MAC 5747 Geometria Computacional

Primitivas

São Paulo, 28 de março de 2000

Sumário

1. Introdução . . . 1
2. Representação de um ponto . . . 1
3. Representação de um polígono . . . 2
4. Cálculo da área de um triângulo . . . 2
5. Cálculo da área de um polígono . . . 2
6. Predicado *Left* . . . 2
7. Predicado *Lefton* . . . 3
8. Predicado *Collinear* . . . 3
9. Predicado *IntersecProp* (intersecção própria) . . . 3
10. Predicado *Between* . . . 3
11. Predicado *Intersect* . . . 3
12. Predicado *Diagonalie* . . . 4
13. Predicado *InCone* . . . 4
14. Primitiva *Diagonal* . . . 4
15. Exercícios . . . 5

1. Introdução

A seguir encontram-se implementações em `C` de algumas operações primitivas que serão muito úteis em diversos algoritmos geométricos que veremos nesta disciplina (teste de intersecção, triangularização de polígonos, partição de polígonos, fecho convexo, ...). Os códigos em `C` forma escritos por Joseph O'Rourke e fazem parte do livro *Computational Geometry in C*, Cambridge University Press, First Edition, 1994.

2. Representação de um ponto

Todos os pontos serão representados por um vetor de dimensão apropriada (a dimensão usada no código abaixo é 2) ao invés de usarmos registros. A representação de pontos através de vetores é mais fácil de ser generalizada (podemos fazer um *loops* que percorre todos os componentes do ponto através de um simples **for**).

```
#define X   0
#define Y   1
#define DIM  2      /* dimensão do espaço */
  typedef enum {
    FALSE, TRUE
  } bool;
  typedef int tPointi[DIM];      /* tipo ponto inteiro */
  typedef double tPointd[DIM];      /* tipo ponto real */
```

3. Representação de um polígono

Um polígono pode ser representado por um vetor de pontos ou uma lista ligada (linear ou circular, simples ou duplamente ligada) de pontos. Listas ligadas permitem alocação dinâmica de memória bem como inserções e remoções simples em tempo constante (i.e., tempo $O(1)$). Entretanto, no código abaixo usaremos a estrutura de vetor para guardar os pontos de um polígono: isso tornará o código mais claro, pois vetores facilitam a estrutura de *loops* e incremento de índices.

```
#define PMAX  1000      /* número máximo de pontos em um polígono */
  typedef tPointi tPolygoni[PMAX];      /* polígono de pontos inteiros */
```

4. Cálculo da área de um triângulo

A função *Area2* retorna duas vezes o valor da *área sinalizada* $\mathcal{A}(T)$ do triângulo terminado pelos pontos a, b e c, ou seja $T := \triangle(a, b, c)$. Se a, b, c aparecem na fronteira do triângulo nesta ordem ao percorremos a fronteira no sentido anti-horário o valor da área sinalizada $\mathcal{A}(T)$ será positivo, caso contrário será negativo. Lembremos que

$$2\mathcal{A}(T) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix} .$$

```
int Area2(tPointi a, b, c)
{
  return a[X] * b[Y] - a[Y] * b[X] + a[Y] * c[X] - a[X] * c[Y] + b[X] * c[Y] - c[X] * b[Y];
}
```

5. Cálculo da área de um polígono

Retorna o dobro da área sinalizada $\mathcal{A}(P)$ de um polígono P.

```
int AreaPoly2(int n, tPolygoni P)
{
  int i;
  int sum = 0;      /* soma das áreas parciais */
  for (i = 1; i < n - 1; i++) sum += Area2(P[0], P[i], P[i + 1]);
  return sum;
}
```

6. Predicado *Left*

A intersecção de dois segmentos de reta pode ser decidida através do uso do predicado *Left*, que determina se um ponto está à esquerda ou à direita de uma reta orientada. Veremos mais adiante como *Left* será usado para decidirmos se dois segmentos de reta dados se intersectam; pretendemos decidir se os dois segmentos dados se intersectam sem, de fato, cálcular a possível intersecção.

Uma reta orientada é determinada por um vetor $\vec{ab}$, onde a e b são pontos. Se um ponto c está à esquerda dessa reta orientada então a tripla (a, b, c) forma um 'circuito' no sentido anti-horário. Um ponto c está à esquerda da reta orientada determinada pelo vetor $\vec{ab}$ se e somente se a orientação do triângulo $\triangle(a, b, c)$ é positiva.

```
bool Left(tPointi a, b, c)
{
  return Area2(a, b, c) > 0;
}
```

7. Predicado *Lefton*

O quê acontece se o ponto c está sobre a reta orientada determinada por $\vec{ab}$? Neste caso a área do $\triangle(a,b,c)$ é nula. O predicado *LeftOn* devolve verdadeiro se o ponto c estiver à esquerda ou sobre a reta orientada determinada pelo vetor $\vec{ab}$.

bool *LeftOn*(**tPointi** a, b, c)
{
  return *Area2*$(a, b, c) \geq 0$;
}

8. Predicado *Collinear*

Este Predicado é verdadeiro se os pontos a, b e c são colineares, ou seja, se a área do triângulo $\triangle(a,b,c)$ for zero.

bool *Collinear*(**tPointi** a, b, c)
{
  return *Area2*$(a, b, c) \equiv 0$;
}

9. Predicado *IntersecProp* (intersecção própria)

A função *IntersecProp* retorna `TRUE` se o segmento de reta ab intersecta o segmento cd propriamente: existe um único ponto em comum no interior relativo de ambos os segmentos. Para determinar a existência de uma intersecção própria faremos uso o predicado *Left*.

bool *IntersectProp*(**tPointi** a, b, c, d) { /* Elimina casos impróprios */
  if ((*Collinear*$(a,b,c) \vee$ *Collinear*$(a,b,d) \vee$ *Collinear*$(c,d,a) \vee$ *Collinear*(c,d,b)) **return** `FALSE`;
  return *Xor*(*Left*(a,b,c), *Left*$(a,b,d)) \wedge$ *Xor*(*Left*(c,d,a), *Left*$(c,d,b))$; }
    /* Xor faz o ou exclusivo entre os dois argumentos */
  bool *Xor*(**bool** x, **bool** y)
  { /* Os argumentos são negados para garantir que os valores são 0/1 */
    return $\neg x \oplus \neg y$;
  }

10. Predicado *Between*

Retorna `TRUE` se e somente se os pontos a, b e c são colineares e c é um ponto do segmento fechado ab. Se o segmento ab não é vertical então o ponto c está no no segmento ab se e somente se a x-coordenada de c está no intervalo determinado pelas x=coordenadas de a e b. Caso ab seja um segmento vertical o mesmo teste é feito para as y-coordenadas de a, b e c.

bool *Between*(**tPointi** a, **tPointi** b, **tPointi** c)
{ **tPointi** ba, ca;
  if ($\neg$ *Collinear*(a,b,c)) **return** `FALSE`; /* rejeição rápida */
  if ($a[X] \neq b[X]$) /* Se ab não é vertical então verifica se c_x está entre a_x e b_x. */
    return $((a[X] \leq c[X]) \wedge (c[X] \leq b[X])) \vee ((a[X] \geq c[X]) \wedge (c[X] \geq b[X]))$;
  else
    return $((a[Y] \leq c[Y]) \wedge (c[Y] \leq b[Y])) \vee ((a[Y] \geq c[Y]) \wedge (c[Y] \geq b[Y]))$;
}

11. Predicado *Intersect*

Retorna `TRUE` se e somente se o segmento ab e o segmento cd intersectam-se, propriamente ou impropriamente.

```
bool Intersect(tPointi a, b, c, d)
{ if (IntersectProp(a, b, c, d)) return TRUE;
  else if (Between(a, b, c) ∨ Between(a, b, d) ∨ Between(c, d, a) ∨ Between(c, d, b))
          return TRUE;
  else return FALSE;
}
```

12. Predicado *Diagonalie*

Retorna TRUE se v_iv_j é uma diagonal própria interna ou externa do polígono P, ou $\mid i - j \mid = 2$ e v_i e v_j são colineares com o vértice do meio.

```
bool Diagonalie(int i, int j, int n, tPolygoni P)
{
  int k;
  int k1;      /* k + 1 */
  for (k = 0; k < n; k++) {      /* para cada aresta v_k v_{k+1} de P */
    k1 = (k + 1) % n;
    if (¬((k ≡ i) ∨ (k1 ≡ i) ∨ (k ≡ j) ∨ (k1 ≡ j)))      /* Pula arestas incidente a v_i ou v_j */
      if (Intersect(P[i], P[j], P[k], P[k1]))
            return FALSE;
  }
  return TRUE;
}
```

13. Predicado *InCone*

Sejam $v_0, v_1, \ldots, v_{n-1}$ os vértices de um polígono listados na ordem que ocorrem ao percorrermos a fronteira do poligono no sentido anti-horária e suponha que $s = v_iv_j$, $i \neq j$, não intesecta arestas do polígono P que não sejam incidentes a v_i nem a v_j. Se s é interior ao polígono P nas vizinhanças de v_i e v_j então s é interior ao polígono P. Assim o teste para decidir se um segmento s é interior a um polígono pode ser divido em: (i) verificar se $s = v_iv_j$ intersecta arestas de P; e (ii) testar se o segmento s é interior a P na vizinhança de v_i e v_j. Podemos testar se s está no interior de P na vizinhança de v_i ou v_j em tempo constante, como pode ser visto abaixo.

```
bool InCone(int i, int j, int n, tPolygoni P)
{
  int i1;      /* i + 1 */
  int in1;     /* i - 1 */
  i1 = (i + 1) % n;
  in1 = (i + n - 1) % n;
  if (LeftOn(P[in1], P[i], P[i1]))
       /* se P[i] é um vértice convexo (v_{i+1} está à esquerda ou sobre o segmento v_{i-1}v_i) */
    return Left(P[i], P[j], P[in1]) ∧ Left(P[j], P[i], P[i1]);
  else      /* Assuma v_{i-1}, v_i, v_{i+1} não colineares. P[i] é vértice reflexo. */
    return ¬(Lefton(P[i], P[j], P[i1]) ∧ LeftOn(P[j], P[i], P[in1]));
}
```

14. Primitiva *Diagonal*

Returna TRUE se e somente se v_iv_j é uma diagonal do polígono P.

```
bool Diagonal(int i, int j, int n, tPolygoni P)
{
  return InCone(i, j, n, P) ∧ Diagonalie(i, j, n, P);
}
```

15. Exercícios

O exercícios a seguir forma cópiados do livro *Computational Geometry in C* por J. O'Rourke.

1. Descreva em C a implementação de um polígono através de listas circulares duplamente ligadas. Reescreva as funções acima usando a sua implementação.

2. (Integer Overflow). Em um computador onde o tipo **int** é restrito a $\pm 2^{31}$, qual o valor máximo que as coordenadas de a, b e c podem ter para que evitemos *integer overflow* na funçãp *Area2*.

3. Qual a redundância existente no código da função *IntersectProp*? Como remover esta redundância?

4. Descreva o que predicado *IntersectProp* faz se o comando **if** é removido. Note que depois desta remoção o predicado *Intersect* ainda funciona.

5. (Ineficiência de *Intersect*). Simule *Intersect* e determine o número máximo de chamadas à *Area2* que podem ser feitas. Apresente uma nova implementação que evita chamadas duplicadas.

6. (*InCone*). Prove que ab está no cone no vértice a se e somente se no máximo uma das seguintes chamadas de *Left* é false: *Left*(a, a_+, b), *Left*(a, b, a_-), *Left*(a, a_-, a_+), onde a_+ e a_- são os vértices sucessor e predecessor de a na fronteira de P, respectivamente.

7. (*Diagonal*) Mostre que uma das chamadas de *InCone* e *Diagonal* pode ser removida ser alterar o resultado.

Índice Remissivo

Aqui vai uma lista de identificadores usados e onde eles aparecem. Entradas grifadas indicam os lugares de definições. Mensagens de erros também são mostradas.

a: 4, 6, 7, 8, 9, 10, 11.
ab: 10.
AreaPoly2: 5.
Area2: 4, 5, 6, 7, 8, 15.
b: 10.
ba: 10.
Between: 10, 11.
bool: 2, 6, 7, 8, 9, 10, 11, 12, 13, 14.
c: 10.
ca: 10.
Collinear: 8, 9, 10.
Diagonal: 14, 15.
Diagonalie: 12, 14.
`DIM`: 2.
`FALSE`: 2, 9, 10, 11, 12.
i: 5, 12, 13, 14.
InCone: 13, 14, 15.
IntersecProp: 9.
Intersect: 11, 12, 15.
IntersectProp: 9, 11, 15.
in1: 13.
i1: 13.
j: 12, 13, 14.
k: 12.
k1: 12.
Left: 6, 9, 13, 15.
LeftOn: 7, 13.
Lefton: 13.
n: 5, 12, 13, 14.
P: 5, 12, 13, 14.
`PMAX`: 3.
sum: 5.
tPointd: 2.
tPointi: 2, 3, 4, 6, 7, 8, 9, 10, 11.
tPolygoni: 3, 5, 12, 13, 14.
`TRUE`: 2, 9, 11, 12.
X: 2.
x: 9.
Xor: 9.
Y: 2.
y: 9.