Politechnika Krakowska

Wydział Fizyki, Matematyki i Informatyki

ZADANIE 4

Zaawansowane techniki programowania

Prowadzący: dr inż. Jerzy Jaworowski

II stopień - rok 1, semestr 1

wykonanie:

Piotr Adam Tomaszewski

Nr albumu: 104896

Wstęp

Celem zadania było obliczenie objętości graniastosłupa prostego o podstawie P oraz wysokości h. Na samym początku należało utworzyć połączenie z bazą danych SQL z wykorzystaniem usługi JNDI. Następnie należało pobrać wartości wierzchołków w przestrzeni 3D z bazy danych i zapisać je w programie. Pole graniastosłupa należało wyznaczyć z wykorzystaniem otoczki wypukłej rzutu punktów x i y na płaszczyznę. Wysokość graniastosłupa należało wyznaczyć ze średniej wartości współrzędnych z.

Organizacja plików

IGameRemote.java

Interfejs ***IGameRemote*** posiadający deklarację metody, która pozwala na zarejestrowanie użytkownika w systemie.

Metoda ***register*** rejestrująca użytkownika w systemie - zwraca $true$ jeżeli proces rejestracji zakończył się poprawnie. Jeżeli rejestracja zakończyła się niepowodzeniem, metoda register zwraca wartość $false$.

@param **hwork** – numer zadania

@param **album** – numer albumu studenta

@return wartość logiczna czy połączenie przebiegło pomyślnie

```
@Remote
public interface IGameRemote {
    /**
     * Metoda register rejestrująca użytkownika w systemie - zwraca true
     * jeżeli proces rejestracji zakończył się poprawnie.
     * Jeżeli rejestracja zakończyła się niepowodzeniem, metoda register
     * zwraca wartość false.
     * @param hwork - numer zadania
     * @param album - numer albumu studenta
     * @return wartość logiczna czy połączenie przebiegło pomyślnie
     */
public boolean register(int hwork, String album);
}
```

Rysunek 1 Implementacja interfejsu IGameRemote

ICuboidRemote

Interfejs ***ICuboidRemote*** zawierający deklaracje metod wykorzystywanych w klasie Cuboid niezbędny do użycia bean'a przekazującego listę punktów z servletu do klasy Cuboid.

```java
@Remote
public interface ICuboidRemote {
    double actions (int i, int n, float sX, float sY, double field);
    double general (List<Float> arr);
    void prepare (List<Float> arr, int i);
    void graham(Cuboid.Point2d[] tan, int n);
    void addElemStact(int nr);
    void remElemStact();
    int det(int a,int b,int c);
    void stackAddRem(int i, Cuboid.Point2d[] tan, int l);
    double countFields(int i, double field, int stcSize);
}
```

Rysunek 2 Implementacja interfejsu ICuboidRemote

Solver.java

Klasa ***Solver*** będąca servletem pozwalającym na połączenie między aplikacją a bazą danych.

```java
public class Solver extends HttpServlet {
    @EJB
    private ICuboidRemote cuboid;
```

Rysunek 3 Implementacja servletu Solver

Metoda ***getConnection*** łącząca się z bazą danych za pomocą nazwy JNDI na serwerze Glassfish.

@param **ds** - nazwa JNDI (parametr poprany metodą GET)

@return - połączenie z bazą

@throws **SQLException** - zapewnia informacje o błędzie dostępu do bazy danych

@throws **NamingException** - zapewnia informacje o błędzie gdy nazwa JNDI jest niepoprawna

```java
private Connection getConnection(String ds) throws
        SQLException,NamingException {
    Connection connection = null;
    InitialContext context = new InitialContext();
    DataSource dataSource = (DataSource) context.lookup(ds);
    connection = dataSource.getConnection();

    return connection;
}
```

Rysunek 4 Implementacja metody getConnection

Metoda ***addPointsToList*** pobierająca wartości punktów z bazy i zapisująca je do listy.

@param **rs** - zbiór rekordów zwróconych po wykonaniu zapytania w bazie danych

@throws **SQLException** - zapewnia informacje o błędzie dostępu do bazy danych

```
private void addPointsToList(ResultSet rs) throws SQLException{
    for (;;){
                if (rs.next()){
                    points.add(rs.getFloat("x"));
                    points.add(rs.getFloat("y"));
                    points.add(rs.getFloat("z"));
                }else break;
            }
}
```

Rysunek 5 Implementacja metody addPointsToList

Metoda ***getPoints*** w której pobierane są wyniki zapytania w bazie danych i przekazywane są do zapisania na liście.

@param **connection** - referencja do otwartego połączenia z bazą danych

@param **sql** - treść zapytania niezbędna do pobrania rezultatu

@param **response** - odpowiedź servletu

@throws **Exception** -przechwycenie zdarzenia, które może zakłócić poprawne działanie programu

@throws **IOException** – w przypadku wystąpienia błędu wejścia/wyjścia

```
private void getPoints(Connection connection, String sql,
        HttpServletResponse response) throws  Exception,IOException {
    Statement stmt;
    ResultSet rs;
    if (connection != null) {
            stmt =  connection.createStatement();
            rs = stmt.executeQuery(sql);
            addPointsToList(rs);
    }
    response.setContentType("text/html;charset=UTF-8");
    response.getWriter().format(Locale.ENGLISH, "%.5f",
            cuboid.general(points));
}
```

Rysunek 6 Implementacja metody getPoints

Metoda ***doGet***, dzięki której po przekazaniu parametru w żądaniu URL, następuje rejestracja użytkownika i jeśli przebiegnie pomyślnie wykonywane są metody określające połączenie z bazą i pobranie danych z bazy.

@param **request** - żądanie servletu

@param **response** - odpowiedź servletu

@throws **ServletException** – w przypadku wystąpienia błędu związanego z servletem

@throws **IOException** – w przypadku wystąpienia błędu wejścia/wyjścia

```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    try {
        String connect = "java:global/ejb-project/GameManager!"
                +"pl.jrj.game.IGameRemote";

        InitialContext con = new InitialContext();
        pl.jrj.game.IGameRemote game = (pl.jrj.game.IGameRemote)
                con.lookup(connect);

        if (game.register(4, "104896")){
            getPoints(getConnection(request.getParameter("ds"))
                    ,"SELECT * FROM fttable; ",response);
        }
     //   processRequest(request, response);
    } catch (NamingException ex) {
        Logger.getLogger(Solver.class.getName()).
                log(Level.SEVERE, null, ex);
    } catch (Exception e) {
        Logger.getLogger(Solver.class.getName()).
                log(Level.SEVERE, null, e);
    }
}
```

Rysunek 7 Implementacja metody doGet

Cuboid.java

Klasa ***Cuboid*** implementująca metody interfejsu ICuboidRemote służące do obliczenia objętości graniastosłupa prostego mając jego wierzchołki w przestrzeni 3D przy użyciu m.in. algorytmu Grahama. Posiada pola:

stack - reprezentujący stos zaimplementowany jako lista;

points - tablica zawierająca zbiór punktów w przestrzeni 2D wykorzystywana w algorytmie Grahama.

pocz - wartownik wskazujący ilość elementów na stosie

z - suma wartości indeksów z w przestrzeni 3D

```
@Stateless
public class Cuboid implements ICuboidRemote {
    List<Integer> stack = new ArrayList<Integer>();
    Point2d[] points;
    int pocz=0;
    float z = 0;
```

Rysunek 8 Implementacja klasy Cuboid

Główna metoda ***general*** wywoływana przez bean z serwletu określająca wykonywanie kolejnych metod w klasie.

@param **arr** - lista wszystkich pobranych punktów z bazy danych

@return - objętość graniastosłupa

```
@Override
public double general (List<Float> arr){
    double wynik;
    prepare(arr,0);
    wynik=actions(0,arr.size()/3,points[0].getX(),points[0].getY(),0);
    return wynik;
}
```

Rysunek 9 Implementacja metody general

Metoda ***prepare*** przyporządkowująca wartości punktów z listy do tablicy obiektów w przestrzeni 2D. Metoda ma na celu odseparowanie ciągu liczb na odpowiadające im współrzędne X i Y. Współrzędna Z jest od razu sumowana aby wyznaczyć jej średnią.

@param **arr** - lista wszystkich pobranych punktów z bazy danych

@param **i** - zmienna pomocnicza do iteracji po elementach

```
@Override
public void prepare(List<Float> arr, int i) {
    points = new Point2d[arr.size()/3];
    int j=0;
    int arrSize = arr.size();
    for (i=0;i<arrSize;i+=3){

        float x = arr.get(i);
        float y = arr.get(i+1);
        z+= arr.get(i+2);
        points[j] = new Cuboid.Point2d(x,y);
        j++;
    }
}
```

Rysunek 10 Implementacja metody prepare

Metoda ***countFields*** pozwalająca obliczyć pole figury zrzutowanej na przestrzeń dwuwymiarową za pomocą wzoru do analitycznego obliczania pól.

@param **i** - zmienna pomocnicza wykorzystywana do iteracji

@param **field** - zmienna pomocnicza przechowująca wartość pola

@param **stcSize** - rozmiar stosu

@return - ostateczna wartość pola

```
@Override
public double countFields(int i, double field, int stcSize) {
    for (i = 0;i<stcSize;i++ ){
        if (i==0){
                field+= (points[stack.get(i)].getX())*
                        (points[stack.get(i+1)].getY()-
                       points[stack.get(stack.size()-1)].getY());
            }else if (i==stack.size()-1){
                field+= (points[stack.get(i)].getX())*
                        (points[stack.get(0)].getY()-
                       points[stack.get(i-1)].getY());
            }else{
                field+= (points[stack.get(i)].getX())*
                        (points[stack.get(i+1)].getY()-
                       points[stack.get(i-1)].getY());
            }
    }
    return field;
}
```

Rysunek 11 Implementacja metody countFields

Metoda ***actions*** wykonująca kolejne kroki pozwalające wyznaczyć pole graniastosłupa przy wykorzystaniu otoczki wypukłej. Po załadowaniu elementów do tablic następuje sortowanie ich w tablicy. Potem wywoływane są metody odpowiedzialne za algorytm Grahama i następuje wypisanie objętości graniastosłupa.

@param **i** - zmienna pomocnicza wykorzystywana do iteracji

@param **n** - ilość wierzchołków graniastosłupa

@param **sX** - współrzędna x punktu początkowego

@param **sY** - współrzędna y punktu początkowego

@param **field** - zmienna przechowująca wartość pola

@return - objętość graniastosłupa

```
@Override
public double actions(int i, int n, float sX, float sY, double field) {
    Point2d[] tan = new Point2d[n-1];
    Arrays.sort(points,new Compar());
    for (i = 1; i < n;i++){
        tan[i-1]=new Cuboid.Point2d(i,(
                (points[i].getX()-sX)/(points[i].getY()-sY)));
    }
    Arrays.sort(tan,new Compar());
    graham(tan,n);

    field = (Math.abs(countFields(0,0,stack.size())))/2;
    return (field*(z/n));
}
```

Rysunek 12 Implementacja metody actions

Metoda ***stackAddRem*** algorytmu Grahama pobierająca każdorazowo wartość iteracyjną z tablicy (getX) i dodająca do stosu. W zależności od wyniku wyznacznika wartości są usuwane ze stosu (usuwanie punktów, które spowodowały by zbudowanie otoczki wklęsłej).

@param **i** - zmienna pomocnicza wykorzystywana do iteracji

@param **tan** - tangens nachylenia punktu do osi X

@param **l** - rozmiar tablicy tangens

```
@Override
public void stackAddRem(int i, Point2d[] tan, int l) {
    for (i = l; i >=0;i--){
            int nr=(int) tan[i].getX();
            if (pocz==1) addElemStact(nr);
            else{
                 while (det(stack.get(pocz-2),stack.get(pocz-1),nr)==0
                         && pocz!=1)
                     remElemStact();
                addElemStact(nr);
            }
    }
}
```

Rysunek 13 Implementacja metody stackAddRem

Metoda ***graham*** implementująca w głównej części algorytm Grahama określający kierunki w których przechodzimy do kolejnego punktu tak, by zbudować otoczkę wypukłą. Do pomocy wykorzystywany jest stos.

@param **tan** - tangens nachylenia punktu do osi X

@param **n** - ilość wierzchołków graniastosłupa

```
@Override
public void graham(Point2d[] tan, int n) {
    int dett;
   addElemStact(0);
   stackAddRem(0,tan,tan.length-1);
   while (det(stack.get(pocz-2),stack.get(pocz-1),0)==0 && pocz!=1)
       remElemStact();
    if (pocz!=n)
        addElemStact(0);
}
```

Rysunek 14 Implementacja metody graham

Metoda ***addElemStact*** dodająca do stosu wartość i inkrementująca ilość znajdujących się na nim elementów.

@param **nr** - wartość do dodania na stos

```
@Override
public void addElemStact(int nr) {
    stack.add(nr);
    pocz++;
}
```

Rysunek 15 Implementacja metody addElemStack

Metoda ***remElemStact*** usuwająca wartości ze stosu i dekrementująca ilość znajdujących się na nim elementów.

```
@Override
public void remElemStact() {
    stack.remove(stack.size()-1);
    pocz--;
}
```

Rysunek 16 Implementacja metody remElemStack

Metoda ***det*** zwracająca wartość wyznacznika ze współrzędnych punktów zawartych w tablicy współrzędnych graniastosłupa

@param **a** - indeks trzeciej współrzędnej

@param **b** - indeks drugiej współrzędnej

@param **c** - indeks pierwszej współrzędnej

@return określenie czy wyznacznik jest dodatni czy ujemny

```
@Override
public int det(int a, int b, int c) {
    double p=points[a].getX()*points[b].getY()+
             points[b].getX()*points[c].getY()+
             points[c].getX()*points[a].getY()-
             points[c].getX()*points[b].getY()-
             points[a].getX()*points[c].getY()-
             points[b].getX()*points[a].getY();
    if (p>=0)
        return 1;
    else
        return 0;
}
```

Rysunek 17 Implementacja metody det

Klasa wewnętrzna ***Compar*** implementująca interfejs i udostępniająca metodę compare, która przyjmuje 2 obiekty i zawiera utworzony schemat postępowania w przypadku różnych lub równych obiektów.

```
private class Compar implements Comparator<Point2d>{

    @Override
    public int compare(Point2d o1, Point2d o2) {
        Float y1 = ((Point2d) o1).getY();
        Float y2 = ((Point2d) o2).getY();
        int sComp = y1.compareTo(y2);

        if (sComp!=0){
            return sComp;
        } else{
            Float x1 = ((Point2d) o1).getX();
            Float x2 = ((Point2d) o2).getX();
            return x1.compareTo(x2);
        }
    }

}
```

Rysunek 18 Implementacja klasy Compar

Klasa ***Point2d*** reprezentująca zbiór punktów x, y w przestrzeni dwuwymiarowej.

```
public class Point2d {
    float x;
    float y;

    public Point2d(){}

    public Point2d (float x, float y){
        this.x = x;
        this.y = y;
    }

    public float getX() {
        return x;
    }
```

Rysunek 19 Implementacja klasy Point2d

Algorytm Grahama – wyznaczanie otoczki wypukłej

Zapewnia obliczenie otoczki wypukłej zestawu punktów na płaszczyźnie. Działa w czasie O(n log n) i używa O(n) dodatkowej pamięci.

Na początku sortujemy zbiory punktów względem wartości Y następnie po wartości X. Potem bierzemy pierwszą wartość z posortowanego zbioru. Następnie tworzymy listę punktów z wartością kątową miedzy punktem początkowym a każdym kolejnym punktem i sortujemy ją. W kolejnym kroku będziemy rozpatrywali 3 punkty (ozn. A, B, C) zaczynając od tego o najmniejszej wartości współrzędnej x oraz y. Jeśli punkt B leży na zewnątrz prowizorycznego trójkąta zbudowanego na wierzchołku początkowym, oraz wierzchołku A i C to on należy do otoczki wypukłej. Jeśli B leży wewnątrz to B nie należy do otoczki wypukłej - należy usunąć ten punkt ze stosu i cofnąć się o jedną pozycję (o ile jest różna od zera).