Wydział Elektryczny Politechniki Warszawskiej

Grafika Komputerowa

Coffeecam - wirtualna kamera

Autorzy:
Bartosz Pieńkowski
Barnaba Turek

15 grudnia 2011

Streszczenie

Celem serii projektów z przedmiotu Grafika Komputerowa jest utworzenie prostego silnika graficznego (i zapoznanie się od strony graficznej z podstawowymi zadaniami grafiki komputerowej, takimi jak rzutowanie, eliminacja elementów zasłoniętych i cieniowanie).

Zdecydowaliśmy się zaimplementować projekt w języku coffeescript.

Pierwsza część projektu polega na utworzeniu wirtualnej kamery. Kamera ma obsługiwać zmianę pozycji, ogniskowej i kąta, w którym jest skierowana. Druga część projektu polegała na dodaniu algorytmu zasłaniania obiektów. Zastosowaliśmy Algorytm malarski. W trzeciej części projektu dodaliśmy cieniowanie metodą Blinna-Phonga.

1 Wirtualna kamera

1.1 Struktura projektu

Projekt podzielony jest na pliki .coffee, z których każdy zostaje skomplikowany do pliku .js (javascript). Zawartość każdego pliku wynikowego jest opakowana w anonimową funkcję, aby zniwelować ryzyko konfliktu nazw z innymi skryptami.

Zmienne i funkcje potrzebne do komunikacji pomiędzy modułami dostępne są za pomocą globalnej zmiennej coffecam.

Nie wszystkie obiekty przedstawione na diagramie 1 są w istocie klasami - z wyjątkiem klas **Camera**, **Controller**, **Polygon**, **Sphere**. nie potrzebowaliśmy przechowywać żadnego stanu, a tylko logicznie podzielić udostępniane funkcje.

camera.coffee W tym pliku zdefiniowana jest klasa **Camera**. Obiekty tej klasy reprezentują wirtualną kamerę. Udostępniają takie metody jak **move()**, **rotateX()**, **rotateY()**, **rotateZ()**, **update()**. Stanem, przechowywanym przez obiekty tej klasy jest np. transformacja potrzebna do zamienienia koordynatów z układu współrzędnych sceny na układ współrzędnych kamery. Ponadto obiekty tej klasy przechowują informacje i udostępniają metody dotyczące źródła światła.

common.coffee W tym pliku zawierają się funkcje pomocnicze, używane w pozostałych plikach, takie jak **normalize** i **point**.

controller.coffee obiekt tej klasy reprezentuje kontroler użytkownika. Pozwala on w przejrzysty sposób powiązać zdarzenia związane z wciskaniem przez użytkownika klawiszy z odpowiadającymi im akcjami.

scene.coffee W tym pliku zdefiniowane są obiekty, które możemy narysować, takie jak **Polygon** i **Sphere**. Obiekty te reprezentują wspólny, prosty interfejs złożony z metod **draw()** i **transform()**. Plik udostępnia także funkcje pozwalające w czytelny sposób zbudować prostopadłościan z wielokątów, oraz funkcje odpowiadające za konstrukcję sceny.

site.coffee Plik odpowiada za połączenie pozostałych plików, inicjalizacje potrzebnych obiektów i zapewnienie im zależności (zgodnie ze wzorcem *dependency injection*.

1.2 Działanie kamery

Obiekt kamery przechowuje macierz transformacji, początkowo ustawioną na macierz jednostkową (z dokładnością do znaku, ze względu na skrętności układów)[1]. Macierz transformacji obliczana jest na nowo po każdym ruchu kamery. Obliczenie nowej macierzy polega na pomnożeniu aktualnej macierzy przez żądaną transformacje.

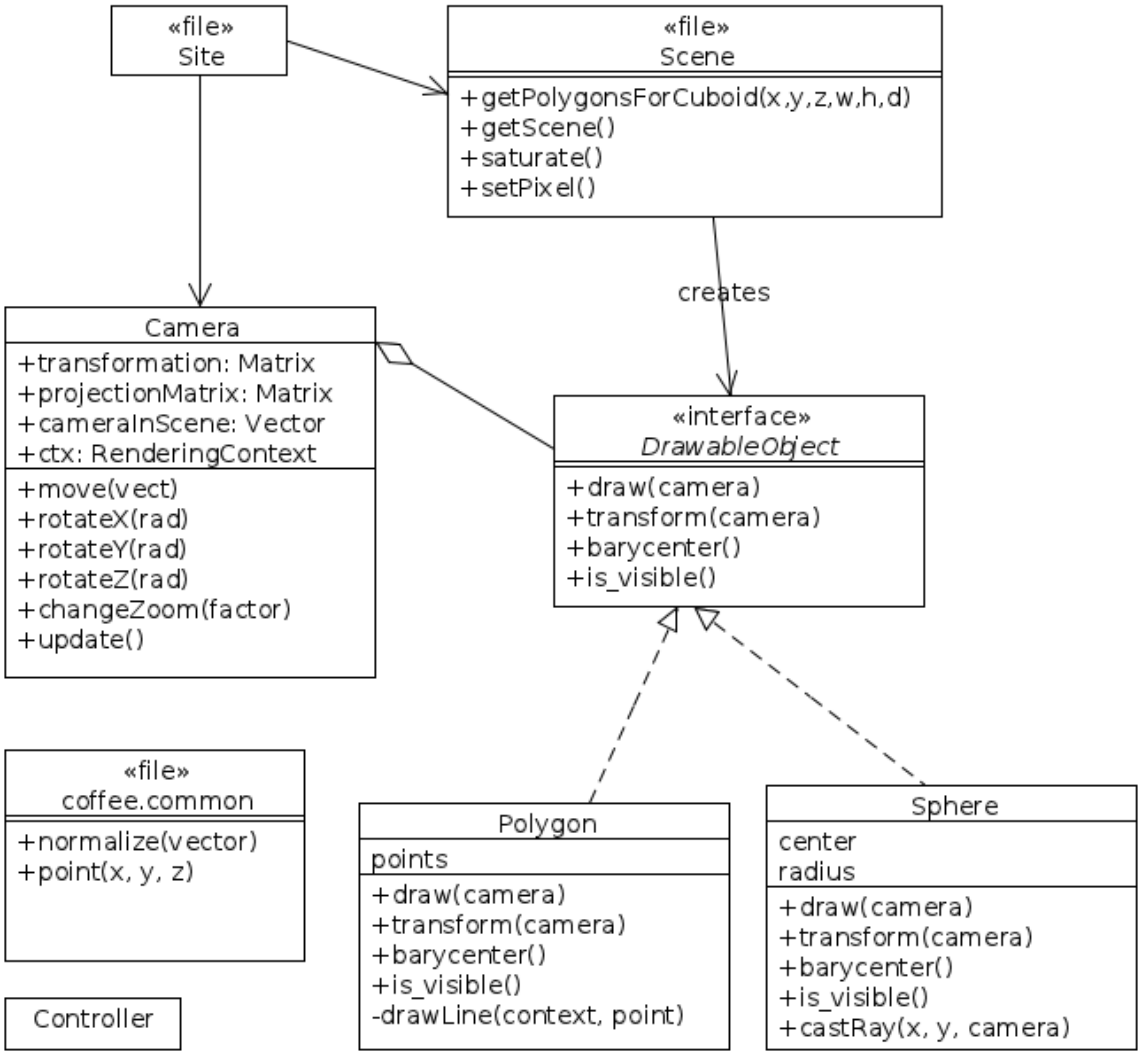
W pliku kamery stworzone są funkcje pomocnicze, pozwalające wyliczyć macierz konkretnej transformacji. Zastosowaliśmy wzorzec projektowy *dekorator*, aby dodać do funkcji obliczających macierze inne zachowania – i w rezultacie otrzymać metory pozwalające faktycznie poruszyć kamerą.

Te zachowania to pomnożenie aktualnej macierzy translacji przez dekorowaną macierz, obliczenie nowej pozycji kamery w koordynatach sceny (co przyspiesza działanie algorytmu malarskiego) oraz obliczenie położenia źródła światła w układzie współrzędnych kamery (co przyspiesza działanie algorytmu *Blinna-Phonga*). Na końcu dekorator wykonuje funkcję update, co pozwala uprościć obsługę kamery do wywołania jednej funkcji na jedną akcję.

Dostępne transformacje to przesunięcia i obroty kamery (Transformacje utworzylismy na podstawie wykładu[2]).

Po wykonaniu każdej operacji na kamerze wywoływana jest metoda **update**. Metoda ta polega przetransformowaniu, posortowaniu pod względem odległości od kamery a następnie narysowaniu wszystkich elementów na ekranie.

Rysunek 1: Struktura projektu



Macierz transformacji mnożona jest przez macierz rzutowania (Obliczaną kiedy kamera jest tworzona i przy zmianie ogniskowej)[1] tylko raz. Każda z rysowanych obiektów sam wie, jak się narysować. W przypadku klasy **Polygon**, która nie wymaga cieniowania, transformacja polega na prostym pomnożeniu wszystkich punktów przez macierz transformacji i rzutowania. W przypadku klasy **Sphere**, jest to nieco bardziej złożone. Obliczane są wtedy wszystkie zmienne niezbędne, do narysowania sfery (z wyjątkiem wartości koloru dla samych punktów).

Na tak przygotowanych obiektach wywoływana jest metoda draw, która rysuje je w oknie przeglądarki.

1.3 Scena

Obiekty, które kamera wyświetla to wielokąty i sfery. Wczesne wersje programu obsługiwały tylko wielokąty, stąd rozbudowany sposób budowania sceny ("ulice" i "domki") dotyczy tylko ich. Sfery zostały dodane w ostatniej iteracji projektu.

1.4 Polygon

Wielokąt to podstawowy obiekt należący do sceny. Klasa udostępnia z grubsza ten sam interfejs, co **Sphere** (różnicą jest konstruktor). Program zakłada, że kolejne punkty opisujące wielokąt podane są w sposób uporządkowany.

Wielokąty same w sobie nie są używane do tworzenia sceny. Zamiast nich używana jest funkcja, która zwraca od razu tablicę wielokątów, reprezentującą prostopadłościan – **get0olygonsForCuboid** – pewnego rodzaju fabryka obiektów.

1.5 Sphere

Sfera to obiekt dodany w trzeciej iteracji. Sfera została dodana, ponieważ jest dużo ciekawszym obiektem do cieniowania niż wielokąt.

Obiekty tej klasy implementują ten sam interfejs, co wielokąty, jednak ich zasada działania jest inna.

Oprócz parametrów opisujących zajmowaną przestrzeń, obiekty tego typu inicjuje się 7 dodatkowymi parametrami.

diffusePower Poziom tego współczynnika opisuje jasność związaną z rozproszonym odbiciem od sfery.

spectralPower Poziom tego współczynnika opisuje jasność związaną z kierunkowym odbiciem (czyli dotyczy bardziej błyszczących obiektów).

spectralHardness Poziom tego współczynnika określa, jak bardzo rozmyte jest odbicie kierunkowe. Bardzo wysokie poziomy tego współczynnika powodują, że uzyskujemy przestrzeń bliższą lustru - punktowe źródło światła będzie reprezentowane jako świecący punkt na powierzchni sfery.

ambientPower Poziom tego wspólczynnika odpowiada za oświetlenie tła, niezależne od kierunku i odległości, z której pada światło (ani od niczego innego.

Pozostałe trzy parametry opisują składowe odpowiadające konkretnym kolorom.

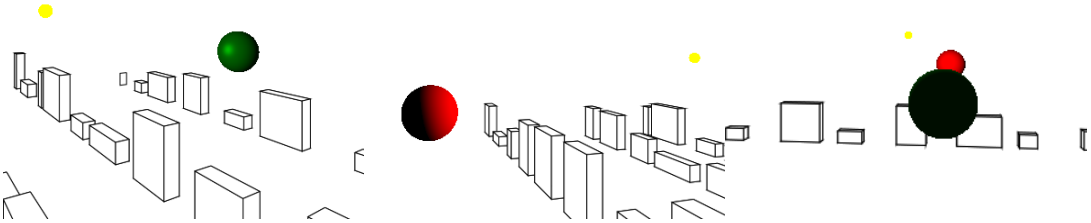
1.5.1 Cykl renderowania obiektu typu `Sphere`

Ponieważ w naszym przypadku sfery nie są opisane za pomocą wielokątów, tak jak to się najczęściej implementuje, sposób ich renderowania jest nieco bardziej złożony. Z drugiej strony rozwiązanie takie pozwoliło nam pominąć proces opisywania sfery za pomocą wielokątów.

Po wywołaniu metody *transform()* przeprowadzany jest szereg transformacji, które przygotowują sferę do rysowania. Korzystając z macierzy transformacji i rzutowania udostępnianych przez obiekt `Camera`, ustalane są współrzędne środka sfery na rzutni. Następnie ustalana jest długość promienia po zrzutowaniu.

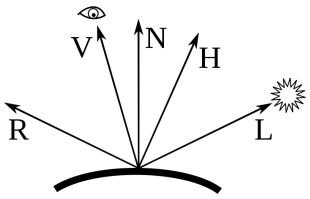
Za pomocą tych dwóch wartości wyliczany jest prostokąt, określający rysowaną sferę na rzutni. Współrzedna Z obu punktów charakterystycznych tego prostokąta określana jest na 1 - współrzędną graniczną, powyżej której rzutowane obiekty sa obcinane (bo znajdują się przed rzutnią). Po pomnożeniu odwrotności macierzy rzutowania przez uzyskane w ten sposób punkty uzyskiwane są punkty, które określają współrzędne prostokąta opisującego kulę po rzutowaniu.

Z wartości współrzednych po zrzutowaniu wiemy też, ile pikseli zajmie rzut sfery na rysowanym obrazie.



(a) Błyszczący obiekt (b) Matowy obiekt (c) Pod światło

Po wywołaniu metody `draw()`, przechodzimy przez punkt na rzutni odpowiadający każdemu z poszukiwanych pikseli. Dla każdego takiego punktu wyznaczamy prostą przechodzącą przez niego i środek układu współrzędnych (czyli położenie oka kamery).



Rysunek 2: Wektory biorące udział w obliczeniu jasności punktu w modelu Blinna-Phonga

Następnie rozwiązujemy równanie kwadratowe, by znaleźć punkt przecięcia tej prostej ze sferą. Dla tego punktu obliczane są wektory: normalny (N), kierunku światła (L) i kierunku patrzenia (V), oraz H, a na podstawie tych wektorów, zgodnie z algorytmem cieniowania Blinna-Phonga wyznaczane są kolory poszczególnych pikseli.

2 Eliminacja powierzchni zasłoniętych

Celem drugiego etapu projektu było zaimplementowanie wybranego algorytmu eliminacji powierzchni zasłoniętych.

Ze względu na prostotę sceny zdecydowaliśmy na algorytm malarski. Działanie tego algorytmu polega na rysowaniu obiektów w kolejności ustalonej przez ich odległość od kamery.

2.1 Implementacja

Algorytm nie zajął zbyt wiele miejsca i nie uznaliśmy za stosowne dodawać kolejnej klasy do obiektu. Dodaliśmy kod do klasy *Camera*.

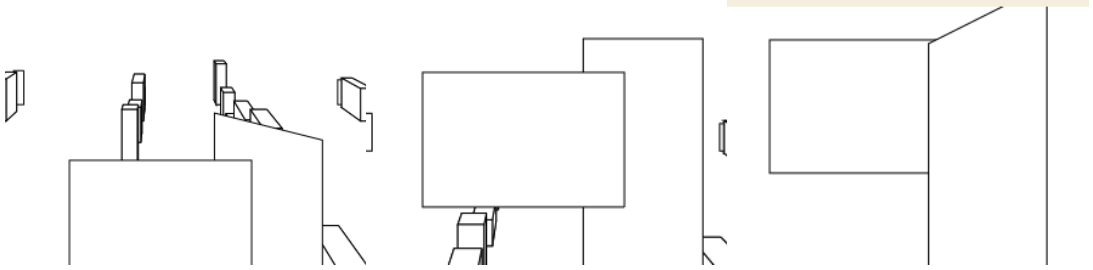
Działanie algorytmu składa się z trzech kroków. W pierwszym kroku obliczamy współrzędne kamery w układzie odniesienia sceny.

Ponieważ kamera zawsze znajduje się w punkcie

$$P_{cam} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

i w każdej klatce zna swoją pozycję względem początku układu współrzędnych sceny (daną macierzą transformacji `@transformation`), możemy łatwo obliczyć położenie kamery w układzie współrzędnych sceny.

Transformacja, której szukamy (transformująca położenie z układu kamery do układu sceny) jest transformacją odwrotną do `@transformation`, zatem współrzędne kamery w układzie współrzędnych



(a) Krok 1 (b) Krok 2 (c) Krok 3

Rysunek 3: Problem z tą implementacją. Kolejne kroki przedstawiają te same wielokąty, na które patrzymy z tego samego kierunku.

możemy policzyć w następujący sposób:

$$\mathtt{P}'_{\mathtt{cam}} = @\mathtt{transformation}^{-1} \times \mathtt{P}_{\mathtt{cam}}$$

Następnym krokiem jest obliczenie odległości środków ciężkości rysowanych wielokątów od punktu P'_{cam}. W tym celu obliczamy środek ciężkości wielokąta `barycenter`, gdzie P_k to kolejne wierzchołki wielokąta:‘

$$\mathtt{barycenter} = \frac{1}{\mathtt{polygon.length}} \sum_{\mathtt{k=0}}^{\mathtt{polygon.length}-1} \mathtt{P}_{\mathtt{k}}$$

i jego odległość od P'_{cam}:

$$\mathtt{distance} = |\mathtt{barycenter} - \mathtt{P}'_{\mathtt{cam}}|$$

Dla sfery środek ciężkości znajduje się w jej środku.

Tak obliczone odległości przypisujemy do wielokątów, które następnie sortujemy wg. tych odległości (malejąco).

Taka implementacja algorytmu usuwania powierzchni zasłoniętych ma swoje problemy. Istnieje wiele szczególnych przypadków, w których środki ciężkości nie będą odpowiadały rzeczywistej kolejności obiektów. Jeden z takich przypadków prezentuje rysunek 3 - dwa wielokąty, zmieniają kolejność w której się zasłaniają, choć nie powinny.

Zdecydowaliśmy się jednak na to rozwiązanie, ponieważ jest łatwe w implementacji i dość wydajne. Poza tym nasz oryginalny zamysł programu przewidywał tylko prostopadłościany, w których algorytm oparty o środki ciężkości sprawuje się dobrze.

3 Pozostałe informacje o projekcie

Statyczne części projektu (strony, style) są tworzone za pomocą frameworku staticmatic, na podstawie źródeł (W językach HAML i SASS).

W projekcie wykorzystaliśmy javascriptową bibliotekę Sylvester wspierającą operacje na macierzach i wektorach.

Kod źródłowy projektu (wraz z aktualną dokumentacją) dostępny jest na serwisie Github (głównie katalog /src/coffee).

Projekt jest dostępny do testowania na następujących serwerach:

- Serwer 1[1]
- Volt[2] (Niekoniecznie aktualna wersja)

[1] `http://barnex.mooo.com`

[2] `http://volt.iem.pw.edu.pl/~pienkowb/`

Literatura

[1] Song Ho Ahn, OpenGL Projection Matrix, http://www.songho.ca/opengl/gl_projectionmatrix.html

[2] Dariusz Sawicki, Grafika komputerowa i wizualizacja, http://wazniak.mimuw.edu.pl/index.php?title=Grafika_komputerowa_i_wizualizacja