Politechnika Śląska w Gliwicach
Wydział Informatyki, Elektroniki i Informatyki

Programowanie komputerów

Projekt: Modelowanie sceny trójwymiarowej

Autor	Łukasz Proksa
prowadzący	dr Piotr Pecka
rok akademicki	2014 / 2015
kierunek	informatyka
rodzaj studiów	SSI
semestr	4
termin laboratoriów	wtorek, 8:30 – 10:00
grupa dziekańska	5
termin oddania	15.06.2015
data oddania	15.06.2015

Spis treści

1. Treść zadania

Program ma umożliwiać modelowanie sceny trójwymiarowej, to znaczy: umożliwiać rendering aktualnej sceny, dodawać i przesuwać obiekty na scenie, zmieniać ich kształty. Dodatkową zaletą będzie graficzny interfejs użytkownika, udostepniający wymienione funkcjonalności w przyjazny dla użytkownika sposób.

2. Analiza zadania

Zadanie jest złożone, chcąc podzielić je na mniejsze fragmenty spokojnie możemy powiedzieć, że GUI jest niezależne od głównych składników programu i może zostać zaprojektowane jako osobny moduł w późniejszej fazie projektu. Głównym celem programu jest rendering, czyli wygenerowanie obrazu na podstawie obiektów umieszczonych w trójwymiarowej przestrzeni. Można więc powiedzieć, że wejściem dla programu jest matematyczny opis takiej sceny, w której obiekty są w jakiś sposób reprezentowane, a wyjściem jest „zdjęcie" tej sceny wykonane z określonego miejsca i kierunku (przez kamerę).

Na początek nasuwają się dwa pytania: w jaki sposób reprezentować obiekty sceny w programie i w jaki sposób utworzyć zdjęcie? O problemie utworzenia zdjęcia możemy pomyśleć jako o problemie obliczenia koloru każdego jego piksela. Kolor ten zależy przede wszystkim od obiektu na który piksel „się patrzy", ale nie tylko – także od jasności oświetlenia. To na który obiekt piksel się patrzy nazywa się **problemem widoczności** (*ang. visibility problem*). Istnieje wiele sposób jego rozwiązania, po krótce wspomnę tylko o dwóch:

Sposób A) Painter's algorithm - nazwa pochodzi od malarzy, którzy malując obrazy stosują czasem tę technikę. Polega ona malowaniu obiektów w kolejności od znajdujących się najdalej do znajdujących się najbliżej kamery. Efektywność tego sposobu pozostawia jednak wiele do życzenia, ponieważ obiekty często są zamalowywane przez te leżące bliżej, przez co czas poświęcony na namalowanie tych poprzednich jest stracony. Drugi sposób jest nieco efektywniejszy.

Sposób B) Z-buffer / depth-buffer (*ang. bufor głębi*) dla każdego piksela określamy który z obiektów sceny jest znajduje się najbliżej przed nim. W tym celu sprawdzamy które obiekty przecinają prostą poprowadzoną od piksela w głąb sceny i dla każdego z takich obiektów spamiętujemy odległość w jakiej znajduje się od piksela (w tzw. buforze głębi). Na koniec wybieramy ten znajdujący się najbliżej.

Jednak co z obiektami przezroczystymi? Co jeśli obiektem leżącym najbliżej okazało się okno? Piksel nie będzie miał przecież koloru okna jako takiego; do obliczenia koloru, musimy wiedzieć co znajduje się za oknem, zatem ta metoda zawodzi. Można by oczywiście modyfikować tę metodę, jednak pozostają jeszcze inne pytania: co z intensywnością światła? Ten sam obiekt oświetlony ma inny kolor niż znajdujący się w cieniu. Co z obiektami takimi jak lustro? Podobnie jak z oknem, nie namalujemy koloru lustra jako takiego; musimy wiedzieć jakie obiekty się w nim odbijają. Czyli prosta którą pierwotnie pociągnęliśmy od piksela w głąb sceny załamuje się – musimy wiedzieć pod jakim kątem by określić w jaki obiekt trafi następnie.

Od takiego myślenia już niedaleko do trzeciej metody, którą ostatecznie wybrałem do swojego projektu, a nazywa się metodą **śledzenia promieni** (więcej w pkt. 3.5).

Powróćmy teraz do pytania drugiego, czyli **jak reprezentować obiekty na scenie**? Zazwyczaj w silnikach graficznych dokonuje się triangulacji, czyli wszystkie obiekty, nawet te najbardziej złożone, są modelowane tylko przy użyciu trójkątów. Jakość takiego odwzorowania zależy tylko od ich ilości - odpowiednio duża pozwala zamodelować nawet kulę tak, by deformacje były niewidoczne. Oczywiście im większa rozdzielczość generowanego obrazu, tym więcej trójkątów potrzeba by zachować jakość

obrazu – co pociąga za sobą wzrost kosztów obliczeń. Triangulacja obiektów daje nam dwa duże plusy. Po pierwsze nie mamy żadnego ograniczenia na obiekty jakie jesteśmy w stanie renderować programie. Po drugie trójkąty bardzo łatwo reprezentować w programie (wystarczą tylko trzy punkty) i efektywnie przetwarzać:

a) przy użyciu **współrzędnych barycentrycznych** szybko można obliczyć, czy jakiś punkt znajduje się wewnątrz trójkąta;
b) wektor normalny do płaszczyzny trójkąta pozwala rozróżnić dwie „strony„ trójkąta, co przydaje się przy teksturowaniu (trójkąt ma przód i tył, teksturą pokrywa się tylko przód), lub określaniu czy jest oświetlony (jeśli światło pada na tylną stronę, to znaczy że jest w cieniu);
c) na podstawie wektora normalnego i wektora padania, szybko obliczymy wektor odbicia (co przydaje się przy obiektach typu „lustro” lub „szyba”);
d) możemy **mapować wektory normalne**, czyli na płaskiej powierzchni trójkąta symulować wypukłości powierzchni (symulowanie „wody”)

Ponadto trzy punkty to najmniejsza ilość jaką potrzeba by wyznaczyć płaszczyznę w przestrzeni, zatem czwarty punkt byłby zbędny i marnowałby pamięć. Ktoś mógłby zarzucić, że przecież czwarty punkt mógłby nie znajdować się na tej samej płaszczyźnie co trzy pozostałe, tym samym modelując jakaś wypukłość powierzchni, którą my przy naszej 3-punktowej reprezentacji właśnie stracimy. To nie prawda, znamy już na to rozwiązanie - wystarczy ten czworokąt striangulować (przy użyciu dowolnej z jego przekątnych). Otrzymaliśmy dwa trójkąty, wypukłość powierzchni została w modelu zachowana.

3. Algorytmy i struktury danych

3.1. Wygodna tablica wielowymiarowa MArray<T>

Wadą tablic statycznych jest konieczność podania z góry ich rozmiaru. Wadą wskaźników jest to, że trzeba pamiętać o zarządzaniu pamięcią. Zaletą tablic jest ich prostota i natychmiastowy dostęp do *i*-tego elementu. Chciałem zachować tą zaletę, pozbywając się tych dwóch wad, dlatego postanowiłem utworzyć klasę, która oddzieli warstwę logiczną (interfejs jaki posiada tablica) od fizycznej (konieczność zarządzania pamięcią). Tak powstał szablon klasy MArray<T>. W konstruktorze podaje się ilość wymiarów tablicy i wielkość każdego z nich. Klasa za nas zaalokuje odpowiednią ilość pamięci, a w destruktorze ją zwolni. Mamy stały dostęp do i-tego elementu: dla n-wymiarowej tablicy, wywołujemy operator() z n-elementową ilością argumentów, i-ty argument to indeks i-tego wymiaru. Przykład dla 3 wymiarowej tablicy intów, w której każdy z wymiarów ma wielkość 10:

```
MArray<int> array(2, 10, 10, 10);
int element = array(1,5,0);
```

```
int array[10][10][10];
int element[1][5][0];
```

Przewagą lewej strony nad prawą jest to, że wielkości wymiarów mogą zostać określone dopiero w czasie działania programu (*ang. at runtime*), natomiast po prawej wielkość tablicy musi być określona w momencie kompilacji.

3.2. Tablica dynamiczna Vector<T>

Tablica dynamiczna o rozmiarze dopasowującym się ilości elementów w niej przechowywanych. Ponieważ jest szablonem *klasy*, tablica może być dowolnego typu (może to być tablica dowolnych obiektów, wskaźników na obiekty, tablica tablic (dwuwymiarowa), itp). Jej użycie jest bardzo wygodne i w programie jest często stosowana. Czas dostępu do *i*-tego elementu jest stały, ponadto jeżeli obiekty w tablicy są uporządkowane (określony jest na obiektach operator mniejszości), przyspiesza to operację wyszukiwania elementu ze złożoności liniowej do logarytmicznej. Wyszukiwanie odbywa się

za pomocą algorytmu prostego wyszukiwania binarnego[1]. Dostępna jest wtedy również operacja sortowania wektora opisana w pkt. 3.3

Zdefiniowane są operacje dodawania i usuwania elementu z końca wektora. Operacja dodawania sprawdza, czy aktualna wielkość tablicy jest wystarczająca do przechowania nowego elementu. Jeśli nie, alokowana jest większa (w programie wybrano, że dwukrotnie większa) tablica, elementy ze starej tablicy przepisywane są do nowej, na końcu dodawany jest nowy element.

Operacja usuwania po usunięciu elementu sprawdza, czy wielkość tablicy nie jest zbyt duża, jak na ilość elementów w niej przechowywanych (w programie przyjęto warunek „czy jest dwukrotnie większa"). Jeśli tak, to alokowana jest dwukrotnie mniejsza tablica, elementy ze starej przepisywane są do nowej tablicy. Operacja ta ma na celu optymalizacje pamięciową wektora, żeby nie zajmował dużo pamięci gdy nie jest to potrzebne.

3.3. Algorytm sortujący Insertion-sort

Algorytm sortowania przez wstawianie sortuje *n*-elementową tablicę w czasie $O(n^2)$. Posiada więc gorszą klasę złożoności czasowej w porównaniu do algorytmu *merge-sort* dostępnego w bibliotece STL. Natomiast jego zaletą jest działanie w miejscu – algorytm nie potrzebuje dodatkowej pamięci.

Poniższy pseudokod sortuje niemalejąco tablicę *A[1..n]* przez wstawianie:

```
InsertionSort(A)
1. for (i ← 2, i <= n, i ← i+1) do
2.    val ← A[i]
3.    pos ← 1
4.    while (A[pos] < val) do
5.      pos ← pos + 1
6.    for (j ← i, j > pos, j ← j-1) do
7.      A[j] ← A[j-1]
8.    A[pos] ← val
9. return A
```

Zewnętrzny *for* w linii 1 iteruje po każdym elemencie w tablicy *A*. Dla każdego elementu w liniach 4-5 wyznaczane jest należne mu miejsce w tablicy *A* i spamiętywane w zmiennej *pos*. Następnie w liniach 6-7 elementy większe przesuwane są w prawo tak, by na pozycji *A[pos]* było wolne miejsce. W linii 8 następuje przypisanie *val*.

Dowód poprawności:
Linia 1: Przed każdym wykonaniem pętli *for* prefiks tablicy *A[1..i-1]* jest posortowany. Warunek początkowy spełnia ten warunek, bo prefiks liczy jeden element. W szczególności jeśli tablica *A* liczy jeden element, jest już posortowana i *for* nie wykona się.
Linia 4: Przed każdą iteracją pętli *while* elementy na indeksach *1..pos-1* są mniejsze od *val.* Na początku tak jest, bo podzbiór ten jest pusty. Po każdej iteracji podzbiór ten powiększa się o jeden element, lub *while* kończy działanie, jeśli element *A[pos]* nie spełnia tego warunku. Zatem *pos* jest indeksem w *A* pod którym ma znaleźć się *val.*
Linia 6: *For* realizuje proste przesunięcie elementów na pozycjach *pos..i* o jeden w prawo.
Linia 8: Po przypisaniu, prefiks tablicy *A[1..i]* jest posortowany, zatem spełniony jest niezmiennik zewnętrznego *for.*

Złożoność pamięciowa i obliczeniowa:

[1] Wyszukiwanie binarne - ang. *binary search.*

Złożoność pamięciowa wynosi $O(n)$ ponieważ pamiętamy *n*-elementową tablicę. Złożoność obliczeniowa wynosi pesymistycznie $O(n^2)$.

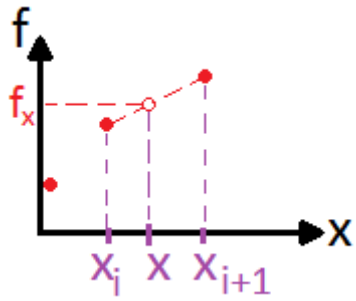
3.4. Algorytm filtrowania dwuliniowego

Stosowane jest podczas mapowania wektorów normalnych lub nakładania na obiekt tekstury w celu poprawienia jakości. Oba przypadki są podobne, bo zarówno tekstura jak i wektory normalne są zapisane w obrazie (pliku *.jpg, *.png lub innym). Zatem w obu przypadkach zachodzi konieczność odwzorowania (*ang. mapping*) trójwymiarowej powierzchni obiektu na powierzchnię dwuwymiarową. Czyli punkt p(x,y,z) odwzorujemy w punkt p'(u,v) (mniejsza z tym jak). Ważne, że powierzchnia obiektu jest ciągła, zatem otrzymane współrzędny u i v niekoniecznie będą liczbami całkowitymi. A przecież obrazy składają się z pikseli o współrzędnych całkowitych. Jak więc określić kolor piksela o współrzędnych (u,v) = (2,456; 1,43) skoro znamy tylko kolory pikseli (2,1), (3,1), (2,2), (3,2)? Istnieją co najmniej dwa sposoby:

a) Filtrowanie punktowe (*ang. point sampling*) – polega na zaokrągleniu współrzędnych (u,v) do najbliższych liczb całkowitych. Jest najszybsze i daje najgorszą jakość. Defekty widać zwłaszcza gdy tekstura znajduje się blisko.
b) Filtrowanie dwuliniowe (*ang. bilinear filtering*) – jest nieco wolniejsze, ale daje znacznie lepsze rezultaty, gwarantuje płynne przejścia kolorów między pikselami w każdą stronę.

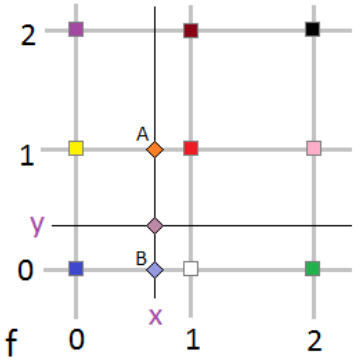
W programie znajdują się implementacje obu tych algorytmów, jednak dwuliniowe jest filtrowaniem domyślnym ze względy na jakość obrazu.

Aby zrozumieć filtrowanie dwuliniowe, trzeba najpierw zrozumieć interpolacje liniową funkcji jednej zmiennej. Dane są punkty $x_0, x_1, \dots, x_n$ i wartości funkcji w tych punktach $f_o, f_1, \dots, f_n$. Zadaniem interpolacji jest obliczenie $f(x)$ dla wszystkich $x \in < x_0, x_n >$ to znaczy nawet dla x dla których wartości f nie zostały wprost podane. Załóżmy że chcemy obliczyć wartość funkcji f dla argumentu x takiego że $x_i < x < x_{i+1}$. Musimy zatem zgadnąć jak przebiega funkcja pomiędzy tymi punktami i musimy zrobić to jak najlepiej. My założymy, że funkcja pomiędzy tymi punktami zmienia się liniowo (stąd nazwa – interpolacja liniowa). Dla naszych celów jest to wystarczająco dobre oszacowanie. Na rysunku po prawej wartość funkcji f w punkcie x została oszacowana na podstawie znajomości wartości funkcji f w punktach x_i i x_{i+1}. Jednakże interpolacja liniowa funkcji jednej zmiennej nie wystarcza do filtrowania tekstur, ponieważ tekstury są dwuwymiarowe; potrzebujemy czegoś więcej.



Filtrowanie dwuliniowe jest uogólnieniem filtrowania liniowego dla funkcji dwóch zmiennych. Jego zrozumienie nie powinno przysporzyć większych kłopotów, intuicyjnie polega na kilkukrotnym zastosowaniu interpolacji liniowej funkcji jednej zmiennej, tak jak pokazano na przykładzie po prawej.

Wartość funkcji f w punkcie (x, y) została obliczona na podstawie wartości funkcji w czterech otaczających punktach: $(0,0)$, $(1,0)$, $(0,1)$, $(1,0)$. Najpierw zastosowano interpolacje liniową dla funkcji jednej zmiennej tak aby policzyć wartość w punkcie A (x jest zmienną, $y = const$). Analogicznie dla punktu B. Następnie po raz trzeci zastosowano interpolacje liniową funkcji jednej zmiennej dla punktów A i B, tak by policzyć wartość w punkcie docelowym ($x = const$, a y jest zmienną).



Na obrazku wartościami funkcji były kolory, zatem jest to przykład filtrowania dwuliniowego tekstury. Mogą to być jednak inne wartości, np. wektory normalne płaszczyzny obiektu. W praktyce przy implementacji ważne jest jedynie, aby na tych wartościach była określona operacja odejmowania i mnożenia przez skalar.

3.5. Algorytm śledzenia promienia, czyli jak określić kolor piksela?

W punkcie 2 zastanawialiśmy się nad metodami obliczenia koloru pojedynczego piksela, co doprowadziło nas do metody zwanej metodą śledzenia promieni (*ang. ray tracing*). Jej zaletą jest to, że wspiera różne, skomplikowane efekty wizualne, takie jak:

- Odbicie idealnie kierunkowe – obiekty typu „lustro", ale nie tylko. W wodzie również odbijają się inne obiekty (**S** od ang. *specular*)
- Transmisja idealnie kierunkowa z refrakcją - obiekty typu „szkło", przez które promień przenika i załamuje się (**T** od ang. *transmittance*)
- Odbicie idealnie rozproszone – na zwykłej, nierównej powierzchni (**D** od ang. *diffuse*)

W programie została zaimplementowana podstawowa **metoda Whitteda** (*ang. Whitted-style ray tracing*), która ponadto jest łatwa i przyjemna w implementacji. Implementacja obsługuje ścieżki typu

$$E(S|T)^*DL$$

gdzie **E** to obserwator, **L** to źródło światła, a **S**, **T** i **D** to efekty wizualne opisane powyżej. Taki zapis oznacza, że ścieżka zaczyna się od obserwatora, efekty **S** i **T** mogą zajść dowolną liczbę razy (w programie ograniczoną parametrem *maxTraceDepth* – maksymalna głębokość ścieżki), efekt **D** zachodzi dokładnie raz przed trafieniem w źródło światła. Obserwatorem jest kamera, a dokładniej źródłem każdego promienia jest pojedynczy piksel generowanego obrazu. Funkcja śledząca promień zwraca kolor piksela. W czasie śledzenia promienia, funkcja ta wywołuje się rekurencyjnie przy każdym odbiciu / efekcie wizualnym, dlatego nie jest z góry znany jej czas działania. Zależy od ustawienia obiektów na scenie. Kolor piksela symbolizuje energię jaka dociera do niego ze źródła światła. Kolory źródła światła i materiałów są w formacie RGB[2]. Poniżej przykłady ścieżek:

Przykład 1) Ścieżka typu EDL. Załóżmy że **L** jest czerwone (FF,00,00) a materiał biały (FF,FF,FF). Kolor jaki zobaczy **E** to będzie kolor **L** przefiltrowany w wyniku odbicia z materiałem. Ponieważ materiał jest biały (nie pochłania żadnej składowej koloru, wszystkie odbija) to kolor jaki zobaczy **E** będzie dokładnie taki sam jak kolor **L**, czyli czerwony.

Przykład 2) Ścieżka typu EDL. Załóżmy że **L** jest czerwone (FF,00,00) a materiał zielony (00,FF,00). Kolor jaki zobaczy **E** to będzie kolor **L** przefiltrowany w wyniku odbicia z materiałem. Ponieważ materiał jest zielony (to znaczy że pochłania wszystkie składowe koloru, odbija tylko składową zieloną), to **E** nie zobaczy żadnego koloru (ponieważ źródło światła nie zawiera składowej zielonej)! A dokładniej **E** zobaczy kolor czarny (00,00,00). Może się to wydawać dziwne, ale tak jest. W rzeczywistości kolory: zielony, czerwony, niebieski itp. składają się również z innych składowych. Podobnie źródło światła - nie jest nigdy dokładnie zielone, czerwone, itp. bo zawiera również inne składowe. **Jednak dla uproszczenia w dalszych przykładach kolory nadal będą zero / jedynkowe**.

Przykład 3) Ścieżka typu ESDL. Załóżmy że **L** jest koloru (FF,FF,FF); materiał **D** koloru (FF,FF,00), materiał **S** koloru (FF,00,00). Dla wygody przeanalizujemy ścieżkę od strony **L** w kierunku **E**. Biały promień

[2] RGB – *(ang. Red, Green, Blue*) skrótowiec oznaczający kolory składowe obrazu: czerwoną, zieloną, niebieską, standardowo właśnie w takiej kolejności.

najpierw trafia w materiał **D**, który nie przepuszcza składowej niebieskiej. Następnie promień odbija się zwierciadlanie od powierzchni **S**, tracąc przy tym wszystkie składowe oprócz składowej czerwonej. Zatem obserwator zobaczy kolor czerwony.

Dokładniejszy opis śledzenia promieni można znaleźć wraz z implementacją klasy *Ray* w punkcie 5.7.

4. Specyfikacja zewnętrzna

Program uruchamiany jest z linii poleceń, przyjmuje dwa argumenty opcjonalne:

- *samplesPerPixel* (domyślnie 1) – oznacza ilość próbek na piksel; ilość promieni jaka zostanie wystrzelona z piksela w głąb ceny w celu określenia jego koloru. Wartość większa niż 1 działa jak **antyaliasing**[3], kolejne promienie są wystrzelane pod minimalnie innym kątem, kolor piksela jest średnią arytmetyczną kolorów zwróconych przez każdy z promieni. W ten sposób wygładzane są brzegi obiektów. Bo jeśli wystrzelany jest tylko jeden promień, istnieje szansa że w wyniku błędów operacji zmiennoprzecinkowych, jeden piksel zdoła jeszcze trafić w obiekt, a piksel nad (lub inny z otaczających) już nie, chociaż z kształtu obiektu wynikałoby że trafić powinien. Sledząc kilka promieni (np. 4) jest szansa że np. 2 trafią w obiekt, a 2 nie (czyli jest to brzeg obiektu). Wyciągnięcie wartości średniej z tych kolorów ładnie wygładzi nam brzegi. Parametr ten ma ogromny wpływ na czas obliczeń, który jest do niego wprost proporcjonalny. **Czterokrotne zwiększenie wartości parametru, powoduje czterokrotne wydłużenie obliczeń**!
- *maxTraceDepth* (domyślnie 5) – oznacza maksymalna głębokość rekursji; maksymalną ilość efektów wizualnych, jakie mogą zajść na drodze od obserwatora do źródła światła (porównaj wzór z pkt. 3.5). Domyślna wartość 5 jest na tyle duża, że pozwala zaobserwować większość zwykłych efektów, takich jak: cienie, przezroczystość obiektu, odbicia lustrzane, cień za lustrem, przedmiot odbity w lustrze widziany przez szkło (i jego cień). Zwiększenie wartości nie wpływa istotnie na jakość obrazu, natomiast jego zmniejszenie ma duże walory edukacyjne – pozwala obserwować od jakiego momentu możliwe staje się obserwowanie określonych zjawisk.

Jeśli chcemy podać *maxTraceDepth*, musimy podać *samplesPerPixel*. Program zakłada poprawność wszystkich danych wejściowych.

5. Specyfikacja wewnętrzna

Wszystko co odpowiada za rendering zostało przeze mnie napisane od zera. Używam własnej funkcji śledzącej promienie, obiekty na scenie są reprezentowane moją klasą. Do wyświetlenia wygenerowanego obrazu używam biblioteki GTK i metody *PutPixel* (co ciekawe w GTK nie ma wbudowanej takiej metody, jej odpowiednikiem jest namalowanie prostokąta o wielkości 1 x 1 px).

5.1. Struktura plików projektu

Tworząc projekt starałem się zachować przejrzystą i wygodną strukturę plików i katalogów projektu.

- W katalogu *containers* znajdują się kontenery obiektów. Zazwyczaj są to szablony klas tak, by mogły przechowywać różne typy danych (jak MArray<T> czy Vector<T>).
- W katalogu *ifce* znajdują się nagłówki kilku interfejsów używanych w programie (takich jak *Transformable*, czy *FilteringStrategy*). O *FilteringStrategy* można przeczytać więcej w pkt. 5.10. Umieszczenie wszystkich interfejsów w jednym katalogu być może nie jest najlepszym

[3] Antyaliasing – techniki zmniejszające ilość błędów zniekształcających obraz. Np. dla efektu „schodkowania" krawędzi antyaliasing stworzy efekt wygładzonych krawędzi.

pomysłem, ze względu na różne ich zastosowanie. W projekcie są jednak tylko 3 i nie jest przewidywane gwałtowne zwiększenie ich ilości, więc nie ma problemu.

- W katalogu *math* znajdują się klasy odpowiadające matematycznym pojęciom: macierz (*matrix*), wektor (*vector*, w projekcie *vec* by odróżnić od kontenera *vector<T>*) i punkt (*point*). Klasy te posiadają gettery / settery oraz przeciążone operacje arytmetyczne, ułatwiające zapisywanie wyrażeń matematycznych, potrzebnych np. przy przekształceniach macierzowych, obliczeniach wektorów normalnych, równania płaszczyzny, pola trójkąta. Reprezentowanie punktu w postaci klasu Point zamiast trójki liczb ułatwia też przekazywanie argumentów do funkcji (łatwiej i szybciej zadeklarować funkcję z jednym argumentem niż trzema). Ponadto **zdaniem Robert C. Martina zwiększa to czytelność kodu**[4], bo dobra metoda powinna przyjmować jeden, lub co najwyżej dwa argumenty (mile widziany jest też ich brak).
- W katalogu *utest* znajdują się testy jednostkowe. Testy dołącza się do pliku *main* dyrektywą *include*. Więcej o testach można przeczytać w rozdziale 6 poświęconym testowaniu.
- W katalogu *world* umieściłem klasy związane z modelowaniem sceny: definicje kolorów (*color*), materiałów (*material*), klasy śledzącej promienie (*ray*), przechowującej informacje o zderzeniu (hit), model kamery (*camera*) i źródła światła (*light_source*) i samą scenę (*scene*). W osobnym podkatalogu *gameobjects* znajdują się definicje obiektów sceny.
- W *utilities* znajdują się klasy dostarczające pewne funkcjonalności, lecz nie pasujące nigdzie indziej, np. dwie implementacje interfejsu *FilteringStrategy* (*bilinear_filtering* oraz *point_sampling*), klasa pośrednicząca w rysowaniu po rzeczywistym urządzeniu wyjściowym (*drawer*). Aktualnie jest używana tylko do rysowania obrazu (dwuwymiarowa tablica *Color*) na powierzchni *cairo_surface_t* biblioteki GTK[5].

Plik *main.cpp* znajduje się w głównym katalogu. Używałem środowiska Eclipse, zatem automatycznie dodane zostały katalogi *Debug* i *Release* i kilka plików konfiguracyjnych. Kod programu był wersjonowany przy użyciu ułatwiającego pracę programu Git[6], zatem doszedł ukryty katalog *.git* i plik *.gitignore*.

5.2. Klasa „Color” i przestrzeń nazw

W metodzie śledzenia promieni Whitteda i matematycznych wzorach często pojawia się pojęcie koloru, dlatego należało je w programie zamodelować. W kodzie reprezentuje je klasa *Color*. Udostępnia prosty interfejs – *gettery* i *settery*, operator pozwalające kolory mnożyć ze sobą (przydatne przy określaniu koloru promienia odbijającego się od różnych powierzchni, opisane w pkt. 3.5). Ogólnie interfejs klasy jest zupełnie zwyczajny, nie robi nic ponad to, czego można się spodziewać po klasie o nazwie *Color* - zatem spełniona jest jedna z zasad R. Martina mówiąca o tym, że klasa powinna robić to, czego się po niej spodziewasz.

Sama klasa to nie wszystko – potrzebujemy jej obiektów. Dopiero one reprezentują konkretne kolory. W programie chcemy mieć pewną ilość predefiniowanych kolorów. W ten sposób tworząc np. źródło światła o kolorze zielonym, lub materiał o kolorze niebieskim, nie musimy na nowo tworzyć obiektu klasy kolor (kto by spamiętał konkretne wartości RGB kolorów), lecz wykorzystamy raz utworzone, predefiniowane kolory. Należy zauważyć dwie rzeczy. Po pierwsze: wystarczy nam jeden obiekt reprezentujący konkretny kolor, nie ma sensu tworzyć dwóch obiektów reprezentujących kolor zielony,

[4] Czytelniejszy kod jest łatwiejszy w utrzymaniu i znajdowaniu błędów, opisane w książce „Czysty kod” zamieszczonej w bibliografii.
[5] GTK – (*Gimp Toolkit*) biblioteka graficzna dostępna w wielu językach i systemach operacyjnych.
[6] Git – rozproszony system kontroli wersji, pierwotnie stworzony by wspierać projekt jądra Linuxa. Umożliwia wersjonowanie (zapamiętywanie) konkretnych etapów rozwoju projektu.

bo będą identyczne. Konsekwencją będzie przekazywanie kolorów przez referencję lub wskaźnik (na dodatek stałą referencję i wskaźnik, bo predefiniowanych kolorów nie będziemy modyfikować). W ten sposób nie będziemy marnowali pamięci. I po drugie – wynika z tego, że kolory (czyli te obiekty) muszą być widoczne z wielu miejsc programu. Rozważałem dwa sposoby na osiągnięcie tego:

a) **Wzorzec projektowy „Singleton”** – rozwiązanie eleganckie pod względem obiektowym. Na początku programu, do obiektu singleton zostałyby załadowane obiekty predefiniowanych kolory. Obiekt singleton byłby globalnie dostępny. W ten sposób obiekt potrzebujący kolor np. „żółty” poprosiłby singleton o udostępnienie mu odpowiedniego obiektu klasy *Color*. Rozwiązanie to odrzuciłem za radą Prowadzącego. Obliczenia na kolorach pojawiają się w programie bardzo często, narzut czasowy związany z dostępem do singleton maiłby duży wpływ na czas renderowania sceny.
b) Zdefiniowanie globalnie widocznych zmiennych – to rozwiązanie wybrałem. W pliku *main* definiowane są kolory ze słowami `static const`. Aby zapewnić unikalność nazw (a także by IDE[7] mogło podpowiadać mi możliwe uzupełnienia) obiekty umieściłem w przestrzeni nazw *color*. W ten sposób pisząc `color::dark` IDE jako jedno z możliwych uzupełnień podpowiada mi `color::darkBlue`, co jest wygodne.

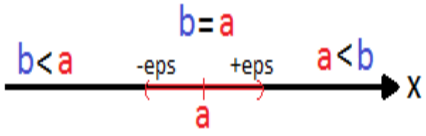
5.3. Klasa „Material” i przestrzeń nazw

Sam kolor to nie wszystko. Powierzchnia materiału budowana jest nie z koloru, a z materiału. Taki materiał ma swoje właściwości związane z odbiciem idealnie rozproszonym, odbiciem idealnie kierunkowym, transmisją idealnie kierunkową z refrakcją (ogólnie z efektami wizualnymi obsługiwanymi przez program, omówionymi w punkcie 3.5). Powiemy że materiał odbija z rozproszeniem, jeśli kolor odbicia rozproszonego jest różny od czarnego, to znaczy jeśli materiał przepuszcza chociaż jedną ze składowych RGB w stopniu większym od zera. Inaczej wszelkie obliczenia związane z odbiciem rozproszonym poszłyby na marne – co nam po obliczeniu koloru, jaki dochodzi do tego materiału od źródła światła, skoro i tak nie przepuści żadnej składowej. Powiemy że materiał wspiera dany efekt, jeśli kolor z nim związany jest inny niż czarny. Wykonując takie sprawdzenie, zaoszczędzimy czas obliczeniowy w takich przypadkach.

Sprawa z materiałami ma się podobnie jak z kolorami. Konkretna instancja klasy *Material* to konkretny materiał. W cały programie wystarczy nam tylko jedna instancja reprezentująca np. właściwości marmuru. Dlatego predefiniowane materiały zdefiniowałem jako globalne obiekty z użyciem słów kluczowych `static const` w przestrzeni nazw *material*.

5.4. Funkcje dla operacji na liczbach zmiennoprzecinkowych.

Porównywania dwóch liczb zmiennoprzecinkowych nie można wykonywać wprost. Istnieje bardzo mała szansa, żeby dwie takie liczby były sobie dokładnie równe. W sytuacji gdy chcielibyśmy liczbę a uznać za równą b, zazwyczaj różnią się o pomijalnie mały błąd obliczeń zmiennoprzecinkowych. Utrudnia to porównywanie, ponieważ za każdym razem trzeba pamiętać o dodaniu i odjęciu pewnego niewielkiego epsilonu od jednej z liczb. Należy zwrócić uwagę ma mały niuans. Mogłoby się wydawać, że problem jest tylko operatorem równości, natomiast pozostałe, np. operator mniejszości będą działały poprawnie i nie wymagają poprawek. Otóż nie. Skoro $b == a \Leftrightarrow a - eps \leq b \leq a + eps$ to



[7] IDE - (*ang. Integrated Development Environment*) zintegrowane środowisko programistyczne, zazwyczaj składa się z kompilatora, zaawansowanego edytora tekstowego, środowiska do debugowania programu. Utawia pracę programisty.

porównanie $b < a$ nie jest prawidłowym sprawdzeniem, czy b jest mniejsze od a. Tak naprawdę $b < a \Leftrightarrow b < a - eps$. Zobrazowałem to na ilustracji po lewej. Analogicznie z pozostałymi operatorami.

Aby to ułatwić, utworzyłem globalnie widoczne funkcje porównujące, zwracają typ *bool*, za argument przyjmują dwie liczby typu *double*. Poniżej kod funkcji:

```
inline bool eq(double lhs, double rhs, double eps = EPS)
{
   return (lhs - eps <= rhs) && (rhs <= lhs + eps);
}
```

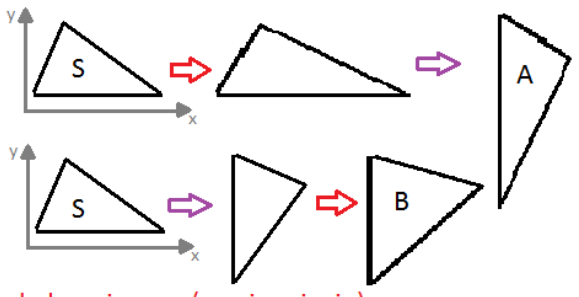
Aby wywoływania nie spowalniały programu, funkcje są oznaczone słowem kluczowym `inline`. Żeby nie doszło no konfliktu nazw, funkcje umieściłem w przestrzeni nazw db. W ten sposób mimo krótkich nazw, przeznaczenie funkcji jest zrozumiałe. Przykładowe wywołanie:

```
if (db::eq(a, b)) {
       // a i b są równe
}
```

Spotkałem się ze zdaniem, że typ *double* jest zbyt dokładny jak na obliczenia związane z grafiką i niepotrzebnie wydłuża czas obliczeń. Być może jest to powód dla którego proces renderowanie trwa w moim programie dość długo (moim subiektywnym zdaniem 3 sekundy na rendering najgorszej jakości przy jeden próbce na pixel to za dużo). Nie mam jednak z czym tego porównać, dlatego w ramach testów planuję zmienić wszystkie typy *double* na *float* i sprawdzić szybkość i jakość działania.

5.5. Wygodne transformacje na macierzach dzięki superklasie Matrix (i klasach podrzędnych)

Wszystkie operacje modyfikowania obiektów: przesunięcie ich na scenie w dowolnym kierunku, skalowanie (rozciąganie), obracanie (rotacja względem osi X, Y, Z) sprowadzają się do transformacji na macierzach. Skutkiem wszystkich tych operacji jest zmiana współrzędnych punktów obiektu. Zmiany tych współrzędnych możemy łatwo obliczyć mnożąc te punkty (trójka współrzędnych x, y, z to też macierz) przez macierz transformacji. Zatem modyfikowanie kształty obiektu sprowadza się do utworzenia odpowiedniej macierzy transformacji. Problem utworzenia macierzy transformacji nie jest taki trudny. Każda z operacji podstawowych opisanych powyżej ma odpowiadającą sobie macierz transformacji. Jeśli chcemy dokonać kilka operacji na raz, wystarczy te macierze ze sobą połączyć (a dokładniej przemnożyć). Mnożenie macierzy nie jest przemienne, dlatego ma znaczenie która operacja jest wykonywana najpierw. Tak samo w rzeczywistości efekty będzie różny, gdy najpierw rozciągniemy trójkąt wzdłuż osi OX a następnie obrócimy zgodnie z ruchem wskazówek zegara o 90*, niż gdybyśmy zrobili to w odwrotnej kolejności. Ilustruje to obrazek powyżej. Z trójkąta S otrzymano dwa różne trójkąty A i B w wyniku wykonania tych samych operacji, lecz w odwrotnej kolejności.



Spróbujmy wyprowadzić wzory na macierze transformacji. Zacznijmy od najprostszej – macierzy translacji (czyli przesunięcia obiektu względem osi) dla dwóch wymiarów. Początkowy punkt P zostanie przesunięty o a jednostek wzdłuż osi OX i b jednostek osi OY. Można to zapisać tak:

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \qquad T = \begin{bmatrix} a \\ b \end{bmatrix} \qquad P' = P + T = \begin{bmatrix} x + a \\ y + b \end{bmatrix}$$

Dlaczego zapis macierzowy? Ponieważ ujednolica on przekształcenia; każde przekształcenie elementarne może zostać przedstawione za pomocą macierzy. Gdyby zrezygnować z postaci macierzowej, wtedy translacja byłaby dodawaniem wartości do współrzędnych, skalowanie byłoby mnożenie współrzędnych przez skalar, a rotacja… byłaby skomplikowana. Złożenie tak zapisanych operacji w celu wykonania jednej, złożonej, na raz byłoby niepotrzebnie skomplikowane. A macierze składa się bardzo łatwo – wystarczy je ze sobą przemnożyć. No właśnie – chcielibyśmy, aby operacją na macierzach było mnożenie (a jak na razie mamy dodawanie). Można chwilę pogłówkować, można sięgnąć po gotowy wzór, ostatecznie powyższe równanie można przekształcić tak:

$$P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad T = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \qquad P' = T * P = \begin{bmatrix} x + a \\ y + b \\ 1 \end{bmatrix}$$

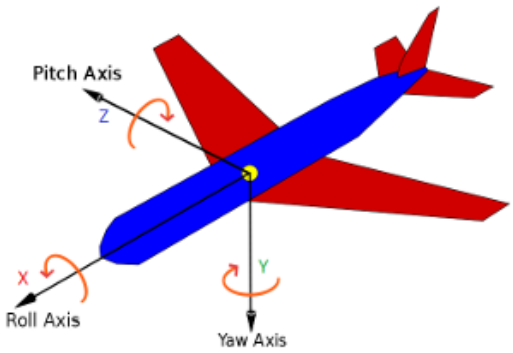
Kto nie wierzy, niech sprawdzi. Wynik jest taki sam, ale teraz mamy mnożenie macierzy. Możemy teraz przejść o wymiar dalej:

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad T = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad P' = T * P = \begin{bmatrix} x + a \\ y + b \\ z + c \\ 1 \end{bmatrix}$$

Poniżej macierz transformacji dla skalowania. Skalowaniem względem każdej z osi możemy wykonać za jednym razem.

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Poniżej macierz transformacji dla rotacji. Tutaj niestety rotację możemy wykonać na raz tylko dla jednej osi, stąd aż trzy macierze – odpowiednio dla rotacji względem osi X, Y, Z.



Na rysunku została zilustrowana „rotacja obiektu względem wybranej osi”.

$$R_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -sin(\alpha) & 0 \\ 0 & sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_Y = \begin{bmatrix} cos(\alpha) & 0 & sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\alpha) & 0 & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_Z = \begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 & 0 \\ sin(\alpha) & cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Poniżej przykład operacji złożonej z transformacji elementarnych. Obiekt jest powiększany 3-krotnie, następnie przesuwany o -5 jednostek wzdłuż osi X, następnie obracany wokół osi Y o 60*, a na koniec przesuwany o 12 jednostek wzdłuż osi X i Y.

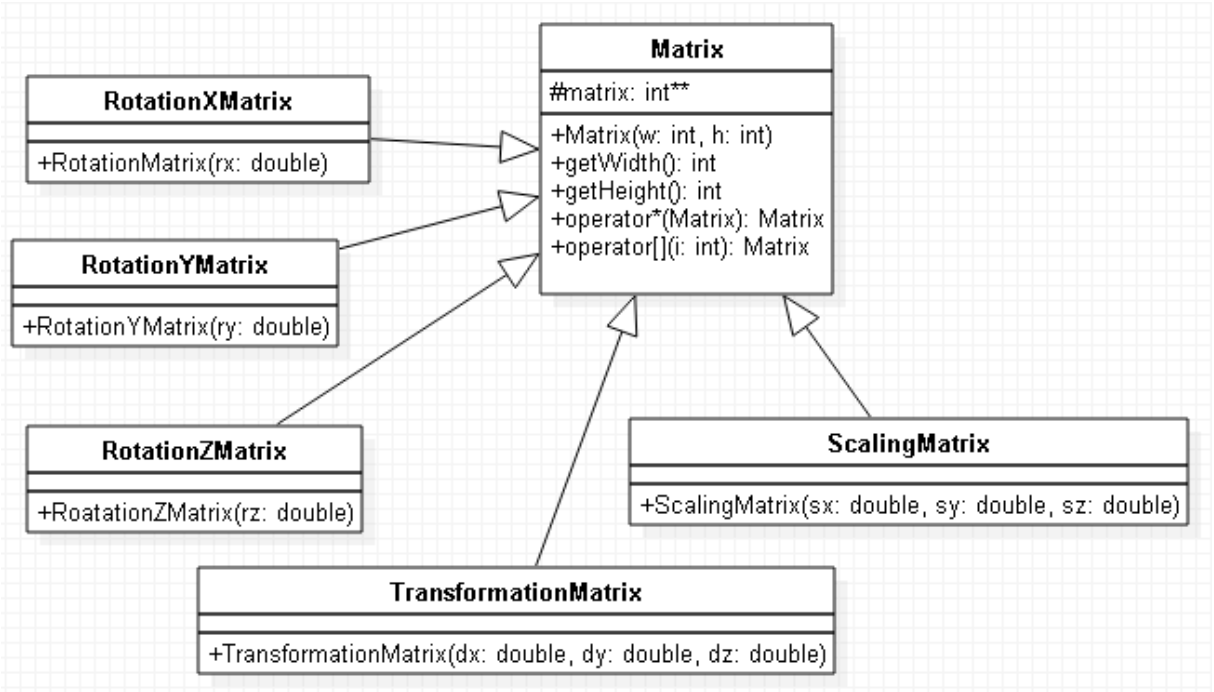
$$M = \begin{bmatrix} 1 & 0 & 0 & 12 \\ 0 & 1 & 0 & 12 \\ 0 & 0 & 1 & 12 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\frac{\pi}{3}) & 0 & sin(\frac{\pi}{3}) & 5 \\ 0 & 1 & 0 & 0 \\ -sin(\frac{\pi}{3}) & 0 & cos(\frac{\pi}{3}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P' = M * P$$

M jest macierzą wypadkową. **Warto zauważyć, że operacje które mają zostać wykonane na początku, powinny znaleźć się najbardziej po prawej stronie** (ponieważ znajdą się najbliżej macierzy P, czyli zmodyfikują ją jako pierwsze, co widać powyżej). Zatem tworząc macierz *M*, kolejne operacje domnaża się od lewej strony.

W programie utworzyłem superklasę *Transformable* oznaczającą, że dany obiekt może być transformowany poprzez podanie mu macierzy transformacji. Każda klasa dziedzicząca po tej klasie musi implementować metodę *transform(Matrix* m)*. O ile programista niczego nie zepsuł, macierz ta zawsze będzie czterowymiarowa. Po superklasie Transformable dziedziczą klasy *Point* i *Gameobject*. Jak już sotało wspomniane, modyfikacja obiektu to tak naprawdę modyfikacja współrzędnych punktów, z których się składa. Dlatego wywołanie metody *transform(Matrix* m)* obiektu *Gameobject* powoduje wywołanie metody *transform(Matrix* m)* punktów z których się składa. Diagram klas można zobaczyć w punkcie 5.9.

W pliku math/matrix.hpp znajduje się deklaracje klasy bazowej Matrix i klas pochodnych. Klasy pochodne utworzyłem po to, żeby tworzenie i inicjowanie wartościami konkretnych macierzy elementarnych było szybkie i sprawne.



Przykład z poprzedniego punktu wyglądały tak:

```
Point p(x, y, z);
Matrix* m1 = new ScalingMatrix(3.0, 3.0, 3.0);
Matrix* m2 = new TransformationMatrix(-3.0, 0.0, 0.0);
Matrix* m3 = new RotationYMatrix(PI / 3.0);
Matrix* m4 = new TransformationMatrix(12.0, 12.0, 12.0);
Matrix* m = m4 * m3 * m2 * m1;
p.transform(m);
// delete
```

5.6. Główna klasa projektu - „Scene”

Klasa *Scene* przechowuje obiekty sceny w tablicy dynamicznej. Główną metodą jest *render()* generująca obraz jako dwuwymiarową tablicę kolorów. Taką tablicę można narysować na ekranie statyczną metodą klasy *Drawer*. W konstruktorze tworzony jest model kamery w postaci dwóch wektorów jednostkowych: *up*, *right*, położenia w przestrzeni i rozdzielczości X i Y. W konstruktorze tworzone jest też jedyne źródło światła. Nie byłoby problemem utworzenie większej liczby świateł, **jednakże metoda Whitteda narzuca ograniczenie, że muszą to być źródła punktowe**. Klasa zawiera cztery wygodne `typedefy`, które stosowałem w projekcie:

```
typedef Scene&              io; // referencja do odczytu i zapisu
typedef const Scene&        i;  // referencja tylko do odczytu
typedef Scene*              p;  // wskaźnik do odczytu i zapisu
typedef const Scene*        cp; // wskaźnik tylko do odczytu
```

W ten sposób zamiast pisać: `void foo(const Scene& s)` mogę napisać: `void foo(Scene::i s)`. Metoda *render()* jest bardzo prosta i wygląda tak:

```
void Scene::render(Image::io image) {
	camera.render(*this, image);
}
```

Zatem scena nie wie jak wyrenderować obraz, deleguje to zadanie do kamery**. Delegowaniem zadań zachowałem zasadę pojedynczej odpowiedzialności** – klasa *Scene* zarządza tylko obiektami na scenie, a wszelkie bardziej wyrafinowane zadania zleca swoim „podwładnym”. Chcąc zmienić sposób przechowywania obiektów – modyfikujemy klasę *Scene*. Chcąc zmienić sposób renderingu – modyfikujemy klasę *Camera*. **Czyli klasy mają tylko jeden powód do zmian.**

5.7. Klasa śledząca promienie - „Ray”

Każdy obiekt klasy Ray symbolizuje promień wystrzelony z określonego piksela obrazu, którego kolor chcemy obliczyć. Klasa implementuje metodę *nearestHit()* zwracającą informacje o najbliższym zderzeniu. W celach optymalizacji, kolizje są obliczone dla obiektów znajdujących się w pobliżu promienia (zwróconych metodą *nearbyObjects()* klasy *Scene*). Na razie zwraca ona wszystkie obiekty, w przyszłości można zastosować bardziej skomplikowaną strukturę, np. KD-drzewo dla obiektów sceny. Metoda *reflectedSpeculary()* zwraca promień odbity zwierciadlanie w podanym punkcie, natomiast *transmitted()* – transmisję kierunkową z refrakcją (zobacz pkt. 3.5 – efekty wizualne). Klasa jest dość prosta, nie zawiera żadnych sztuczek i mam nadzieję, że kod jest samo komentujący się, dlatego nie będę tutaj pisał więcej.

5.8. Informacje o kolizji – klasa „Hit”

Klasa zawiera informacje o kolizji promienia z obiektem. Nie jest ważne, jaki konkretnie jest to obiekt – klasa jest uzależniona tylko od interfejsu *GameObject*, który posiada metody umożliwiające pobranie potrzebnych danych (takich jak obliczenie odległości kolizji, pobranie wektora normalnego). Jeśli

materiał z którego budowany jest obiekt posiada tekstury, obliczane są współrzędne tego teksela na teksturze (jeśli nie, takie obliczenie nie jest wykonywane w celach optymalizacyjnych). Jeśli materiał posiada odwzorowanie wektorów normalnych, obliczane są jednostkowe wektory pomocnicze Ex i Ez, wyznaczające powierzchnię do której prostopadły jest wektor normalny (Ex, Ey, Ez tworzą osie układu współrzędnych, gdzie Ey to wektor normalny). Względem tego układu wyznaczane jest przesunięcie wektora normalnego zapisane w teksturze:

- składowa *czerwona* – składowa X wektora
- składowa *zielona* – składowa Y wektora
- składowa *niebieska* – składowa Z wektora, wartości składowych czerwonej i zielonej wybierane są najpierw, wartość składowej niebieskiej dobrana jest tak, by wektor pozostał wektorem jednostkowym.

Jeśli *czerw*.= *ziel*. = 0, to zmodyfikowany wektor normalny nie różni się od pierwotnego wektora normalnego. Technika mapowania wektorów normalnych pozwala dobrze symulować jedynie niewielkie nierówności powierzchni, dlatego składowe *czerwona* i *zielona* są niewielkie – stąd w teksturach wektorów normalnych dominuje kolor niebieski.

5.9. Reprezentacja obiektu na scenie, czyli interfejs „Gameobject” i klasy dziedziczące

Obiekty różnych kształtów posiadają wiele wspólnych cech, stąd naturalnie powstała klasa bazowa *GameObject*. Reprezentuje ona interfejs, jaki musi implementować każdy typ obiektu. Typy obiektów to klasy dziedziczące po *GameObject*: *Triangle* i *Sphere*. Są to typy proste, jednak w przyszłości planuję dodać typ obiektu złożony z wielu trójkątów (czyli strinagulowany, tak jak zostało to opisane w pkt. 2). Każdy typ obiektu musi implementować następujące metody:

- ***getNormal(const Point& p***) – zwraca wektor normalny do powierzchni w punkcie p. Dla różnych typów obiektów, sposób obliczenia tego wektora może być inny (można to zobaczyć w kodzie na przykładzie klas *Triangle* i *Sphere*).
- ***intersectDistance(const Ray& ray)*** – zwraca odległość po jakiej promień *ray* trafi w ten obiekt (jeśli nie trafi, metoda zwraca Hit::INFINITY, czyli nieskończoną odległość zderzenia). Metoda używana przy śledzeniu promienia do określenia z którym obiektem promień zderzy się jako pierwszy (ze wszystkich obiektów wybierany jest ten o najmniejszej odległości zderzenia). Metoda spełnia warunek 90-10, to znaczy należy do tych 10% kodu które są wykonywane przez 90% czasu działania programu. Stąd efektywność tej metody ma duży wpływ na czas renderingu.
- ***mapSurfaceToTexture(const Point& p)*** – odwzorowuje punkt na powierzchni obiektu, w punkt na powierzchni tekstury (texel[8]). Trudność polega na tym, że powierzchnia obiektu jest trójwymiarowa, stąd punkt o 3 współrzędnych należy przekształcić w punkt o 2 współrzędnych. Tak jakby powierzchnię obiektu próbować rozłożyć na 2 wymiarowej powierzchni – nie da się tego zrobić bez pewnych zniekształceń (porównaj „mapa świata”).
    - Implementacja Sphere – każdy punkt na powierzchni kuli traktowany jest jako punkt o współrzędnych (w, l), gdzie: w – szerokość geograficzna $\in < -\frac{\pi}{2}; +\frac{\pi}{2} >$, l – długość geograficzna $\in < 0; 2\pi >$. Wyjątkiem są południki – ich długość geograficzna jest

[8] Texel to „texture element” (ang. element tekstury), podobnie jak pixel to „picture element” (ang. element obrazu).

nieokreślona, natomiast szerokość to $+\frac{\pi}{2}$ albo $-\frac{\pi}{2}$. Następnie współrzędne w i l są konwertowane tak by mieściły się w przedziale <0, 1>

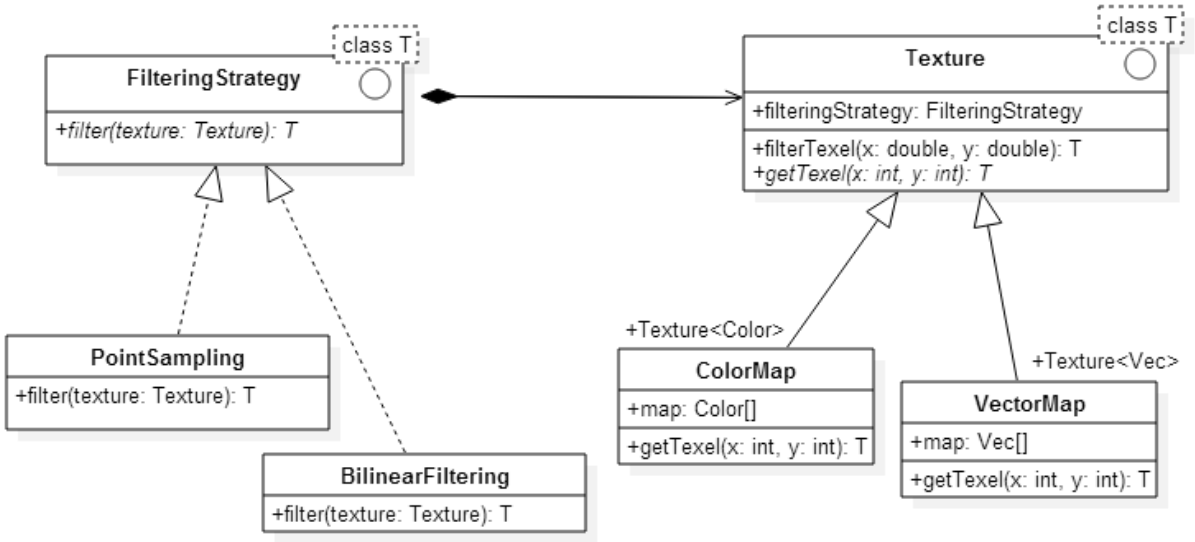
- Implementacja Triangle – gdy promień trafia punkt P na trójkącie, obliczane są jego współrzędne barycentryczne. Chociaż tymi współrzędnymi jest trójka liczb, wystarczy pamiętać tylko dwie pierwsze a i b, ponieważ spełniają one równanie $a + b + c = 1$. Współrzędna barycentryczne a mówi nam tak jakby „jak bardzo punkt P oddalony jest od wierzchołka A", analogicznie b. Mając a i b policzone względem wierzchołków trójkąta na scenie A, B, C, możemy je zastosować by obliczyć współrzędne wierzchołka P' na teksturze (względem punktów T1, T2, T3, bo każdy trójkąt posiada trójkę punktów określających położenie jego wierzchołków na teksturze).

5.10. Filtrowanie tekstur z użyciem interfejsów „FilteringStrategy" i „Texture".

Na początku miałem klasę ColorMap (reprezentuje teksturę obiektu) która posiadała metodę filtrującą kolor teksela. W zależności od przekazanego argumentu, w instrukcji *if* dokonywany był wybór – czy filtrować punktowo, czy dwuliniowo (filtrowanie zostało opisane w pkt. 3.4). Następnie postanowiłem dodać klasę VectorMap (reprezentuje zmodyfikowane wektory normalne obiektu) w której potrzebowałem takiej samej metody filtrującej. Wystarczyłoby zrobić „kopiuj-wklej" i zmienić typy zmiennych, jednak kiedyś usłyszałem, że w programowaniu nie istnieje coś takiego jak „kopiuj-wklej". **Poza tym byłoby to naruszenie zasady DRY**[9].

Wyciągnąłem więc elementy wspólne *ColorMap* i *VectorMap* do klasy bazowej *Texture*. Można ją rozumieć tak: *ColorMap* w teksturze (obrazie) zwiera informacje o kolorze obiektu, natomiast *VectorMap* zawiera w teksturze zmodyfikowane wektory normalne.

Filtrowanie punktowe i dwuliniowe są różnymi sposobami wykonania tej samej czynności – **zastosowałem więc wzorzec projektowy Strategia**. Klasy *PointSampling* i *BilinearFiltering* implementują abstrakcyjną metodą *filter(const Texture* texture)* interfejsu *FilteringStrategy*.



W ten sposób kod odpowiedzialny za filtrowanie został napisany tylko raz dla każdego z typów. Metoda *filter(const Texture* texture)* spodziewa się obiektu typu *Texture* (który jest interfejsem) - to dobrze,

[9] DRY (ang. Don't Repeat Yourself = Nie Powtarzaj Się) zasada mówiąca o tym, aby eliminować w kodzie wszelkie powtórki np. tworząc funkcje.

że metoda jest powiązana z interfejsem, a nie konkretną klasą (jak np. *ColorMap*). Korzyści z takiego rozwiązania są też przy rozwoju aplikacji. Chcąc dodać nowy rodzaj tekstury (przechowujący w obrazie jakieś inne informacje o obiekcie) wystarczy dziedziczyć po *Texture*, filtrowanie jest gotowe do użycia. Chcąc dodać nowy rodzaj filtrowania (np. trójliniowe, anizotropowe) wystarczy dziedziczyć po *FilteringStrategy*. W żadnym przypadku nie trzeba modyfikować istniejących klas – **jest to zasada otwarte-zamknięte**[10] (kod jest jest otwarty na rozbudowę, lecz zamknięty na modyfikacje).

5.11. Ładowanie tekstur z użyciem zewnętrznej biblioteki *Corona*

Do ładowania i przetwarzania tekstur z plików .jpg, .png itp. została użyta bardzo lekka biblioteka *Corona*, składająca się z kilkunastu plików, o łącznej długości kodu nie przekraczającej 2500 linii. Stanowi ona cienką warstwę nad bibliotekami języka C do operacji na plikach .jpg, .png itp. Dostarcza minimalistyczne API:

- *Corona::loadFile(string)* – otwiera plik o podanej nazwie
- *Corona:: `OpenImage` (File*, PixelFormat)* – traktuje plik jako uchwyt do obrazu, próbuje utworzyć mapę pikseli (tablicę bajtów) o zadanym formacie (jeśli format obrazu jest innych, dokonuje konwersji „najbliższej oryginałowi”).
- I kilka innych, pełne API można zobaczyć w pliku *Corona.h*

Biblioteka ta posiada dokładnie to czego potrzebowałem i nic ponad to. Wiele się nauczyłem przeglądając jej kod źródłowy i dołączony tutorial.

Fragment z klasy *ColorMap* otwierający plik z teksturą:

```
corona::File* file = corona::OpenFile(fname.c_str(), false);
corona::Image* image = corona::OpenImage(file, corona::PF_R8G8B8);
```

Określając format w jakim chcemy mieć obraz `corona::PF_R8G8B8` mamy pewność, że każdy piksel tablicy obrazu składa się z 3 bajtów; każdy bajt to składowa: czerwona, zielona, niebieska. Zakładając że plik istnieje, przejście po tablicy pikseli może wyglądać tak:

```
byte* pixels = (byte*)image->getPixels();
for (int y = 0; y < yResolution; ++y) {
	for (int x = 0; x < xResolution; ++x) {
		byte red = *pixels++;
		byte green = *pixels++;
		byte blue = *pixels++;
		// tutaj kod
	}
}
```

6. Testowanie

Program składa się z modułów, które mogą być testowane niezależnie. Zastosowałem wstępującą metodę testowania. Zaletą tej metody jest prostota (w przeciwieństwie do metody zstępującej nie trzeba pisać zaślepek) i możliwość szybkiego wykrycia błędów. Wadą jest możliwość późnego wykrycia poważnych błędów konstrukcyjnych (dopiero na etapie łączenia większych modułów). Pisanie programu renderującego metodą Whitteda jest na tyle standardowym tematem, że nie spodziewałem się takich błędów, dlatego wybrałem tą metodę testowania.

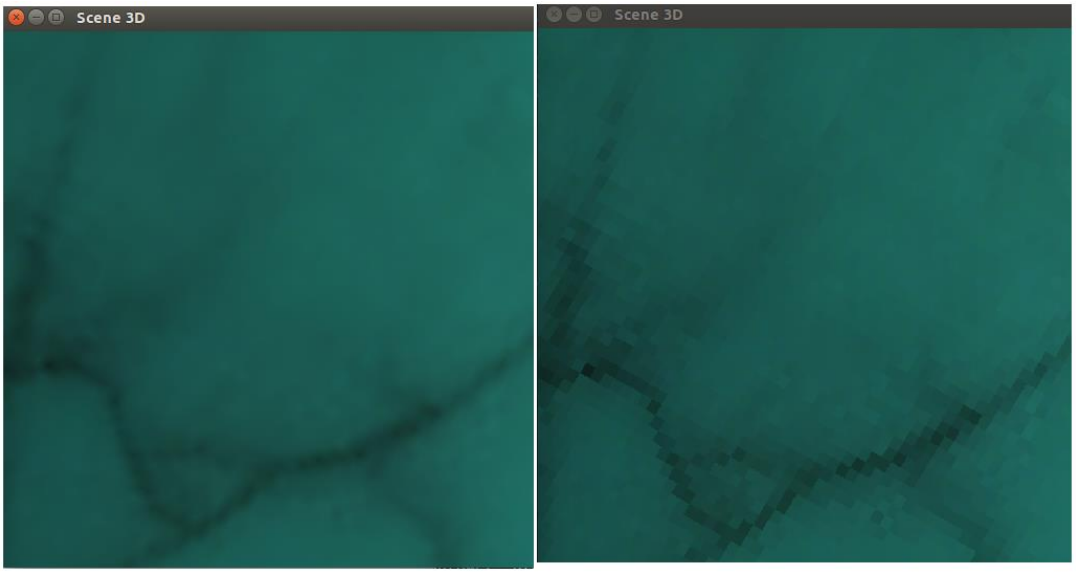
[10] Zasada „otwarte – zamknięte” - kod jest otwarty na rozbudowę, lecz zamknięty na modyfikację. Rozbudowa programu sprowadza się do dodania nowych klas, bez konieczności modyfikowania już istniejących.

W katalogu *utest* znajdują się wszystkie testy jednostkowe. Zostały napisane przede wszystkim dla klas matematycznych i kontenerów – łatwo w nich o błąd. Gdyby opuścić etap testowania, program na 99% nie zadziałałby prawidłowo po pierwszym uruchomieniu, a znalezienie przyczyny (zapewne wielu) w programie liczącym kilka tysięcy linii, byłoby skrajnie trudne. Dlatego chciałem mieć pewność, że te podstawowe struktury są napisane prawidłowo**. Dopiero wtedy przystąpiłem do pisania właściwej części programu**, czyli algorytmu śledzenia promieni i struktur z nim związanych. Testy uruchamia się poprzez dołączenie wybranego pliku nagłówkowego i uruchomienie metody *run()*. Testowanie na pewno oszczędziło mi wiele czasu.

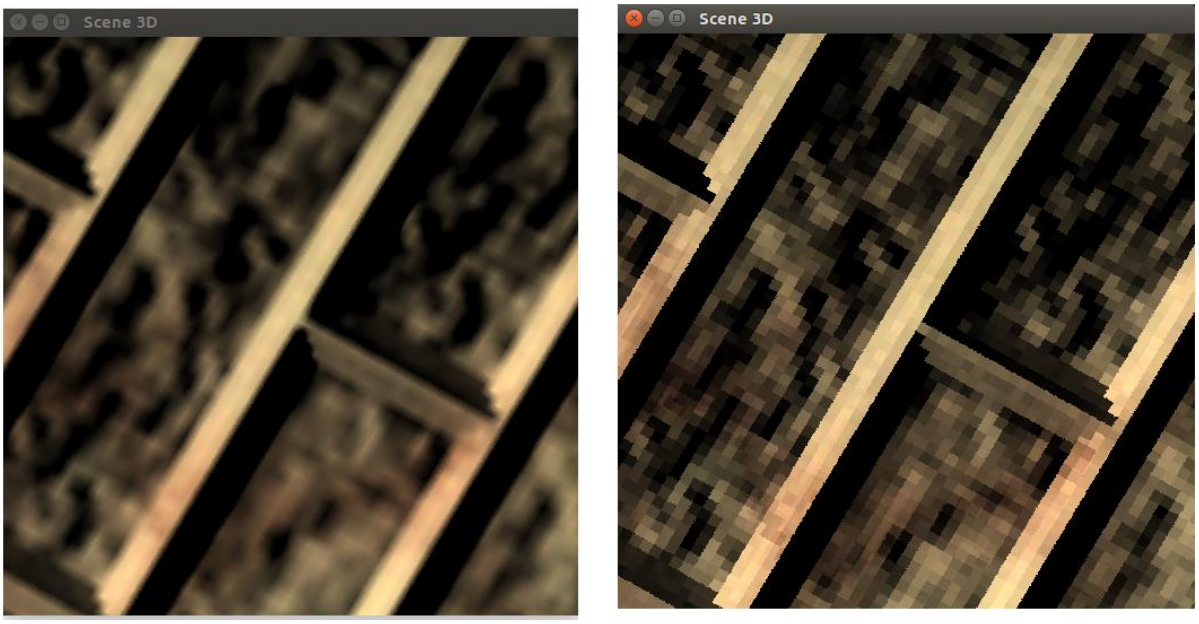
6.1. Filtrowanie punktowe kontra filtrowanie dwuliniowe

To że filtrowanie punktowe daje najgorsze rezultaty można przeczytać w dowolnym artykule o grach komputerowym. Różnica ta jest szczególnie widoczna dla obiektów znajdujących się blisko kamery. Poniżej zestawienie. Filtrowanie dwuliniowe (po lewej) vs. próbkowanie punktowe (po prawej):

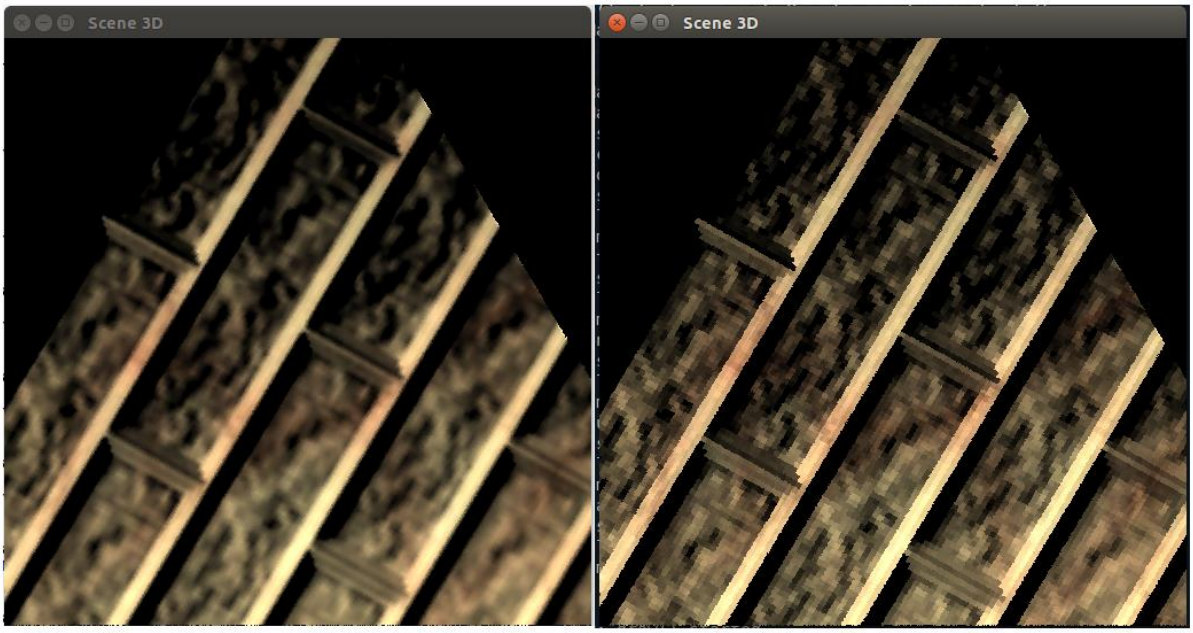
a) tylko tekstura



b) tekstura + wektory normalne (blisko)



c) tekstura + wektory normalne (daleko)

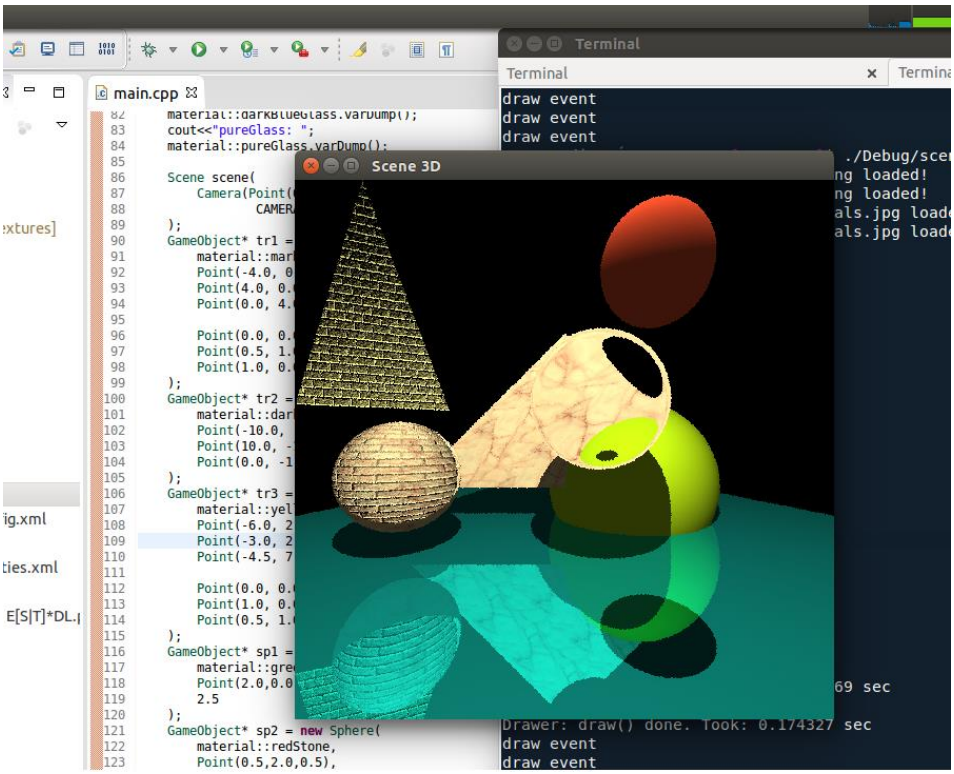


Wszystkie obrazy zostały wykonane bez antyaliasingu, czyli jedna próbka (promień) na piksel.

6.2. Czas renderingu dla różnej jakości docelowego obrazu

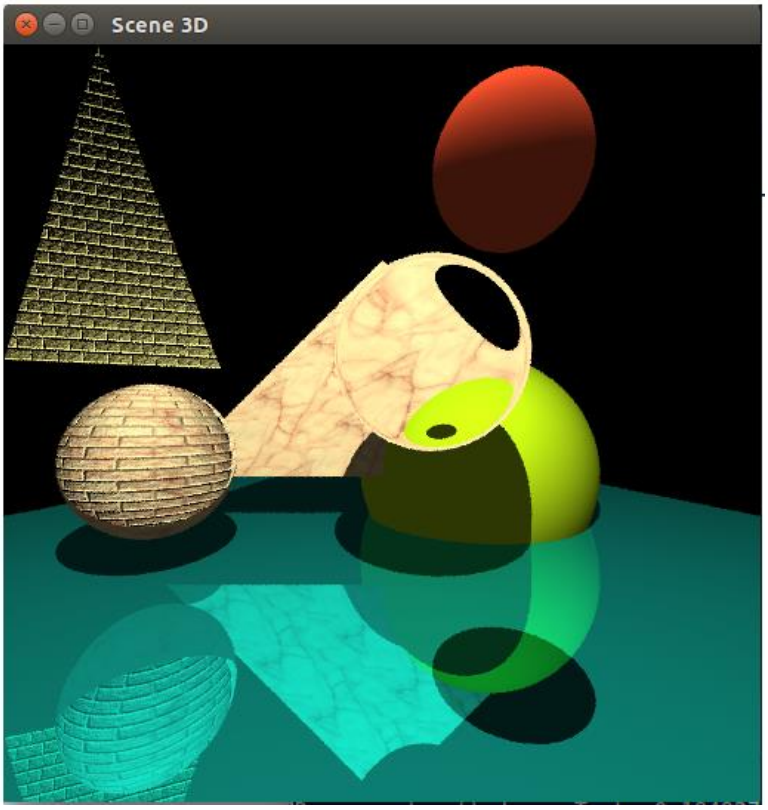
Rendering wykonywany był na laptopie Intel i3-4000 2.40 GHz, 8 GB RAM, z systemem 64 bitowy Linux Ubuntu 14.04 LTS. Tak naprawdę największe znaczenie ma procesor. Wszystkie obrazy mają rozdzielczość 512 x 512 px.

a) 1 SPP[11], czas: 4.87 sek.

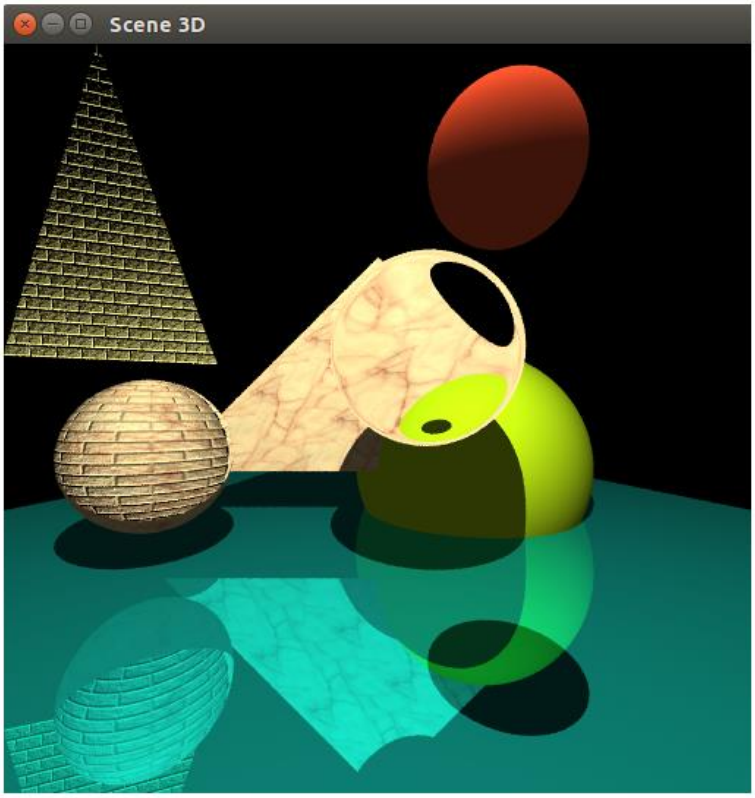


[11] SPP - (ang. Samples Per Pixel) – próbek (śledzonych promieni) na piksel, więcej o tym parametrze można przeczytać w punkcie 4.

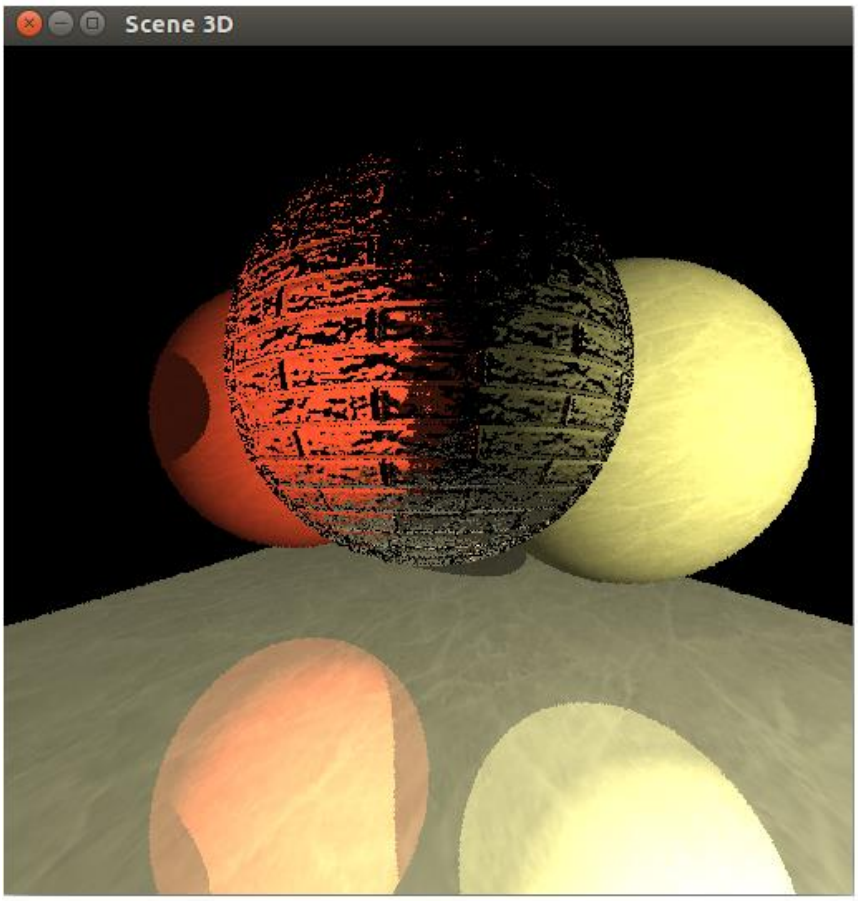
b) 4 SPP, czas: 12 sek.

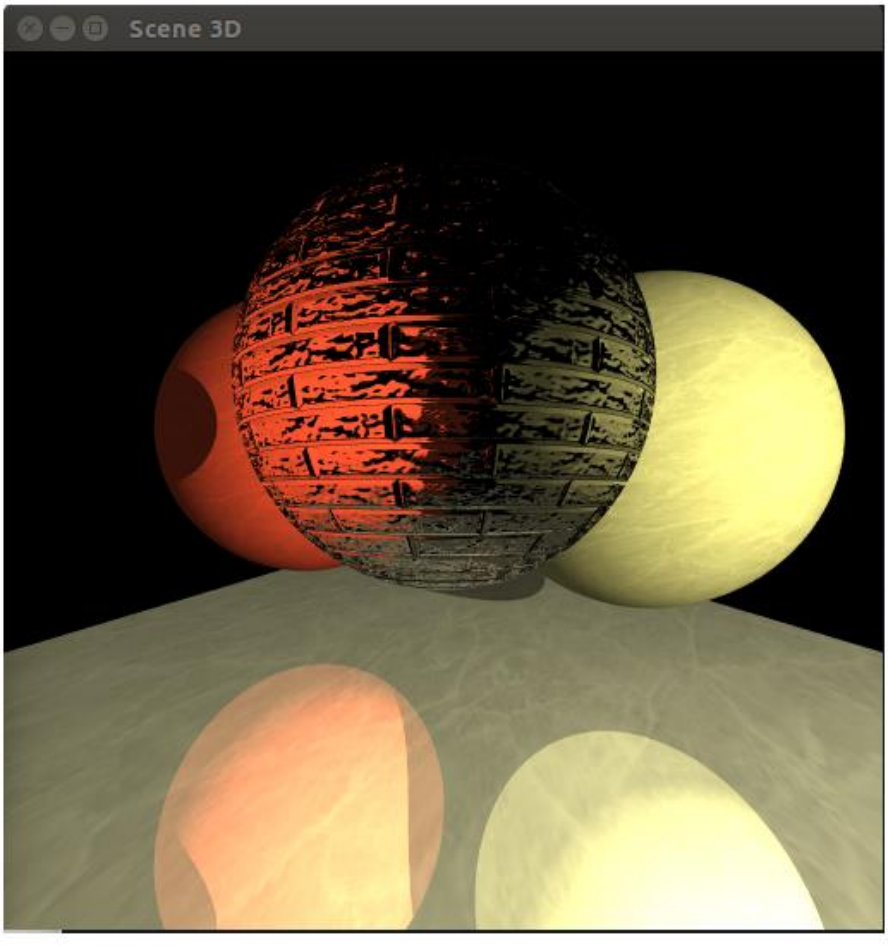


c) 12 SPP, czas: 58 sek.



Poniżej porównanie obrazu najniższej jakości, z najwyższą osiągniętą jakością (więcej SPP nie renderowałem). Obraz gorszy to 1 SPP, czas: 5 sek. Obraz lepszy to 48 SPP, czas: 140 sek (ponad 2 minuty. Czytelnikowi pozostawiam ocenę który jest który.


Scene 3D


Scene 3D

7. Wnioski

Udało się zrealizować główną funkcjonalność projektu – rendering sceny. Ponadto mamy różne rodzaje obiektów, materiały posiadają różne właściwości, program obsługuje tekstury i mapowanie wektorów normalnych, które jest częstą techniką szybkiego symulowania drobnych nierówności. Program generuje obraz zadowalającej jakości w krótkim czasie. Kosztem czasu, możemy też wygenerować obraz (moim zdaniem) bardzo dobrej jakości. W projekcie udało się wykorzystać wzorzec projektowy Strategia, oraz udało mi się przestrzegać dobre zasady programowania obiektowego (a przynajmniej te które znam: DRY, „otwarty-zamknięty", projektowanie interfejsów a nie konkretnych klas, odwrócenia zależności, pojedynczej odpowiedzialności klasy, delegowania zadań). Pracę nad projektem ułatwiał system wersjonowania Git w połączeniu z darmowym repozytorium umieszczonym na bitbucket.org Takie rozwiązanie zastosowałem już nie pierwszy raz, ma ono wiele zalet. Po pierwsze, nie ma znaczenia, na jakim komputerze pracuję nad projektem, a po drugie, wszelkie błędy mogę łatwo wycofać.

Nie udało się zrealizować GUI, które umożliwiałoby przemieszczanie i modyfikowanie obiektów na scenie w czasie działania programu. Takie było początkowe założenie, które musiałem jednak zweryfikować po testach programu. Użytkownicy są zazwyczaj przyzwyczajeni do natychmiastowych odpowiedzi programu. Tutaj, skoro rendering najniższej jakości trwa co najmniej 3 sekundy, po każdej interakcji użytkownik musiałby czekać te kilka sekund na odpowiedź. Moim zdaniem takie GUI nie wnosi dodatkowej wartości do projektu i nie opłaca się poświęcać na nie czasu.

Możliwy rozwój projektu:

- W przyszłości planuję najczęściej wykonywane metody (10% kodu wykonywane jest przez 90% programu) zamienić na metody typu `inline` i porównać czasy wykonania.
- Aby przyspieszyć rendering dla dużych scen (wiele obiektów) należy zminimalizować liczbę obiektów, dla których liczone są kolizje względem aktualnego promienia. Można to osiągnąć przechowując obiekty w bardziej skomplikowanej strukturze, jak KD-tree. Jak na razie nie było to problemem, ponieważ renderowanie sceny zawierały mniej niż 10 obiektów.
- Można również zamiast algorytmu śledzenia promieni metodą Whitteda użyć algorytmu śledzenia ścieżek (*monte carlo path tracing*), który umożliwia modelowania nie punktowych źródeł światła i ogólnie bardziej realistycznego obrazu.

8. Bibliografia

Cormen, T. H., Leiserson, C. E., Rivest, R. L. i Stein, C. (brak daty). *Wprowadzenie do algorytmów.* (K. Diks, M. Jurdziński, A. Malinowski, D. Rytter i W. Rytter, Tłumacze) Warszawa: Wydawnictwa Naukowo - Techniczne.

Eric Freeman, E. F. (2011). *Rusz głową! Wzorce projektowe.* (P. K. Grzegorz Kowalczyk, Tłum.) Gliwice: Helion.

Kraus, R. (2015, kwiecień 15). *Własny silnik graficzny*. Pobrano z lokalizacji Wrocławski Portal Informatyczny: https://informatyka.wroc.pl/node/415

Martin, R. C. (2010). *Czysty kod. Podręcznik dobrego programisty.* luty: 19.