

Spotkanie (A) – (A+3)

Transformacje macierzowe

Czytanie i tworzenie plików z obrazami

Ćwiczenie 0 – O co chodzi

Będziemy tworzyć algorytm w Javie, do przekształcania obrazków. Brzmi trochę skomplikowanie, ale to bardzo proste: wgrywamy obrazek, podajemy 4 liczby, i dostajemy taki sam obrazek ale rozciągnięty, skurczony lub obrócony w dowolną stronę.

Gotowy projekt powinien wyglądać mniej więcej tak:

Oryginalny obrazek	Macierz [0, 1; 1, 0]	Macierz [1, -0.25; 0.25, -1.25]
		

Uczestnicy grupy mogą być w dowolnym wieku, byleby wiedzieli, czym są wektory, i słyszeli kiedykolwiek słowo “macierz”; oraz mieli wcześniej jakkolwiek styczność z Javą.

Nie nawet muszą wiedzieć, co to są “przekształcenia macierzowe” - żeby to wyjaśnić, można pokazać kilka krótkich animacji, np. z tego filmu: <https://youtu.be/kYB8IZa5AuE>

▶ Linear transformations and matrices | Chapter 3, Essence of linear algebra

i omówić 2 lub 3 proste przykłady transformacji. Najważniejsze, żeby rozumieli:

- że przekształcenie macierzowe to “mnożenie” wektorów przez macierz;
- że przekształcenie przez macierz $[1,0; 0,1]$ niczego nie zmienia;
- przekształcenia przez jakie macierze to odbicia lustrzane;
- przekształcenia przez jakie macierze to “rozciągnięcia” lub “skurzenia” (poziome lub pionowe).

Jako “wektory” będziemy traktować piksele, a transformacja przez macierz to po prostu zmiana pozycji pikseli. Nie są to zajęcia z matematyki, więc można podać gotowy wzór, jak obliczyć nową pozycję przekształconego wektoru/piksela:

$$[x, y] * [a, b; c, d] = [ax + by; cx + dy]$$

Następny krok to wspólnie przegadanie logiki tego projektu. Nie jest to kurs z programowania obiektowego, więc wystarczy napisanie jednej klasy. Spiszemy na tablicy listę rzeczy (wraz z kolejnością ich wykonywania) potrzebnych do zaprogramowania takiego skryptu, który będzie przekształcał obrazki. Przykładowa lista:

- 1) Inputy od użytkownika
 - a) 4 liczby opisujące macierz (int czy nie int? 4 zmienne czy 2 tablice? a może tablica tablic?)
 - b) Nazwa pliku z obrazkiem
- 2) Plik z obrazkiem
 - a) Odczytanie pliku z obrazkiem początkowym
 - b) Stworzenie nowego pliku (w którym będzie transformowany obrazek)
- 3) Transformacja pikseli
 - a) Przejście przez każdy piksel obrazka wejściowego
 - b) Obliczenie jego nowej pozycji (po transformacji)
 - c) Umieszczenie piksela na nowej pozycji w obrazku wyjściowym
- 4) Ewentualne komunikaty o błędach

Ćwiczenie 1 - Inputy od użytkownika

Musimy zdecydować, w jaki sposób będziemy brać potrzebne dane. Jeśli nikt nie będzie miał pomysłu, jak to zrobić, to można zasugerować podawanie argumentów w wierszu poleceń (w języku Java wydaje się to prostszą opcją, niż tworzenie obiektu klasy Scanner). Przykładowo:

Unset

```
C:\Users\Admin\Desktop> java Transformacja.java Obrazek.png 1 0 0 1
```

W tym wierszu podane argumenty to: “Obrazek.png”, “1”, “0”, “0”, “1”. Można zapytać: gdzie są one przypisywane? Jak do nich się dostać, jak je “wydobyć”, żeby móc coś z nimi zrobić?

Jeśli nikt nie wie, to można dać dzieciom chwilę, żeby spojrzały na dotychczasowy kod (kod pustej klasy z metodą main) i znalazły jakąś zmienną.

Java

```
public static void main(String[] args)
```

Oczywiście chodzi o tablicę String[], która jest zawsze argumentem metody main - to do niej przypisywane są argumenty z wiersza poleceń. Możemy łatwo się do nich dostać, indeksując zmienną args od 0 do 4. Na przykład:

```
Java
String nazwaPliku = args[0];
```

I tak dalej. Niech dzieci same spróbuje przypisać elementy args do zmiennych, które potrzebujemy; a następnie wyświetlić je po kolej w konsoli, żeby sprawdzić, czy wszystko działa:

```
Java
System.out.println(zmienna);
```

Po tym etapie istnieje możliwość, że część z osób dostała taki błąd:

```
Unset
error: incompatible types: String cannot be converted to int
```

A to dlatego, że niestety język Java wymaga zgodności typów zmiennych. Jakiego typu są zmienne podane w wierszu poleceń? A jakiego typu są zmienne, które utworzyliśmy? Chcąc wydobyć wartość liczbową z napisu, musimy użyć konwersji typów:

```
Java
int x1 = Integer.parseInt(args[1]); // jeżeli używamy intów
double x1 = Double.parseDouble(args[1]); // jeżeli używamy np. double
```

Teraz, gdy dotychczasowy kod już wszystkim działa, możemy zastanowić się nad jego estetyką/użytecznością. Bardzo możliwe, że część osób utworzyła 4 różne zmienne liczbowe, np: x1, x2, y1, y2. Czy to na pewno dobry pomysł? Mamy 4 linijki kodu, które są bardzo podobne do siebie - czy nie da się tego jakoś zautomatyzować? Czy gdybyśmy mieli 100 zmiennych, to byśmy potrzebowali napisać 100 linijek kodu? A jak inaczej możemy gdzieś przypisać kilka wartości liczbowych?

Oczywiście chodzi o użycie w tym celu tablicy. Wtedy przypisywanie zmiennych do elementów tablicy można wykonać używając pętli *for*:

```
Java
double[][] macierz = new double[2][2];
int indeks = 1;

for (int i=0; i<2; i++)
    for (int j=0; j<2; j++)
    {
        macierz[i][j] = Double.parseDouble(args[indeks]);
        indeks++;
    }
```

W tym przypadku stworzyliśmy tablicę 2 tablic, każda o 2 elementach. Zmienne *i* oraz *j* to indeksy tychże tablic, a zmienna *indeks* to nr argumentu z wiersza poleceń (startując od zera).

Przechodzimy dwoma pętlami *for* przez wszystkie elementy tablicy 2x2 (*macierz*), i do każdego jej elementu przypisujemy *args[indeks]*, po czym zwiększamy *indeks* o 1.

Niech dzieci same zaproponują ten kawałek kodu. Pewnie część z nich utworzy dwie osobne tablice, np. *double[] x* oraz *double[] y*. Natomiast wtedy pozostaje zadać to samo pytanie, czy jest to najlepsze możliwe rozwiązanie. Jeśli nikt wcześniej na to nie wpadnie – można zasugerować tablicę 2x2 (ewentualnie tablica 1x4 też ujdzie, ale wtedy pytanie, czy jest ona czytelna dla nas, programistów).

Ćwiczenie 2 – Plik z obrazkiem

Do tej pory nasz kod pisaliśmy tylko w obrębie metody *main*. Możemy założyć, że wygląda ona tak:

```
Java
public static void main(String[] args)
{
    String nazwaPliku = args[0];

    double[][] macierz = new double[2][2];
    int indeks = 1;

    for (int i=0; i<2; i++)
        for (int j=0; j<2; j++)
    {
        macierz[i][j] = Double.parseDouble(args[indeks]);
        indeks++;
    }
}
```

Czy wszystko powinno być tylko w jednej metodzie? Czy może jednak do obsługi pliku z obrazkiem lepiej będzie napisać osobną metodę?

Oczywiście zaproponujmy napisanie osobnej metody. Niech wszyscy zastanowią się, jakie argumenty powinna ona przyjmować (jeżeli mieli wcześniej więcej styczności z Javą, to również jaki typ ma zwracać – ale jeśli nie, to możemy zaproponować po prostu static void, bez wchodzenia w szczegóły).

Ustaliliśmy wcześniej, że potrzebna jest nam nazwa pliku wejściowego oraz tablica liczb, zatem oczywiście to powinny być argumenty metody. Można też podać nazwę pliku wyjściowego, ale nie trzeba.

Java

```
public static void transformacja(String nazwaPliku, double[][] macierz)
```

Potrzebujemy jakiś przykładowy obrazek – niech każdy narysuje na szybko jakiś prosty w Paintie.

Nie spodziewam się, żeby ktokolwiek znał klasy i metody potrzebne do obsługi pliku z obrazkiem. Można podrzuścić im wskazówkę – ale niech sami powiedzą, gdzie co trzeba wpisać:

Java

```
File file = new File(fileName);
BufferedImage obrazek = ImageIO.read(file);
```

Tylko że te 2 linijki zadziałają na kompilator jak płachta na byka: od razu wyrzuci 4 błędy. Dlaczego? Co mogło pójść nie tak? Co to znaczy cannot find symbol? No tak, trzeba zaimportować klasy:

Java

```
import java.io.File;
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
```

I teraz powinno działać. Ale czy na pewno? Co to znaczy *unreported exception*? (Tutaj, jak bardzo chcemy się w to zagłębiać – znowu zależy, jak zaawansowana w Javie jest grupa. Można po prostu powiedzieć, że niektóre metody wymagają “specjalnego traktowania” na wypadek wyjątku, czyli gdyby coś mogło pójść nie tak.)

Potrzebny jest więc jeszcze jeden import:

```
Java
import java.io.IOException;
```

Oraz nasz kod trzeba “objąć specjalnym traktowaniem”, w taki sposób:

```
Java
try
{
    File file = new File(fileName);
    BufferedImage obrazek = ImageIO.read(file);
    // Reszta naszego kodu tutaj...
}
catch (IOException wyjatek)
{
    System.err.println("Niestety, coś poszło nie tak. :(");
}
```

Mamy już czytanie obrazka, pozostało jeszcze utworzyć obiekt z nowym obrazkiem.
Podpowiedź jak to zrobić:

```
Java
BufferedImage image = new BufferedImage(width,height,BufferedImage.TYPE_INT_ARGB);
```

Trzeba tylko upewnić się, że nikt nie nazwie obiektu z nowym obrazkiem tak samo, jak obiekt z początkowym obrazkiem. Skąd weźmiemy wymiary nowego obrazka? Czy mogą być takie same jak wymiary początkowego? A skąd je weźmiemy - odczytamy w Paintie i przepiszemy? A jeśli będziemy chcieli czytać inny obrazek o innych wymiarach? Proponuję odczytać wymiary początkowego, ale nie w Paintie, tylko w Javie.
Podpowiedź:

```
Java
image.getWidth(); //szerokość
image.getHeight(); //wysokość
```

Gdzie przypiszemy te wartości? Jak myślicie, jakiego one są typu?

Oczywiście wymiary transformowanego obrazka nie zawsze będą takie same jak wymiary początkowego; ale na tym etapie proponuję przyjąć, że tak jest, i wrócić do tego potem (bo póki co może to być zbyt skomplikowane).

Grupa Kontrolna
Scenariusz, Spotkanie (A) – (A+3)

A czy gdybyśmy chcieli w przyszłości zmienić wymiary transformowanego obrazka, to będziemy mogli to łatwo zrobić, zmieniając maksymalnie dwie linijki w kodzie? Oczywiście, że tak, dlatego najlepiej utwórzmy osobne zmienne:

```
Java
// Obrazek początkowy
int szerokoscPoczatkowa = obrazek.getWidth();
int wysokoscPoczatkowa = obrazek.getHeight();

// Obrazek transformowany
int szerokoscTransformed = szerokoscPoczatkowa;
int wysokoscTransformed = wysokoscPoczatkowa;
```

Możemy skompilować kod i zobaczyć, co się stanie. Nic się nie stało? Dlaczego? Z dwóch powodów:

- a) Obrazek nie został zapisany do pliku. Popatrzmy na kod, który już mamy do tej pory. Która linijka tworzyła nowy obiekt z plikiem? Czy możemy jej użyć ponownie? Możemy, tylko musimy jeszcze zapisać w nim nowy obrazek:

```
Java
ImageIO.write(nowyObrazek, "png", nowyPlik);
```

- b) Metoda *transformacja* nie została wywołana w metodzie *main*. Jeżeli grupa miała wcześniej styczność z Javą, to chyba nie trzeba tłumaczyć, że “odpalane” jest tylko to, co znajduje się w metodzie *main*. Niech grupa sama spróbuje dopisać tę linijkę:

```
Java
transformacja(nazwaPliku, macierz);
```

Działa? Czemu obrazek jest pusty? A może chociaż wymiary się zgadzają? Tak, zgadzają się; ale jest pusty, bo jeszcze nic w nim nie “narysowaliśmy”. To będzie następne ćwiczenie.

Ćwiczenie 3 – Malowanie pikseli

Oto nasz dotychczasowy kod w metodzie transformacja:

```
Java
public static void transformacja(String nazwaPliku, double[][] macierz)
{
    try
    {
        // Odczytaj obrazek z pliku oryginalnego
        File plikOryginalny = new File(nazwaPliku);
        BufferedImage obrazek = ImageIO.read(plikOryginalny);

        // Obrazek początkowy
        int szerokoscPoczatkowa = obrazek.getWidth();
        int wysokoscPoczatkowa = obrazek.getHeight();

        // Obrazek transformowany
        int szerokoscTransformed = szerokoscPoczatkowa;
        int wysokoscTransformed = wysokoscPoczatkowa;

        BufferedImage nowyObrazek = new
        BufferedImage(szerokoscTransformed, wysokoscTransformed, BufferedImage.TYPE_INT_ARGB);
        File nowyPlik = new File("nowy" + nazwaPliku);
        ImageIO.write(nowyObrazek, "png", nowyPlik);
    }
    catch (IOException e)
    {
        System.err.println(e);
    }
}
```

Jednak zanim zacznijemy “transformować” początkowy obrazek, warto nauczyć się, jak w ogóle działa tworzenie obrazów w Javie. Podpowiedzią będzie poniższa metoda:

```
Java
image.setRGB(x, y, RGB);
```

Maluje ona w obrazie *image* piksel o współrzędnych (x,y) na kolor o wartości RGB (typ int) [Red Green Blue - akronim używany przez grafików komputerowych]. Dla przykładu, RGB koloru białego w tym formacie to -1. Zadanie dla dzieci na teraz jest takie, żeby zamalowali nowo utworzony obraz cały na biało (wszystkie piksele mają mieć wartość RGB równą -1).

Grupa Kontrolna
Scenariusz, Spotkanie (A) – (A+3)

Zadanie jest dosyć łatwe, spodziewam się, że grupa wpadnie na jakiś kod podobny do poniższego:

```
Java
for (int x=0; x<szerokoscTransformed; x++)
    for (int y=0; y<wysokoscTransformed; y++)
        nowyObrazek.setRGB(x, y, -1);
```

Są to 2 pętle for, przechodzące przez wszystkie piksele obrazu nowyObrazek i ustawiające ich wartość RGB. Tylko, żebyśmy po uruchomieniu faktycznie dostali biały obrazek, musimy ten kod umieścić przed stworzeniem pliku, ale po stworzeniu obiektu nowyObrazek.

A gdybyśmy chcieli, żeby nowy obrazek był taki sam jak początkowy? To jak to zrobić?

Oczywiście nie musimy znać na pamięć wartości RGB dla wszystkich kolorów, pomocna będzie ta metoda (odczytuje wartość RGB piksela o współrzędnych (x,y)):

```
Java
image.getRGB(x, y);
```

Spodziewam się, że każdy da radę napisać jakiś kod podobny do poniższego:

```
Java
for (int x=0; x<szerokoscTransformed; x++)
    for (int y=0; y<wysokoscTransformed; y++)
        nowyObrazek.setRGB(x, y, obrazek.getRGB(x, y));
```

W zależności od tego, jak dobrze sobie radzi z tym grupa, można jeszcze zaproponować obrócenie obrazka poziomo, pionowo, lub tak i tak jednocześnie. To bardzo ważne, żeby już na tym etapie wszyscy dobrze zrozumieli logikę, jaka stoi za malowaniem obrazu metodą setRGB.

Przykładowy kod dla obrotu lustrzanego (poziomego):

```
Java
for (int x=0; x<szerokoscTransformed; x++)
    for (int y=0; y<wysokoscTransformed; y++)
        nowyObrazek.setRGB(szerokoscTransformed-1 - x, y, obrazek.getRGB(x, y));
```

Czemu od szerokoscTransformed odejmujemy 1? Ile ma wynosić współrzędna pozioma dla $x == 0$? A ile dla x maksymalnego? Ile w ogóle wynosi maksymalne x ? szerokoscTransformed, czy mniej?

No dobrze, a gdybyśmy chcieli rozciągnąć obrazek, żeby zrobić go np. 2 razy szerszym – to jakbyśmy to zrobili? Pomińmy na razie mnożenie macierzowe, tylko spróbujmy samemu znaleźć kod. To już zadanie nieco trudniejsze, ale jeśli wszyscy wszystko zrozumieli do tej pory, to powinno im się udało napisać podobny kod:

```
Java
int szerokoscTransformed = 2*szerokoscPoczatkowa;

for (int x=0; x<szerokoscPoczatkowa; x++)
    for (int y=0; y<wysokoscPoczatkowa; y++)
        nowyObrazek.setRGB(2*x, y, obrazek.getRGB(x, y));
```

Trzeba tylko pamiętać, że wtedy pętla for dla x musi być dla wartości od 0 do szerokości początkowego obrazka, a nie transformowanego. Zresztą – niech sami się przekonają, wpisując $x < szerokoscTransformed$; i niech powiedzą, dlaczego wystąpił błąd: Coordinate out of bounds.

Można też oczywiście zrobić tak:

```
Java
for (int x=0; x<szerokoscTransformed; x++)
    for (int y=0; y<wysokoscTransformed; y++)
        if (x % 2 == 0)
            nowyObrazek.setRGB(x, y, obrazek.getRGB(x/2, y));
```

Wtedy, zamiast mnożyć współrzędną piksela z początkowego obrazka, pomijamy co drugi piksel w obrazku transformowanym (a biorąc kolor – dzielimy współrzędną przez 2).

Ćwiczenie 4 – Współrzędne pikseli

Zróbjmy jeszcze jedno ćwiczenie, zanim zaczniemy się zajmować przekształceniemi macierzowymi. Będzie ono proste i szybkie, ale kluczowe, żeby zrozumieć współrzędne pikseli w klasie *BufferedImage*.

Mamy utworzony nowy obrazek o wymiarach takich samych, jak obrazek początkowych. Zamalujmy jego lewą dolną ćwiartkę na biało, a resztę zostawmy pustą (niech grupa spróbuje sama).

Spodziewam się, że duża część osób napisze taki kod:

```
Java
for (int x=0; x<szerokoscTransformed/2; x++)
    for (int y=0; y<wysokoscTransformed/2; y++)
        nowyObrazek.setRGB(x, y, -1);
```

Co dostajemy po skompilowaniu go? Niestety – figę z makiem, bo zamalowana jest nie lewa dolna ćwiartka, a lewa góra. Dlaczego tak? Co to oznacza, jak liczone są współrzędne?

Jeśli grupa się jeszcze nie domyśla odpowiedzi, to można poprosić o zamalowanie tylko jednego punktu – o współrzędnych $x = 0, y = 0$. Gdzie znajduje się ten piksel? Będzie ledwo go widać, ale można przyzoomować i poszukać białej kropki.

Teraz powinno być już jasne, że współrzędne y w Javie są liczone inaczej niż na matematyce, bo od góry do dołu – a nie od dołu do góry. No dobrze, to jak w takim razie zamalować tę lewą dolną ćwiartkę? Tutaj grupa może potrzebować chwili na zastanowienie się, ale myślę, że w kilka minut uda się większości osób napisać taki kod:

```
Java
for (int x=0; x<szerokoscTransformed/2; x++)
    for (int y=0; y<wysokoscTransformed/2; y++)
        nowyObrazek.setRGB(x, wysokoscTransformed-1 - y, -1);
```

Lub też taki:

```
Java
for (int x=0; x<szerokoscTransformed/2; x++)
    for (int y=wysokoscTransformed-1; y>=wysokoscTransformed/2; y--)
        nowyObrazek.setRGB(x, y, -1);
```

Jednak czy będzie to dla nas problemem, w kontekście przekształceń macierzowych, że będziemy mieli współrzędną y inaczej liczoną? Chyba wystarczy przypomnieć (animacje z ćwiczenia 0), jak te przekształcenia wyglądają, żeby się przekonać, że tak – będzie to utrudnienie, lepiej żeby współrzędne pikseli zgadzały się ze współrzędnymi matematycznymi.

Jak to zrobić? Chyba już wszyscy powinni wiedzieć:

```
Java
int yDolGora = wysokoscTransformed-1 - y;
```

Jednak czy to wystarczy? Jak wygląda odbicie lustrzane? Co się dzieje z wektorami o dodatnich współrzędnych? Przypomnijmy sobie animację z ćwiczenia 0 - dodatnie z ujemnymi zamieniają się miejscami.

Gdzie w takim razie musi być punkt (0,0), żeby działały u nas również odbicia lustrzane? Najlepiej na środku - a jak to zrobić? Niech dzieci zastanowią się.

Podpowiedź: jakie współrzędne powinny być wtedy na rogach obrazka?

Lewa krawędź	Prawa krawędź
$x = -\text{ szerokosc}/2$	$x = \text{ szerokosc}/2$

Górna krawędź	$y = \text{ wysokosc}/2$
Dolna krawędź	$y = -\text{ wysokosc}/2$

Warto dać chwilę, na zastanowienie się. Natomiast nie jest to najistotniejsza część projektu, a tym bardziej nie jest zbyt programistyczna, więc jeśli nikt nie będzie miał pomysłu, to można podpowiedzieć:

```
Java
int x00NaSrodku = x - szerokosc/2;
int y00NaSrodku = yDolGora - wysokosc/2;
```

Po skompilowaniu kodu jednak może wystąpić błąd:

```
Unset
error: incompatible types: possible lossy conversion from double to int
```

Co to znaczy? Kto ma pomysł, co tu może się nie zgadzać? No jasne, typy zmiennych - napisaliśmy przecież, że to są inty; jednak dzielenie przez 2 nie zawsze da inta. To jak zrobić konwersję typu? Podpowiedź:

```
Java
double d = 1.0;
int i = (int)d;
```

A jeśli byśmy chcieli zrobić na odwrót? Żeby współrzędne liczone od (0,0) na środku zamienić na takie, jakie były pierwotnie? Niech grupa sama się zastanowi, to nie powinno być trudne:

Java

```
int xSpowrotem = x0NaSrodku + szerokosc/2;  
int ySpowrotemDolGora = y0NaSrodku + wysokosc/2;  
int ySpowrotem = wysokosc-1 - ySpowrotemDolGora;
```

Proponuję sprawdzić, wyświetlając wynik w konsoli, czy na pewno każdy ma dobry kod.

Ćwiczenie 5 – Mnożenie przez macierz

No dobrze, to pozostaje już tylko część ostatnia, czyli przekształcanie obrazka zgodnie z macierzą.

Nie są to zajęcia z algebry liniowej, więc nie będziemy wyprowadzać wzoru na współrzędne transformowanego wektora. Na pewno warto spytać, czy ktokolwiek z grupy wie, jak się mnoży wektor przez macierz - ale jeśli nie, to podajmy gotowy wzór:

$$[x, y] * [a, b; c, d] = [ax + by; cx + dy]$$

Po ćwiczeniach 3. i 4. grupa już powinna wiedzieć, jak się za to zabrać - dajmy im chwilę, żeby sami napisali kod. Nie powinno być to trudne, żeby wpadli na coś takiego:

```
Java  
for (int x=0; x<szerokoscPoczatkowa; x++)  
{  
    for (int y=0; y<wysokoscPoczatkowa; y++)  
    {  
        double xT = (x * macierz[0][0]) + (y * macierz[0][1]);  
        double yT = (x * macierz[1][0]) + (y * macierz[1][1]);  
        nowyObrazek.setRGB(xT, yT, obrazek.getRGB(x,y));  
    }  
}
```

(Warto zasugerować, żeby stworzyli zmienne xT oraz yT - wtedy kod będzie czytelniejszy, niż gdyby pchać wszystko do jednej linijki.)

Spróbujmy to skompilować (i podać w wierszu poleceń jakieś dane macierzy). Co się stanie?

Unset

```
error: incompatible types: possible lossy conversion from double to int
```

Co to znaczy? Kto ma pomysł, co tu może się nie zgadzać? No jasne, typy zmiennych - współrzędne przecież muszą być intami. To jak je zamienić? Podpowiedź:

```
Java
double d = 1.0;
int i = (int)d;
```

Niech grupa sama spróbuje poprawić kod.

Po tej poprawce, niektórym będzie działać. Ale niektórzy dostaną Coordinate out of bounds! - to zależy od tego, jaką macierz wpisali. Co to znaczy? Dlaczego coś się nie zgadza?

Jeśli nikt nie ma pomysłu, to dowiedzmy się, dla jakich pikseli coś nie gra. Skąd możemy to wiedzieć? Czy komputer może nam powiedzieć, przy jakich dokładnie współrzędnych coś się wywala? Jak to zrobić? Co trzeba napisać? Tymi pytaniami próbuję zasugerować wyświetlenie komunikatu w konsoli, zawierając w nim zmienne x , y , xT , yT . Na przykład - wewnątrz pętli:

```
Java
System.out.println(x+ " , " +y + "    ->    " + (int)xT+ " , " +(int)yT);
```

Kiedy coś przestaje działać? Dla jakich pikseli? Co się z nimi dzieje? Oczywiście wychodzą one poza wymiary obrazka, bo ten nowy jest za mały.

Możliwe, że na tym etapie niektórzy w grupie będą chcieli wymiary nowego obrazka podać po prostu 2 razy większe, niż w obrazku początkowym. Jednak czy to rozwiąże problem, jeśli podamy macierz np. [10, 0; 0, 20]? Dlatego sugerowałbym zostawienie wymiarów obrazka na koniec, bo to może być trochę skomplikowane.

Jak inaczej możemy obejść problem zbyt małego obrazka transformowanego? Co zrobić z pikselami, które wychodzą poza obrazek? Ominąć je - ale jak? Spodziewam się, że ktoś zaproponuje instrukcję warunkową:

```
Java
if ((int)xT < szerokoscTransformed && (int)yT < wysokoscTransformed)
    nowyObrazek.setRGB((int)xT, (int)yT, obrazek.getRGB(x,y));
```

Dla dodatnich macierzy będzie działać. Ale dla ujemnych współczynników? Co się stanie? Znowu - sprawdźmy, dla których dokładnie pikseli coś nie gra. No i mamy odpowiedź, współrzędne nie mogą być ujemne. Trzeba dodać jeszcze dwa warunki

zatem: $xT \geq 0 \ \&\& \ yT \geq 0$.

Nadal nie gra? A może coś ze współrzędnymi jest nie tak? Pamiętamy ostatnie ćwiczenie? Musimy użyć obliczonych wcześniej nowych zmiennych. Niech grupa sama zmodyfikuje kod, warto podpowiedzieć, żeby teraz już nie inicjalizali zmiennych w środku pętli `for`:

```
Java
int yDolGora, x00NaSrodku, y00NaSrodka;
double xT, yT;

for (int x=0; x<szerokoscPoczatkowa; x++)
{
    for (int y=0; y<wysokoscPoczatkowa; y++)
    {
        yDolGora = wysokoscPoczatkowa - y;
        x00NaSrodka = x - (int)szerokoscPoczatkowa/2;
        y00NaSrodka = yDolGora - (int)wysokoscPoczatkowa/2;

        xT=(x00NaSrodka * macierz[0][0])+(y00NaSrodka * macierz[0][1]);
        yT=(x00NaSrodka * macierz[1][0])+(y00NaSrodka * macierz[1][1]);

        if ((int)xT<szerokoscTransformed && (int)yT<wysokoscTransformed && xT>=0 && yT>=0)
            nowyObrazek.setRGB((int)xT,(int)yT,obrazek.getRGB(x,y));
    }
}
```

Nadal jednak otrzymujemy pusty obrazek dla ujemnych współczynników macierzy. Dlaczego?

Spójrzmy spowrotem na wartości xT oraz yT dla macierzy $[1, 0; 0, 1]$, która – jak wiemy – nie powinna niczego zmieniać. Jakie one są dla lewej dolnej ćwiartki? Czy współrzędne pikseli w metodzie `setRGB` mogą być ujemne? Co trzeba z tym zrobić? Spowrotem zamienić na początkowy sposób liczenia pikseli. Mamy już gotowy kod z poprzedniego ćwiczenia. Niech grupa sama pomyśli, w którym miejscu dokładnie trzeba go wstawić. Podpowiedź: na pewno nie przed transformacją współrzędnych.

Java

```
int yDolGora, x00NaSrodku, y00NaSrodku;  
double xT, yT;  
  
for (int x=0; x<szerokoscPoczatkowa; x++)  
{  
    for (int y=0; y<wysokoscPoczatkowa; y++)  
    {  
        yDolGora = wysokoscPoczatkowa-1 - y;  
        x00NaSrodku = x - (int)szerokoscPoczatkowa/2;  
        y00NaSrodku = yDolGora - (int)wysokoscPoczatkowa/2;  
  
        xT=(x00NaSrodku * macierz[0][0])+(y00NaSrodku * macierz[0][1]);  
        yT=(x00NaSrodku * macierz[1][0])+(y00NaSrodku * macierz[1][1]);  
  
        xT = xT + szerokoscTransformed/2;  
        yT = yT + wysokoscTransformed/2;  
        yT = wysokoscTransformed-1 - yT;  
  
        if ((int)xT<szerokoscTransformed && (int)yT<wysokoscTransformed && xT>=0 && yT>=0)  
            nowyObrazek.setRGB((int)xT,(int)yT,obrazek.getRGB(x,y));  
    }  
}
```

Warto dać chwilę wszystkim na pobawienie się samemu kodem: zobaczenie samemu, jak działają przekształcenia macierzowe. Może ktoś sam będzie w stanie wymyślić, jak obliczyć wymiary nowego obrazka. Jednak to jest zagadnienie, które mało ma wspólnego z programowaniem, a o wiele bardziej z algebrą liniową. Dlatego myślę, że - o ile grupa nie będzie wręcz żądała, żeby to policzyć samemu - śmiało można po prostu podrzucić gotowy wzór:

Java

```
int szerokoscTransformed = (int)(szerokoscPoczatkowa *  
Math.abs(macierz[0][0]) + wysokoscPoczatkowa * Math.abs(macierz[0][1]));  
  
int wysokoscTransformed = (int)(szerokoscPoczatkowa * Math.abs(macierz[1][0])  
+ wysokoscPoczatkowa * Math.abs(macierz[1][1]));
```