## 6. МЕТОДЫ ПОСТРОЕНИЯ ВЫПУКЛОЙ ОБОЛОЧКИ

Данный раздел посвящен алгоритмам построения звездчатых и выпуклых геометрических фигур. Показано, как эти алгоритмы применяются для разработки конструкторов классов многоугольников и многогранников соответствующих типов.

Сначала изучаются тесты на принадлежность точки многоугольнику. Затем рассматриваются методы построения звездчатых многоугольников, выпуклых многоугольников и многогранников. Описываются различные алгоритмы построения выпуклых оболочек. Предлагаются программные реализации этих методов.

## 6.1. ТЕСТЫ ПРИНАДЛЕЖНОСТИ ТОЧКИ МНОГОУГОЛЬНИКУ

Будем рассматривать многоугольники, заданные как лежащие в декартовой плоскости замкнутые ломаные, состоящие из отрезков *p0p1*, *p1p2*,…,*pn-1p0*. Каждый многоугольник определен последовательностью точек (*p0*, *p1*,…,*pn-1*), которые являются его вершинами. Сформулируем ***основную задачу*** данного параграфа.

*Даны* многоугольник *P*, вершинами которого служат точки (*p0*, *p1*,…,*pn-1*), и точка *t*. *Определить*, находится ли точка *t* внутри многоугольника *P*.

Рассмотрим эту задачу сначала для *простого многоугольника* — многоугольника, у которого никакая пара непоследовательных ребер не имеет общих точек. Например, любой выпуклый многоугольник является простым. Отметим, что многоугольник, приведенный на рис. 6.1, является простым.

Согласно теореме Жордана, простой многоугольник разбивает плоскость на две части — внутреннюю и внешнюю. Если точка *t* имеет координаты (*xt*, *yt*), то проведем от нее горизонтальный луч, состоящий из точек (*x*, *y*), таких, что *x* > *xt* и *y* = *yt*. Согласно стандарту GKS, точка *t* называется *внутренней*, если число точек пересечения луча со сторонами многоугольника нечетно. В противном случае она называется *внешней*.

***Тест принадлежности методом луча.*** Рассмотрим алгоритм, основанный на определении внутренней точки, данном в стандарте GKS. Пусть луч пересекается с многоугольником в точках, являющихся вершинами многоугольника (рис. 6.1).







Рис. 6.1. Пример расположения точки и многоугольника

В этом примере луч пересекается с границей многоугольника в четном числе точек, хотя точка является внутренней. Если же подсчитывать число пересечений луча со сторонами многоугольника, то точка *q* и точка *r* будут учитываться дважды, и вновь получаем четное число пересечений со сторонами многоугольника.

Анализ этого примера приводит к выводу, что точку *q* следует учитывать два раза или не учитывать совсем, а точку *r* — один раз. Следовательно, при определении, пересекаются ли горизонтальный луч и отрезок, мы не должны учитывать точки отрезка, имеющие *максимальную ординату*.

В примере (рис. 6.1) получаем, что число точек пересечения со сторонами равно 1 + 1 + 1 + 0.

Алгоритм проверки на принадлежность точки *t* многоугольнику *p0*, *p1*,…,*pn-1* будет состоять из следующих действий:

1. Установить четность *parity* = 0.
2. Организовать цикл по *i* = 0, 1, 2,…,*n*-1, в теле которого, в случае, когда пересечение луча, проведенного вправо от точки *t*, с отрезком *pip(i+1)%n* не пусто, применяется операция изменения четности *parity* = 1 – *parity*.
3. После окончания работы цикла, если *parity* = 1, то точка расположена внутри, а в случае *parity* = 0 — снаружи многоугольника.

При программной реализации проверка на пересечение луча и отрезка будет осуществляться в подпрограмме (функции), возвращающей 1, если пересечение не пусто, и 0 — в противном случае. Если общей точкой луча и отрезка является точка отрезка с максимальной ординатой, то возвращается 0. В частности, пересечение луча с горизонтальным отрезком всегда считается пустым. Луч, исходящий из точки (*x*, *y*), будет пересекаться с отрезком, соединяющим точки (*x1*, *y1*) и (*x2*, *y2*), если min (*y1*, *y2*) ≤ *y* < max (*y1*, *y2*) и, более того, точка пересечения горизонтальной прямой с отрезком находится справа от точки (*x*, *y*).

Для реализации алгоритма определим класс точки как структуру, состоящую из координат. Разработаем с помощью этой структуры подпрограмму проверки на пересечение горизонтального луча с отрезком и определим многоугольник как класс, состоящий из массива вершин многоугольника и теста на принадлежность точки многоугольнику.

struct Point

{

float x, y; // координаты точки

};

int intersect(Point p, Point p1, Point p2)

{

// проверка пересечения луча с отрезком

if(p1.y == p2.y) return 0;

if((p1.y < p2.y ? p2.y:p1.y) <= p.y) return 0; // max(y1,y2)<=y

if((p1.y < p2.y ? p1.y:p2.y) > p.y) return 0; // min(y1,y2)>y

if(p2.y - p1.y>0) // если y2>y1, то векторное произведение > 0

return ((p.x - p1.x) \* (p2.y - p1.y) –

(p2.x - p1.x) \* (p.y - p1.y) > 0);

else

return ((p.x - p1.x) \* (p2.y - p1.y) –

(p2.x - p1.x) \* (p.y - p1.y) < 0);

}

class Polygon

{

int n; // количество вершин

Point \*p; // массив вершин

public:

int isin(Point t); // тест на принадлежность

};

// реализация алгоритма

int Polygon::isin(Point t)

{

int i, parity = 0;

for(i = 0; i < n; i++)

if(intersect(t,

p[i], p[(I + 1)%n]))

parity = 1 - parity;

return parity;

}

Примеры использования теста на принадлежность будут даны позже.

Если включить в цикл действия, которые выполняет функция intersect(), в подпрограмму isin(), то получим следующий текст подпрограммы проверки на принадлежность точки многоугольнику:

int Polygon::isin(Pnt t)

{

int i,j, parity = 0;

 for(i = 0, j = n-1; i < n; j = i++)

 {

 if (( ( (p[i].y<=t.y)&&(t.y<p[j].y) )||

 ((p[j].y<=t.y)&&(t.y<p[i].y)))&&

 (t.x<(p[j].x-p[i].x)\*(t.y-p[i].y)/

 (p[j].y-p[i].y)+p[i].x))

 parity=1-parity;

 }

 return parity;

}

В случае, когда многоугольник не является простым, этот тест не всегда дает желаемый результат.

Рис. 6.2. Точки многоугольника *a0a1a2a3a4* являются внешними



















Например, точки внутреннего выпуклого многоугольника *a0a1a2a3a4*(рис. 6.2) всегда будут внешними по отношению к многоугольнику *p0p1p2p3p4*, хотя в некоторых случаях их удобно считать внутренними. В этих случаях можно применить алгоритм, описанный ниже.

***Тест на принадлежность методом углов.*** Задан многоугольник, состоящий из вершин *p0*, *p1*,…,*pn-1*, и точка *t*. В этом алгоритме проверки на принадлежность точки *t* многоугольнику, точка *t* берется за начало новой системы координат. В результате вся плоскость будет разбита на 4 области–квадраты относительно начала координат *t*.

Пусть *ci* = *code* (*pi*)  {0, 1, 2, 3} обозначает четверть, содержащую точку *pi*. Введем число, характеризующее изменение номера четверти при переходе от точки *pi* к точке *pi+1*. Если *i*+ 1 = *n*, то вместо *pi+1* берется точка *p0*. Положим:

 0, если *ci* = *ci+1*,

 1, если *ci+1*= (*ci* + 1) mod 4

mi =  – 1, если *ci+1*= (*ci* – 1) mod 4,

 ± 2, если *ci+1* = (*ci* + 2) mod 4.

В последнем случае если *t* находится слева от направленного отрезка *pipi+1*, то *mi* = + 2, а если справа, то *mi* = – 2. Индексом *indiP* точки *t* относительно многоугольника *P* называется число .

***Теорема.*** Индекс *indiP* равен 0 тогда и только тогда, когда точка *t* расположена вне многоугольника *P*.

***Пример.*** Рассмотрим точку *t* и многоугольник, изображенные на рис. 6.3. Точки *p* первой четверти имеют код *code* (*p*) = 0, второй — *code* (*p*) = 1, третьей — *code* (*p*) = 2, четвертой — *code* (*p*) = 3.

Рис. 6.3. Вычисление индекса точки *t*

























t

Получаем *m0* = 2, *m1* = 0, *m2* = 1, *m3* = 1, *m4* = – 2, *m5* = – 2, следовательно, индекс равен:

*indtP* = (2 + 0 + 1 + 1 – 2 –2) / 4 = 0,

и точка *t* является внешней.

*Алгоритм определения принадлежности точки методом углов* состоит в переборе всех сторон многоугольника и вычислении индекса точки.

Предполагая, что структура точки содержит все необходимые операции, разработаем функцию, реализующую тест на принадлежность методом углов.

int Polygon::isin(Point t)

{

int i, ind = 0;

// вначале индекс = 0

Point q = p[n - 1];

// начальная сторона p(n-1)p(0)

for(i = 0; i < n; i++)

{

if(t.code(q) == t.code(p[i])); // приращение индекса = 0

else if((t.code(p[i]) - t.code(q) + 3)%4 == 0)

ind++; // приращение индекса = 1

else if((t.code(p[i]) - t.code(q) + 1)%4 == 0)

ind--; // приращение индекса = -1

else if((p[i] - q) \* (t - q) > 0) ind += 2;

// приращение индекса = 2

else ind -= 2; // приращение индекса = -2

q = p[i];

}

ind = ind / 4;

if(ind == 0) return 0;

else return 1;

}

Чтобы эта функция работала, необходимо дополнить класс точки функцией вычисления кода и операциями разности и векторного произведения для плоских векторов. Под векторным произведением плоских векторов *v* \* *w* подразумевается число, равное |*v*| · |*w*| · sin*α*, где *α* – угол между векторами *v* и *w*. Учитывая эти дополнения, получим структуру точки:

struct Point

{

float x, y; // координаты точки

int code(Point q); // код четверти точки q в предположении,

 // что (x,y) – начало координат

float operator\*(Point q); // векторное произведение

Point operator-(Point q); // разность векторов

};

int Point::code(Point q)

{

if(q.x – x >= 0 && q.y – y >= 0) return 0;

if(q.x – x < 0 && q.y – y >= 0) return 1;

if(q.x – x >= 0 && q.y – y < 0) return 3;

if(q.x – x < 0 && q.y – y < 0) return 2;

}

Point Point::operator-(Point q) // разность

{

Point t;

t.x = x - q.x; t.y = y - q.y;

return t;

}

float Point::operator\*(Point q)

{

return x \* q.y – y \* q.x; // векторное произведение

}

## 6.2. ПОСТРОЕНИЕ ЗВЕЗДЧАТОГО МНОГОУГОЛЬНИКА

Дадим определение звездчатого многоугольника. Пусть точки *p* и *q* принадлежат некоторому многоугольнику *P*, т.е. расположены внутри многоугольника или на его границе. Будем говорить, что точка *q* *видна* из точки *p*, если прямолинейный отрезок *pq* состоит из точек, принадлежащих многоугольнику *P*.

*s*

*q*

*p*

Рис. 6.4. Видимость точек

*Например*, на рис. 6.4 точка *q* видна из точки *s*, а точка *p* не видна из точки *s*. Множество точек многоугольника, из которых видны все точки этого многоугольника, называется *ядром многоугольника*. Отметим, что ядро выпуклого многоугольника составляют все его точки. Многоугольник называется *звездчатым*, если его ядро не пустое. Заметим, что ядро звездчатого многоугольника всегда является выпуклым множеством.

Опишем *алгоритм построения звездчатого многоугольника*. Предположим, что мы строим звездчатый многоугольник, который должен содержать точки *p0*, *p1*,…, *pm–1*. Центром тяжести точек *p0*, *p1*,…, *pm–1* называется точка *pC = (xC, yC)*, координаты которой равны , , где (*xi*, *yi*) — координаты точки *pi*, а *m —* количество точек. Построим многоугольник, центр тяжести которого *p* = *pC* принадлежит ядру.

С этой целью отсортируем точки *p0*, *p1*,…, *pm–1* по возрастанию угла наклона отрезков *ppi* вокруг точки *p*. Соединяя затем *pi* с *pi+1* , где *pm* = *p0*, для всех *i* = 0, 1,…*m*-1***,*** получим искомый звездчатый многоугольник.

Приведем определение класса, объектами которого будут звездчатые многоугольники. Конструктор будет строить звездчатые многоугольники по описанному алгоритму.

Предположим, что структура точки состоит из двух полей, содержащих координаты точки, и всех необходимых для реализации алгоритма операций.

Определим класс звездчатого многоугольника.

class SPolygon

{

int color; // цвет области

int n; // число вершин

Point \*p; // массив вершин

Point pC; // центр тяжести

public:

friend class Wnd; // класс для вывода точек области на экран

SPolygon(float \*x, float \*y, int m, int cl);

};

SPolygon::SPolygon(float \*x, float \*y, int m, int cl)

:color(cl) // цвет

{

int i, j;

Point t;

n = m; pC.x = 0; pC.y = 0;

for(i = 0; i < m; i++)

{

pC.x += x[i]; pC.y += y[i];

p[i].x = x[i]; p[i].y = y[i];

}

// сортируем точки по возрастанию угла

// вокруг центра тяжести методом вставок

for(i = 1; i < m; i++)

{

t = p[i];

for(j = i - 1; (j >= 0) && ((t - pC) < (p[j] - pC)); j--)

p[j + 1] = p[j];

p[j + 1]=t;

}

}

Этот класс будет главной частью программы, в которой создается и выводится на экран звездчатый многоугольник. Приведем программную реализацию. Для создания приложения достаточно перенести на главную форму **Form1** компонент **Image** со страницы **Additional**, а затем набрать следующий текст **Unit1.cpp** и запустить компиляцию, сборку и выполнение, выбрав пункт меню **Run.** Текст программы **Unit1.cpp**:

Листинг 6.1. Построение звездчатого полигона

//---- построение звездчатого полигона ----------------------------

#include <vcl.h>

#pragma hdrstop

#include "Unit1.h"

//---------------------------------------------------------------------------

#pragma package(smart\_init)

#pragma resource "\*.dfm"

TForm1 \*Form1;

#include <math.h>

#define PI 3.14159

class Wnd;

struct Pnt // структура точки

{

 float x, y; // координаты точки

 int code(Pnt q); // код четверти точки q

 float operator\*(Pnt q); // ориентированная площадь

 Pnt operator-(Pnt q); // разность векторов

};

int Pnt::code(Pnt q) // код четверти

{ // начало координат - в точке (x, y)

 if(q.x - x >= 0 && q.y - y >= 0) return 0; // первая четверть

 if(q.x - x < 0 && q.y - y >= 0) return 1; // вторая четверть

 if(q.x - x < 0 && q.y - y < 0) return 2; // третья четверть

 return 3; // во всех остальных случаях

}

float Pnt::operator\*(Pnt q)

{

 return x \* q.y - y \* q.x; // ориентированная площадь параллелограмма

}

Pnt Pnt::operator-(Pnt q)

{

 Pnt t; t.x = x - q.x; t.y = y - q.y; return t;// разность векторов

}

int operator<(Pnt p, Pnt q)

{

Pnt t; // сравнение углов радиус-векторов

 t.x = 0; t.y = 0; // точки p и q относительно (0, 0)

 if(t.code(p) < t.code(q)) return 1;

 if(t.code(p) > t.code(q)) return 0;

 if(p \* q == 0) return p.x \* p.x + p.y \* p.y < q.x \* q.x + q.y \* q.y;

 else return p \* q > 0;

}

class SPolygon // звездчатый многоугольник

{

 TColor color; // цвет точек многоугольника

 int n; // количество вершин

 Pnt \*p; // массив вершин

 Pnt pC; // центр тяжести

 public:

 friend class Wnd;

 SPolygon(float \*x, float \*y, int m, TColor cl);

 SPolygon(const SPolygon &ob);

 int isin(Pnt t);

 ~SPolygon()

 {

 delete []p;

 }

};

// Конструктор копирования

SPolygon::SPolygon(const SPolygon &ob) // необходим для передачи объекта

{ // в качестве параметра

 int i;

 n = ob.n;

 p = new Pnt[n]; // выделим память

 for(i=0; i < ob.n; i++) p[i]=ob.p[i]; // производим копирование

}

SPolygon::SPolygon(float \*x, float \*y, int m, TColor cl)

:color(cl) // цвет точек

{

int i, j;

Pnt t;

 p=new Pnt[m];

 n = m; pC.x = 0.; pC.y = 0.;

 for(i = 0; i < m; i++)

 {

 pC.x += x[i]; pC.y += y[i]; // вычисление координат

 p[i].x = x[i]; p[i].y = y[i]; // центра тяжести

 }

 pC.x=pC.x/m; pC.y=pC.y/m;

 for(i = 1; i < m; i++)

 {

 t = p[i]; // сортировка методом вставок

 for(j = i - 1; (j >= 0) && ((t - pC) < (p[j] - pC)); j--)

 p[j + 1] = p[j]; // по величине угла

 p[j + 1] = t;

 }

}

// тест на принадлежность методом углов

int SPolygon::isin(Pnt t)

{

// 0 - если точка t не принадлежит многоугольнику,

// не 0 - в других случаях

int i, ind = 0; // индекс точки

Pnt q = p[n - 1];

 for(i = 0; i < n; i++)

 {

 if(t.code(q) == t.code(p[i])) ; // ничего не делать

 else if((t.code(p[i]) - t.code(q) + 3)%4 == 0) ind++;

 else if((t.code(p[i]) - t.code(q) + 1)%4 == 0) ind--;

 else if((p[i] - q) \* (t - q) > 0) ind += 2;

 else ind -= 2;

 q = p[i];

 }

 if(ind==0) return 0; else return 1;

}

class Wnd // класс окна

{

 float xmin, ymin, xmax, ymax; // окно

 public:

 Wnd(float x0, float y0, float x1, float y1):

 xmin(x0), ymin(y0), xmax(x1), ymax(y1){}

 Wnd& operator<<(SPolygon);

 Wnd& operator<<(Pnt);

};

Wnd& Wnd::operator<<(SPolygon f)

// операция вывода многоугольника в окно

{

int i;

TPoint \*parray = new TPoint [f.n];

 Form1->Image1->Canvas->Brush->Color=f.color; // цвет точек многоугольника

 for(i = 0; i < f.n; i++)

 {

 parray[i] = Point(

 ((f.p[i]).x - xmin)\*(Form1->Image1->Width) / (xmax - xmin),

 (ymax - (f.p[i]).y) \*(Form1->Image1->Height) / (ymax - ymin));

 }

 Form1->Image1->Canvas->Polygon(parray, f.n-1);

 // вывод с помощью стандартной функции

 return \*this;

}

Wnd& Wnd::operator<<(Pnt t)

{

 Form1->Image1->Canvas->Pixels

 [(t.x - xmin) \* (Form1->Image1->Width) / (xmax - xmin)]

 [(ymax - t.y) \* (Form1->Image1->Height) / (ymax - ymin)]=clBlack;

 return \*this;

}

//---------------------------------------------------------------------------

\_\_fastcall TForm1::TForm1(TComponent\* Owner)

 : TForm(Owner)

{

// главная программа

Wnd w(-300, -300, 300, 300);

float rad = 300, xx[10], yy[10]; // радиус и координаты

int i; Pnt t; // счетчик и точка

 for(i = 0; i < 5; i++)

 {

 xx[i] = rad \* cos(2 \* PI \* i / 5); // берутся точки на

 yy[i] = rad \* sin(2 \* PI \* i / 5); // окружности радиуса rad

 xx[i + 5] = rad \* cos(2 \* PI \* i / 5 + PI / 5) / 2;// и на окружности

 yy[i + 5] = rad \* sin(2 \* PI \* i / 5 + PI / 5) / 2;// радиуса rad/2

 }

 SPolygon sp(xx,yy,10,(TColor)RGB(0,255,100)); // объект - многоугольник

 w<<sp; // вывод многоугольника на экран

 float sx, sy; // проверка теста на принадлежность

 for(sx = -300; sx <= 300; sx += 5) // пробегаются все

 {

 for(sy = -300; sy <= 300; sy += 5)// точки окна

 {

 t.x = sx; t.y = sy; // если точка не принадлежит

 if(sp.isin(t) == 0) w<<t; // окну, то выводится на экран

 }

 }

}

//---------------------------------------------------------------------------

Результаты работы программы показаны на рис. 6.5. Поскольку при выполнении операции w<<sp вывода многоугольника sp в окно w объект sp передается по значению, вызывается конструктор копирования объекта sp.Если не определить конструктор копирования явно, то будет выполняться побитное копирование указателя на массив точек. При выходе из подпрограммы, выполняющей операцию w<<sp, будет освобождаться область, адресом которой является этот указатель, и массив точек может измениться. Для того чтобы исключить эту возможность, мы определили явно конструктор копирования Spolygon(const SPolygon &ob).



Рис. 6.5. Звездчатый многоугольник

Отметим, что текст программы начинается с определения класса окна Wnd, в котором для вывода многоугольника переопределяется операция <<. Затем определяется структура точки и операции, необходимые для построения звездчатого многоугольника. Также определяется класс, объектами которого являются звездчатые многоугольники, и составная функция этого класса, реализующая тест на принадлежность методом углов.

## 6.3. ПОСТРОЕНИЕ И ИЗОБРАЖЕНИЕ ВЫПУКЛОЙ ОБОЛОЧКИ

***Выпуклой оболочкой множества точек*** называется наименьшее выпуклое множество, содержащее эти точки. Выпуклая оболочка множества точек {*p0*, *p1*,…, *pn–1*} будет состоять из всех точек *p*, для которых существуют неотрицательные действительные числа *ti* ≥ 0, 1 ≤ *i* ≤ *n*, такие, что  и *p* = *t0p0* + *t1p1* +…+*tn-1pn-1* (здесь точки *pi* рассматриваются как радиус-векторы). Например, выпуклой оболочкой двух точек будет соединяющий их прямолинейный отрезок. Выпуклой оболочкой не лежащих на одной прямой трех точек будет треугольник, вершинами которого являются эти точки.

***Построение выпуклой оболочки на плоскости.*** Алгоритм Дейкстры строит выпуклый многоугольник по шагам. Сначала берется тройка точек. Она составляет начальный выпуклый многоугольник. Затем последовательно присоединяются остальные точки. При каждом присоединении многоугольник достраивается до выпуклого многоугольника, содержащего очередную точку, описанным далее образом.

Напомним, что точка *q* многоугольника называется *видимой* *из точки* *p*, не принадлежащей многоугольнику, если отрезок *pq* не содержит точек многоугольника, кроме точки *q*. Сторона многоугольника называется *видимой* *из точки* *p*, если она состоит из видимых точек. Пусть *P —* наименьший выпуклый многоугольник, содержащий первые *k* точек входного массива, *p —* (*k* + 1)-я точка. Если *p* принадлежит многоугольнику *P*, то многоугольник *P* не изменится. В противном случае (рис. 6.6) удаляются видимые стороны многоугольника *P* и добавляются две стороны, которые являются прямолинейными отрезками, соединяющими точку *p* с двумя крайними точками оставшихся сторон.











Рис. 6.6. Добавление точки *p* к многоугольнику *P*

Последовательно добавляя точки входного массива, получим выпуклую оболочку этих точек.

Для программной реализации этого метода построения выпуклой оболочки определим класс выпуклого многоугольника как производный от класса звездчатого многоугольника. Разработаем составную функцию добавления точки. Конструктор будет строить выпуклый многоугольник, многократно обращаясь к функции добавления (предполагается, что закрытые члены класса SPolygon заменены на защищенные).

class CPolygon:public SPolygon

{

public:

void Insert(Point t) // добавление точки

{

int i; int j;

if(isin(t)) return; // t принадлежит

int \*del = new int [n]; // n – число вершин

Point \*q = new Point [n + 1]; // формируемый

// многоугольник

j0 = j1 = 0;

for(i = 0; i < n; i++)

{

if((t - p[i]) \* (p[(i + 1)%n] - p[i]) >= 0)

del[i] = 1; // отмечаем видимые стороны

else del[i] = 0;

}

for(i = 0; i < n; i++)

if(del[i] == 1 && del[(I + 1)%n] == 0) break;

j = 0; i = (i + 1)%n; // i – номер последней

// невидимой стороны

while(del[i] == 0)

{

q[j++] = p[i]; i = (i + 1)%n; // перепись вершин

}

q[j] = p[i];

q[j + 1] = t; // добавление вершин

delete []p;

delete []del;

p = new Point [j + 2]; n = j + 2;

for(i = 0; i < n; i++)

p[i] = q[i];

delete []q;

}

CPolygon(float \*x, float \*y, int m, int cl);

};

// Конструктор

CPolygon::CPolygon(float \*x, float \*y, int m, int cl):color(cl)

{

// выпуклая точка массива точек (x[i], y[i])

int i; Point t;

p = new Point [3]; n = 3; // начальный треугольник

for(i = 0; i < 3; i++)

{

p[i].x = x[i]; p[i].y = y[i];

}

if((p[1] - p[0]) \* (p[2] - p[1]) <0 )

{

 // ориентация против часовой стрелки

t = p[1]; p[1] = p[2]; p[2] = t;

}

for(i = 3; i < m; i++)

{

t.x = x[i]; t.y = y[i];

Insert(t); // добавление точки

}

}

Предполагается, что первые три точки не лежат на одной прямой.

Приведем полный текст программы, в которой строится выпуклая оболочка методом Дейкстры. Тест на принадлежность точки многоугольнику реализуется с помощью метода горизонтального луча.

Листинг 6.2. Построение выпуклой оболочки методом Дейкстры

//---------------------------------------------------------------------------

#include <vcl.h>

#pragma hdrstop

#include "Unit1.h"

//---------------------------------------------------------------------------

#pragma package(smart\_init)

#pragma resource "\*.dfm"

#include <math.h>

#include <stdlib.h>

#define PI 3.14159

TForm1 \*Form1;

int getmaxx()

{

 return Form1->Image1->Width;

}

int getmaxy()

{

 return Form1->Image1->Height;

}

void line(int x0, int y0, int x1, int y1)

{

 Form1->Image1->Canvas->MoveTo(x0,y0);

 Form1->Image1->Canvas->LineTo(x1,y1);

}

struct Pnt // класс точки

{

 double x, y; // координаты точки

 int code(Pnt q); // код четверти, в которой лежит точка q

 double operator\*(Pnt q); // векторное произведение

 Pnt operator-(Pnt q); // разность векторов

};

int Pnt::code(Pnt q) // код четверти, в которой лежит точка q

{

// начало координат находится в точке (x, y)

 if(q.x - x >= 0 && q.y - y >= 0) return 0;

 if(q.x - x < 0 && q.y - y >= 0) return 1;

 if(q.x - x >= 0 && q.y - y < 0) return 3;

 if(q.x - x < 0 && q.y - y < 0) return 2;

}

double Pnt::operator\*(Pnt q)

{

 return x \* q.y - y \* q.x; // векторное произведение

}

Pnt Pnt::operator-(Pnt q) // разность векторов

{

 Pnt t; t.x = x - q.x; t.y = y - q.y; return t;

}

int operator<(Pnt p, Pnt q)

{

 Pnt t; // сравнение углов радиус-векторов p и q

 t.x = 0; t.y = 0; // коды четвертей вычисляются

// относительно (0,0)

 if(t.code(p) < t.code(q)) return 1;

 if(t.code(p) > t.code(q)) return 0;

 return (p \* q > 0); // вращение от p к q направлено

// против часовой стрелки

}

int intersect(Pnt p, Pnt p1, Pnt p2)

{

// тест на пересечение луча и отрезка

 if(p1.y == p2.y) return 0;

 if((p1.y < p2.y ? p2.y : p1.y) <= p.y) return 0;

 if((p1.y < p2.y ? p1.y : p2.y) > p.y) return 0;

 if(p2.y - p1.y > 0) return

 ((p.x - p1.x)\*(p2.y - p1.y)-(p2.x - p1.x)\*(p.y - p1.y) > 0);

 else return

 ((p.x - p1.x)\*(p2.y - p1.y)-(p2.x - p1.x)\*(p.y - p1.y) < 0);

}

class SPolygon

{

 protected:

 int color, n;

 Pnt \*p, pC;

 public:

 SPolygon(TColor cl):color(cl){} // конструктор по умолчанию

 SPolygon(double \*x, double \*y, int m, TColor cl);

 SPolygon(const SPolygon &ob); // конструктор копирования

 int isin(Pnt t);

 void show(double xmin, double ymin,

 double xmax, double ymax);

};

// Конструктор копирования

SPolygon::SPolygon(const SPolygon &ob) // необходим для передачи объекта

{ // в качестве параметра

 int i;

 n = ob.n;

 p = new Pnt[n]; // выделим память

 for(i=0; i < ob.n; i++) p[i]=ob.p[i]; // производим копирование

}

SPolygon::SPolygon(double \*x, double \*y, int m, TColor cl):color(cl)

{

int i, j;

Pnt t;

 p = new Pnt [m]; n = m;

 pC.x = 0; pC.y = 0;

 for(i = 0; i < m; i++)

 {

 pC.x += x[i]; pC.y += y[i];

 }

 pC.x = pC.x / m; pC.y = pC.y / m;

 for(i = 0; i < m; i++)

 {

 p[i].x = x[i]; p[i].y = y[i];

 }

// Сортируем точки по возрастанию угла вокруг центра

// тяжести методом вставок

 for(i = 1; i < m; i++)

 {

 t = p[i];

 for(j = i - 1; (j >= 0) && ((t - pC) < (p[j] - pC)); j--)

 p[j + 1] = p[j];

 p[j + 1] = t;

 }

}

int SPolygon::isin(Pnt t)

{

int i, parity = 0;

 for(i = 0; i < n; i++)

 if(intersect(t, p[i], p[(i + 1)%n]))

 parity = 1 - parity;

 return parity;

}

void SPolygon::show(double xmin, double ymin,

 double xmax, double ymax)

{

int i;

TPoint \*parray = new TPoint [n];

 Form1->Image1->Canvas->Brush->Color=color;

 for(i = 0; i < n; i++)

 {

 parray[i] = Point(

 (p[i].x - xmin) \* getmaxx() / (xmax - xmin),

 (ymax - p[i].y) \* getmaxy() / (ymax - ymin));

 }

 Form1->Image1->Canvas->Polygon(parray,n-1);

int x = (pC.x - xmin) \* getmaxx() / (xmax - xmin);

int y = (ymax - pC.y) \* getmaxy() / (ymax - ymin);

 for(i = 0; i < n; i++)

 {

 line(parray[i].x, parray[i].y, x,y);

 }

}

class CPolygon:public SPolygon

{

 public:

 int isin(Pnt r) {return SPolygon::isin(r);};

 void Insert(Pnt t)

 {

 int i; int j0, j1, j;

 int \*del = new int [n];

 Pnt \*q = new Pnt [n + 1];

 if(isin(t)) return;

 j0 = j1 = 0;

 for(i = 0; i < n; i++)

 {

 if((t - p[i]) \* (p[(i + 1)%n] - p[i]) >= 0)

 del[i] = 1;

 else del[i] = 0;

 }

 for(i = 0; i < n; i++)

 if(del[i] == 1 && del[(i + 1)%n] == 0) break;

 j = 0; i = (i + 1)%n;

 while(del[i] == 0)

 {

 q[j++] = p[i];

 i = (i + 1)%n;

 }

 q[j] = p[i];

 q[j + 1] = t;

 delete []p;

 p = new Pnt [j + 2]; n = j + 2;

 for(i = 0; i < n; i++)

 p[i] = q[i];

 delete []q;

}

CPolygon(double \*x, double \*y, int m, TColor cl);

};

CPolygon::CPolygon(double \*x, double \*y, int m, TColor cl):SPolygon(cl)

{

// выпуклая оболочка методом Дейкстры

int i; Pnt t; p = new Pnt [3]; n = 3;

 for(i = 0; i < 3; i++)

 {

 p[i].x = x[i]; p[i].y = y[i];

 }

 if((p[1] - p[0]) \* (p[2] - p[1]) < 0)

 {

 t = p[1]; p[1] = p[2]; p[2] = t;

 }

 for(i = 3; i < m; i++)

 {

 t.x = x[i]; t.y = y[i];

 Insert(t);

 }

}

//---------------------------------------------------------------------------

\_\_fastcall TForm1::TForm1(TComponent\* Owner)

 : TForm(Owner)

{

double rad = 150; //радиус окружности и вершины семиугольника

int i, mm = 7;

double \*x, \*y;

 x = new double [2 \* mm]; y = new double [2 \* mm];

 for(i = 0; i < mm; i++)

 {

 // вершины большого семиугольника

 x[i] = rad \* cos(2 \* PI \* i / mm);

 y[i] = rad \* sin(2 \* PI \* i / mm);

 // вершины маленького семиугольника

 x[i + mm] = rad \* cos(2 \* PI \* i / mm + PI / mm) / 2;

 y[i + mm] = rad \* sin(2 \* PI \* i / mm + PI / mm) / 2;

 }

 randomize();

 for(i = 0; i < 2 \* mm; i++)

 x[i] += random(100) - 50;

CPolygon dom2(x, y, 2 \* mm, (TColor)RGB(0,200,200)); // построение выпуклой

// оболочки

SPolygon dom3(x, y, 2 \* mm, (TColor)RGB(255,0,255));// построение звездчатого

// полигона

dom2.show(-300, -300, 300, 300); // вывод выпуклой оболочки

dom3.show(-300, -300, 300, 300); // вывод звездчатого полигона

}

//---------------------------------------------------------------------------

В результате работы программы на экран будут выведены выпуклая оболочка 14 точек и звездчатый многоугольник, содержащий эти точки. На рис. 6.7 показаны примеры результатов вывода.



Рис. 6.7. Звездчатые многоугольники и их выпуклые оболочки

***Выпуклая оболочка в трехмерном пространстве.*** Отметим, что рассмотренный выше алгоритм был предложен Дейкстрой для построения выпуклой оболочки в трехмерном пространстве. Он основан на приведенной ниже идее добавления точек к выпуклому многограннику.

Пусть многогранник *P* задан списком составляющих его граней. Построим список граней многогранника *P'*, который будет выпуклой оболочкой вершин многогранника *P* и новой точки *q*, не принадлежащей *P*. Грани задаются как массив вершин соответствующих многоугольников. Грань называется *видимой из точки* *q*, если угол между ее вектором внешней нормали и вектором, соединяющим любую точку этой грани (рис. 6.8) с точкой *q*, меньше π / 2. В противном случае будем называть грань *невидимой*. Каждое ребро принадлежит некоторым двум граням. Мы будем допускать случаи, при которых две грани находятся в одной плоскости. Ребро, принадлежащее видимой грани и невидимой грани, называется *контурным*. Если соединить точку *q* с вершинами контурного ребра, то получится треугольник, который будет гранью многогранника *P'*. Следовательно, чтобы получить список граней многогранника *P'*, нужно сначала удалить все видимые грани. Затем добавить к оставшемуся списку граней все треугольники, полученные соединением точки *q* с вершинами контурных ребер.

Рис. 6.8. Видимая из точки *q* грань





***Пример 6.1.*** Построим выпуклую оболочку пяти точек, первые четыре из которых не лежат в одной плоскости. Обозначим эти пять точек через *p0*, *p1*, *p2*, *p3*, *q*. Перенесем начало координат в точку *p0*. Поскольку точки *p0*, *p1*, *p2* и *p3* не лежат в одной плоскости, то вектора , ,  будут составлять базис пространства. Следовательно, существуют числа *x*, *y*, *z*, такие, что .















Рис. 6.9. Тетраэдр *p0p1p2p3*

Это равенство можно рассматривать как систему линейных уравнений с ненулевым определителем. Решая эту систему, сведем задачу к случаям расположения точки относительно тетраэдра, заданного системой неравенств: *x* ≥ 0, *y*≥ 0, *z*≥ 0, *x*+ *y* + *z* ≤ 1 (рис. 6.9). Обозначим решение системы через (*x*, *y*, *z*). Возможны перечисленные ниже пятнадцать случаев, каждому из которых соответствует список граней выпуклой оболочки.

1. *x*+ *y* + *z* > 1, *x* ≥ 0, *y* ≥ 0, *z* ≥ 0 (видимая грань *p1p2p3*), список граней: *p0p p2*, *p0p2p3*, *p0p1p3*, *qp1p2*, *qp2p3*, *qp3p1*.
2. *x*+ *y* + *z* ≤ 1, *x* < 0, *y* ≥ 0, *z* ≥ 0 (видимая грань *p0p2p3*), список граней: *p0p1p3*, *p0p2p3*, *p1p2p3*, *p0p2q*, *p0p3q*, *p2p3q*.
3. *x*+ *y* + *z* ≤ 1, *x* ≥ 0, *y* < 0, *z* ≥ 0 (видимая грань *p0p1p3*), список граней: *p0p1p2*, *p0p2p3*, *p1p2p3*, *p0p1q*, *p0p3q*, *p1p3q*.
4. *x*+ *y* + *z* ≤ 1, *x* ≥ 0, *y* ≥ 0, *z* < 0 (видимая грань *p0p1p2*), список граней: *p0p1p3*, *p0p2p3*, *p1p2p3*, *p0p1q*, *p0p2q*, *p1p2q*.
5. *x*+ *y* + *z* > 1, *x* ≤ 0, *y* ≥ 0, *z* ≥ 0 (видимые грани *p0p2p3* и *p1p2p3*), список граней: *p0p1p2*, *p0p1p3*, *p1p2q*, *p0p2q*, *p0p3q*, *p1p3q*.
6. *x*+ *y* + *z* > 1, *x* ≥ 0, *y* < 0, *z* ≥ 0 (видимые грани *p0p1p3* и *p1p2p3*), список граней: *p0p1p2*, *p0p2p3*, *p0p1q*, *p1p2q*, *p2p3q*, *p0p3q*.
7. *x*+ *y* + *z* > 1, *x* ≥ 0, *y* ≥ 0, *z* ≤ 0 (видимые грани *p0p1p2* и *p1p2p3*), список граней: *p0p1p3*, *p0p2p3*, *p0p1q*, *p1p3q*, *p0p2q*, *p2p3q*.
8. *x*+ *y* + *z* < 1, *x* ≤ 0, *y* ≤ 0, *z* > 0 (видимые грани *p0p1p3* и *p0p2p3*), список граней: *p0p1p2*, *p1p2p3*, *p0p1q*, *p1p3q*, *p0p2q*, *p2p3q*.
9. *x*+ *y* + *z* < 1, *x* ≤ 0, *y* > 0, *z* ≤ 0 (видимые грани *p0p1p2* и *p0p2p3*), список граней: *p0p1q*, *p1p2q*, *p2p3q*, *p0p3q*.
10. *x*+ *y* + *z* < 1, *x* > 0, *y* ≤ 0, *z* ≤ 0 (видимые грани *p0p1p2* и *p0p1p3*), список граней: *p0p2p3*, *p1p2p3*, *p0p2q*, *p0p3q*, *p1p2q*, *p1p3q*.
11. *x*+ *y* + *z* > 1, *x* < 0, *y* < 0, *z* ≥ 0 (все грани, кроме *p0p1p2*, видимые), список граней: *p0p1p2*, *p0p1q*, *p0p2q*, *p1p2q*.
12. *x*+ *y* + *z* > 1, *x* < 0, *y* ≥ 0, *z* < 0 (все грани, кроме *p0p1p3*, видимые), список граней: *p0p1p3*, *p0p1q*, *p0p3q*, *p1p3q*.
13. *x*+ *y* + *z* > 1, *x* ≥ 0, *y* < 0, *z* < 0 (все грани, кроме *p0p2p3*, видимые), список граней: *p0p2p3*, *p0p2q*, *p0p3q*, *p2p3q*.
14. *x* < 0, *y* < 0, *z* < 0 (все грани, кроме *p1p2p3*, видимые), список граней: *p1p2p3*, *p1p2q*, *p1p3q*, *p2p3q*.
15. *x*+ *y* + *z* ≤ 1, *x* ≥ 0, *y* ≥ 0, *z* ≥ 0 (точка *q* принадлежит тетраэдру), список граней: *p0p1p2*, *p0p1p3*, *p0p2p3*, *p1p2p3*.

***Изображение выпуклого тела.*** Пусть задано выпуклое тело с помощью координат вершин и списка многоугольников, которые служат гранями этого выпуклого тела. Рассмотрим систему координат (*X*, *Y*, *Z*), выражающихся с помощью параллельной проекции через мировые координаты (*x*, *y*, *z*). В этой системе координат плоскость *OXY* будет совпадать с плоскостью экрана, а ось *OZ* будет перпендикулярна плоскости экрана (переход от координат (*X*, *Y*) к экранным производится с помощью масштабирования.)

Система координат (*X*, *Y*, *Z*) называется *проекционной*. В этой системе координат вершины выпуклого тела (*xi*, *yi*, *zi*) будут иметь некоторые координаты (*Xi*, *Yi*, *Zi*).

Для каждой грани запишем уравнение плоскости в проекционной системе координат, содержащей эту грань *akX* + *bkY* + *ckZ* + *dk* = 0, где *k* – номер грани, 1 ≤ *k* ≤ *K*. Предположим, что вектор нормали *N* = (*ak*, *bk*, *ck*) направлен наружу выпуклого тела. Тогда если *Z*-компонента вектора *N*, равная *ck*, не больше нуля, то данная грань невидима. Если же *ck* > 0, то грань видима. Таким образом, получаем схему алгоритма вывода ребер видимых граней выпуклого тела.

1. Преобразовать координаты вершин в проекционные координаты.
2. Для каждой грани *k****,*** 1 ≤ *k* ≤ *K*, найти уравнение плоскости *akX* + *bkY* + *ckZ* + *dk* = 0, проходящей через точки граней. Это можно сделать, например, с помощью формул усреднения Мартина-Ньюэла, позволяющих записать уравнение плоскости, проходящей через точки (*Xi*, *Yi*, *Zi*), при 0 ≤ *i* < *n*, где *j* зависит от *i* следующим образом: *j* = (*i* + 1) mod *n*.

,,.

Коэффициент *dk* вычисляется с помощью любой точки плоскости (*X0*, *Y0*, *Z0*), а именно: *dk* = – *akX0* – *bkY0* – *ckZ0*.

1. Вычислить значения линейных функций *akX* + *bkY* + *ckZ* + *dk*  в центре тяжести вершин тела. Если значение *k*-й функции в центре тяже­сти больше 0, то изменить коэффициенты на противоположные: *ak* = – *ak*, *bk* = – *bk*, *ck* = – *ck*, *dk* = – *dk*.. Координаты центра тяжести тела можно найти как среднее арифметическое координат всех вершин этого тела.
2. Вычислить коэффициенты *ck*. Если *ck* > 0, то вывести (или установить признак видимости) грань с номером *k*.

Такой подход можно применить и в случае центральной проекции. Но центральная проекция не переводит плоскость в плоскость, если учитывать третью координату, что мешает нахождению проекционных координат точек грани по проекционным координатам вершин этой грани, необходимому во многих алгоритмах реалистичного изображения тел.

Для нахождения видимых граней в центральной проекции сначала находятся уравнения плоскостей в мировых координатах (*x*, *y*, *z*), содержащих грани. Пусть *fk* (*x*, *y*, *z*) = *akx* + *bky* + *ckz* + *dk* = 0 — уравнение *k*-й грани. Если значение *fk* (*x*, *y*, *z*) в центре тяжести больше нуля, то следует изменить коэффициенты на противоположные. После этого вектор (*ak*, *bk*, *ck*) будет вектором внешней нормали. Далее берется произвольная точка *p*, принадлежащая *k*-й грани, и соединяется с точкой *qV*, в которой находится наблюдатель. Если угол между вектором внешней нормали (*ak*, *bk*, *ck*) и вектором  является острым, то грань будет видимой, и эту грань следует вывести на экран. В противном случае *k*-я грань невидима. Для каждой грани расчеты и вывод на экран выполняются независимо друг от друга.

***Программа пострения выпуклой оболочки в трехмерном пространстве.*** Определим класс выпуклого тела. Конструктор строит выпуклое тело по четырем некомпланарным точкам. Выпуклая оболочка получается добавлением остальных точек к этому телу. Точка обзора перемещается по окружности, находящейся на высоте *zV*. Это позволяет увидеть созданное выпуклое тело с различных сторон. Определим структуры данных для разработки приложения.

* class Pnt3 — класс, объектами которого являются точки трехмерного пространства. Над соответствующими этим точкам радиус-векторами определим векторные операции. В частности, точки

Pnt3 o(0, 0, 0), e1(1, 0, 0), e2(0, 1, 0), e3(0, 0, 1);

задают начало координат и базис.

* Двухсвязный список ребер, структура элемента которого определена как

struct edge

{

Pnt3 p0, p1;

struct edge \*prev, \*next;

}

вместе с функциями добавления и удаления элемента списка. Помимо этих функций определим функцию, имеющую прототип

edge \*xoredge(edge \*first, Pnt3 q0, Pnt3 q1);

добавляющая в список ребро, соединяющее точки q0 и q1, если такого ребра в списке нет.

Если в списке есть ребро с началом q1 и концом q0, то оно удаляется. В этом случае ребро, соединяющее точки q0 и q1, не добавляется.

* Класс выпуклого тела определяется как список треугольных граней

class convex

{

 face \*ff;

 public:

 void append(Pnt3 pt0, Pnt3 pt1, Pnt3 pt2);

 void delface(face \*cur);

 int addpnt(Pnt3 v);

 void display();

 convex(Pnt3 pt0, Pnt3 pt1, Pnt3 pt2, Pnt3 pt3);

 }

Для вывода движущегося изображения используются сообщения таймера. Определяется виртуальное графическое окно для записи в него изображений. Изображение записывается в него при приеме сообщения от таймера. Выводится с помощью функции CopyRect(), копирующей это окно на экран.

Для создания проекта выполним действия:

1. Запустим Borland C++ Builder с помощью меню Windows **Пуск->Программы**. Затем сохраним с помощью выбора опций **File->Save As** текущий файл **Unit1.cpp** в папку **Convex3d**.
2. Перенесем на пустую форму **Form1** со страницы **Additional** компонент **Image**, со страницы **System —** компонент **Timer**. Затем перенесем на форму **Form1** со страницы **Standart** компонент **Button**.
3. Щелкнув дважды курсором мыши на компоненте **Button**, наберем текст подпрограммы

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

 Timer1->Enabled=!Timer1->Enabled;

}

1. Аналогично наберем текст подпрограммы обработки сообщений таймера.
2. Наберем текст прграммы **Unit1.cpp** и запустим проект на компиляцию, сборку и выполнение, выбрав пункт **Run** главного меню.

Листинг 6.3. Построение трехмерной выпуклой оболочки

//---------------------------------------------------------------------------

#include <vcl.h>

#pragma hdrstop

#include <math.h>

#include "Unit1.h"

#include <stdio.h>

//---------------------------------------------------------------------------

#pragma package(smart\_init)

#pragma resource "\*.dfm"

#define radius 25

TForm1 \*Form1;

Graphics::TBitmap \*Bitmap; // виртуальное графическое окно

int maxX, maxY; // размеры окна

float pxmax, pxmin, pymax, pymin;// максимальные и минимальные координаты

float xv, yv, zv, d, cosa, sina;

class Pnt3

{

 float x, y, z;

public:

 Pnt3(float x0, float y0, float z0):x(x0), y(y0), z(z0) {}

 Pnt3() {}

Pnt3 operator&(Pnt3 q);

Pnt3 operator-(Pnt3 q);

Pnt3 operator+(Pnt3 q);

float operator\*(Pnt3 q);

bool operator==(Pnt3 q)

{

 return ((x==q.x)&&(y==q.y)&&(z==q.z));

}

friend Pnt3 operator\*(float a, Pnt3 p);

friend float vol(Pnt3 p0, Pnt3 p1, Pnt3 p2, Pnt3 p3);

friend float px(Pnt3 p);

friend float py(Pnt3 p);

friend float pz(Pnt3 p);

friend float length(Pnt3 p);

};

Pnt3 Pnt3::operator&(Pnt3 q)

{

Pnt3 r(y\*q.z-z\*q.y, z\*q.x-x\*q.z, x\*q.y-y\*q.x); return r;

}

Pnt3 Pnt3::operator-(Pnt3 q)

{

Pnt3 r(x-q.x, y-q.y, z-q.z); return r;

}

Pnt3 Pnt3::operator+(Pnt3 q)

{

Pnt3 r(x+q.x, y+q.y, z+q.z); return r;

}

float Pnt3::operator\*(Pnt3 q)

{

 return x\*q.x+y\*q.y+z\*q.z;

}

Pnt3 operator\*(float a, Pnt3 p)

{

Pnt3 r(a\*p.x, a\*p.y, a\*p.z); return r;

}

float vol(Pnt3 p0, Pnt3 p1, Pnt3 p2, Pnt3 p3)

{

 return ( ((p1-p0)&(p2-p0))\*(p3-p0));

}

float length(Pnt3 p)

{

 return sqrt(p.x\*p.x+p.y\*p.y+p.z\*p.z);

}

Pnt3 o(0,0,0), u(0,0,1), e1(1,0,0), e2(0,1,0), e3(0,0,1);

float px(Pnt3 p)

{

 return d\*((p.x-xv)\*sina-(p.y-yv)\*cosa)/

 ((p.x-xv)\*cosa+(p.y-yv)\*sina) ;

}

float py(Pnt3 p)

{

 return -d\*(p.z-zv)/((p.x-xv)\*cosa+(p.y-yv)\*sina);

}

float pz(Pnt3 p)

{

 return d/((p.x-xv)\*cosa+(p.y-yv)\*sina);

}

int ex(Pnt3 p)

{

 return (px(p) - pxmin) \* (maxX + 1) / (pxmax - pxmin);

}

int ey(Pnt3 p)

{

 return (pymax - py(p)) \* (maxY + 1) / (pymax - pymin);

}

void fillpoly(int \*x, int \*y, int n, TColor cpen, TColor cbrush)

{

int i;

TColor savepen, savebrush;

TPoint \*p= new TPoint [n];

 for (i=0; i<n; i++)

 {

 p[i]= Point(x[i], y[i]);

 }

 savepen=Bitmap->Canvas->Pen->Color;

 savebrush=Bitmap->Canvas->Brush->Color;

 Bitmap->Canvas->Pen->Color=cpen;

 Bitmap->Canvas->Brush->Color=cbrush;

 Bitmap->Canvas->Polygon(p,n-1);

 Bitmap->Canvas->Pen->Color=savepen;

 Bitmap->Canvas->Brush->Color=savebrush;

}

void filltriangle(Pnt3 p0, Pnt3 p1, Pnt3 p2)

{

int xp[3], yp[3];

Pnt3 v(1,2,3);

float cosg = ( v \*((p1-p0)&(p2-p0)) )/

 length (v)/length((p1-p0)&(p2-p0));

 xp[0]= ex(p0); yp[0]= ey(p0);

 xp[1]= ex(p1); yp[1]= ey(p1);

 xp[2]= ex(p2); yp[2]= ey(p2);

 if (cosg<0) cosg=0;

 cosg=(1.+2\*cosg)/3;

 fillpoly(xp, yp, 3, (TColor)RGB(255\*cosg,255\*cosg,255\*cosg),

 (TColor)RGB(255\*cosg,255\*cosg,255\*cosg));

}

struct face

{

 Pnt3 p0, p1, p2;

 struct face \*prev, \*next;

};

struct edge

{

 Pnt3 p0, p1; // вершины ребра

 struct edge \*prev, \*next;

};

edge \*append(edge \*first, Pnt3 q0, Pnt3 q1)

{

edge \*e = new edge, \*cur =first;

 e->p0 =q0; e->p1 = q1;

 e ->prev = e ->next = NULL;

 if (cur==NULL) return e;

 while(cur->next) cur = cur->next;

 cur->next=e;

 e->prev=cur;

 return first;

}

edge \*deledge(edge \*first, edge \*e)

{

 if (e==first)

 {

 first= first->next; delete e;

 if (first) first->prev=NULL; return first;

 } else

 if (e->next==NULL)

 {

 (e->prev)->next = NULL; delete e; return first;

 } else

 {

 (e->prev)->next=e->next;

 (e->next)->prev=e->prev; delete e; return first;

 }

}

edge \*xoredge(edge \*first, Pnt3 q0, Pnt3 q1)

{

edge \*cur = first; int i=0;

 if (first==NULL) // если список пуст

 {

 first = append(first, q0, q1); return first;

 }

 while (cur)

 {

 if (((cur->p0)==q1)&&((cur->p1)==q0))

 {

 first = deledge(first, cur); return first;

 } else cur = cur -> next;

 }

 first = append(first, q0, q1); return first;

}

class convex

{

 face \*ff;

 public:

 void append(Pnt3 pt0, Pnt3 pt1, Pnt3 pt2);

 void delface(face \*cur);

 int addpnt(Pnt3 v);

 void display();

 convex operator=(convex c);

 convex(){ff=NULL;};

 convex(Pnt3 pt0, Pnt3 pt1, Pnt3 pt2, Pnt3 pt3);

 convex(Pnt3 \*pt, int n);

} ;

void convex::append(Pnt3 pt0, Pnt3 pt1, Pnt3 pt2)

{

face \*ft= new face, \*fcur=ff;

 ft->p0=pt0; ft->p1=pt1; ft->p2=pt2;

 ft->prev=NULL; ft->next=NULL;

 if (ff==NULL) ff=ft;

 else

 {

 while(fcur->next) fcur=fcur->next;

 fcur->next= ft;

 ft->prev=fcur;

 }

}

void convex::delface(face \*cur)

{

 if (cur==NULL) return;

 if (cur==ff)

 {

 ff=ff->next;

 delete cur; ff->prev=NULL; return;

 }

 if (cur->next==NULL)

 {

 (cur->prev)->next=NULL;

 delete cur; return;

 }

 (cur->prev)->next=cur->next;

 (cur->next)->prev=cur->prev; delete cur;

}

int convex::addpnt(Pnt3 v)

{

face \*cur =ff;

 while (cur)

 {

 if (vol(cur->p0,cur->p1,cur->p2,v)>0) break;

 cur = cur->next;

 }

 if(!cur) return 0;

 face \*curnext; cur = ff;

 while(cur)

 {

 curnext = cur ->next;

 if (vol(cur->p0,cur->p1,cur->p2,v)>0) delface(cur);

 cur = curnext;

 }

 edge \*fe=NULL; cur =ff;

 while (cur)

 {

 fe=xoredge(fe,cur->p1,cur->p0);

 fe=xoredge(fe,cur->p2,cur->p1);

 fe=xoredge(fe,cur->p0,cur->p2);

 cur = cur->next;

 }

 while(fe)

 {

 append(fe->p0,fe->p1,v);

 fe=fe->next;

 }

}

void convex::display()

{

face \*fcur=ff;

Pnt3 v(xv,yv,zv);

Bitmap->Canvas->FillRect(Rect(0,0,maxX,maxY));

 for(;fcur;fcur=fcur->next)

 if (vol(fcur->p0,fcur->p1,fcur->p2,v)>0) // выводим, если грань видима

 filltriangle(fcur->p0, fcur->p1, fcur->p2);

Form1->Image1->Canvas->CopyRect(Rect(0,0,maxX,maxY),Bitmap->Canvas,

 Rect(0,0,maxX,maxY));

}

convex convex::operator=(convex c)

{

face \*fcur=ff;

 while(ff)

 {

 delete fcur; ff=ff->next; fcur=ff;

 }

 fcur=c.ff;

 while (fcur)

 {

 append (fcur->p0, fcur->p1, fcur->p2);

 fcur=fcur->next;

 }

}

convex::convex(Pnt3 pt0, Pnt3 pt1, Pnt3 pt2, Pnt3 pt3)

{

float volume;

Pnt3 p[4];

 ff=NULL;

 volume=vol(pt0, pt1, pt2, pt3);

 if (fabs(volume)<0.00001)

 {

 Form1->Image1->Canvas->TextOut(10,10,"Complanar Points");

 return;

 } else

 {

 if (volume<0)

 {

 p[0]=pt0; p[1]=pt1; p[2]=pt2; p[3]=pt3;

 } else

 {

 p[3]=pt0; p[2]=pt1; p[1]=pt2; p[0]=pt3;

 }

 append(p[1],p[2],p[3]);

 append(p[0],p[3],p[2]);

 append(p[0],p[1],p[3]);

 append(p[0],p[2],p[1]);

 }

}

convex::convex(Pnt3 \*pt, int n)

{

int i, i1, i2, i3;

Pnt3 p[4], temp;

float v, volume=0.;

 ff=NULL;

 for (i=0;i<n;i++)

 for (i1=i+1;i1<n;i1++)

 for (i2=i1+1;i2<n;i2++)

 for (i3=i2+1;i3<n;i3++)

 {

 v=vol(pt[i],pt[i1],pt[i2],pt[i3]);

 if(fabs(v)>fabs(volume))

 {

 volume = v;

 p[0]=pt[i];p[1]=pt[i1];p[2]=pt[i2]; p[3]=pt[i3];

 }

 }

 if (fabs(volume)<0.00001)

 {

 Form1->Image1->Canvas->TextOutA(10,10, "Complanar Points"); return;

 } else

 {

 if(volume>0)

 { // инверсия

 temp=p[3]; p[3]=p[0]; p[0]=temp;

 temp=p[2]; p[2]=p[1]; p[1]=temp;

 }

 append(p[1],p[2],p[3]);

 append(p[0],p[3],p[2]);

 append(p[0],p[1],p[3]);

 append(p[0],p[2],p[1]);

 }

 for (i=0; i<n; i++) addpnt(pt[i]);

}

convex cb;

\_\_fastcall TForm1::TForm1(TComponent\* Owner)

 : TForm(Owner)

{

 d=100; zv=d/2.5;

 cosa=1; sina=sqrt(1-cosa\*cosa);

 xv=d\*cosa; yv=d\*sina;

maxX = Form1->Image1->Width;

maxY = Form1->Image1->Height;

pxmax=d/2; pxmin=-d/2;

pymax=0.;

pymin=-(pxmax-pxmin)\*maxY/maxX;

Bitmap = new Graphics::TBitmap;

Bitmap->Width=maxX;

Bitmap->Height=maxY;

/\*

convex cv(o, radius\*e1, radius\*e2, radius\*e3);

 cv.addpnt(radius\*(e1+e2+e3));

 cv.addpnt(radius\*(e1+e2));

 cv.addpnt(radius\*(e2+e3));

 cv.addpnt(radius\*(e1+e3));

 cv.addpnt(12.5\*e1+37.5\*e3);

 cv.addpnt(12.5\*e1+25\*e2+37.5\*e3);

\*/

Pnt3 p[10];

 p[0]=o; p[1]=radius\*e1; p[2]= radius\*e2; p[3]= radius\*e3;

 p[4]=radius\*(e1+e2+e3);

 p[5]=radius\*(e1+e2);

 p[6]= radius\*(e2+e3);

 p[7]= radius\*(e1+e3);

 p[8]= 0.5\*radius\*e1+1.5\*radius\*e3;

 p[9]= 0.5\*radius\*e1+radius\*e2+1.5\*radius\*e3;

convex cv(p, 10);

 cb=cv;

 Timer1->Interval=200;

}

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

 Timer1->Enabled=!Timer1->Enabled;

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::Timer1Timer(TObject \*Sender)

{

static int i;

 cosa=cos(0.1\*i); sina=sin(0.1\*i); i++;

 xv=d\*cosa; yv=d\*sina;

 cb.display();

}

//---------------------------------------------------------------------------

Результаты работы программы приведены на рис. 6.10.



Рис. 6.10. Вращение выпуклой оболочки вокруг вертикальной оси

## 6.4. МЕТОДЫ ПОСТРОЕНИЯ ВЫПУКЛОЙ ОБОЛОЧКИ

Рассмотрим алгоритмы построения выпуклой оболочки множества точек в плоскости.

***Метод заворачивания подарка.*** Этот метод построения выпуклой оболочки основан на том, что отрезок, соединяющий две точки заданного конечного множества, является ребром выпуклой оболочки тогда и только тогда, когда все другие точки заданного множества расположены на этом отрезке или с одной стороны от него. Отметим, что если отрезок *ab*, соединяющий точки *a* и *b*, является ребром выпуклой оболочки, то существует еще одно ребро, с вершиной в точке *b*.

Пусть задано конечное множество точек плоскости. Обозначим это множество *M*. Предположим, что выбрана точка (*xa*, *ya*), которая заведомо является одной из вершин выпуклой оболочки. Для этого годится точка, имеющая наименьшую абсциссу. Если таких точек несколько, то выбираем среди них точку с наименьшей ординатой. Вокруг этой точки поворачивается исходящий из нее луч по направлению движения часовой стрелки до тех пор, пока он не встретит некоторую точку (*xb* *yb*) из множества *M*. Отрезок, соединяющий точку (*xa*, *ya*) с точкой (*xb*, *yb*), будет стороной строящегося выпуклого многоугольника (рис. 6.11).

Для поиска следующего ребра продолжим вращение луча по часовой стрелке. На этот раз луч будет вращаться вокруг точки *b* до встречи со следующей точкой *c*, принадлежащей множеству *M*. Затем луч вращается вокруг точки *c*  *M* до встречи с новой точкой *d*, как это показано на рис. 6.11. Процесс продолжается до тех пор, пока не будет достигнута точка *a*. На рис. 6.11 показан пример построения выпуклой оболочки, в результате которого будет получен многоугольник *abcde*.











Рис. 6.11. Метод заворачивания подарка

Данный метод построения является частным случаем метода заворачивания подарка и называется *методом обхода Джарвиса*. Метод заворачивания подарка является более общим и годится для построения выпуклой оболочки в пространстве размерности более двух. Приведем программную реализацию описанного алгоритма.

Для создания ее проекта достаточно загрузить Borland C++ Builder, перенести компонент **Image** на главную форму **Form1**, набрать следующий далее текст **Unit1.cpp** и запустить приложение на компиляцию, сборку и выполнение с помощью выбора пункта **Run** главного меню.

Листинг 6.4. Выпуклая оболочка методом заворачивания подарка

//---------------------------------------------------------------------------

// выпуклая оболочка методом заворачивания подарка

#include <vcl.h>

#pragma hdrstop

#include "Unit1.h"

//---------------------------------------------------------------------------

#pragma package(smart\_init)

#pragma resource "\*.dfm"

#include <math.h>

#include <stdlib.h>

#define PI 3.14159

TForm1 \*Form1;

int maxX, maxY;

struct Pnt // класс точки

{

 float x, y; // координаты точки

 int code(Pnt q); // код четверти, в которой лежит точка q

 float operator\*(Pnt q); // векторное произведение

 Pnt operator-(Pnt q); // разность векторов

};

int Pnt::code(Pnt q) // код четверти, в которой лежит точка q

{

// начало координат находится в точке (x, y)

 if(q.x - x >= 0 && q.y - y >= 0) return 0;

 if(q.x - x < 0 && q.y - y >= 0) return 1;

 if(q.x - x >= 0 && q.y - y < 0) return 3;

 if(q.x - x < 0 && q.y - y < 0) return 2;

}

float Pnt::operator\*(Pnt q)

{

 return x \* q.y - y \* q.x; // векторное произведение

}

Pnt Pnt::operator-(Pnt q) // разность векторов

{

Pnt t; t.x = x - q.x; t.y = y - q.y; return t;

}

float fabs(Pnt p)

{

 return sqrt(p.x\*p.x+p.y\*p.y);

}

int operator<(Pnt p, Pnt q)

{

Pnt t; // сравнение углов радиус-векторов p и q

 t.x = 0; t.y = 0; // коды четвертей вычисляются относительно (0,0)

 if(t.code(p) < t.code(q)) return 1;

 if(t.code(p) > t.code(q)) return 0;

 return (p \* q > 0); // вращение от p к q направлено

// против часовой стрелки

}

class CPolygon

{

 int color, n;

 Pnt \*p;

 public:

 CPolygon(float \*x, float \*y, int m, TColor cl);

 CPolygon(const CPolygon &ob); // конструктор копирования

 int isin(Pnt t);

 void show(float xmin, float ymin, float xmax, float ymax);

};

// Конструктор копирования

CPolygon::CPolygon(const CPolygon &ob) // необходим для передачи объекта

{ // в качестве параметра

 int i;

 n = ob.n; color = ob.color;

 p = new Pnt[n]; // выделим память

 for(i=0; i < ob.n; i++) p[i]=ob.p[i]; // производим копирование

}

void CPolygon::show(float xmin, float ymin, float xmax, float ymax)

{

int i;

TPoint \*parray = new TPoint [n];

 Form1->Image1->Canvas->Brush->Color=color;

 for(i = 0; i < n; i++)

 {

 parray[i] = Point(

 (p[i].x - xmin) \* maxX / (xmax - xmin),

 (ymax - p[i].y) \* maxY / (ymax - ymin));

 }

 Form1->Image1->Canvas->Polygon(parray,n-1);

}

CPolygon::CPolygon(float \*x, float \*y, int m, TColor cl):color(cl)

{

// построение выауклой оболочки

int a, i, j = 0, k = 0; Pnt \*pp = new Pnt [m + 1];

Pnt temp, \*s = new Pnt [m + 1];

 for(a = 0, i = 1; i < m; i++) //выбор точки с наименьшей абсциссой

 if((x[i] < x[a]) || (x[i] == x[a] && (y[i] < y[a]))) a = i;

 for(i = 0; i < m; i++)

 {

s[i].x = x[i]; s[i].y = y[i]; // перестановка s[a] и s[k]

 }

 s[m].x = x[a]; s[m].y = y[a];

 for(k = 0; k < m; k++)

 {

 temp = s[a]; s[a] = s[k]; s[k] = temp; pp[j++] = s[k];

a = k + 1;

 for(i = k + 2; i <= m; i++)

 { // нахождение точки лежащей слева от остальных

if(((s[a] - s[k]) \* (s[i] - s[k]) > 0.0) ||

 ((s[a] - s[k]) \* (s[i] - s[k]) == 0.0 && fabs(s[a] - s[k])

< fabs(s[i] - s[k]))) a = i;

 }

 if(a == m) break;

 }

 n = j; p = new Pnt [n];

 for(i = 0; i < n; i++) p[i] = pp[i];

 delete []s; delete []pp;

}

int CPolygon::isin(Pnt t)

{

int i,j, parity = 0;

 for(i = 0, j=n-1; i < n; j=i++)

 {

 if (( ( (p[i].y<=t.y)&&(t.y<p[j].y) )||

 ((p[j].y<=t.y)&&(t.y<p[i].y)))&&

 (t.x<(p[j].x-p[i].x)\*(t.y-p[i].y)/

 (p[j].y-p[i].y)+p[i].x))

 parity=1-parity;

 }

 return parity;

}

//---------------------------------------------------------------------------

\_\_fastcall TForm1::TForm1(TComponent\* Owner): TForm(Owner)

{

 randomize();

 maxX= Form1->Image1->Width;

 maxY= Form1->Image1->Height;

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

float rad = 150, \*x, \*y; // радиус окружности и

// вершины 2mm-угольника

int i;

int mm = random(10)+2;

 x = new float[2 \* mm]; y = new float [2 \* mm];

 for(i = 0; i < mm; i++)

 {

// вершины большого 2mm-угольника

x[i] = rad \* cos(2. \* PI \* i / mm);

y[i] = rad \* sin(2. \* PI \* i / mm);

// вершины маленького 2mm-угольника

x[i + mm] = rad \* cos(2. \* PI \* i / mm + PI / mm) / 2;

y[i + mm] = rad \* sin(2. \* PI \* i / mm + PI / mm) / 2;

 }

 for(i = 0; i < 2 \* mm; i++) x[i] += random(100) - 50;

 Form1->Image1->Canvas->FillRect(TRect(0,0,maxX,maxY));

CPolygon cpoly(x, y, 10, clLtGray); // построение выпуклой оболочки

cpoly.show(-300, -300, 300, 300); // вывод выпуклой оболочки

Form1->Image1->Canvas->Brush->Color = (TColor)clWhite;//восстановление кисти

Pnt f = {0., 0.};

 for(i = 0; i < 2 \* mm; i++)

 {

f.x = x[i]; f.y = y[i];

if(cpoly.isin(f))

Form1->Image1->Canvas->Rectangle((int)(x[i] + 300.) \* maxX / 600 - 1,

(int)((300. - y[i]) \* maxY / 600 - 1),

(int)((x[i] + 300.) \* maxX / 600) + 1,

(int)((300. - y[i]) \* maxY / 600) + 1);

 }

 delete []x; delete []y;

}

//---------------------------------------------------------------------------

Результат работы программыприведен на рис. 6.12.

  

Рис. 6.12. Выпуклая оболочка, построенная методом заворачивания подарка

Оценим сложность алгоритма построения выпуклой оболочки методом заворачивания подарка.

Так как все *m* точек множества *M* могут быть вершинами его выпуклой оболочки, а рассматриваемый алгоритм затрачивает на нахождение каждой точки оболочки линейное время, то время выполнения в худшем случае равно *O* (*m*2). Если в действительности число вершин выпуклой оболочки равно *h*, то время выполнения алгоритма Джарвиса будет равно *O* (*hm*). Поэтому этот алгоритм особенно эффективен в тех случаях, когда заранее известно, что число сторон выпуклой оболочки ограничено некоторой константой *h*.

***Метод обхода Грэхема.*** Этот метод основан на теореме о том, что последовательные вершины выпуклого многоугольника можно расположить в порядке, соответствующем возрастанию угла относительно любой внутренней точки. В качестве внутренней точки можно взять центр тяжести множества точек *M*, выпуклую оболочку которых требуется найти. Перейдем к системе координат с началом в этой внутренней точке. Упорядочим лексикографически все точки множества *M* в соответствии со значениями полярного угла и расстояния от начала координат. Сравнение расстояний выполняется лишь в случае, когда две точки имеют один и тот же полярный угол. В результате точки будут располагаться вокруг внутренней точки, как это показано на рис. 6.13. Легко видеть, что если точка из *M* не является вершиной выпуклой оболочки, то она будет внутренней для некоторого треугольника *Opq*, где *p* и *q*— последовательные вершины выпуклой оболочки.





Рис. 6.13. Метод обхода Грэхема

В примере, показанном на рис. 6.13, точка *p2* является внутренней точкой треугольника. Вершина *pi0* является начальной. Эта вершина выбирается среди точек, имеющих наименьшую ординату. Она должна быть самой правой среди этих точек. Вследствие такого выбора начальная вершина будет принадлежать множеству вершин искомого выпуклого многоугольника.

Затем тройки последовательных точек из упорядоченного массива многократно проверяются в порядке обхода против часовой стрелки с целью определить, образуют они или нет угол, больший или равный π. Если внутренний угол *pi-1pipi+1* больше или равен π, то говорят, что *pi-1pipi+1* образуют *правый поворот*, иначе они образуют *левый поворот*. На рис. 6.13 тройка точек *p0p1p2* образует левый поворот, а *p1p2p3*— правый поворот. Это свойство легко проверяется с помощью векторного произведения плоских векторов: в первом случае векторное произведение (*pi – pi-1*) × (*pi+1 – pi-1*) меньше нуля, а во втором векторное произведение будет больше нуля. Напомним, что для векторов в плоскости *Oxy* векторное произведение будет вектором, параллельным оси *Oz*, и мы в этом случае под векторным произведением понимаем вещественное число, равное величине проекции этого вектора на ось *Oz*. Из выпуклости многоугольника следует, что при его обходе будут делаться только левые повороты. В зависимости от величины угла, образуемого текущей тройкой точек, возможны следующие два варианта продолжения просмотра:

* если *pi-1pipi+1* образуют правый поворот, то удаляем вершину *pi* из массива;
* если *pi-1pipi+1* образуют левый поворот, то увеличиваем *i* на 1.

Просмотр завершается, когда, обойдя все вершины, вновь приходим к начальной вершине *pi0*.

Анализ показывает, что такой просмотр выполняется за линейное время *O* (*m*). Тем не менее, сложность алгоритма будет зависеть от сложности метода сортировки точек перед просмотром. Если выбрать быструю сортировку Хоара или турнирную сортировку, то получим, что время работы алгоритма методом обхода Грэхема равно *O* (*m* log *m*).

Приведем программную реализацию метода обхода Грэхема.

Листинг 6.5. Метод Грэхема

// метод Грэхема

//---------------------------------------------------------------------------

#include <vcl.h>

#pragma hdrstop

#include "Unit1.h"

//---------------------------------------------------------------------------

#pragma package(smart\_init)

#pragma resource "\*.dfm"

#include <math.h>

#include <stdlib.h>

#define PI 3.14159

TForm1 \*Form1;

int getmaxx()

{

 return Form1->Image1->Width;

}

int getmaxy()

{

 return Form1->Image1->Height;

}

struct Pnt // класс точки

{

 double x, y; // координаты точки

 int code(Pnt q); // код четверти, в которой лежит точка q

 double operator\*(Pnt q); // векторное произведение

 Pnt operator-(Pnt q); // разность векторов

 int operator!=(Pnt q);

};

int Pnt::code(Pnt q) // код четверти, в которой лежит точка q

{

// начало координат находится в точке (x, y)

 if(q.x - x >= 0 && q.y - y >= 0) return 0;

 if(q.x - x < 0 && q.y - y >= 0) return 1;

 if(q.x - x >= 0 && q.y - y < 0) return 3;

 if(q.x - x < 0 && q.y - y < 0) return 2;

}

double Pnt::operator\*(Pnt q)

{

 return x \* q.y - y \* q.x; // векторное произведение

}

Pnt Pnt::operator-(Pnt q) // разность векторов

{

 Pnt t; t.x = x - q.x; t.y = y - q.y; return t;

}

double fabs(Pnt p)

{

 return sqrt(p.x\*p.x+p.y\*p.y);

}

int operator<(Pnt p, Pnt q)

{

 Pnt t; // сравнение углов радиус-векторов p и q

 t.x = 0; t.y = 0; // коды четвертей вычисляются

// относительно (0,0)

 if(t.code(p) < t.code(q)) return 1;

 if(t.code(p) > t.code(q)) return 0;

 return (p \* q > 0); // вращение от p к q направлено

// против часовой стрелки

}

int Pnt::operator!=(Pnt q)

{

 return x!=q.x || y!=q.y;

}

int intersect(Pnt p, Pnt p1, Pnt p2)

{

// тест на пересечение луча и отрезка

 if(p1.y == p2.y) return 0;

 if((p1.y < p2.y ? p2.y : p1.y) <= p.y) return 0;

 if((p1.y < p2.y ? p1.y : p2.y) > p.y) return 0;

 if(p2.y - p1.y > 0) return

 ((p.x - p1.x)\*(p2.y - p1.y)-(p2.x - p1.x)\*(p.y - p1.y) > 0);

 else return

 ((p.x - p1.x)\*(p2.y - p1.y)-(p2.x - p1.x)\*(p.y - p1.y) < 0);

}

class CPolygon

{

 int color, n;

 Pnt \*p, pC;

 public:

 CPolygon(double \*x, double \*y, int m, TColor cl);

 CPolygon(const CPolygon &ob); // конструктор копирования

 int isin(Pnt t);

 void show(double xmin, double ymin,

 double xmax, double ymax);

};

// Конструктор копирования

CPolygon::CPolygon(const CPolygon &ob) // необходим для передачи объекта

{ // в качестве параметра

 int i;

 n = ob.n; color = ob.color;

 p = new Pnt[n]; // выделим память

 for(i=0; i < ob.n; i++) p[i]=ob.p[i]; // производим копирование

}

void CPolygon::show(double xmin, double ymin,

 double xmax, double ymax)

{

int i;

TPoint \*parray = new TPoint [n];

 Form1->Image1->Canvas->Brush->Color=color;

 for(i = 0; i < n; i++)

 {

 parray[i] = Point(

 (p[i].x - xmin) \* getmaxx() / (xmax - xmin),

 (ymax - p[i].y) \* getmaxy() / (ymax - ymin));

 }

 Form1->Image1->Canvas->Polygon(parray,n-1);

}

CPolygon::CPolygon(double \*x, double \*y, int m, TColor cl):color(cl)

{

int i, j; int i0 = 0, v, w, f = 0;

Pnt t;

 p = new Pnt [m]; n = m;

 pC.x = 0; pC.y = 0;

 for(i = 0; i < m; i++)

 {

 pC.x += x[i]; pC.y += y[i];

 }

 pC.x = pC.x / m; pC.y = pC.y / m;

 for(i = 0; i < m; i++)

 {

 p[i].x = x[i]; p[i].y = y[i];

 }

 for(i = 1; i < m; i++) // сортируем точки по возрастанию угла

 // вокруг центра тяжести методом вставок

 {

 t = p[i];

 for(j = i - 1; (j >= 0) && ((t - pC) < (p[j] - pC)); j--)

 p[j + 1] = p[j];

 p[j + 1] = t;

 }

// построение звездчатого многоугольника

 for(j = 0; j < n; j++)

 {

 if(p[j].y < p[i0].y) i0 = j; // точка с наименьшей ординатой

 else if(p[j].y == p[i0].y && p[j].x > p[i0].x)

 i0 = j; // самая правая

 }

 v = i0;

Pnt p0 = p[i0];

 if(v > 0) w = (v - 1)%n; else w = n - 1; f = 0;

Pnt p1 = p[w];

 while((p[(v + 1)%n] != p0) || (f == 0))

 {

 if(p[(v + 1)%n] != p1); else f = 1;

 if((p[(v + 1)%n] - p[v]) \* (p[(v + 2)%n] - p[v]) > 0) v = (v + 1)%n;

 else

 {

 for(i = (v + 1)%n; i < n; i++) p[i]=p[(i + 1)%n]; //i<n

 n = n - 1; // удаление v + 1

 if(v == 0) v = n - 1;

 else v = (v - 1)%n;

 }

 }

}

//

int CPolygon::isin(Pnt t)

{

int i,j, parity = 0;

 for(i = 0, j=n-1; i < n; j=i++)

 {

 if (( ( (p[i].y<=t.y)&&(t.y<p[j].y) )||

 ((p[j].y<=t.y)&&(t.y<p[i].y)))&&

 (t.x<(p[j].x-p[i].x)\*(t.y-p[i].y)/

 (p[j].y-p[i].y)+p[i].x))

 parity=1-parity;

 }

 return parity;

}

//---------------------------------------------------------------------------

\_\_fastcall TForm1::TForm1(TComponent\* Owner)

 : TForm(Owner)

{

double rad = 150, \*x, \*y; // радиус окружности и

// вершины 2mm-угольника

int i; randomize();

int mm = 10;

 x = new double[2 \* mm]; y = new double [2 \* mm];

 for(i = 0; i < mm; i++)

 {

 // вершины большого 2mm-угольника

 x[i] = rad \* cos(2. \* PI \* i / mm);

 y[i] = rad \* sin(2. \* PI \* i / mm);

 // вершины маленького 2mm-угольника

 x[i + mm] = rad \* cos(2. \* PI \* i / mm + PI / mm) / 2;

 y[i + mm] = rad \* sin(2. \* PI \* i / mm + PI / mm) / 2;

 }

 randomize();

 for(i = 0; i < 2 \* mm; i++) x[i] += random(100) - 50;

CPolygon cpoly(x, y, 10, clLtGray); // построение выпуклой оболочки

cpoly.show(-300, -300, 300, 300);// вывод выпуклой оболочки

Pnt f = {0., 0.};

 for(i = 0; i < 2 \* mm; i++)

 {

 f.x = x[i]; f.y = y[i];

 if(cpoly.isin(f))

 Form1->Image1->Canvas->Rectangle((int)(x[i] + 300.) \* getmaxx() / 600 - 1,

 (int)((300. - y[i]) \* getmaxy() / 600 - 1),

 (int)((x[i] + 300.) \* getmaxx() / 600) + 1,

 (int)((300. - y[i]) \* getmaxy() / 600) + 1);

 }

 delete []x; delete []y;

}

//---------------------------------------------------------------------------

Программа выводит случайные точки и их выпуклые оболочки. Результат работы программы приведен на рис. 6.14.



Рис. 6.14. Выпуклые оболочки, построенные методом Грэхема

Для создания проекта необходимо выполнить следующие действия:

1. Перенести компонент **Image**, со страницы **Additional** на главную форму **Form1**.
2. Перенести компонент **Button** со страницы **Standart** наглавную форму **Form1**.
3. Щелкнув дважды курсором мыши на кнопке **Button,** набрать приведенный выше текст модуля **Unit1.cpp**.
4. Запустить проект на компиляцию, сборку и выполнение, выбрав пункт **Run** главного меню.

***Метод Эндрю.*** В методе обхода Грэхема имеется один недостаток: при упорядочении точек используется сравнение углов. Чтобы исправить этот недостаток, Эндрю предложил следующую модификацию метода обхода Грэхема, в которой вместо сравнения углов применяется сравнение абсцисс точек.

Пусть задано конечное множество *M* точек плоскости, состоящее из *m* элементов. Сначала находятся его левая крайняя точка *l* и правая крайняя точка *r* этого множества. Для определенности среди точек с наименьшей абсциссой выбирается точка с наименьшей ординатой, а среди точек с наибольшей абсциссой — точка с наибольшей ординатой. Построим прямую, проходящую через точки *l* и *r* (рис. 6.15). Множество *M* разбивается на два подмножества, первое из которых состоит из точек, расположенных выше прямой, и называется *верхним подмножеством*, а второе состоит из остальных точек и называется *нижним подмножеством*.

*Верхнее подмножество*

*Нижнее подмножество*

Рис. 6.15. Разбиение, определенное крайними точками

Строится выпуклая оболочка верхнего подмножества вместе с точками *l* и *r*. Затем строится выпуклая оболочка нижнего подмножества и точек *l* и *r*. В результате получаем две ломанные, которые вместе составляют выпуклый многоугольник, ограничивающий искомую выпуклую оболочку.

Выпуклая оболочка верхнего подмножества и точек *l* и *r* строится в два этапа:

1. Верхние точки вместе с точками *l* и *r* сортируются по возрастанию абсцисс. Среди точек с одинаковой абсциссой оставляется лишь верхняя.
2. Производится просмотр троек *pipi+1pi+2* верхних точек. Если вектор *pipi+1* расположен слева от *pipi+2*, то увеличивается *i*, в противном случае точка *pi+1* удаляется из списка (или массива) верхних точек.

Приведем программную реализацию метода обхода Эндрю..

Листинг 6.5. Метод Эндрю

// метод Эндрю

//---------------------------------------------------------------------------

#include <vcl.h>

#pragma hdrstop

#include "Unit1.h"

//---------------------------------------------------------------------------

#pragma package(smart\_init)

#pragma resource "\*.dfm"

TForm1 \*Form1;

#include <time.h>

#include <math.h>

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

void line(int x0, int y0, int x1, int y1)

{

 Form1->Image1->Canvas->MoveTo(x0,y0);

 Form1->Image1->Canvas->LineTo(x1,y1);

}

void putpixel(int x, int y, int c)

{

 Form1->Image1->Canvas->Pixels[x][y]=RGB(0,c,0);

}

// Структура, описывающая координаты одной точки

class PNT

{

public:

float X;

float Y;

// Функция сравнивает два элемента

int operator==(PNT& tmp)

{

if((X == tmp.X) && (Y == tmp.Y)) return 1;

else return 0;

};

// Функция присваивает одному классу значения другого

PNT& operator=(PNT& tmp)

{

X = tmp.X; Y = tmp.Y;

return \*this;

};

};

// Класс, описывающий выпуклую оболочку

class Convex

{

private:

PNT \*tchK; // Массив точек

int len; // Количество точек

// Минимальные и максимальные возможные координаты

float MIN\_X, MIN\_Y;

float MAX\_X, MAX\_Y;

// Вспомогательные функции

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Нахождение мин/макс элемента

// при r = 1 - минимальный

// при r = -1 - максимальный

int min\_max(int r);

// Функция определяет присутствие такого элемента в массиве

// Возвращает индекс найденного элемента,

// в противном случае -1

int is\_exist(int index);

// Функция убирает из массива заданный элемент

// со сдвигом элементов влево

void del(int index);

// Функция сортирует массив в следующем порядке

// Первый элемент - левый нижний

// Далее все элементы, лежащие сверху отрезка

// в порядке возрастания их координат по ос X

// Следующий - правый верхний

// И дальше все элементы, лежащие на отрезке или ниже

// в порядке уменьшения их координат X

void sort(int \*min, int \*max);

// Функция возвращает массив точек, находящихся

// при r = 1 - выше отрезка

// при r = -1 - на отрезке или ниже его

void uslovie(int min, int max, PNT \*mas, int \*lens, int r);

// Функция сортирует массив

// при r = 1 - по возрастанию

// при r = -1 - по убыванию

void sortmas(PNT \*mas, int lens, int r);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public:

// Конструкторы

Convex()

{

MIN\_X = -310; MAX\_X = 310; MIN\_Y = -230; MAX\_Y = 230;

tchK = NULL;

};

Convex(int left, int top, int right, int bottom)

:MIN\_X(left), MIN\_Y(top), MAX\_X(right), MAX\_Y(bottom)

{ tchK = NULL; };

// Деструктор

~Convex(){ free(tchK);};

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Функция создает ошибку с выходом из программы

void error\_s(int i, char \*err)

{

//cout<<"\nRaise error "<<i<<" Message: "<<err<<"\n";

//exit(1);

};

// Функция заполняет массив точек заданными значениями

void put(PNT \*mas, int lens);

// Функция заполняет массив точек произвольными значениями

void put\_rnd(int lens);

// Функция возвращает количество точек массива

int length(void);

// Функция возвращает указанный элемент

PNT operator[] (int index)

{

if(index > len - 1) error\_s(1,"Exit to range massiv");

return tchK[index];

};

// Функция производит все операции по созданию оболочки

void create(void);

// Функция возвращает смещение по Х и по Y

int absX(int rX) { return MAX\_X + rX;};

int absY(int rY) { return MAX\_Y - rY;};

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

};

// Тела функций

void Convex::put(PNT \*mas, int lens)

{

// Если массив не пуст, удаляем его

if(tchK != NULL) free(tchK);

// Создаем новый массив

tchK = (PNT\*) malloc(lens \* sizeof(PNT));

// Заполняем его пользовательскими данными

for(int i = 0; i < lens; i++)

tchK[i] = mas[i];

// Устанавливаем длину массива

len = lens;

};

void Convex::put\_rnd(int lens)

{

randomize();

// Если массив не пуст, очищаем его

if(tchK != NULL) free(tchK);

// Создаем новый массив

tchK = (PNT\*) malloc(lens \* sizeof(PNT));

// Заполняем массив сгенерированными величинами

for(int i = 0; i < lens; i++)

{

tchK[i].X = MIN\_X + ((float)rand() \* (MAX\_X - MIN\_X) /

(float)RAND\_MAX);

tchK[i].Y = MIN\_Y + ((float)rand() \* (MAX\_Y - MIN\_Y) /

(float)RAND\_MAX);

};

// Устанавливаем длину массива

len = lens;

};

int Convex::length(void)

{ return len;};

int Convex::min\_max(int r)

{

int tmp = tchK[0].X, tmp1, tmpindex = 0;

// Находим мин/макс элемент по оси Х

for(int i = 0; i < len; i++)

{

if(tmp \* r > tchK[i].X \* r)

{

tmp = tchK[i].X;

tmpindex = i;

};

};

// Из найденных мин/макс элементов по Х

// находим мин/макс элемент по оси Y

tmp1 = tchK[tmpindex].Y;

for(int i = tmpindex; i < len; i++)

{

if((tmp1 \* r > tchK[i].Y \* r) && (tmp == tchK[i].X))

{

tmp1 = tchK[i].Y;

tmpindex = i;

};

};

// Возвращаем индекс найденного мин/макс элемента

return tmpindex;

}

int Convex::is\_exist(int index)

{

for(int i = 0; i < len; i++)

{

if(index == i) continue;

if(tchK[index] == tchK[i]) return i;

};

return -1;

}

void Convex::del(int index)

{

PNT \*tmp;

int r = 0;

tmp = (PNT\*) malloc((len - 1) \* sizeof(PNT));

for(int i = 0; i < len; i++)

{

if(index == i) {r = -1; continue;};

tmp[i + r] = tchK[i];

};

free(tchK);

tchK = tmp;

len--;

}

void Convex::sort(int \*min, int \*max)

{

PNT \*tmprez, \*tmp;

int lens;

int tec, i;

tmprez = (PNT\*) malloc(len \* sizeof(PNT) + 1);

tmp = (PNT\*) malloc(len \* sizeof(PNT));

// Находим все точки, находящиеся выше отрезка

uslovie(\*min, \*max, tmp, &lens, 1);

sortmas(tmp, lens, 1);

// Записываем в результирующий массив первый элемент

tmprez[0] = tchK[\*min];

// Записываем в массив все точки выше отрезка

for(i = 0, tec = 1; i < lens; i++, tec++)

tmprez[tec] = tmp[i];

// Записываем в массив крайний верхний правый элемент

tmprez[tec++] = tchK[\*max];

//Записываем в массив все точки ниже отрезка

uslovie(\*min, \*max, tmp, &lens, -1);

sortmas(tmp, lens, -1);

// Получаем новые min и max

\*min = 0;

\*max = tec-1;

for(i = 0; i < lens; i++, tec++)

tmprez[tec] = tmp[i];

tmprez[tec] = tmprez[0];

len += 1;

// Присваиваем массиву класса результирующий массив

free(tchK);

tchK = tmprez;

}

void Convex::uslovie(int min, int max, PNT \*mas,

int \*lens, int r)

{

// Коэффициенты прямой по двум точкам min и max

float a = (float)(tchK[max].Y - tchK[min].Y) /

(float)(tchK[max].X - tchK[min].X);

float b = (float)(tchK[max].X \* tchK[min].Y -

tchK[min].X \* tchK[max].Y) /

(float)(tchK[max].X - tchK[min].X);

\*lens = 0;

for(int j = 0; j < len; j++)

{

if((j == min) || (j == max)) continue;

// Проверяем на условие в зависимости от значения r

if((float)tchK[j].Y \* r > (a \* (float)tchK[j].X + b) \* r)

{

mas[\*lens] = tchK[j];

\*lens += 1;

};

};

}

void Convex::sortmas(PNT \*mas, int lens, int r)

{

PNT max;

int kon;

for(int i = 0; i < lens; i++)

{

kon = i;

max = mas[i];

for(int j = i + 1; j < lens; j++)

if(max.X \* r > mas[j].X \* r) {max = mas[j]; kon = j;};

if(kon != i) {mas[kon] = mas[i]; mas[i] = max;};

};

}

void Convex::create(void)

{

int i, rc;

int min, max;

int tmplen = len;

if(tchK == NULL) error\_s(2,"Massiv is empty");

// Удаляем повторяющиеся элементы

for(i = 0; i < len; i++)

while((rc = is\_exist(i)) != -1) del(rc);

// Проверяем, чтобы у нас массив состоял как минимум из трех точек

max = min\_max(-1);

min = min\_max(1);

if(len < 3) error\_s(3,"Haves menee treh tochek");

// Сортируем массив согласно требованию алгоритма

sort(&min, &max);

////setcolor(15);

line(10, MAX\_Y + 10, 2 \* MAX\_X + 10, MAX\_Y + 10);

line(MAX\_X + 10, 10, MAX\_X + 10, 2 \* MAX\_Y + 10);

for(i = 0; i < len; i++)

putpixel(absX(tchK[i].X) + 10, absY(tchK[i].Y) + 10, 15);

// Отсеиваем точки, находящиеся сверху отрезка

if(max - min > 1)

{

for(i = max; i > 1; i--)

{

if((tchK[i-1].X-tchK[i].X)\*(tchK[i-2].Y-tchK[i-1].Y) -

(tchK[i-2].X-tchK[i-1].X)\*(tchK[i-1].Y-tchK[i].Y)<0)

{

del(i - 1);

if(i < max) i++;

max--;

};

}

};

if(max < len - 1)

{

for(i = len - 1; i > max + 1; i--)

{

if((tchK[i-1].X-tchK[i].X)\*(tchK[i-2].Y-tchK[i-1].Y) -

(tchK[i-2].X-tchK[i-1].X)\*(tchK[i-1].Y-tchK[i].Y)<0)

{

del(i - 1);

if(i < len) i++;

};

};

};

for(i = 0; i < len - 1; i++)

{

line(absX(tchK[i + 1].X) + 10,

absY(tchK[i + 1].Y) + 10,

absX(tchK[i].X) + 10,

absY(tchK[i].Y) + 10);

putpixel(absX(tchK[i].X) + 10, absY(tchK[i].Y) + 10, 15);

};

line(absX(tchK[min].X) + 10,

absY(tchK[min].Y) + 10,

absX(tchK[max].X) + 10,

absY(tchK[max].Y) + 10);

}

//---------------------------------------------------------------------------

\_\_fastcall TForm1::TForm1(TComponent\* Owner)

 : TForm(Owner)

{

char ch;

Convex obj;

obj.put\_rnd(15);

obj.create();

}

//---------------------------------------------------------------------------

Программа выводит случайные точки и их выпуклые оболочки. Результат работы программы приведен на рис. 6.16.

 

Рис. 6.16. Выпуклые оболочки, построенные методом Эндрю

Для создания проекта необходимо выполнить следующие действия:

1. Перенести компонент **Image**, со страницы **Additional** на главную форму **Form1**.
2. Перенести компонент **Button** со страницы **Standart** наглавную форму **Form1**.
3. Щелкнув дважды курсором мыши на кнопке **Button,** набрать приведенный выше текст модуля **Unit1.cpp**.
4. Запустить проект на компиляцию, сборку и выполнение, выбрав пункт **Run** главного меню.

## 6.5. ЗАДАЧИ

1. Пусть *P*— простой многоугольник с *n* вершинами *(xi,* *yi),*0 ≤ *i*< *n*. Доказать, что его площадь равна абсолютной величине суммы , где индексы *i*+1 берутся по модулю *n* , т. е. *xn* = *x0* и *yn* = *y0*.

1. Написать подпрограмму, находящую пересечение двух выпуклых многоугольников, лежащих в декартовой плоскости и заданных координатами вершин.

***Указание.*** Применить алгоритм Шеймоса-Хоея. Через вершины заданных многоугольников провести горизонтальные прямые. В результате вся плоскость будет разбита на горизонтальные полосы, а каждый из многоугольников будет разбит на трапеции и треугольники. Эти полосы просматриваются сверху вниз, и в каждой полосе вычисляется пересечение соответствующих трапеций (см. [11]).

1. (Триангуляция простого многоугольника.) Многоугольник, заданный вершинами (*P0* ,*P1* ,…,*Pn-1* ) называется *простым*, если его стороны, кроме соседних, не имеют общих точек. *Триангуляцией* многоугольника называется его разбиение на треугольники, вершины которых принадлежат множеству {*P0* , *P1* , … , *Pn-1* }. Разработать алгоритм и написать программу, строящую триангуляцию простого многоугольника.

***Указание.*** Решение этой задачи описано в книге Л. Аммерала [1]. Сначала выбирается порядок обхода вершин *P0 , P1 ,… ,* *P n* = *P 0*, при котором многоугольник находится слева от отрезков границы. Затем рассматриваются треугольники *Pi-1 Pi Pi+1*, ориентированные против часовой стрелки. Среди них выбирается треугольник *P i-1P iPi+1*, для которого длина отрезка *Pi-1Pi+1* минимальна. Далее производится переход к разбиению оставшегося многоугольника *P0P1…Pi-1 Pi+1…Pn-1* на треугольники. Реализация алгоритма приведена в [1, с. 60-65].

1. (Теорема о картинной галерее.) Пусть простой многоугольник задан вершинами (*P0* ,*P1* ,…,*Pn-1* ) . Согласно теореме Жордана замкнутая ломаная, состоящая из отрезков *P0P1* , *P1P2* , …, *Pn-1P0 ,*разбивает плоскость на две связные области. Одна из этих областей ограничена. Точки, принадлежащие этой ограниченной области, вместе с точками замкнутой ломаной, называются *точкам многоугольника.* Точка *N* многоугольника называется *видимой* из точки *M*, если отрезок *MN* состоит из точек многоугольника. Доказать, что существуют  точек многоугольника, из которых видны все точки многоугольника.

***Указание.*** Доказательство приведено в книге Рурка [18]. Сначала производится триангуляция простого многоугольника. Рассматривается граф, вершинами которого служат точки *P0 , P1 , … ,* *P n-1*, а ребрами – стороны треугольников, разбивающих многоугольник. Легко видеть, что вершины этого графа можно раскрасить в три цвета, таким образом, что любые две вершины, имеющие общее ребро, будут раскрашены в различные цвета. Далее применяется принцип k карманов, согласно котрому, если n предметов положить в k ящиков, то в одном из ящиков будет не более  предметов. Из этого принципа вытекает, что существует цвет, в который будут раскрашены не более, чем  вершин. Эти вершины будут составлять множество точек, из которых видны все точки многоугольника.

1. Написать программу, строящую выпуклую оболочку множества точек методом Эндрю.