Geometria trójwymiarowa biblioteki OpenGL

March 20, 2019

1 Introduction

Celem jest stosowanie geometrii 3D biblioteki OpenGL.

W domyślnym układzie współrzędnych dodatni kierunek osi z wskazuje kierunek prostopadły do obrazu.

W domyślnym układzie współrzędnych dla OpenGL obraz pokazuje obszar przestrzeni 3D, w którym x, y i z mają zakres od minus jeden do jednego. Aby pokazać inny region, musimy zastosować transformację. Na razie użyjemy tylko współrzędnych leżących między -1 a 1.

Uwaga dotycząca programowania: OpenGL może być zaimplementowany w wielu różnych językach programowania, ale specyfikacja API zakłada, że językiem jest C. W większości specyfikacja C tłumaczy się bezpośrednio na inne języki. Główne różnice wynikają ze szczególnych cech typu array w języku C. Przykłady będą zgodne ze składnią C, z kilkoma uwagami o tym, jak rzeczy mogą się różnić w innych językach. Ponieważ używamy API C, będzimy stosować „function", a nie „subroutine" lub „method".

2 Prymitywy OpenGL

OpenGL może rysować tylko kilka podstawowych kształtów, w tym punkty, linie i trójkąty. Nie ma wbudowanej obsługi krzywych lub zakrzywionych powierzchni; muszą być przybliżone przez prostsze kształty. Podstawowe kształty określa się jako prymitywne. Prymityw w OpenGL jest zdefiniowany przez jego wierzchołki. Wierzchołek jest po prostu punktem w 3D, podanym przez jego współrzędne x, y i z. Przejdźmy od razu i zobaczmy, jak narysować trójkąt. To wymaga kilku kroków:

```
glBegin (GL_TRIANGLES);
glVertex2f (-0,7, -0,5);
glVertex2f (0,7, -0,5);
glVertex2f (0, 0,7);
glEnd ();
```

Każdy wierzchołek trójkąta jest określony przez wywołanie funkcji `glVertex2f`. Wierzchołki muszą być określone między wywołaniami `glBegin` i `glEnd`. Parametr

`glBegin` informuje, który typ prymitywu jest rysowany. Prymityw `GL_TRIANGLES` pozwala na narysowanie więcej niż jednego trójkąta: wystarczy określić trzy wierzchołki dla każdego trójkąta, który chcesz narysować.

(Te funkcje faktycznie wysyłają polecenia do GPU. OpenGL może zapisywać partie poleceń do przesyłania razem, a rysunek nie będzie faktycznie wykonywany, dopóki polecenia nie zostaną przesłane. Aby tak się stało, funkcja `glFlush()` musi być wywołana. W niektórych przypadkach ta funkcja może być wywoływana automatycznie przez API OpenGL, ale może się zdarzyć, że natrafisz na czasy, kiedy będziesz musiał ją wywołać samodzielnie.)

W przypadku OpenGL wierzchołki mają trzy współrzędne. Funkcja `glVertex2f` określa współrzędne x i y wierzchołka, a współrzędna z jest ustawiona **na zero**.

Istnieje również funkcja `glVertex3f`, która określa wszystkie trzy współrzędne. „2" lub „3" w nazwie określa liczbę parametrów przekazywanych do funkcji. „f" na końcu nazwy wskazuje, że parametry są typu `float`.

Najprostszym prymitywem jest `GL_POINTS`, który po prostu renderuje punkt na każdym wierzchołku prymitywu. Domyślnie punkt jest renderowany jako pojedynczy piksel. Rozmiar prymitywów punktowych można zmienić, wywołując

```
glPointSize (size);
```

gdzie parametr `size` jest typu float i określa średnicę punktu renderowania, w pikselach. Domyślnie punkty są kwadratami. Możesz uzyskać okrągłe punkty, wywołując

```
glEnable (GL_POINT_SMOOTH);
```

Funkcje `glPointSize` i `glEnable` zmieniają **stan** OpenGL. Stan zawiera wszystkie ustawienia wpływające na renderowanie. Spotkamy wiele funkcji zmieniających stan. Funkcje `glEnable` i `glDisable` mogą być używane do włączania i wyłączania wielu funkcji. Zasadą jest, że każda funkcja renderowania, która wymaga dodatkowych obliczeń, jest domyślnie wyłączona. Jeśli chcesz tę funkcję, musisz ją włączyć, wywołując `glEnable` za pomocą odpowiedniego parametru.

Istnieją trzy prymitywy dla segmentów linii rysunkowych: `GL_LINES`, `GL_LINE_STRIP` i `GL_LINE_LOOP`.

`GL_LINES` rysuje odłączone segmenty linii; podaj dwa wierzchołki dla każdego segmentu, który chcesz narysować. Pozostałe dwa prymitywy rysują połączone sekwencje odcinków linii. Jedyną różnicą jest to, że `GL_LINE_LOOP` dodaje dodatkowy segment linii od końcowego wierzchołka z powrotem do pierwszego wierzchołka.

Szerokość prymitywów linii można ustawić, wywołując `glLineWidth(width)`.

Następny zestaw prymitywów służy do rysowania trójkątów. Są trzy z nich: `GL_TRIANGLES`, `GL_TRIANGLE_STRIP` i `GL_TRIANGLE_FAN`.

Trzy pozostałe prymitywy, które zostały **usunięte z nowoczesnego OpenGL**, to `GL_QUADS`, `GL_QUAD_STRIP` i `GL_POLYGON`.

Aby quad był poprawnie renderowany w OpenGL, wszystkie wierzchołki quada muszą leżeć w tej samej płaszczyźnie. To samo dotyczy prymitywów

wielokątów. Podobnie, aby być poprawnie renderowanym, kwadraty i wielokąty muszą być wypukłe. Ponieważ OpenGL nie sprawdza, czy warunki te są spełnione, użycie quadów i wielokątów jest podatne na błędy. Ponieważ te same kształty można łatwo wytworzyć za pomocą prymitywów trójkąta, nie są one naprawdę potrzebne.

3 Kolory OpenGL

OpenGL ma dużą kolekcję funkcji, których można użyć do określenia kolorów dla rysowanej geometrii. Funkcje te mają nazwy w postaci `glColor*`, gdzie „*" oznacza przyrostek, który podaje liczbę i typ parametrów. Dla realistycznej grafiki 3D OpenGL ma bardziej skomplikowane pojęcie koloru, które wykorzystuje inny zestaw funkcji. Ale na razie pozostaniemy przy glColor *.

Na przykład funkcja `glColor3f` ma trzy parametry typu float. Parametry dają czerwone, zielone i niebieskie składniki koloru jako liczby z zakresu od 0,0 do 1,0. (W rzeczywistości dozwolone są wartości spoza tego zakresu, nawet wartości ujemne. Gdy w obliczeniach używane są wartości kolorów, wartości poza zakresem zostaną użyte w podany sposób. Gdy kolor rzeczywiście pojawia się na ekranie, jego wartości składowe są zaciskane do zakres od 0 do 1. Oznacza to, że wartości mniejsze od zera są zmieniane na zero, a wartości większe niż jeden są zmieniane na jeden.)

Możesz dodać czwarty komponent do koloru za pomocą `glColor4f()`. Czwarty komponent, znany jako **alfa**, nie jest używany w domyślnym trybie rysowania, ale możliwe jest skonfigurowanie OpenGL do używania go jako stopnia przezroczystości koloru, podobnie jak użycie komponentu alfa w interfejsach API grafiki 2D. Aby włączyć przezroczystość, potrzebujesz dwóch poleceń:

```
glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Pierwsze polecenie umożliwia użycie komponentu alfa. Można go wyłączyć, wywołując `glDisable (GL_BLEND)`. Gdy opcja `GL_BLEND` jest wyłączona, alfa jest po prostu ignorowana. Drugie polecenie mówi, w jaki sposób zostanie użyty komponent alfa koloru. Wyświetlane tutaj parametry są najpowszechniejsze; wprowadzają przejrzystość w zwykły sposób. Chociaż przezroczystość działa dobrze w 2D, znacznie trudniej jest prawidłowo używać przezroczystości w 3D.

Jeśli chcesz użyć liczb całkowitych w zakresie od 0 do 255, możesz użyć `glColor3ub()` lub `glColor4ub`, aby ustawić kolor. W tych nazwach funkcji „ub" oznacza „unsigned byte". Unsigned byte jest ośmiobitowym typem danych o wartościach z zakresu od 0 do 255. Oto kilka przykładów poleceń do ustawiania kolorów rysunkowych w OpenGL:

```
glColor3f (0,0,0); // Rysuj na czarno.

glColor3f (1,1,1); // Rysuj na bia␣lo.
```

```
glColor3f (1,0,0); // Rysuj w pełnej intensywności czerwonym.

glColor3ub (1,0,0); // Rysuj w kolorze nieco innym niż
                    // czarny. (Sufiks ,,ub'' lub ,,f'' jest ważny!)

glColor3ub (255,0,0); // Rysuj w pełnej intensywności czerwonym.

glColor4f (1, 0, 0, 0,5); // Rysuj w przezroczystym czerwonym kolorze,
                          // ale tylko jeśli OpenGL
                    // został skonfigurowany do przezroczystości.
                    // Domyślnie to samo co rysowanie zwykłym czerwonym.
```

Typową operacją jest wyczyszczenie obszaru rysunku przez wypełnienie go kolorem tła. Można to zrobić, rysując duży kolorowy prostokąt, ale OpenGL ma potencjalnie skuteczniejszy sposób, aby to zrobić. Funkcja

```
glClearColor (r, g, b, a);
```

ustawia kolor, który ma być użyty do wyczyszczenia obszaru rysunku. (To ustawia tylko kolor; kolor nie jest używany, dopóki nie wydasz polecenia, aby wyczyścić obszar rysunku.) Parametry są wartościami zmiennoprzecinkowymi w zakresie od 0 do 1. Nie ma wariantów tej funkcji; musisz podać wszystkie cztery komponenty koloru i muszą one mieścić się w zakresie od 0 do 1. Domyślnym kolorem czyszczenia jest wszystkie zero, to znaczy czarny z komponentem alfa równym zero. Polecenie wykonania czyszczenia to:

```
glClear (GL_COLOR_BUFFER_BIT);
```

Prawidłowym określeniem tego, co nazywam obszarem rysowania, jest **bufor koloru**, gdzie „bufor" jest ogólnym terminem odnoszącym się do obszaru pamięci. OpenGL używa kilku buforów oprócz bufora koloru („Bufor głębi"). Komendy `glClear` można użyć do wyczyszczenia kilku różnych buforów jednocześnie, co może być bardziej wydajne niż ich usuwanie osobno, ponieważ czyszczenie można wykonać równolegle. Parametr do glClear informuje go, który bufor lub bufory mają zostać wyczyszczone. Aby usunąć kilka buforów na raz, połącz stałe reprezentujące je. Na przykład,

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Jest to forma `glClear`, która jest zwykle stosowana w grafice 3D, gdzie bufor głębokości odgrywa zasadniczą rolę. W przypadku grafiki 2D bufor głębokości zwykle nie jest używany, a właściwym parametrem dla glClear jest tylko `GL_COLOR_BUFFER_BIT`.

4 glColor i glVertex z tablicami

Widzieliśmy, że istnieją wersje `glColor*` i `glVertex*`, które przyjmują różne liczby i typy parametrów. Istnieją również wersje, które pozwalają umieścić wszystkie dane dla polecenia w jednym parametrze tablicy. Nazwy takich wersji

kończą się na „v”. Na przykład: `glColor3fv`, `glVertex2iv`, `glColor4ubv` i `glVertex3dv`. „V” oznacza „wektor”, co oznacza zasadniczo jednowymiarową tablicę liczb. Na przykład w wywołaniu funkcji `glVertex3fv(coords)` coords będą tablicą zawierającą co najmniej trzy liczby zmiennoprzecinkowe.

Przykład w C:

```
float coords[] = { -0.5, -0.5,  0.5, -0.5,  0.5, 0.5,  -0.5, 0.5 };

glBegin(GL_TRIANGLE_FAN);
glVertex2fv(coords);      // Uses coords[0] and coords[1].
glVertex2fv(coords + 2);  // Uses coords[2] and coords[3].
glVertex2fv(coords + 4);  // Uses coords[4] and coords[5].
glVertex2fv(coords + 6);  // Uses coords[6] and coords[7].
glEnd();
```

Przykład w Java:

```
float[] coords = { -0.5F, -0.5F,  0.5F, -0.5F,  0.5F, 0.5F,  -0.5F, 0.5F };

gl2.glBegin(GL2.GL_TRIANGLES);
gl2.glVertex2fv(coords, 0);  // Uses coords[0] and coords[1].
gl2.glVertex2fv(coords, 2);  // Uses coords[2] and coords[3].
gl2.glVertex2fv(coords, 4);  // Uses coords[4] and coords[5].
gl2.glVertex2fv(coords, 6);  // Uses coords[6] and coords[7].
gl2.glEnd();
```

5 Bufor głębokości

Istotnym w 3D jest to, że jeden obiekt może znajdować się za innym obiektem. Gdy tak się dzieje, tylny obiekt jest ukryty przez przedni obiekt. Kiedy tworzymy obraz świata 3D, musimy upewnić się, że obiekty, które mają być ukryte za innymi obiektami, w rzeczywistości nie są widoczne na obrazie. To jest problem ukrytej powierzchni.

Rozwiązanie może wydawać się dość proste: wystarczy narysować obiekty w kolejności od tyłu do przodu. Jeśli jeden obiekt znajduje się za innym, tylny obiekt zostanie zakryty później, gdy zostanie narysowany przedni obiekt. Nazywa się to **algorytmem malarza**. Jest to zasadniczo to, do czego jesteś przyzwyczajony w 2D. Niestety, wdrożenie nie jest takie proste. Po pierwsze, możesz mieć obiekty, które się przecinają, tak aby część każdego obiektu była ukryta przez drugą. Bez względu na to, w jakiej kolejności narysujesz obiekty, pojawią się punkty, w których widoczny jest niewłaściwy obiekt. Aby to naprawić, musisz pociąć obiekty na kawałki, wzdłuż skrzyżowania i traktować te elementy jako oddzielne obiekty. W rzeczywistości mogą występować problemy, nawet jeśli nie ma przecinających się obiektów: możliwe jest posiadanie trzech nie przecinających się obiektów, w których pierwszy obiekt ukrywa część drugiego, drugi ukrywa część trzeciego, a trzeci ukrywa część pierwszego . Algorytm malarza nie powiedzie się bez względu na kolejność rysowania trzech obiektów.

Rozwiązaniem jest przecięcie przedmiotów na kawałki, ale teraz nie jest tak oczywiste, gdzie wyciąć.

Mimo że te problemy można rozwiązać, jest jeszcze inny problem. Prawidłowa kolejność rysowania może się zmienić, gdy punkt widzenia zostanie zmieniony lub gdy zastosowana zostanie transformacja geometryczna, co oznacza, że prawidłowa kolejność rysowania musi być ponownie obliczana za każdym razem, gdy to nastąpi. W animacji oznaczałoby to dla każdej klatki.

Więc **OpenGL nie używa algorytmu malarza**. Zamiast tego używa techniki zwanej **testem głębi**. Test głębokości rozwiązuje problem ukrytej powierzchni bez względu na kolejność wciągania obiektów, dzięki czemu można je rysować w dowolnej kolejności! Termin „głębokość" ma tutaj związek z odległością od widza do obiektu. Obiekty na większej głębokości są dalej od widza. Obiekt o mniejszej głębokości ukryje obiekt z większą głębią. Aby zaimplementować **algorytm testu głębi**, OpenGL przechowuje wartość głębi dla każdego piksela na obrazie. Dodatkowa pamięć używana do przechowywania tych wartości głębokości tworzy **bufor głębokości**, o którym wspomniałem wcześniej.

Podczas procesu rysowania bufor głębokości jest używany do śledzenia tego, co jest aktualnie widoczne na każdym pikselu. Kiedy drugi obiekt jest narysowany na tym pikselu, informacje w buforze głębokości mogą być użyte do decydowania, czy nowy obiekt znajduje się przed lub za obiektem, który jest obecnie widoczny. Jeśli nowy obiekt znajduje się z przodu, kolor piksela jest zmieniany, aby pokazać nowy obiekt, a bufor głębokości jest również aktualizowany. Jeśli nowy obiekt znajduje się za bieżącym obiektem, dane nowego obiektu są odrzucane, a bufory kolorów i głębi pozostają niezmienione.

Domyślnie test głębokości nie jest włączony, co może prowadzić do bardzo złych wyników podczas rysowania w 3D. Możesz włączyć test głębokości wywołując

```
glEnable (GL_DEPTH_TEST);
```

Można go wyłączyć, wywołując `glDisable (GL_DEPTH_TEST)`. Jeśli zapomnisz włączyć test głębokości podczas rysowania w 3D, obraz, który otrzymasz, będzie prawdopodobnie mylący i nie będzie miał sensu fizycznie. Możesz też mieć dość bałaganu, jeśli zapomnisz wyczyścić bufor głębokości, używając polecenia glClear pokazanego wcześniej w tej sekcji, w tym samym czasie, gdy wyczyścisz bufor kolorów.

6 Przekształcenia 3D

```
glScalef(2,2,2);        // Uniform scaling by a factor of 2.

glScalef(0.5,1,1);      // Shrink by half in the x-direction only.

glScalef(-1,1,1);       // Reflect through the yz-plane.
                        // Reflects the positive x-axis onto negative x.
```

```
glTranslatef(5,0,0);    // Move 5 units in the positive x-direction.

glTranslatef(3,5,-7.5); // Move each point (x,y,z) to (x+3, y+5, z-7.5).

glRotatef(90,1,0,0);    // Rotate 90 degrees about the x-axis.
                        // Moves the +y axis onto the +z axis
                        //    and the +z axis onto the -y axis.

glRotatef(-90,-1,0,0);  // Has the same effect as the previous rotation.

glRotatef(90,0,1,0);    // Rotate 90 degrees about the y-axis.
                        // Moves the +z axis onto the +x axis
                        //    and the +x axis onto the -z axis.

glRotatef(90,0,0,1);    // Rotate 90 degrees about the z-axis.
                        // Moves the +x axis onto the +y axis
                        //    and the +y axis onto the -x axis.

glRotatef(30,1.5,2,-3); // Rotate 30 degrees about the line through
                        //    the points (0,0,0) and (1.5,2,-3).
```

7 Modelowanie hierarchiczne

W modelowaniu hierarchicznym obiekt może być zdefiniowany we własnym lokalnym układzie współrzędnych, zwykle używając (0,0,0) jako punktu odniesienia. Obiekt można następnie przeskalować, obrócić i przetłumaczyć, aby umieścić go we współrzędnych świata lub w bardziej złożonym obiekcie. Aby to zaimplementować, potrzebujemy sposobu ograniczenia efektu transformacji modelowania do jednego obiektu lub części obiektu. Można to zrobić za pomocą stosu transformacji. Przed narysowaniem obiektu popchnij kopię aktualnej transformacji na stos. Po narysowaniu obiektu i jego podobiektów, przy użyciu wszelkich niezbędnych transformacji tymczasowych, przywróć poprzednią transformację, wyrzucając ją ze stosu.

OpenGL utrzymuje stos transformacji i zapewnia funkcje do manipulowania tym stosem. (W rzeczywistości ma kilka stosów transformacji, dla różnych celów, co wprowadza pewne komplikacje)

Ponieważ przekształcenia są reprezentowane jako macierze, stos jest w rzeczywistości stosem macierzy. W OpenGL funkcje działające na stosie to `glPushMatrix()` i `glPopMatrix()`.

Funkcje te nie pobierają parametrów ani nie zwracają wartości. OpenGL śledzi bieżącą macierz, która jest kompozycją wszystkich zastosowanych transformacji. Wywołanie funkcji takiej jak `glScalef` po prostu modyfikuje bieżącą macierz. Gdy obiekt jest rysowany za pomocą funkcji `glVertex*`, współrzędne określone dla obiektu są przekształcane przez bieżącą macierz. Istnieje inna funkcja, która wpływa na bieżącą macierz: `glLoadIdentity()`. Wywołanie

funkcji `glLoadIdentity` ustawia bieżącą macierz jako transformację tożsamości, która w ogóle nie oznacza zmiany współrzędnych i jest zwykłym punktem wyjścia dla serii transformacji.

Gdy wywoływana jest funkcja `glPushMatrix()`, kopia bieżącej macierzy jest wypychana na stos. Zauważ, że nie zmienia to bieżącej macierzy; po prostu zapisuje kopię na stosie. Gdy wywoływana jest `glPopMatrix()`, macierz na szczycie stosu jest wyrzucana ze stosu, i ta macierz zastępuje bieżącą macierz. Należy zauważyć, że `glPushMatrix i glPopMatrix` muszą zawsze występować w odpowiednich parach; `glPushMatrix` zapisuje kopię bieżącej macierzy, a odpowiednie wywołanie `glPopMatrix` przywraca tę kopię. *Pomiędzy wywołaniem glPushMatrix a odpowiadającym mu wywołaniem glPopMatrix mogą istnieć dodatkowe wywołania innych funkcji*, o ile są one prawidłowo sparowane. Zazwyczaj wywołujesz *glPushMatrix przed narysowaniem obiektu* i `glPopMatrix po zakończeniu tego obiektu`.

Rysowanie podobiektów może wymagać dodatkowych par wywołań do tych funkcji!.

Przykład (rysowanie sześcianu)

Rysowanie ściany

```
void square( float r, float g, float b ) {
    glColor3f(r,g,b);
    glBegin(GL_TRIANGLE_FAN);
    glVertex3f(-0.5, -0.5, 0.5);
    glVertex3f(0.5, -0.5, 0.5);
    glVertex3f(0.5, 0.5, 0.5);
    glVertex3f(-0.5, 0.5, 0.5);
    glEnd();
}
```

Rysowanie sześcianu

```
void cube(float size) {  // draws a cube with side length = size

    glPushMatrix();  // Save a copy of the current matrix.
    glScalef(size,size,size); // scale unit cube to desired size

    square(1, 0, 0); // red front face

    glPushMatrix();
    glRotatef(90, 0, 1, 0);
    square(0, 1, 0); // green right face
    glPopMatrix();

    glPushMatrix();
    glRotatef(-90, 1, 0, 0);
    square(0, 0, 1); // blue top face
    glPopMatrix();
```

```
    glPushMatrix();
    glRotatef(180, 0, 1, 0);
    square(0, 1, 1); // cyan back face
    glPopMatrix();

    glPushMatrix();
    glRotatef(-90, 0, 1, 0);
    square(1, 0, 1); // magenta left face
    glPopMatrix();

    glPushMatrix();
    glRotatef(90, 1, 0, 0);
    square(1, 1, 0); // yellow bottom face
    glPopMatrix();

    glPopMatrix(); // Restore matrix to its state
                   // before cube() was called.

}
```

Literatura

Uwaga!

Teoretyczne podstawy przekształceń geometrycznych umieszczone w prezentacji wykładu https://drive.google.com/drive/folders/0B0-jI5UeRsjcVExBTExOcXNlcGc

OpenGL w języku C https://drive.google.com/open?id=1Md8v8VI3TILcCGwm_Z3l8hwRJlyFfgJ0S1yQ-93rGgA

W języku angielskim

- książka interakcyjna: OpenGL http://math.hws.edu/graphicsbook/c3/s2.html (rozdziały 3.1-3.2)

8 Zadanie

Stworzyć dwa obiekty przy użyciu OpenGL. Po uruchomieniu zakończonego programu naciśnięcie jednego z klawiszy numerycznych 1 lub 2 spowoduje wybranie wyświetlanego obiektu. Program już ustawia wartość zmiennej globalnej, `objectNumber`, aby powiedzieć, który obiekt ma zostać narysowany. Użytkownik może obracać obiekt za pomocą klawiszy strzałek, PageUp, PageDown i Home. Podprogram `display()` jest wywoływany, aby narysować obiekt. Podprogram ten z kolei wywołuje `draw()` i właśnie w draw () powinieneś wykonać podstawową pracę. (Miejsce jest oznaczone `TODO`.) Dodaj również kilka nowych podprogramów do programu.

Obiekt 1. Korkociąg wokół osi x | y | z zawierający N obrotów. Punkty są stopniowo powiększane. Ustalić aktualny kolor rysujący na zielony | niebieski | brązowy | ... .

Obiekt 2. Pyramida, wykorzystując dwa wachlarze trójkątów oraz modelowanie hierarchiczne (najpierw tworzymy podprogramę rysowania jednego trójkonta; dalej wykorzystując przekształcenia geometryczne tworzymy pyramidę). Podstawą pyramidy jest wielokąt o N wierzchówkach.