МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНДУСТРИАЛЬНЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВПО «МГИУ»)
КАФЕДРА ИНФОРМАЦИОННЫХ СИСТЕМ И ТЕХНОЛОГИЙ

КУРСОВАЯ РАБОТА

по дисциплине «Методы хранения и обработки информации»
на тему «Вычисление количества вершин выпуклой оболочки, лежащих в 1-окрестности заданного заполненного треугольника. Сумма длин проекций невидимых частей частично видимых рёбер, образующих с горизонтальной плоскостью угол не более $\pi/7$, центр которых находится строго внутри сферы $x^2 + y^2 + z^2 = 4$ »

Группа 2362

Студент М.У. Бирюков

Руководитель работы
к.ф.-м.н., доцент Е.А. Роганов

Москва 2014

Аннотация

Работа посвящена модификации проектов «Выпуклая оболочка» и «Изображение проекции полиэдра». В проекте «Выпуклая оболочка» необходимо вычислить количество вершин выпуклой оболочки, лежащих в 1-окрестности заданного заполненного треугольника. Во проекте «Изображение проекции полиэдра» необходимо вычислить сумму длин проекций невидимых частей частично видимых рёбер, образующих с горизонтальной плоскостью угол не более $\pi/7$, центр которых находится строго внутри сферы $x^2 + y^2 + z^2 = 4$.

Содержание

1. Введение 3
2. Модификация проекта «Выпуклая оболочка» 3
3. Модификация проекта «Изображение проекции полиэдра» 14

1. Введение

Проект «Выпуклая оболочка»[1] решает задачу индуктивного перевычисления выпуклой оболочки последовательно поступающих точек плоскости и таких её характеристик, как периметр и площади выпуклой оболочки. Целью данной работы является вычисление количества вершин выпуклой оболочки, которые находятся в заданном треугольнике или в единичной окрестности заданного треугольника. Решение этой задачи требует знания теории индуктивных функций [2], основ аналитической геометрии, векторной алгебры и языка Ruby [3].

Проект «Изображение проекции полиэдра» [4] — пример классической задачи, для успешного решения которой необходимо знакомство с основами вычислительной геометрии. Задачей, решаемой в данной работе, является модификация эталонного проекта с целью определения суммы длин проекций невидимых частей частично видимых рёбер, образующих с горизонтальной плоскостью угол не более $\pi/7$, центр которых находится строго внутри сферы $x^2 + y^2 + z^2 = 4$. Для этого необходимы хорошее понимание ряда разделов аналитической геометрии и векторной алгебры, основ объектно-ориентированного программирования и языка Ruby.

Общее количество строк в рассмотренных проектах составляет около 1180, из которых более 350 были изменены или добавлены автором в процессе работы над задачами модификации.

2. Модификация проекта «Выпуклая оболочка»

Точная постановка задачи

Вычисляется количество вершин выпуклой оболочки, лежащих в 1-окрестности заданного заполненного треугольника.

Решение данной задачи и модификация кода

Эталонный проект до модификации индуктивно строит выпуклую оболочку по точкам. При добавлении новой точки программа индуктивно перевычисляет периметр и площадь оболочки. Наша цель - модифицировать эталонный проект таким образом, чтобы вычислялось количество вершин выпуклой оболочки, лежащих в 1-окрестности заданного заполненного треугольника..

Для начала модифицируем файл `convex/r2point.rb` добавив в класс `R2Point` методы, которые выполняют следующие действия:

- нахождение расстояния от точки до отрезка;
- проверка принадлежности точки треугольнику;
- проверка принадлежности точки окрестности и треугольнику.

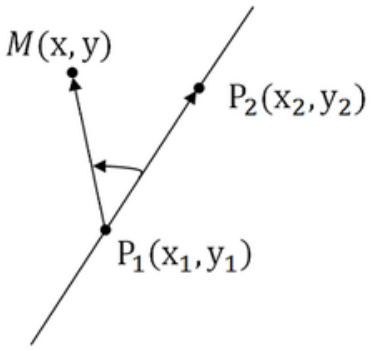
```
def dist_segm(a,b)
  p1=(@x-a.x)*(b.x-a.x)+(@y-a.y)*(b.y-a.y)
  p2=(a.x-b.x)*(@x-b.x)+(a.y-b.y)*(@y-b.y)
  return Math.sqrt((@x-a.x)**2+(@y-a.y)**2) if p1<=0
  return Math.sqrt((@x-b.x)**2+(@y-b.y)**2) if p2<=0
  a=(a.y-b.y)*@x+(b.x-a.x)*@y+(a.x*b.y-b.x*a.y
  b=Math.sqrt((b.x-a.x)**2+(b.y-a.y)**2)
  return (a/b).abs
end
```

Метод `dist_segm(a,b)` основан на векторной алгебре. Первоначально определяется, лежит ли точка левее или правее отрезка, для этого вычисляется скалярное произведение от искомой точки до каждого из карев отрезка. Если одно из этих скалярных произведений меньше или равно нулю, то возвращается расстояние от точки до края отрезка, скалярное произведение с которым было меньше либо равно нулю, причем считается расстояние как гипотенуза прямоугольного треугольника. Если же скалярные произведения больше нуля, то расстояние считается по формуле, которая указана ниже.

$$\mathbf{d}(\mathbf{P},\mathbf{L}) = \frac{(y_0 - y_1)x + (x_1 - x_0)y + (x_0 y_1 - x_1 y_0)}{\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}}$$

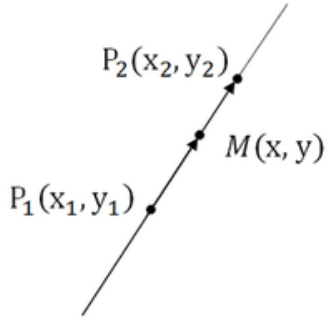
```
def inside_triangle?(a,b,c)
  l=(a.x-@x)*(b.y-a.y)-(b.x-a.x)*(a.y-@y)
  m=(b.x-@x)*(c.y-b.y)-(c.x-b.x)*(b.y-@y)
  n=(c.x-@x)*(a.y-c.y)-(a.x-c.x)*(c.y-@y)
  (l>0 && m>0 && n>0) || (l<0 && m<0 && n<0)
end
```

Данный метод использует псевдоскалярное произведение для определения, находится ли точка внутри треугольника. 2 вершины треугольника образуют отрезок, который можно воспринять как луч. Этот луч делит всю плоскость на 2 полуплоскости. Берутся 2 вершины треугольника и обозреваемая точка, затем относительно них выполняется псевдоскалярное произведение.



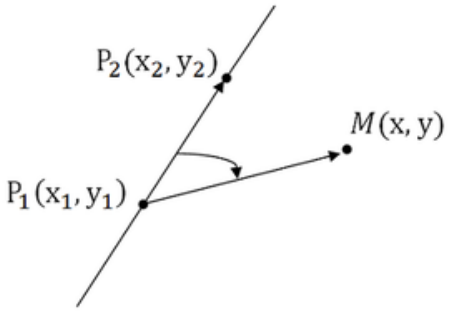
$[P_1P_2, P_1M] > 0$

Точка лежит в верхней полуплоскости



$[P_1P_2, P_1M] = 0$

Точка лежит на прямой



$[P_1P_2, P_1M] < 0$

Точка лежит в нижней полуплоскости

Знак числа, которое получается при вычислении говорит нам, в какой из полуплоскостей относительно стороны треугольника лежит точка. Если все знаки чисел, полученных во время вычисления псевдоскалярных произведений относительно всех сторон треугольника, совпадают, значит точка лежит внутри треугольника, иначе, точка находится вне треугольника.

```
def is_inside?(a,b,c)
  return true if self.inside_triangle?(a,b,c) #если в треугольнике
  return true if (self.dist_segm(a,b)<=1 && self.dist_segm(a,b)>=0) ||
  (self.dist_segm(b,c)<=1 && self.dist_segm(b,c)>=0) ||
  (self.dist_segm(c,a)<=1 && self.dist_segm(c,a)>=0) ||
  (self.dist_segm(a,c)<=1 && self.dist_segm(a,c)>=0)
  return false
end
```

Метод `is_inside?(a,b,c)` выполняет финальную проверку на принадлежность точки треугольнику или окрестности треугольника. Первоначально, он проверяет, лежит ли точка внутри треугольника, вызывая метод `inside_triangle?(a,b,c)`. Если этот метод вернул true, то и метод `is_inside?(a,b,c)` возвращает true, говоря о том, что точка удовлетворяет условию. Если же был получен false, то идет проверка окрестности. Проверяется расстояние от точки до каждой из сторон при помощи метода `dist_segm(a,b)`. Если хотя бы до одной стороны расстояние больше нуля и меньше единицы, то возвращается true, опять же говоря о том, что точка удовлетворяет условию. Во всех остальных ситуациях (если точка не лежит ни в треугольнике, ни в окрестности) возвращается false, что означает, что точка не удовлетворяет условию и увеличивать количество точек, удовлетворяющих условию, не стоит.

Теперь модифицируем файл `convex/convex.rb`. Добавим в класс `Figure` инициализацию объекта, метод создания треугольника, метод проверки на принадлежность треугольнику и метод получения количество вершин выпуклой оболочки, которые удовлетворяют нашему условию.

```
def initialize; @ins=0 end
```

Теперь при инициализации абстрактной фигуры создается переменная экземпляра класса `Figure @ins` . Так как в нашем проекте создается только одна выпуклая оболочка, мы можем использовать именно этот тип переменной.

```
def set_triangle(a,b,c)
  @@a, @@b, @@c = a,b,c
end
```

Задание треугольника производится тремя точками, которые будут храниться в переменных класса `Figure @@a, @@b, @@c` соответственно. Так как при добавлении вершин выпуклой оболочки может произойти создание объекта другого класса, нам придется использовать именно переменные класса, ибо все объекты классов `Void, Point, Segment, Polygon` наследуются от класса `Figure`, а следовательно, наследуют и точки нашего треугольника.

```
def intr?(p) ; (p.is_inside?(@@a,@@b,@@c)) ? 1 : 0 ;end
```

Метод `intr?(p)` проверяет принадлежность точки, получаемой в качестве аргумента, заданному ранее треугольнику используя метод `is_inside?(a,b,c)`, который возвращает true, в случае, если точка лежит либо в заданном треугольнике, либо в его единичной окрестности и false во всех остальных случаях. Метод `intr?(p)`, в случае, если точка попадает в заданный треугольник или его окрестность, возвращает число 1, во всех остальных случаях он возвращает 0.

```
def inside_points; @ins; end
```

Метод `inside_points` возвращает переменную экземпляра `@ins`, которая хранит количесто вершин, лежащих в заданном треугольнике или его единичной окрестности.

Теперь рассмотрим класс `Void`. Так как объект именно этого класса создается при начале работы программы, то задавать треугольник лучше всего именно в нем. Для этого добавим инициализацию объекта класса `Void`.

```
def initialize(a=R2Point.new, b=R2Point.new, c=R2Point.new)
  set_triangle(a,b,c)
end
```

Если при создании объекта класса `R2Point` не указать аргументы, то координаты точки будут запрашиваться через ручной ввод, мы этим воспользуемся. Зададим 3 точки `a, b, c` и передадим их в метод `set_triangle(a,b,c)`, чтобы задать эти точки как переменные класса `Figure`, которые в будущем будут доступны из всех классов.

После задания треугольника можно начать работать с точками выпуклой оболочки. В классе `Point` изменим инициализацию, добавив строчку, которая будет проверять, лежит ли она в заданном ранее треугольнике или его единичной окрестности.

```
class Point < Figure
  def initialize(p)
    @p = p
    @ins=intr?(@p)
  end
  ...
end
```

Если точка удовлетворяет условию, то переменная `@ins` будет равна 1, в противном случае, она будет равна 0, так как метод `intr?(p)` возвращает 1, если точка удовлетворяет условию и 0, если не удовлетворяет.

По тому же принципу изменим инициализацию класса `Segment` и класса `Polygon`

```
class Segment < Figure
  def initialize(p, q)
    @p, @q = p, q
    @ins=intr?(@p)+intr?(@q)
  end
  ...
end

class Polygon < Figure
  attr_reader :points, :perimeter, :area
  ...
  @area      = R2Point.area(a, b, c).abs
  @ins=intr?(a)+intr?(b)+intr?(c)
  end
```

Также нам нужно будет проверить пересекаются ли прямые, на которых лежит ребро выпуклой оболочки и заданный отрезок. Для этого добавим в файл `convex/convex.rb` метод `cross?` в класс `Segment`, который будет проверять данное условие.

```
def cross?(point1,point2)
 if (point1.x==point2.x && self.p.x==self.q.x)
  return false
 else
  an=(point2.y-point1.y)/(point2.x-point1.x)
  bn=(self.q.y-self.p.y)/(self.q.x-self.p.x)
```

```
  if an == bn
   return false
  else
   return true
  end
 end
end
```

Ещё нам необходимо проверить является ли найденная точка пересечением прямых или точкой пересечения отрезков. Чтобы этот случай учитывался добавим в файл `convex/convex.rb` метод `is_on_segments?` в класс `Segment`.

```
def is_on_segments?(point1,point2)
 j=crosspoint(point1,point2).inside?(point1,point2)
 k=crosspoint(point1,point2).inside?(@p,@q)
 if cross?(point1,point2)
  j and k
 else
  false
 end
end
```

Теперь можно приступить к созданию нового метода, который будет вычислять периметр части ребра, лежащего внутри или на границе заданного треугольника. Для этого создадим новый метод `part_perimeter` и добавим его в класс `Segment`.

Для начала проверим с помощью метода `outside_tr?` лежат ли обе точки отрезка внутри или на границе треугольника (рис.1).

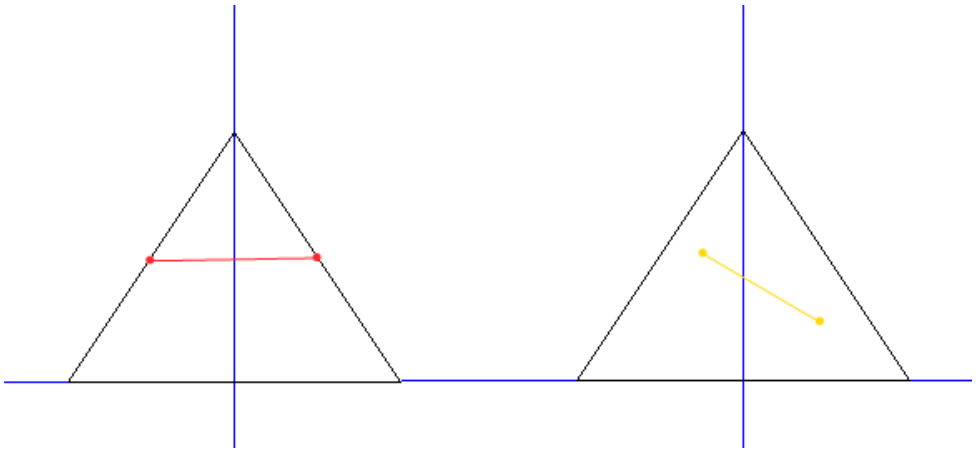


Рис. 1.

Если точки лежат, то возвращаем расстояние от одной точки до другой как периметр. Для этого используем метод `dist`, который вычисляет расстояние по формуле:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Если точки не лежат, то продолжаем проверку. Проверяем следующее условие, когда одна точка лежит снаружи, а другая находится строго внутри треугольника, тогда вычисляем расстояние от точки пересечения одной из сторон треугольника до точки лежащей строго внутри треугольника. Для нахождения точки пересечения используем метод `crosspoint` и выводим расстояние от найденой точки до точки лежащей строго внутри треугольника.

Если одна точка строго снаружи, а другая точка лежит на границе треугольника, тогда проверяем лежит ли одна точка на отрезке, а другая на прямой с помощью методов `on_segment? и on_line?`. Также необходимо учитывать, что обе точки могут лежать на одной прямой (рис. 2). Тогда выводим расстояние прямой лежащей в треугольнике или на его границе.

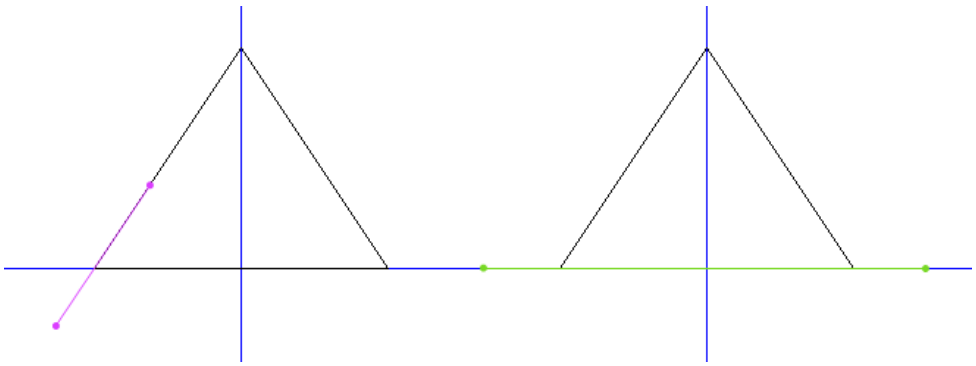


Рис. 2.

Для метода `is_on_segments?` создадим массив куда будем добавлять результат, является ли найденная точка пересечения прямых — точкой пересечения отрезков.

И выводим расстояние от одной точки до другой.

После рассмотренных случаев метод `part_perimeter` примет следующий вид:

```
class Segment < Figure
  ...
def part_perimeter()
  pt=!@p.outside_tr?(@point1,@point2,@point3)
  pt=!@q.outside_tr?(@point1,@point2,@point3)
  pq=@p.outside_tr?(@point1, @point2, @point3)
  qp=@q.inside_tr?(@point1, @point2, @point3)
  pv=@q.outside_tr?(@point1, @point2, @point3)
  qv=@p.inside_tr?(@point1, @point2, @point3)

  if pt and pt
    return @p.dist(@q)
  elsif pq and qp
    if is_on_segments?(@point1, @point2)
  return @p.dist(@q) - crosspoint(@point1, @point2).dist(@p)
elsif is_on_segments?(@point2, @point3)
  return @p.dist(@q) - crosspoint(@point2, @point3).dist(@p)
    elsif is_on_segments?(@point1, @point3)
  return @p.dist(@q) - crosspoint(@point1, @point3).dist(@p)
end
  elsif pv and qv
    if is_on_segments?(@point1, @point2)
  return @p.dist(@q) - crosspoint(@point1, @point2).dist(@q)
    elsif is_on_segments?(@point2, @point3)
  return @p.dist(@q) - crosspoint(@point2, @point3).dist(@q)
    elsif is_on_segments?(@point1, @point3)
  return @p.dist(@q) - crosspoint(@point1, @point3).dist(@q)
end
  else
    if @q.on_segment?(@point1, @point2) && @p.on_line?(@point1, @point2)
      return @p.dist(@q) - [@p.dist(@point1), @p.dist(@point2)].min
    elsif @q.on_segment?(@point2, @point3) && @p.on_line?(@point3, @point2)
      return @p.dist(@q) - [@p.dist(@point3), @p.dist(@point2)].min
    elsif @q.on_segment?(@point1, @point3) && @p.on_line?(@point3, @point1)
      return @p.dist(@q) - [@p.dist(@point3), @p.dist(@point1)].min
    end

      if @p.on_segment?(@point1, @point2) && @q.on_line?(@point1, @point2)
        return @p.dist(@q) - [@q.dist(@point1), @q.dist(@point2)].min
      elsif @p.on_segment?(@point2, @point3) && @q.on_line?(@point3, @point2)
        return @p.dist(@q) - [@q.dist(@point3), @q.dist(@point2)].min
      elsif @p.on_segment?(@point1, @point3) && @q.on_line?(@point3, @point1)
        return @p.dist(@q) - [@q.dist(@point3), @q.dist(@point1)].min
      end
```

```
      if @p.on_line?(@point1, @point2) && @q.on_line?(@point1, @point2)
     return @point1.dist(@point2)
      elsif @p.on_line?(@point3, @point2) && @q.on_line?(@point3, @point2)
    return @point3.dist(@point2)
      elsif @p.on_line?(@point1, @point3) && @q.on_line?(@point1, @point3)
    return @point1.dist(@point3)
  end

      array = []

  if is_on_segments?(@point1, @point2)
    array << crosspoint(@point1, @point2)
      elsif is_on_segments?(@point2, @point3)
    array << crosspoint(@point2, @point3)
      elsif is_on_segments?(@point1, @point3)
    array << crosspoint(@point1, @point3)
  end

      if array.size == 3
       if array[0] != array[1]
     return array[0].dist(array[1])
       elsif array[2] != array[1]
     return array[2].dist(array[1])
       elsif array[0] != array[2]
     return array[0].dist(array[2])
   end
      elsif array.size == 2
        return array[0].dist(array[1])
      else
        return 0.0
      end
    end
  end
...
end
```

Также нам необходимо модифицировать класс `Polygon`, добавив небольшие изменения в методы `initialize и add(t)`, а именно:

Вычисляем периметр части выпуклой оболочки лежащей в заданном треугольнике.

```
class Polygon < Figure
  attr_reader :points, :perimeter, :area, :part_perimeter
  ...
  def initialize(a, b, c, point1, point2, point3)
    ...
    ao=Segment.new(a, b, @point1, @point2, @point3).part_perimeter
    bo=Segment.new(b, c, @point1, @point2, @point3).part_perimeter
    co=Segment.new(a, c, @point1, @point2, @point3).part_perimeter
    @part_perimeter = ao + bo + co;  end
```

Когда добавляем новую точку нужно рассмотреть несколько случаев:

1. если освещённых ребер нет, что означает новая точка попала внутрь или на границу старой выпуклой оболочки и нам нечего не нужно делать;
2. если хотя бы одно освещённое ребро есть, то их нужно удалить и соединить концы оставшейся ломаной с новой точкой.

```
 def add(t)
   ...
  if t.light?(@points.last, @points.first)
    ...
  dl=Segment.new(@points.first,@points.last,@point1,@point2,@point3)
  @part_perimeter -= dl.part_perimeter
    ...
```

Для каждого освещённого ребра нам нужно уменьшить периметр оболочки на его длину.

```
  while t.light?(p, @points.first)
    ...
  det=Segment.new(@points.first, p, @point1, @point2, @point3).part_perimeter
  @part_perimeter -= det
    ...
  end
  ...

  while t.light?(@points.last, p)
   ...
   get=Segment.new(p, @points.last, @point1, @point2, @point3).part_perimeter
   @part_perimeter -= get
   ...
  end
```

Затем нам необходимо добавить сумму длин двух новых рёбер.

```
       ...
    ml=Segment.new(@points.first, t, @point1, @point2, @point3).part_perimeter
    nl=Segment.new(t, @points.last, @point1, @point2, @point3).part_perimeter
    @part_perimeter += ml + nl
    ...
   end
   self
  end
end
```

Но для того чтобы рисовался треугольник нам нужно модифицировать файл `convex/tk_drawer.rb` метод `draw` класса `Figure`:

```
class Figure
  def draw
   ...
    TkDrawer.draw_line(@point1, @point2)
    TkDrawer.draw_line(@point2, @point3)
    TkDrawer.draw_line(@point3, @point1)
  end
end
```

Для вывода результата на экран изменяем в файлах `convex/run_tkconvex.rb` и `convex/run_convex.rb` несколько строк.

```
  fig = Void.new(R2Point.new, R2Point.new, R2Point.new)
    ...
  puts "S = #{fig.area}, P = #{fig.perimeter} perimetr = #{fig.part_perimeter}"
```

Модификация эталонного проекта «Выпуклая оболочка» завершена. Пример работы программы можно увидеть на (рис.3).

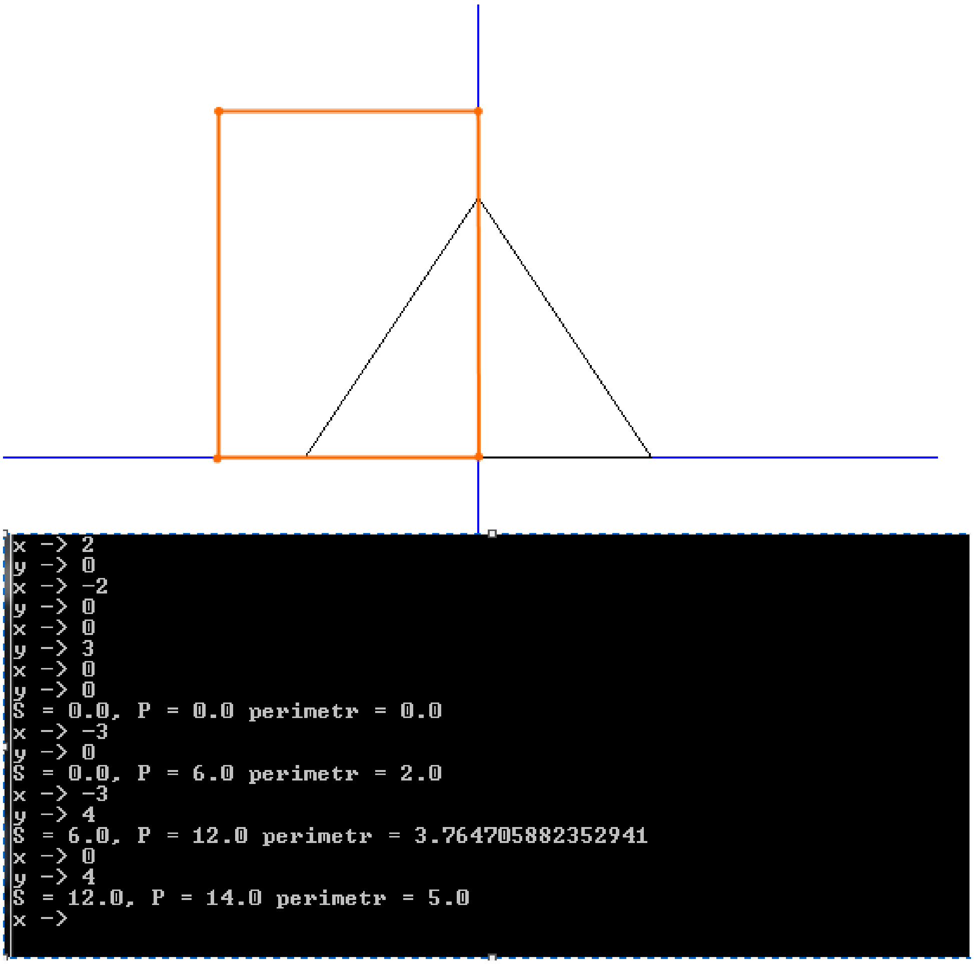


Рис. 3. Работа программы «Выпуклая оболочка»

3. Модификация проекта «Изображение проекции полиэдра»

Точная постановка задачи

Модифицируйте эталонный проект таким образом, чтобы определялась и печаталась следующая характеристика полиэдра: сумма длин проекций полностью невидимых рёбер, центр которых находится строго внутри сферы $x^2 + y^2 + z^2 = 4$.

Решение данной задачи и модификация кода

Для решения данной задачи модифицируем код файла `shadow/polyedr.rb`. Проверка ребра на невидимость, а также проверка центра ребра на попадание в сферу и вычисление периметра суммы длин проекций выполняется в методе `draw` класса `Polyedr`:

```
class Polyedr
  ...

  def draw
    ...
    p=0.0
      edges.each do |e|
        facets.each{|f| e.shadow(f)}
          e.gaps.each{|s| TkDrawer.draw_line(e.r3(s.beg), e.r3(s.fin))}
            if e.invisible?
              if e.func?(c)
                p+=e.perimeter(c)
              end
            end
      end
  return p
  end
end
```

Чтобы проверить, является ли ребро невидимым, нужно проверить является ли массив `@gaps` пустым. Для этого добавим в файл `shadow/polyedr.rb` метод `invisible?` в классе `Edge`:

```
class Edge
  ...
  def invisible?
    @gaps.size==0 ? true : false
  end
  ...
end
```

В файле `common/polyedr.rb` дополним метод `initialize(file)` в классе `Polyedr` заведем константу с коэффициентом гомотетии:

```
class Polyedr
  ...
  def initialize(file)
    ...
    @c=c
    ...
  end
end
```

Чтобы проверить находится ли центр полностью невидемых рёбер строго внутри сферы $x^2 + y^2 + z^2 = 4$ с учетом коэффициента гомотетии. Используем следующую формулу для проверки центра ребра:

$$x^2 + y^2 + z^2 < 4\,.$$

Надо добавить в файле `shadow/polyedr.rb` метод `func?(c)` в класс `Edge`:

```
class Edge
  ...
  def func?(c)
    point=r3(0.5)
    point.x**2+point.y**2+point.z**2 < 4*c**2
  end
  ...
end
```

После чего нам необходимо посчитать суммы длин проекций. Суммы длин проекций с учетом коэффициента гомотетии будем искать с помощью формулы:

$$\frac{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}{\mathrm{c}}\,.$$

Добавим в файле `shadow/polyedr.rb` метод `perimetr(c)` в класс `Edge`, позволяющий вычислить длину проеции.

```
class Polyedr
  ...
  def perimeter(c)
    vx=@fin.x-@beg.x
    vy=@fin.y-@beg.y
    (Math.sqrt(vx**2+vy**2))/c
  end
  ...
end
```

Заключительная модификация в файле `shadow/run_polyedr.rb` — вывод результата на экран:

```
#!/usr/bin/env ruby
require_relative './polyedr'
```

```
require_relative '../common/tk_drawer'
TkDrawer.create
%w(ccc box cube king).each do |name|
  puts '============================================================'
  puts "Начало работы с полиэдром '#{name}'"
  start_time = Time.now
  a=Polyedr.new("../data/#{name}.geom")
  puts "Сумма длин проекций #{a.draw}"
  puts "Изображение полиэдра '#{name}' заняло #{Time.now - start_time} сек."
  print 'Hit "Return" to continue -> '
  gets
end
```

Модификация эталонного проекта «Изображение проекции полиэдра» завершена. Пример работы программы с модифицированным файлом `cube1.geom` можно увидеть на (рис.4) и его содержание представленно ниже.

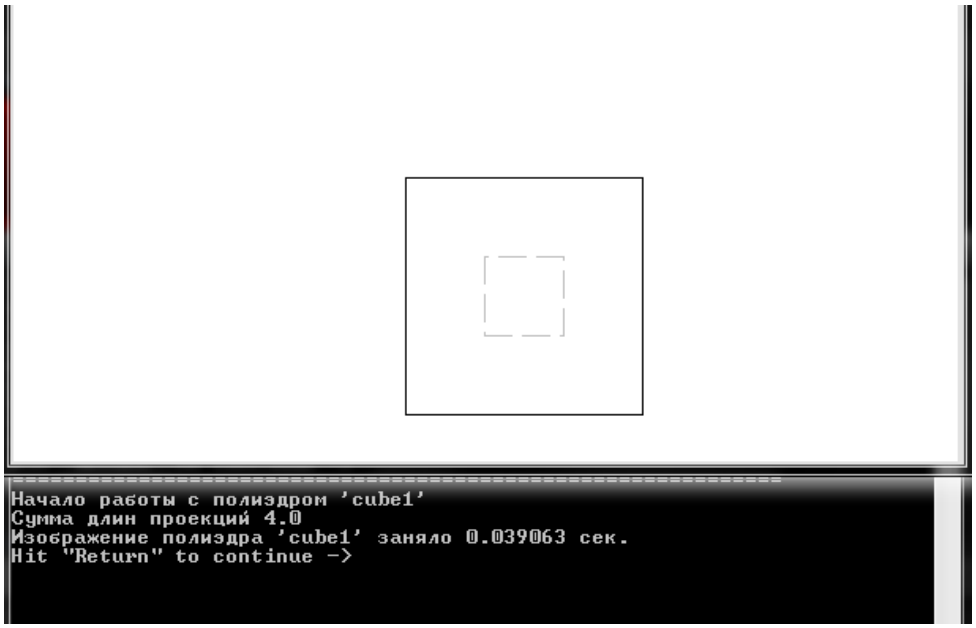


Рис. 4. Работа программы «Изображение проекции полиэдра»

```
50.0  0.0  0.0  0.0
 8  2  8
 0.0  0.0  0.0
 1.0  0.0  0.0
 1.0  1.0  0.0
 0.0  1.0  0.0
-1.0 -1.0  1.0
 2.0 -1.0  1.0
 2.0  2.0  1.0
-1.0  2.0  1.0
4  1  2  3  4
4  5  6  7  8
```

Список литературы и интернет-ресурсов

[1] `http://edu.msiu.ru//files/25029-lecture.html` — Описание проекта «Выпуклая оболочка».

[2] Е.А. Роганов *Основы информатики и программирования.* — М., МГИУ, 2002.

[3] `http://ru.wikipedia.org/wiki/Ruby` — Википедия (свободная энциклопедия) о языке Ruby.

[4] `http://edu.msiu.ru/files/26490-lecture.html`, `http://edu.msiu.ru/files/26929-lecture.html` — Описание проекта «Изображение проекции полиэдра».