Przekształcenia afiniczne

Wprowadzenie

W tym ćwiczeniu będziemy omawiać wykorzystanie podstawowych operacji matematycznych na macierzach w grafice komputerowej.

Jednym z głównych problemów grafiki komputerowej jest dokonywanie prostych przekształceń obrazów. Wyobraźmy sobie, że chcemy zrobić prostą grę platformową, podobną do Super Mario ®.

Składa się ona z wielu małych obrazków nazwanych po angielsku *sprites*. Animacja w takiej grze polega na dwóch podstawowych elementach:

- szybkiej podmianie obrazków na kolejne klatki animacji
- przesuwaniu, skalowaniu i obrocie każdego obrazka w przestrzeni 2D

W tym ćwiczeniu będziemy omawiać drugi z tych problemów - jak wydajnie przesuwać, skalować i obracać przedmioty w przestrzeni. Dla uproszczenia ćwiczenie będzie dotyczyć grafiki 2D, ale bez problemu można ten pomysł rozwinąć do dowolnej ilości wymiarów.

Wstępny program

Zacznijmy od następującego programu:

```
<style> body { background-color:#ccc; } </style>
<script src="//cdnjs.cloudflare.com/ajax/libs/p5.js/0.5.7/p5.js"></script>
<body>
<script type="text/javascript">
var img4;
var img8;
function setup() {
    createCanvas(512,512);
    background(255);
    img4 = createImage(512,512);
    img8 = createImage(512,512);
    img4.loadPixels();
    img8.loadPixels();
    var d = pixelDensity();
    for(var i=0; i<512*512*4*d; i+=4) {
        img4.pixels[i]=240;
        img4.pixels[i+1]=250;
        img4.pixels[i+2]=240;
        img4.pixels[i+3]=255;
        img8.pixels[i]=240;
        img8.pixels[i+1]=240;
        img8.pixels[i+2]=250;
        img8.pixels[i+3]=255;
    }
    img4.updatePixels();
    img8.updatePixels();
}
function draw() {
    if (!keyIsDown(32)) {
        image(img4,0,0);
        text('Image A',10,20);
    } else {
        image(img8,0,0);
        text('Image B',10,20);
  }
}
</script>
</body>
```

Program ten rysuje planszę z dwoma buforami do rysowania. Najpierw jest wywietlany jasnozielony bufor o nazwie *img4*, a po naciśnięciu spacji, wyświetla się bufor jasnoniebieski o nazwie *img8*. Te dwa obrazy umożliwią obserwowanie różnic między dokonywanymi przekształceniami.

Wektorowa reprezentacja obrazów

(zadanie na ocenę 3)

Operacje opisywane w tym ćwiczeniu służą do przekształcania obiektów w układzie współrzędnych, niezależnie od ich rzeczywistego wyglądu. Może to być pojedynczy punkt w przestrzeni, grupa punktów określająca jakiś kształt (np. cztery wierzchołki kwadratu) albo nawet zbiór pikseli w obrazie rastrowym (gdzie każdy piksel ma zdefiniowaną współrzędną x i y). Dla uproszczenia my będziemy przekształcać pojedyncze punkty w przestrzeni 2D.

Zacznijmy więc od definicji takiego punktu w postaci wektora. Wektor to zbiór liczb powiązanych ze sobą uwzględniający ich pozycję. Wektory zapisujemy zazwyczaj w postaci wiersza albo kolumny:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{lub} \quad \begin{bmatrix} x & y & 1 \end{bmatrix}$$

W grafice 2D, wektor reprezentujący punkt w przestrzeni ma 3 wartości. Pierwsze dwie to współrzędne *x* i *y*, a trzecia wartość to zawsze 1. Ze względu na istnienie trzeciej wartości, mamy do czynienia z **przestrzenią afiniczną**, dzięki czemu oprócz obrotu i skalowania, możemy skutecznie modelować również przesunięcia. Z jednej strony jest ona konieczna, ale z drugiej strony może też służyć jako skuteczna metoda do sprawdzania poprawności operacji matematycznych omawianych później.

Jako pierwsze zadanie, należy stworzyć funkcję *makeVector* z argumentami *x* i *y*. Funkcja ta ma zwracać tablicę jednowymiarową, tak jak pokazano wyżej. Tablice w JavaScripcie definiujemy używając następującej składni: `var tab=[1,2,3,4];`

Następnie należy stworzyć funkcję *drawVector* przyjmującą dwa argumenty *img* i *vec*. Pierwszy argument to obraz, na jakim chcemy rysować (*img4* lub *img8*), a drugi argument to wektor, jaki ma być narysowany. Do rysowania możesz użyć funkcji `img.set(x,y,kolor);`. Po użyciu tej funkcji należy uruchomić funkcję `img.updatePixels();` żeby zaktualizować wartość bufora w pamięci.

Jako ostatni krok tego zadania należy zaimplementować wydarzenie `function mouseDragged()`. Wewnątrz tego wydarzenia należy najpierw stworzyć wektor metodą *makeVector*, podając do niej współrzędne *mouseX* i *mouseY*, a potem należy przekazać otrzymany wektor do funkcji *drawVector* razem z obrazem *img4* (na razie tylko tego obrazu będziemy używać).

Po pomyślnym wykonaniu tego zadania powinieneś móc rysować proste kształty na obrazie *img4*.

Macierze przekształceń

(zadanie na ocenę 3.5)

Po wykonaniu poprzedniego zadania można mieć wrażenie, że tworzenie wektorów jest trochę bez sensu, ponieważ można od razu ustawić piksel na obrazie. Niemniej jednak jest pewien dobry powód, dla którego wprowadzamy pojęcie wektora. Okazuje się, że wszystkie wspomniane poprzednio operacje przekształcania można zdefiniować matematycznie jako operacje mnożenia wektora z macierzą. Wektor, tak jak wyżej, ma reprezentować współrzędne obiektu, jaki podlega przekształceniom, a macierz samo przekształcenie.

$$M_{\text{trans}} \cdot \vec{v} = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Zanim jednak przejdziemy do stosowania tych przekształceń, zdefiniujmy najpierw kilka podstawowych macierzy, zaczynając od najprostszej - **macierzy jednostkowej** (ang. identity matrix):

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Macierz ta ma prosty efekt, że nie dokonuje żadnego przekształcenia - jest swoistą jedynką w przestrzeni macierzy. Wydawałoby się, że powinna się ona składać z samych zer albo jedynek, ale jak się okaże, ma ona wygląd taki jak powyżej.

Kolejną, często stosowaną macierzą jest **macierz przesunięcia** lub translacji:

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Podstawiając w miejsce t_x i t_y wartości przesunięcia (o ile ma zostać przesunięty obiekt), macierz ta dokona tego przekształcenia po wykonaniu operacji mnożenia z wektorem reprezentującym obiekt.

Analogicznie możemy zdefiniować **macierz skalowania**:

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

Trochę bardziej skomplikowana jest **macierz obrotu**:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Litera θ reprezentuje tutaj kąt, o jaki należy dokonać obrotu. Warto tutaj jednak pamiętać o tym, że w JavaScript (jak i większości języków programowania) funkcje trygonometryczne przyjmują wartości w radianach a nie w stopniach. Chcąc zatem dokonać obrotu o 30 stopni, trzeba do wartości θ wpisać liczbę ~0.5236. Żeby dokonać konwersji kąta ze stopni na radiany wystarczy podzielić najpierw stopnie przez wartość 180, a potem pomnożyć otrzymaną wartość przez π (w JavaScripcie `Math.PI`).

Na sam koniec warto jeszcze wspomnieć o ostatnim przekształceniu, trochę rzadziej stosowanym niż powyższe, tj. **przekształceniu pochylania** (ang. shear):

$$\begin{bmatrix} 1 & Sh_x & 0 \\ Sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Do zaliczenia tego zadania, należy zaimplementować funkcje tworzące wszystkie powyższe macierze według specyfikacji (analogicznie do *makeVector*, np. *makeIdentity* albo *makeScale*). Na razie nie zobaczymy ich w akcji, ale możesz użyć polecenia `console.log(...);` żeby je wypisać do konsoli. Wypisywanie to możesz umieścić na zewnątrz jakiejkolwiek funkcji, pod koniec skryptu (zostanie wtedy wykonane tylko raz).

Zastosowanie macierzy przekształceń

(zadanie na ocenę 4)

Mając już stworzony wektor i macierze przekształceń, czas zastosować wybrane przekształcenia w naszej aplikacji.

Żeby zastosować przekształcenie, należy dokonać operacji **mnożenia macierzy** z wektorem używając wzoru poniżej:

$$\vec{w} = M \cdot \vec{v}$$

$$\begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$w_i = \sum_{j=1}^{V} m_{i,j} \cdot v_j \qquad \forall i \in 1..W$$

Literka *V* w powyższym wzorze to długość wektora *v*, a *W* długość wektora *w*. Tym samym rozmiar macierzy musi być *W*x*W*.

Zrób funkcję do wykonywania powyższej operacji mnożenia wektora z macierzą, a potem zmodyfikuj funkcję *mouseDragged*tak, żeby po narysowaniu piksela na pierwszym obrazie dokonać mnożenia jego wektora z jedną z powyższych macierzy, a potem narysować otrzymany wektor na obrazku *img8*.

Po zaimplementowaniu tej części ćwiczenia, powinieneś móc narysować obraz i naciskając spację zobaczyć, jak on wygląda po dokonaniu na nim transformaty. Zaobserwuj, jak wygląda/g poszczególne przekształcenia. Na czym dokń a na czym skalowanie? Zrób macierze dla wszystkich przekształceń i zakomentuj wszystkie oprócz jednego, żeby pokazać prowadzącemu dowolne z nich.

(Mała rada: żeby ułatwić sobie debugowanie powyższej funkcji, pamiętaj, że wynikiem operacji musi być poprawny wektor, czyli ostatnia jego wartość musi zawsze wynosić 1. Oprócz tego, pamiętaj, że mnożenie dowolnego wektora z macierzą jednostkową powinno dać ten sam wektor.)

Łączenie przekształceń

(zadanie na ocenę 4.5)

W tej chwili jesteśmy w stanie dokonywać dowolnych przekształceń, ale nadal operacja ta nie jest bardzo wydajna od bezpośredniej modyfikacji wartości współrzędnych. Chcąc wykonać kilka przekształceń po kolei, możemy dokonać najpierw jednego, potem na jego wyniku dokonać kolejnych i tak dalej.

$$M_1 \cdot v_1 = v_2,\ M_2 \cdot v_2 = v_3,\ M_3 \cdot v_3 = v_4 \ldots$$

Okazuje się jednak, że jest to matematycznie identyczne do wykonania operacji mnożenia pierwszego wektora z iloczynem wszystkich macierzy razem:

$$(M_n \cdot \ldots \cdot M_3 \cdot M_2 \cdot M_1) \cdot v_1 = v_n$$

Co więcej, ponieważ macierze przekształceń są z reguły znane na samym początku i nie zmieniają się w trakcie rysowania, to wszystko co jest w nawiasie w powyższym wzorze, można wyliczyć tylko raz i przez to przyspieszyć działanie algorytmów rysowania o rząd wielkości *(np. ze złożoności O(n²) możemy go przyspieszyć do złożoności O(n)).*

Do zaliczenia tego zadania, należy zaimplementować analogiczną funkcję do **mnożenia macierzy** z macierzą:

$$O = M \cdot N$$

$$\begin{bmatrix} o_{1,1} & o_{1,2} & o_{1,3} \\ o_{2,1} & o_{2,2} & o_{2,3} \\ o_{3,1} & o_{3,2} & o_{3,3} \end{bmatrix} = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \cdot \begin{bmatrix} n_{1,1} & n_{1,2} & n_{1,3} \\ n_{2,1} & n_{2,2} & n_{2,3} \\ n_{3,1} & n_{3,2} & n_{3,3} \end{bmatrix}$$

$$o_{i,j} = \sum_{k=1}^{C} m_{i,k} \cdot n_{k,j} \qquad \forall i \in 1..A, j \in 1..B$$

W powyższym wzorze macierz wynikowa *O* ma wymiary *AxB*, macierz *M* ma wymiary *AxC*, a macierz *N* ma wymiary *CxB*.

Do zaliczenia ćwiczenia zmodyfikuj kod w *mouseDragged*, żeby oprócz dokonywania pojedynczych przekształceń, dokonać sekwencji kilku. Zauważ, że kolejność przekształceń ma znaczenie! Inaczej mówiąc, operacja mnożenia macierzy nie jest przemienna (tak jak w przypadku pojedynczych liczb). Zrób przykład, gdzie na *img4* mamy jedną sekwencję przekształceń (np. przesunięcie + obrót + skalowanie), a na *img8* mamy inną sekwencję używającą tych samych przekształceń, ale w innej kolejności (np. skalowanie + przesunięcie + obrót).

(Mała rada: tak jak poprzednio, warto zrobić kilka przykładów do debugowania powyższego kodu. Na pewno warto sprawdzić mnożenie macierzy z macierzą jednostkową, w wyniku którego powinniśmy otrzymać tę samą macierz.)

Interaktywna aplikacja

(zadanie na ocenę 5)

Na najwyższą ocenę należy przygotować aplikację pozwalającą na rysowanie na *img4* i wykonanie przekształceń otrzymanego obrazu w *img8*, z elementami interfejsu użytkownika.

- pokazywać cały czas zawartość macierzy przekształceń, jaka jest używana w danym momencie
- umożliwić użytkownikowi dokonanie resetu macierzy na macierz jednostkową
- umożliwić użytkownikowi zastosowanie przekształcenia (obrotu, skalowania, translacji, pochylenia), dokonując wymnożenie bieżącej macierzy przez wybraną macierz przekształcenia.
- umożliwić użytkownikowi zmiany poszczególnych wartości przekształceń.