Przecinanie figur

20 maja 2019

Zadanie polega na przecięciu (wyznaczeniu części wspólnej) dwóch niekoniecznie wypukłych figur ze sobą. Zakładamy, że figury nie mają samoprzecięć. Z grubsza algorytm będzie polegał na znajdowaniu punktów przecięć dwóch figur, a następnie używania ich do wyznaczenia przecięcia dwóch figur.

Informacje wstępne

Wielokąt przechowujemy jako listę dwukierunkową wierzchołków. Używamy wbudowanej klasy `LinkedList<Vertex>`. Klasa ta nie pozwala niestety na zawijanie wierzchołków (ostatni element wskazuje na pierwszy), dlatego aby samemu dokonać takiego zawijania należy użyć konstrukcji `c = c.Next ?? c.List.First;`. Typ wierzchołka na liście to `LinkedListNode<Vertex>`.

Przypadki szczególne

Algorytm niezbyt dobrze radzi sobie z następującymi przypadkami szczególnymi, dlatego nie należy ich rozważać:

- wierzchołek jednego wielokąta leży na boku drugiego wielokąta,
- boki dwóch wielokątów pokrywają się, czyli mają więcej niż jeden punkt przecięcia.

Za to następujące przypadki szczególne powinny być rozważane:

- wielokąt jest całkowicie zawarty w drugim,
- wielokąty nie mają części wspólnej (wtedy należy zwrócić pustą listę).

Co można

Można jedynie modyfikować metody w pliku `Lab12.cs`. Można dopisywać metody pomocnicze w klasie `Clipper`. Nie można modyfikować plików `Vertex.cs` i `Polygon.cs`.

Etap 1 (1 pkt)

Napisać funkcję `void MakeIntersectionPoints(Polygon source, Polygon clip)`. Funkcja ma za zadanie przetestować wszystkie pary boków figur takie że pierwszy bok pochodzi z pierwszej figury, a drugi z drugiej pod kątem przecięć. Jeśli taka para boków się przecina, należy wyznaczyć punkt przecięcia. Następnie należy taki punkt wstawić w odpowiednie miejsce obu wielokątów (wstawiane punkty przecięć powinny mieć ustawione pole `IsIntersection` na `true`). W szczególności należy pamiętać, że może się okazać, że pomiędzy daną parę oryginalnych wierzchołków figury wstawimy więcej niż jeden punkt przecięcia. Ostatecznie muszą one znajdować się w odpowiedniej kolejności. Tę należy zachować dzięki przechowywaniu informacji, w jakiej części odcinka znajduje się punkt przecięcia (liczba z przedziału $[0, 1]$, pole `Distance`). Co więcej, punkt powinien znać swój odpowiednik w drugiej figurze (przyda się w dalszej części zadania, pole `CorrespondingVertex`). Odpowiednik to punkt przecięcia o tych samych współrzędnych, ale umieszczony w drugiej figurze.

Przydatne pola w klasie `Vertex`:

- `IsIntersection` - czy dany punkt jest punktem przecięcia,

- `Distance` - w jakiej części odcinka znajduje się punkt,
- `CorrespondingVertex` - odpowiednik w drugim wielokącie.

Dana jest już funkcja `GetIntersectionPoints()`, która znajduje punkt przecięcia oraz wylicza odpowiednią wartość pola `Distance`.

Uwaga: funkcja nie powinna modyfikować argumentów wejściowych. Powinna zwracać nowe wielokąty, będące kopiami oryginalnych wielokątów z wstawionymi wierzchołkami.

Uwaga2: Do znajdowania przecięć pomiędzy bokami użyć algorytmu naiwnego (każdy z każdym) o złożoności $O(nm)$, gdzie n i m są liczbami boków obu wielokątów.

Etap 2 (1 pkt)

Napisać funkcję `void MarkEntryExitPoints(Polygon source, Polygon clip)`.

Wyobraźmy sobie, że zaczynamy od pewnego wierzchołka pierwszej figury oraz że ten wierzchołek jest poza drugą figurą. Idziemy wzdłuż boku figury. W końcu natrafimy na punkt przecięcia. Wtedy oznaczamy go jako punkt wejściowy (pole `IsEntry` ustawiamy na `true`), ponieważ wchodzimy w drugą figurę. Gdy napotkamy kolejny punkt przecięcia, będzie to punkt wyjściowy (pole `IsEntry` ustawiamy na `false`). I tak na przemian. Pamiętajmy, że algorytm ulegnie modyfikacji, gdy zaczynamy w wierzchołku który nie jest na zewnątrz.

W tym etapie należy odpowiednio oznaczyć wierzchołki przecięcia w obu figurach. Może się zdarzyć tak, że wierzchołek o tych samych współrzędnych będzie w jednej figurze wejściowym, a w drugiej wyjściowym.

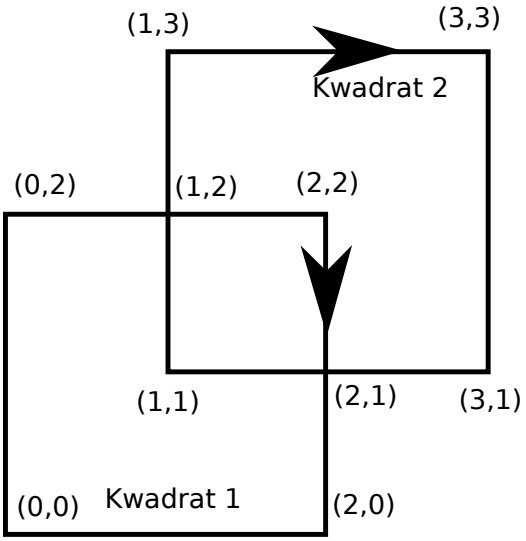
To, czy dany wierzchołek jest wejściowy, czy nie, zależy także od tego, w jakiej kolejności przechowujemy wierzchołki na liście (czy zgodnie z ruchem wskazówek zegara, czy przeciwnie). My oczywiście idziemy naprzód listy, czyli od pierwszego wierzchołka do ostatniego.

Dana jest już funkcja `IsInside()`, która stwierdza, czy dany wierzchołek znajduje się wewnątrz danej figury.

Uwaga: funkcja nie powinna modyfikować argumentów wejściowych. Powinna zwracać nowe wielokąty, będące kopiami oryginalnych wielokątów z wstawionymi wierzchołkami z ustawionymi odpowiednimi polami.

Uwaga2: zakładamy, że funkcja z tego etapu na samym początku wywołuje funkcję z etapu 1. Wywołanie jest już napisane w dostarczonym kodzie.

Przykład



Prześledźmy przykład. Strzałki na kwadratach oznaczają, w którą stronę idą punkty na liście. Zacznijmy w punkcie $(0, 0)$. Idąc do przodu przejdziemy do punktu $(0, 2)$, a następnie $(1, 2)$. Punkt $(1, 2)$ jest punktem przecięcia i przechodząc przez niego wchodzimy w kwadrat 2. Dlatego jest to punkt wchodzący, czyli jego pole `IsEntry` jest równe `true`. Z podobnych powodów punkt $(2, 1)$ jest punktem wychodzącym z punktu widzenia kwadratu 1 (bo idąc po kolei przez listę punktów opuszczamy kwadrat 2).

Inaczej sytuacja ma się dla kwadratu 2. Zacznijmy od punktu $(1, 1)$. Idąc do przodu, docieramy do punktu $(1, 2)$, ale tym razem przechodząc przez niego wychodzimy z kwadratu 1, dlatego też jest to punkt wychodzący (`IsEntry` równe `false`). Z analogicznych powodów punkt $(2, 1)$ jest punktem wchodzącym.

Etap 3 (2 pkt)

Napisać funkcję `List<Polygon> ReturnClippedPolygons(Polygon source, Polygon clip)`. W tym etapie rozpoczynamy od dowolnego wierzchołka przecięcia jednej z figur, a następnie poruszamy się wzdłuż boków, aż nie domkniemy powstałej figury. Domkniętą figurę wrzucamy na zwracaną listę. Jeżeli wierzchołek przecięcia jest wejściowy, należy poruszać się do przodu. W przeciwnym wypadku do tyłu.

Jeżeli podczas poruszania się napotkamy kolejny wierzchołek przecięcia, należy przenieść się na drugą figurę i wzdłuż jej boków kontynuować trasę. Ten nowy wierzchołek przecięcia (jego wersja w drugiej figurze) wyznacza, czy od teraz poruszamy się do przodu czy do tyłu.

Algorytm kończy się, gdy przetworzymy wszystkie wierzchołki przecięcia.

Uwaga: zakładamy, że funkcja z tego etapu na samym początku wywołuje funkcję z etapu 2. Wywołanie jest już napisane w dostarczonym kodzie.

Przykład

Spójrzmy jeszcze raz na przykładowe kwadraty. Zaczynamy od dowolnego punktu przecięcia: np. od $(1, 2)$ na kwadracie 1. W którą stronę (do przodu albo do tyłu) należy się udać po kwadracie 1, aby iść po części wspólnej? Ponieważ punkt $(1, 2)$ jest punktem wchodzącym z punktu widzenia kwadratu 1, to udajemy się do przodu (wgłąb drugiej figury). Gdy dotrzemy do kolejnego punktu przecięcia, $(2, 1)$, wiemy, że dalsza droga po kwadracie 1 nie ma sensu, gdyż wyjdziemy poza część wspólną. Dlatego przechodzimy na kwadrat

2 (używając pola `CorrespondingVertex`). Czy teraz należy poruszać się do przodu czy do tyłu? Ponieważ $(2, 1)$ jest punktem wchodzącym dla kwadratu 2, należy iść naprzód, aż do zamknięcia figury.