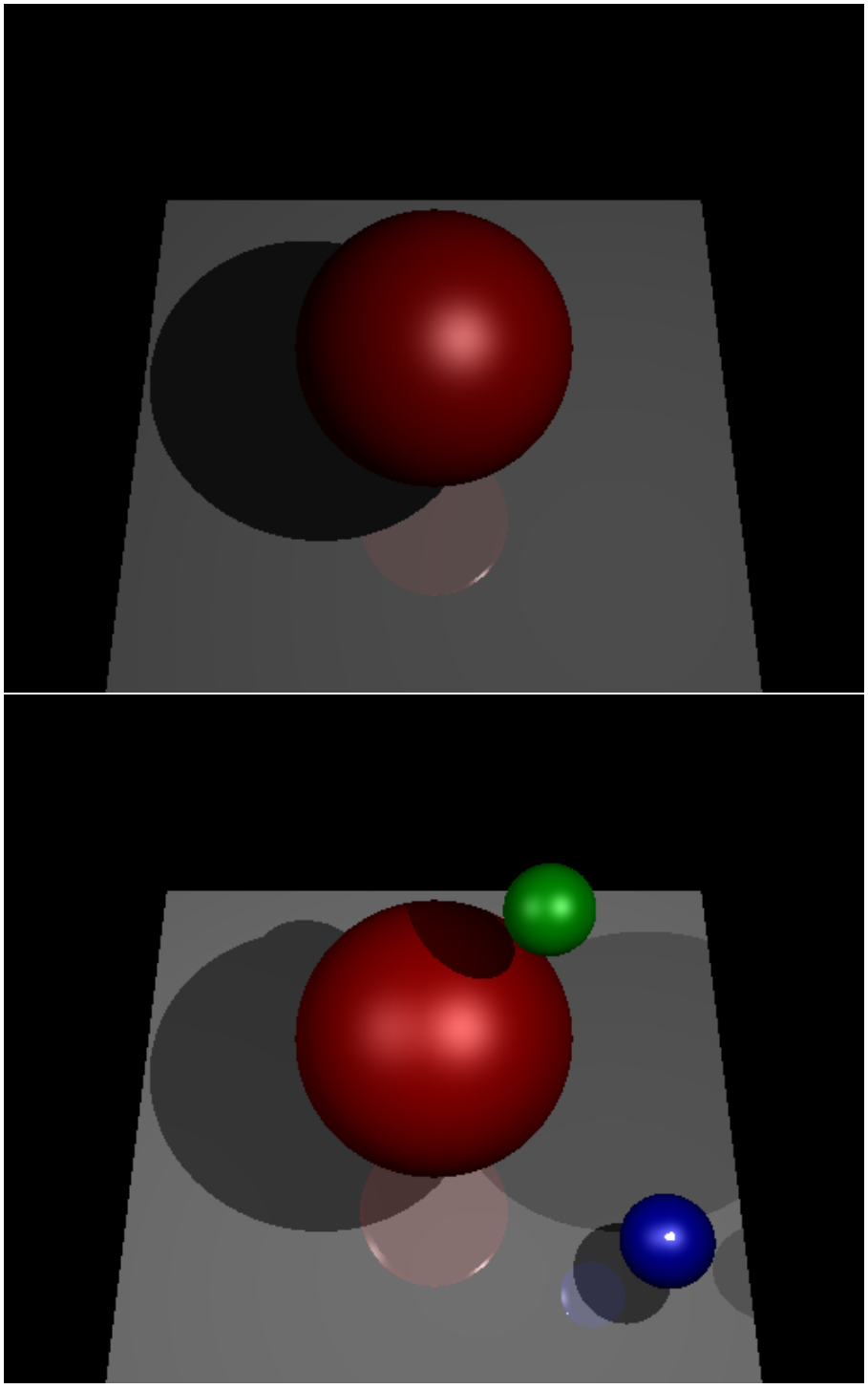
Zaimplementować rekurencyjne śledzenie promieni. Przetestować implementację dla scen o parametrach określonych w Tab.1.9 oraz Tab.1.10. Wzorcowe renderingi prezentowane są na Rys.1.15.

Tabela 1.9: Parametry sceny (reflectance)

Camera	$width = 500$	$height = 400$	$\boldsymbol{eyep} = [0, -4, 10]$	$\boldsymbol{lookp} = [0, 0, 0]$	$\boldsymbol{up} = [0, 1, 0]$	$fov = 50$
Light source	$\boldsymbol{pos} = [-3, -2, 8]$	$\boldsymbol{color} = [0.6, 0.6, 0.6]$				
Sphere	$\boldsymbol{pos} = [0, 0, 0]$ $\boldsymbol{m_{amb}} = [0.1, 0, 0]$	r=1.6 $\boldsymbol{m_{dif}} = [0.6, 0, 0]$	$\boldsymbol{m_{spec}} = [0.7, 0.7, 0.7]$	shininese=30	reflectance=0	
Triangle	$\boldsymbol{v0} = [5, 5, -5]$ $\boldsymbol{m_{amb}} = [0.1, 0.1, 0.1]$	$\boldsymbol{v1} = [-5, 5, -5]$ $\boldsymbol{m_{dif}} = [0.4, 0.4, 0.4]$	$\boldsymbol{v2} = [-5, -5, -5]$ $\boldsymbol{m_{spec}} = [0, 0, 0]$	shininese=0	reflectance=1	
Triangle	$\boldsymbol{v0} = [5, 5, -5]$ $\boldsymbol{m_{amb}} = [0.1, 0.1, 0.1]$	$\boldsymbol{v1} = [-5, -5, -5]$ $\boldsymbol{m_{dif}} = [0.4, 0.4, 0.4]$	$\boldsymbol{v2} = [5, -5, -5]$ $\boldsymbol{m_{spec}} = [0, 0, 0]$	shininese=0	reflectance=1	

Tabela 1.10: Parametry sceny (reflectance2)

Camera	$width = 500$	$height = 400$	$\boldsymbol{eyep} = [0, -4, 10]$	$\boldsymbol{lookp} = [0, 0, 0]$	$\boldsymbol{up} = [0, 1, 0]$	$fov = 50$
Light source	$\boldsymbol{pos} = [-3, -2, 8]$	$\boldsymbol{color} = [0.6, 0.6, 0.6]$				
Light source	$\boldsymbol{pos} = [5, -2, 8]$	$\boldsymbol{color} = [0.3, 0.3, 0.3]$				
Sphere	$\boldsymbol{pos} = [-1, 0, 3]$ $\boldsymbol{m_{amb}} = [0, 0.1, 0]$	r=0.4 $\boldsymbol{m_{dif}} = [0, 0.6, 0]$	$\boldsymbol{m_{spec}} = [0.7, 0.7, 0.7]$	shininese=30	reflectance=0	
Sphere	$\boldsymbol{pos} = [0, 0, 0]$ $\boldsymbol{m_{amb}} = [0.1, 0, 0]$	r=1.6 $\boldsymbol{m_{dif}} = [0.6, 0, 0]$	$\boldsymbol{m_{spec}} = [0.7, 0.7, 0.7]$	shininese=30	reflectance=0	
Sphere	$\boldsymbol{pos} = [-3, -2, -2]$ $\boldsymbol{m_{amb}} = [0, 0, 0.1]$	r=0.6 $\boldsymbol{m_{dif}} = [0, 0, 0.6]$	$\boldsymbol{m_{spec}} = [0.7, 0.7, 0.7]$	shininese=30	reflectance=0	
Triangle	$\boldsymbol{v0} = [5, 5, -5]$ $\boldsymbol{m_{amb}} = [0.1, 0.1, 0.1]$	$\boldsymbol{v1} = [-5, 5, -5]$ $\boldsymbol{m_{dif}} = [0.4, 0.4, 0.4]$	$\boldsymbol{v2} = [-5, -5, -5]$ $\boldsymbol{m_{spec}} = [0, 0, 0]$	shininese=0	reflectance=1	
Triangle	$\boldsymbol{v0} = [5, 5, -5]$ $\boldsymbol{m_{amb}} = [0.1, 0.1, 0.1]$	$\boldsymbol{v1} = [-5, -5, -5]$ $\boldsymbol{m_{dif}} = [0.4, 0.4, 0.4]$	$\boldsymbol{v2} = [5, -5, -5]$ $\boldsymbol{m_{spec}} = [0, 0, 0]$	shininese=0	reflectance=1	



Rysunek 1.15: Przykłady obrazów z wyrenderowanym odbiciem zwierciadlanym.

Rozdział 2

Wizualizacja

Omówienie zagadnień związanych z wizualizacją obrazów generowanych technikami grafiki komputerowej. Rozdział obejmuje:

- Kwantyzację koloru obliczanego przez system renderujący obraz.
- Korekcję gamma jasności koloru pikseli obrazu.

2.1 Kwantyzacja kolorów

Algorytmy grafiki komputerowej, np. ray tracing, renderują obrazy w przestrzeni liniowej, tzn. zakładana jest liniowa zależność wartości koloru od jego fotometrycznej jasności. Dodatkowo wartości kolorów wyrażane są liczbami zmiennoprzecinkowymi (wektor $\boldsymbol{rgb}$ to trzy wartości typu *float* o wartościach z zakresu $< 0,1 >$). Wyświetlenie obrazu na monitorze lub zapisanie go do pliku dyskowego wymaga wcześniejszej konwersji typu *float* na liczbę całkowitą o określonej liczbie bitów, tzn. wykonania **kwantyzacji kolorów**. Obecnie standardem jest wyrażanie kanałów koloru w postaci liczb 8 bitowych:

$$\boldsymbol{ray_{8bits}} = (int)(\boldsymbol{rgb} \cdot 255.0f). \tag{2.1}$$

2.2 Korekcja gamma

Układ wzrokowy człowieka w sposób nieliniowy interpretuje jasność obrazu. Ludzie są czuli nawet na niewielkie zmiany jasności w ciemnym otoczeniu, co umożliwia im widzenie obiektów, np. o zmroku albo przy słabym sztucznym świetle. Różnica jasności jest natomiast znacznie trudniej dostrzegalna przy jasnym oświetleniu słonecznym. Wspomniana cecha wykorzystywana jest do **nieliniowego kodowania wartości koloru**, a dokładniej do nieliniowego kodowania jasności każdej z trzech składowych R, G i B wektora kolorów $\boldsymbol{rgb}$. Większa liczba bitów liczby reprezentującej składową koloru przyporządkowana jest ciemniejszym barwom kosztem zmniejszenia dokładności reprezentacji jaśniejszych wartości. Dzięki operacji kodowania można prawidłowo zapisać wartość koloru na mniejszej liczbie bitów, tzn. zmniejszyć rozmiar obrazów komputerowych bez pogarszania ich percepcyjnej jakości.

Nieliniowe kodowanie kolorów wykonuje się wykorzystując funkcji potęgowej z wykładnikiem *1/gamma* ($\frac{1}{\gamma}$), dlatego nazywane jest **kodowanie gamma** bądź **korekcją gamma**. $gamma$ przyjmuje wartości większe od jedności, a samo kodowanie polega na podniesieniu wartości składowych koloru do $\frac{1}{\gamma}$:

$$\boldsymbol{ray} = \boldsymbol{rgb}^{\frac{1}{\gamma}}. \tag{2.2}$$

Kodowanie powinno zostać wykonane w przestrzeni liczb zmiennoprzecinkowych, tzn. przed kwantyzacją wartości koloru.

Korekcja gamma jest rozwiązaniem powszechnie stosowanym, dlatego producenci wyświetlaczy komputerowych zakładają, że dane obrazu do wyświetlenia zostały wcześniej zakodowane. Dlatego, żeby doprowadzić do linearyzacji wartości składowych koloru są **dekodowane** poprzez wykonanie odwrotnego przekształcenia, tzn.:

$$\boldsymbol{ray_{8bits}} = \boldsymbol{rgb_{8bits}}^{\gamma}, \tag{2.3}$$

przy czym wartość γ jest wartością standaryzowaną, ale na skutek rozkalibrowywania się monitora może podlegać wahaniom. Dlatego poprawnym rozwiązaniem jest pomiar wartości γ wyświetlacza i wykorzystanie tej wartości do kodowania gamma renderowanego obrazu przez wykonaniem jego kwantyzacji.

17. Ćwiczenie - Korekcja gamma

Wykonać pomiar wartości γ wyświetlacza oraz wykorzystać zmierzoną wartości do wykonania korekcji gamma renderowanego obrazu:

1. W metodzie `CImage::plotCalibChart()` zaimplementować rysowanie wzorca kalibracyjnego tak, jak prezentowane jest to na Rys. 2.1. Poszczególne pola powinny mieć wartości kolorów:

$$\begin{aligned} \boldsymbol{rgb} &= [0,0,0], \\ \boldsymbol{rgb} &= [0.25,0.25,0.25], \\ \boldsymbol{rgb} &= [0.5,0.5,0.5], \\ \boldsymbol{rgb} &= [0.75,0.75,0.75], \\ \boldsymbol{rgb} &= [1,1,1]. \end{aligned}$$

2. Wyświetlić wyrenderowany obraz, a następnie zmierzyć luminancję pól koloru wyrażaną jednostką $[cd/m^2]$ za pomocą miernika luminancji.

3. Korzystając z oprogramowania Matlab zaimplementować skrypt o nazwie *draw-gamma.m* rysujący wykres krzywej gamma wyświetlacza. Przyjąć zakres i opisy osi takie jak prezentowane na Rys.2.2.

4. Narysować taki sam wykres jak w poprzednim punkcie, ale po wcześniejszym znormalizowaniu wartości luminancji. Normalizację można wykonać dzieląc każdą zmierzoną wartość przez maksymalną zmierzoną wartość luminancji.

5. Znaleźć parametr γ funkcji potęgowej (ang. power function), które najlepiej aproksymują zmierzone dane:

$$\boldsymbol{y} = x^{\gamma}. \tag{2.4}$$

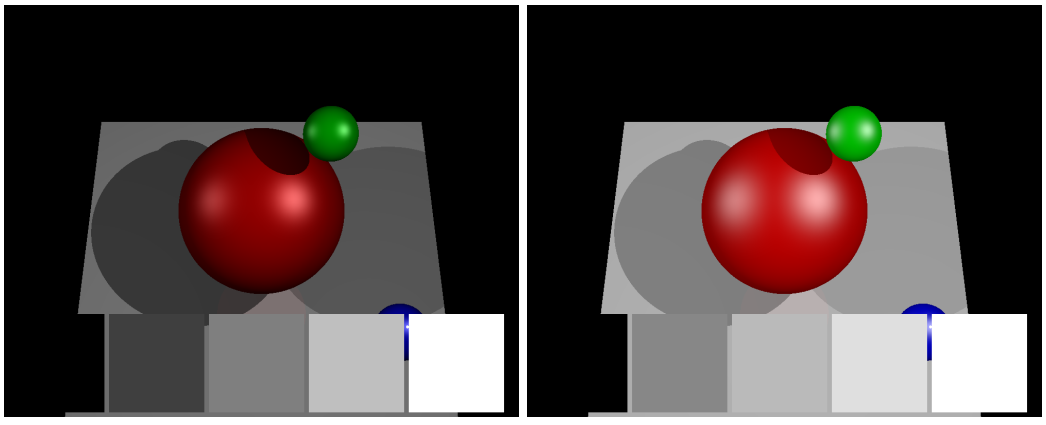
   Do tego celu można użyć narzędzia *cftool* (curveFitter w nowszych wersjach Matlaba). Wywołanie *cftool*: *cftool(kolory prostokątów, znormalizowane zmierzone wartości luminancji)*. Działanie narzędzia *cftool* prezentowane jest na Rys.2.3. Parametr a to wartość gamma.

6. W *draw-gamma.m* narysować krzywą $y = x^{\gamma}$ oraz wykres krzywej wykonującej korekcję gamma, tzn. o wykładniku potęgi równym $1/gamma$ (patrz Rys.2.2).

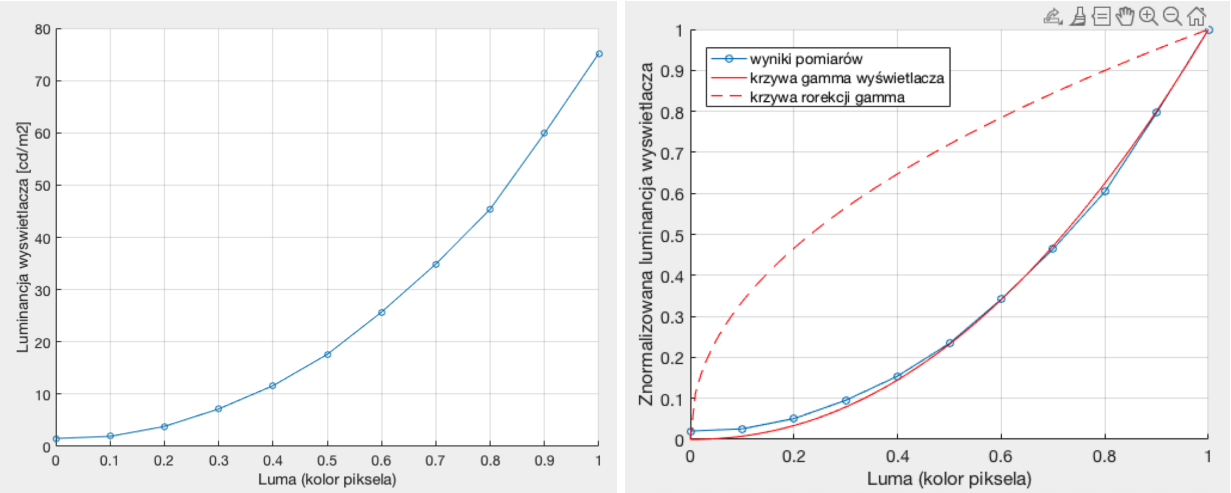
7. Wykorzystać obliczoną wartość γ w metodzie `CImage::float2uchar()` do wykonania korekcji gamma przed kwantyzacją obrazu.

8. Wyświetlić wyrenderowany obraz z uwzględnieniem korekcji gamma, a następnie zmierzyć luminancję pól koloru wyrażaną jednostką $[cd/m^2]$ za pomocą miernika luminancji.

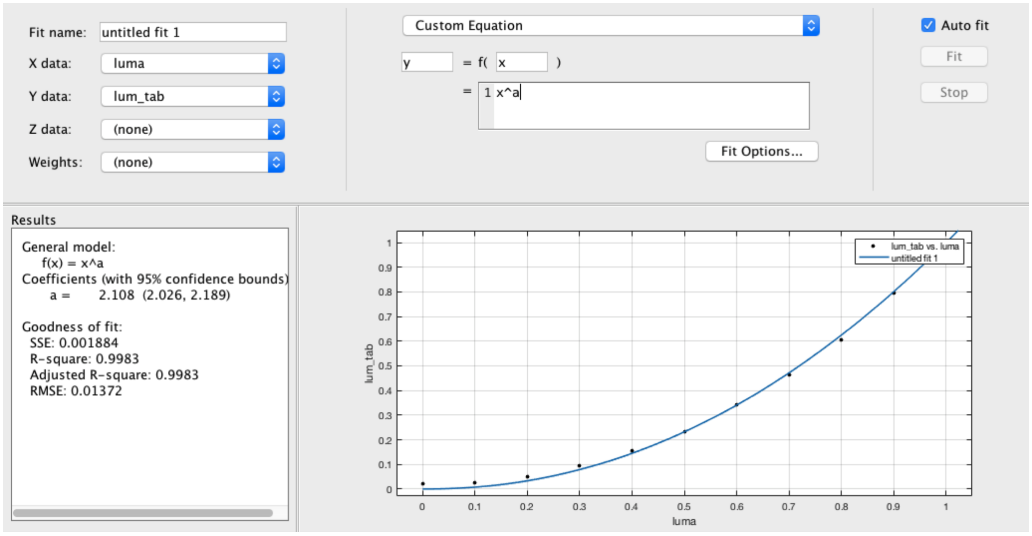
9. W *draw-gamma.m* narysować zmierzoną krzywą. Krzywa powinna mieć przebieg zbliżony do liniowego.



Rysunek 2.1: Obraz przed (po lewej) oraz po wykonaniu korekcji gamma (po prawej).



Rysunek 2.2: Wykres krzywej gamma wyświetlacza.



Rysunek 2.3: Wykres krzywej gamma wyświetlacza.

Dodatek A

Materiały pomocnicze

A.1 Repozytorium GIT

A.1.1 Tworzenie repozytorium

1. Zalogować się do serwisu WI ZUT GIT na stronie https://git.wi.zut.edu.pl. Do logowania należy użyć pełnego adresu poczty ZUT (tzn., adresu e-mail) oraz hasła jak do poczty ZUT.
2. Po zalogowaniu w "Profile and Settings" (prawy-górny róg)->Settings w polu "Full Name" wpisać swoje imię i nazwisko.
3. Zgłosić prowadzącemu zajęcia utworzenie konta w serwisie. Prowadzący na swoim koncie GIT utworzy, indywidualnie dla każdego studenta, repozytorium, dla którego nada temu studentowi prawa wprowadzania zmian.
4. Sklonować repozytorium tworząc jego lokalną kopię na komputerze laboratoryjnym:
   git clone URL
   Adres URL poda prowadzący zajęcia. Adres trzeba zanotować, ponieważ będzie służył do tworzenia lokalnej kopii repozytorium, np. na komputerze domowym lub na komputerze w innej sali.
5. Wprowadzić zmiany w w plikach, np. wpisując swoje imię i nazwisko oraz numer grupy do pliku README.md.
6. Przejść do katalogu z repozytorium i wydać polecenie wyświetlające stan lokalnego repozytorium:
   git status
7. Zatwierdzić zmiany poleceniem:
   git commit -a -m "opis wprowadzonych zmian"
8. Przesłać zmienione pliki na serwer:
   git push
9. Sprawdzić poprawność wprowadzenia zmian przeglądając plik README.md na stronie https://git.wi.zut.edu.pl.

Przesyłanie danych na serwer lub odczytywanie z serwera wymaga stosowania VPN lub korzystania z komputera podłączonego do wewnętrzej sieci Wydziału Informatyki.

A.1.2 Podstawowe komendy GIT

Podstawową zasadą korzystania z repozytorium GIT jest umieszczanie na tym repozytorium wyłacznie plików edytowalnych, tzn. np. plików z kodami źródłowymi oprogramowania. Nie należy natomiast dodawać programu wykonywalnego utworzonego w wyniku kompilacji plików źródłowych. Wystarczające jest umieszcznie pliku z informaacjami jak powinna zostać przeprowadzona kompilacja (np. CMakeLists.txt).

Zawsze przed rozpoczęciem pracy trzeba zaktualizować lokalną kopię repozytorium (plecenie *git pull*), a po zakończeniu pracy wysłać wprowadzone zmiany na serwer.

- Wyświetlenie statusu lokalnej kopii repozytorium: *git status*

- Dodanie pliku do lokalnego repozytorium: *git add [filename]*
- Cofanie zmian wywołanych użyciem *git add*: *git reset*
- Zatwierdzenie zmian w lokalnym repozytorium: *git -a commit -m "[dokładny opis wprowadzonych zmian]"*
- Cofanie zmian po wykonaniu *git commit*: *git reset –soft HEAD^*
- Przesłanie plików z lokalnego repozytorium na serwer: *git push*
- Przesłanie plików z serwera do lokalnego repozytorium: *git pull*

A.2 Transformacje geometryczne

Wektor:

$$\boldsymbol{a} = [a_x, a_y, a_z] = [a_1, a_2, a_3],$$

gdzie a_1, a_2, a_3 to skalarne parametry wektora.

Długość wektora:

$$|\boldsymbol{a}| = \sqrt{a_1^2 + a_2^2 + a_3^2} = \sqrt{\sum_{i=1}^{3} a_i^2}$$

Iloczyn skalarny:

$$\boldsymbol{a} \bullet \boldsymbol{b} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3 = \sum_{i=1}^{3} a_i \cdot b_i$$

Iloczyn wektorowy:

$$\boldsymbol{a} \times \boldsymbol{b} = (a_2 \cdot b_3 - a_3 \cdot b_2)\boldsymbol{i} + (a_3 \cdot b_1 - a_1 \cdot b_3)\boldsymbol{j} + (a_1 \cdot b_2 - a_2 \cdot b_1)\boldsymbol{k}$$

Normalizacja wektora:

$$normal(\boldsymbol{a}) = \left[\frac{a_1}{|\boldsymbol{a}|}, \frac{a_2}{|\boldsymbol{a}|}, \frac{a_3}{|\boldsymbol{a}|}\right]$$

Mnożenie macierz razy wektor:

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} m_{11} \cdot a1 + m_{12} \cdot a_2 \\ m_{21} \cdot a1 + m_{22} \cdot a_2 \end{bmatrix}$$