UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
http://www.icmc.usp.br
Departamento de Ciências de Computação

SCC-210 - Capítulo 10
Geometria

João Luís Garcia Rosa[1]

[1] Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
http://www.icmc.usp.br/~joaoluis

2010

Sumário

1. Geometria
   - Retas
   - Ângulos
2. Trigonometria
   - Ângulos retos
   - Resolução de triângulos
   - Círculos
3. Geometria Computacional
   - Segmentos de reta e interseção
   - Polígonos e computações angulares
   - Triangulação: algoritmos e problemas relacionados

Sumário

1. Geometria
   - Retas
   - Ângulos
2. Trigonometria
   - Ângulos retos
   - Resolução de triângulos
   - Círculos
3. Geometria Computacional
   - Segmentos de reta e interseção
   - Polígonos e computações angulares
   - Triangulação: algoritmos e problemas relacionados

Geometria

- Academia de Platão: “Não deixem nenhum ignorante em geometria entrar aqui!”
- Nas competições: pelo menos um problema de geometria.
- Geometria é uma disciplina visual: desenhar e estudar!
- Parte da dificuldade da programação geométrica: operações fáceis com o lápis podem ser difíceis programar.

Retas e representação

- A menor distância entre dois pontos é uma *linha reta*.
- Retas têm comprimento infinito em ambas as direções.
- *Segmentos* de reta têm comprimento finito.
- *Representação*: retas podem ser representadas como pares de pontos ou como equações.
- Toda reta l é completamente representada como o par de pontos (x_1, y_1) e (x_2, y_2).
- Pode também ser descrita por equações como $y = mx + b$, onde m é o *coeficiente angular* da reta e b é o *cruzamento com* y, ou seja, o único ponto $(0, b)$ onde a reta cruza o eixo do y.
- A reta l tem coeficiente angular $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_2}{x_1 - x_2}$ e cruzamento $b = y_1 - mx_1$.

Retas

- Retas verticais não podem ser descritas por tais equações, pois ocorrerá a divisão por zero ($\Delta x = 0$)
- A equação $x = c$ denota uma reta vertical que cruza o eixo do x no ponto $(c, 0)$.
- Este caso especial, ou *degeneração*, requer atenção extra na programação geométrica.
- Usa-se a fórmula geral $ax + by + c = 0$ porque ela cobre todas as retas no plano.

```
typedef struct {
    double a;          /* x-coefficient */
    double b;          /* y-coefficient */
    double c;          /* constant term */
} line;

typedef double point[2];
```

Retas

- Multiplicando os coeficientes por constantes não nulas resulta em uma representação alternativa para qualquer reta.
- Estabelece-se uma representação canônica, fazendo o coeficiente de y igual a 1 se ele for não zero.
- Senão, faz-se o coeficiente de x igual a 1.

Retas

```
points_to_line(point p1, point p2, line *l)
{
    if (p1[X] == p2[X])
    {
       l->a = 1;
       l->b = 0;
       l->c = -p1[X];
    }
    else
    {
       l->b = 1;
       l->a = -(p1[Y]-p2[Y])/(p1[X]-p2[X]);
       l->c = -(l->a * p1[X]) - (l->b * p1[Y]);
    }
}


point_and_slope_to_line(point p, double m, line *l)
{
    l->a = -m;
    l->b = 1;
    l->c = -((l->a*p[X]) + (l->b*p[Y]));
}
```

Interseção

- Duas retas distintas têm um *ponto de interseção*, a não ser que sejam *paralelas*, que não têm nenhum.
- Retas paralelas têm o mesmo coeficiente angular mas diferentes cruzamentos e por definição nunca se cruzam.

```
#define EPSILON 0.000001 /* a quantity small enough to be zero */

bool parallelQ(line l1, line l2)
{
    return ( (fabs(l1.a-l2.a) <= EPSILON) && (fabs(l1.b-l2.b) <= EPSILON) );
}

bool same_lineQ(line l1, line l2)
{
    return ( parallelQ(l1,l2) && (fabs(l1.c-l2.c) <= EPSILON) );
}
```

Interseção

- Um ponto (x', y') está situado em uma reta l: $y = mx + b$ se ao substituirmos x da fórmula por x', obtemos y'.
- O ponto de interseção das retas l_1: $y = m_1 x + b_1$ e l_2: $y = m_2 x + b_2$ é o ponto onde elas são iguais, ou seja,

$$x = \frac{b_2 - b_1}{m_1 - m_2}, \qquad y = m_1 \frac{b_2 - b_1}{m_1 - m_2} + b_1 \tag{1}$$

Interseção

$$l_1 = a_1x + b_1y + c_1 = 0 \therefore x = \frac{-b_1y - c_1}{a_1} \tag{2}$$

$$l_2 = a_2x + b_2y + c_2 = 0 \therefore y = \frac{-a_2x - c_2}{b_2} \tag{3}$$

$$x = \frac{-b_1(\frac{-a_2x-c_2}{b_2}) - c_1}{a_1} \tag{4}$$

$$xa_1 = \frac{a_2b_1x + b_1c_2 - c_1b_2}{b_2} \tag{5}$$

$$x(a_1b_2 - a_2b_1) = b_1c_2 - c_1b_2 \tag{6}$$

$$x = \frac{b_1c_2 - c_1b_2}{a_1b_2 - a_2b_1} = \frac{b_2c_1 - b_1c_2}{a_2b_1 - a_1b_2} \tag{7}$$

Interseção

```
intersection_point(line l1, line l2, point p)
{
    if (same_lineQ(l1,l2))
    {
      printf("Warning:  Identical lines, all points intersect.\n");
      p[X] = p[Y] = 0.0;
      return;
    }

    if (parallelQ(l1,l2) == TRUE)
    {
      printf("Error:  Distinct parallel lines do not intersect.\n");
      return;
    }

    p[X] = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);

    if (fabs(l1.b) > EPSILON) /* test for vertical line */
      p[Y] = - (l1.a * (p[X]) + l1.c) / l1.b;
    else
      p[Y] = - (l2.a * (p[X]) + l2.c) / l2.b;
}
```

Sumário

1. Geometria
   - Retas
   - Ângulos
2. Trigonometria
   - Ângulos retos
   - Resolução de triângulos
   - Círculos
3. Geometria Computacional
   - Segmentos de reta e interseção
   - Polígonos e computações angulares
   - Triangulação: algoritmos e problemas relacionados

Ângulos

- Quaisquer duas retas não-paralelas intersecionam cada uma num dado ângulo.
- Retas $l_1 : a_1x + b_1y + c_1 = 0$ e $l_2 : a_2x + b_2y + c_2 = 0$, escritas na forma geral, intersecionam no ângulo θ dado por:

$$tg\ \theta = \frac{a_1b_2 - a_2b_1}{a_1a_2 + b_1b_2} \tag{8}$$

- Para retas escritas na forma coeficiente angular-cruzamento:

$$tg\ \theta = \frac{m_2 - m_1}{m_1m_2 + 1} \tag{9}$$

- Duas linhas são *perpendiculares* se elas se cruzam num ângulo reto, ou seja, se o produto de seus coeficientes angulares for -1. Por exemplo, $y = x$ e $y = -x$.

Ângulos

- Sejam α_1 e α_2 os ângulos formados pelas inclinações das retas l_1 e l_2 com o eixo do x, respectivamente.
- Seja $\theta = \alpha_2 - \alpha_1$ o ângulo entre as retas l_1 e l_2:

$$tg\ \theta = tg\ (\alpha_2 - \alpha_1) \tag{10}$$

- Mas

$$tg\ (a-b) = \frac{tg\ (a) - tg\ (b)}{1 + tg\ (a)\ tg\ (b)} \tag{11}$$

pois

$$tg\ (a-b) = \frac{sen\ (a-b)}{cos\ (a-b)} = \frac{sen\ (a)\ cos\ (b) - cos\ (a)\ sen\ (b)}{cos\ (a)\ cos\ (b) + sen\ (a)\ sen\ (b)} \tag{12}$$

Ângulos

$$tg(a-b)=\frac{\frac{sen(a)\ cos(b)}{cos(a)\ cos(b)}-\frac{cos(a)\ sen(b)}{cos(a)\ cos(b)}}{\frac{cos(a)\ cos(b)}{cos(a)\ cos(b)}+\frac{sen(a)\ sen(b)}{cos(a)\ cos(b)}}=\frac{\frac{sen(a)}{cos(a)}-\frac{sen(b)}{cos(b)}}{1+\frac{sen(a)}{cos(a)}\frac{sen(b)}{cos(b)}} \quad (13)$$

$$tg\theta=\frac{tg\alpha_2-tg\alpha_1}{1+tg\alpha_2\ tg\alpha_1}=\frac{m_2-m_1}{1+m_2m_1} \quad (14)$$

pois $m = tg\alpha$.

Ponto mais próximo

- Um subproblema muito útil é identificar o ponto na reta *l* que é mais perto a um dado ponto *p*.
- Este ponto mais perto situa-se na reta que é perpendicular a *l*:

```
closest_point(point p_in, line l, point p_c)
{
    line perp; /* perpendicular to l through (x,y) */

    if (fabs(l.b) <= EPSILON)
    { /* vertical line */
      p_c[X] = -(l.c);
      p_c[Y] = p_in[Y];
      return;
    }

    if (fabs(l.a) <= EPSILON)
    { /* horizontal line */
      p_c[X] = p_in[X];
      p_c[Y] = -(l.c);
      return;
    }

    point_and_slope_to_line(p_in,1/l.a,&perp); /* non-degenerate line */
    intersection_point(l,perp,p_c);
}
```

Sumário

1 Geometria
- Retas
- Ângulos

2 Trigonometria
- Ângulos retos
- Resolução de triângulos
- Círculos

3 Geometria Computacional
- Segmentos de reta e interseção
- Polígonos e computações angulares
- Triangulação: algoritmos e problemas relacionados

Triângulos e Trigonometria

- *Raios* são meias-retas que originam de algum vértice v, chamado de *origem*.
- Todo raio é completamento descrito por uma equação de reta, origem e direção ou origem e um outro ponto no raio.
- Um *ângulo* é a união de dois raios compartilhando um mesmo ponto final.
- *Trigonometria* é o ramo da matemática que trata dos ângulos e de suas medidas.

Ângulos retos

- Duas unidades para medir ângulos: *radianos* e *graus*.
- De 0 a 2π radianos ou de 0 a 360 graus.
- Radianos é melhor computacionalmente, pois as bibliotecas de trigonometria usam radianos.
- Um ângulo *reto* mede 90° ou $\frac{\pi}{2}$ radianos.
- Ângulos retos se formam na interseção de duas retas perpendiculares.
- Eixos de coordenadas retilíneas: dividem o espaço de 360° ou 2π radianos em quatro ângulos retos.

Teorema de Pitágoras e Funções trigonométricas

- Cada par de raios com um ponto final em comum define dois ângulos, um *ângulo interno* de *a* radianos e um *ângulo externo* de $2\pi - a$ radianos.
- Um triângulo é chamado de *retângulo* se contém um ângulo interno reto.
- Triângulos retângulos são fáceis de se trabalhar por causa do *teorema de Pitágoras*.
- As funções trigonométricas *seno* e *cosseno* são definidas como as coordenadas-*x* e *y* de pontos no círculo unitário centrado em (0, 0).
- Portanto os valores do seno e cosseno variam de -1 a 1.
- Além disso, as duas funções são realmente a mesma coisa, já que $cos(\theta) = sen(\theta + \frac{\pi}{2})$.

Outra função trigonométrica

- Uma terceira importante função trigonométrica é a *tangente*, definida como a razão do seno e cosseno.
- Portanto, $tg(\theta) = \frac{sen(\theta)}{cos(\theta)}$, que é bem definida, exceto quando $cos(\theta) = 0$ em $\theta = \frac{\pi}{2}$ e $\theta = \frac{3\pi}{2}$.
- Estas funções são importantes para permitir que sejam relacionados os comprimentos de quaisquer dois lados de um triângulo retângulo T com os ângulos não retos de T.
- Lembre-se de que a hipotenusa é o lado maior em T, oposto ao ângulo reto.
- Os outros dois lados em T são lados oposto e adjacente em relação a um ângulo a (não reto).

Funções trigonométricas

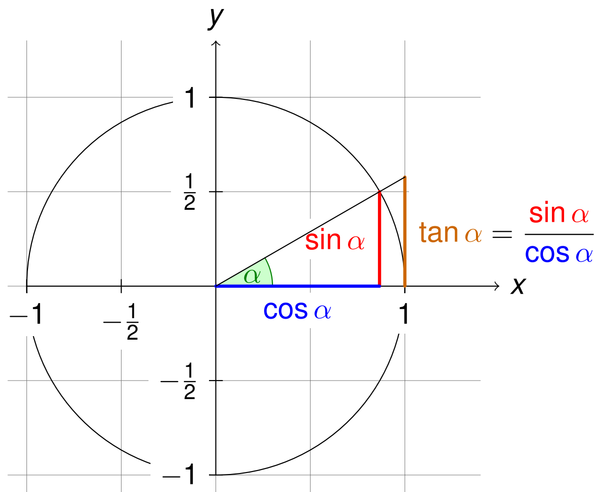
$$cos(a) = \frac{|\ adjacente\ |}{|\ hipotenusa\ |} \tag{15}$$

$$sen(a) = \frac{|\ oposto\ |}{|\ hipotenusa\ |} \tag{16}$$

$$tg(a) = \frac{|\ oposto\ |}{|\ adjacente\ |} \tag{17}$$

- Funções inversas: *arccos*, *arcsen* e *arctg*.
- Funções trigonométricas são computadas usando expansões de séries de Taylor: bibliotecas já as incluem!
- Funções trigonométricas tendem a ser instáveis: $\theta \approx arcsen(sen(\theta))$, para ângulos grandes ou pequenos.

Funções trigonométricas [2]



O ângulo α é 30° no exemplo ($\pi/6$ em radianos). O seno de α, que é a altura da linha vermelha, é

$$\sin\alpha = 1/2.$$

Pelo teorema de Pitágoras tem-se $\cos^2\alpha + \sin^2\alpha = 1$. Portanto, o comprimento da linha azul, que é o cosseno de α, deve ser

$$\cos\alpha = \sqrt{1 - 1/4} = \tfrac{1}{2}\sqrt{3}.$$

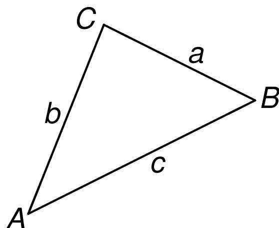
Isto mostra que $\tan\alpha$, que é a altura da linha laranja, é

$$\tan\alpha = \frac{\sin\alpha}{\cos\alpha} = 1/\sqrt{3}.$$

Lei dos Senos

- Duas fórmulas trigonométricas poderosas habilitam a computação de propriedades importantes dos triângulos.
- A *Lei dos Senos* provê o relacionamento entre lados e ângulos em qualquer triângulo.
- Para ângulos A, B e C, e lados opostos a, b e c:

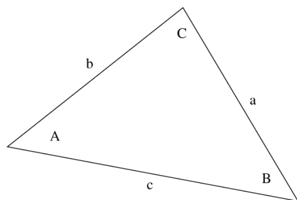
$$\frac{a}{\text{sen } A} = \frac{b}{\text{sen } B} = \frac{c}{\text{sen } C} \tag{18}$$

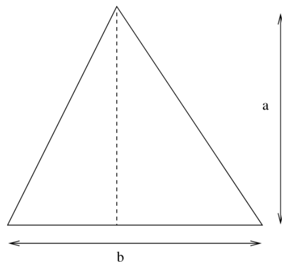


Lei dos Cossenos

- A *Lei dos Cossenos* é uma generalização do teorema de Pitágoras para ângulos não retos.
- Para qualquer triângulo com ângulos A, B e C, e lados opostos a, b e c:

$$a^2 = b^2 + c^2 - 2bc \cos A \qquad (19)$$





Sumário

1. Geometria
   - Retas
   - Ângulos
2. Trigonometria
   - Ângulos retos
   - Resolução de triângulos
   - Círculos
3. Geometria Computacional
   - Segmentos de reta e interseção
   - Polígonos e computações angulares
   - Triangulação: algoritmos e problemas relacionados

Resolver triângulos

- Resolver triângulos é a arte de derivar os ângulos e comprimentos de lados desconhecidos de um dado triângulo. Duas categorias:
  1. *Dados dois ângulos e um lado, ache o resto*: achar o terceiro ângulo é fácil, pois os três ângulos devem somar $180^\circ = \pi$ radianos. A Lei dos Senos permite achar os comprimentos dos lados faltantes.
  2. *Dados dois lados e um ângulo, ache o resto*: Se o ângulo fica entre os dois lados, a Lei dos Cossenos permite que se ache o comprimento do terceiro lado. A Lei dos Senos permite que se resolva os ângulos desconhecidos. De outra forma, pode-se usar a Lei dos Senos e a propriedade da soma dos ângulos para determinar todos os ângulos e então a Lei dos Senos para obter a comprimento do terceiro lado.

Resolver triângulos

- A área $A(T)$ de um triângulo T é dada por $A(T) = (\frac{1}{2})ab$, onde a é a altura e b é a base do triângulo.
- A base é qualquer um dos lados enquanto que a altura é distância do terceiro vértice a esta base.
- A altura pode ser calculada via trigonometria ou teorema de Pitágoras, dependendo do que se conhece sobre o triângulo.
- Uma outra abordagem para computar a área de um triângulo é a partir da sua representação de coordenadas.

Resolver triângulos

- Usando álgebra linear e determinantes, pode-se mostrar que a área $A(T)$ do triângulo $T = (a, b, c)$ é:

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

```
double signed_triangle_area(point a, point b, point 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]) / 2.0 );
}

double triangle_area(point a, point b, point c)
{
    return( fabs(signed_triangle_area(a,b,c)) );
}
```

Sumário

1. Geometria
   - Retas
   - Ângulos
2. Trigonometria
   - Ângulos retos
   - Resolução de triângulos
   - Círculos
3. Geometria Computacional
   - Segmentos de reta e interseção
   - Polígonos e computações angulares
   - Triangulação: algoritmos e problemas relacionados

Representação de círculos

- Um *círculo* é definido como o conjunto de pontos a uma dada distância (ou *raio*) de seu *centro*, (x_c, y_c).
- Um *disco* é um círculo mais seu interior, ou seja, o conjunto de pontos a uma distância no máximo r de seu centro.
- *Representação*: Um círculo pode ser representado de duas formas básicas, ou como triplas de pontos fronteiriços ou pelo seu centro/raio.
- Para muitas aplicações, a representação centro/raio é mais conveniente:

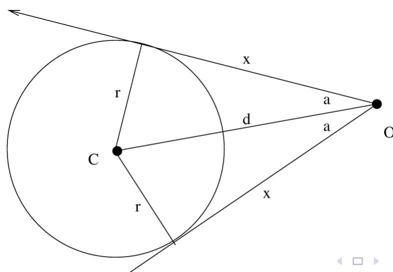
```
typedef struct {
   point c;    /* center of circle */
   double r;    /* radius of circle */
} circle;
```

Equação de um círculo

- A equação de um círculo segue diretamente de sua representação centro/raio.
- Como a distância entre dois pontos é definida como $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, a equação de um círculo de raio r é $r = \sqrt{(x - x_c)^2 + (y - y_c)^2}$ ou, equivalentemente, $r^2 = (x - x_c)^2 + (y - y_c)^2$.
- *Circunferência e área*: Muitas quantidades importantes associadas com círculos são fáceis de computar.
- A área A e a circunferência (comprimento da fronteira) C de um círculo dependem da constante mágica $\pi = 3.1415926$.
- $A = \pi r^2$ e $C = 2\pi r$. O diâmetro é $2r$.

Tangente

- *Tangente* é uma reta x que interseciona o limite de um círculo c em exatamente um ponto.
- Note que o triângulo com lados r, d e x é um triângulo retângulo: pode-se calcular x pelo teorema de Pitágoras.
- De x, pode-se calcular o ponto tangente ou o ângulo a.
- A distância d de O ao centro C é computada usando a fórmula da distância.

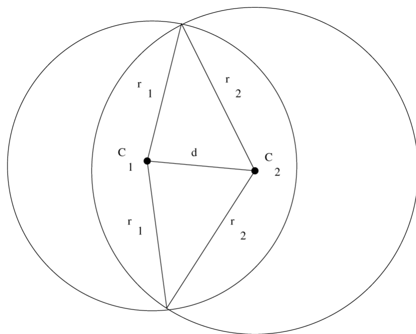


Círculos interagentes

- Dois círculos c_1 e c_2 de raios distintos r_1 e r_2 podem interagir de várias formas.
- Os círculos intersecionarão se e apenas se a distância entre seus centros for no máximo $r_1 + r_2$.
- O círculo menor (por exemplo, c_1) estará completamente contido em c_2 se e apenas se a distância entre seus centros mais r_1 é no máximo r_2.
- O caso restante é quando c_1 e c_2 intersecionam-se em dois pontos.

Círculos interagentes

- Os pontos de interseção formam triângulos com os dois centros cujos comprimentos dos lados são determinados (r_1, r_2 e a distância d entre os centros), tal que ângulos e coordenadas podem ser computados quando necessário:



Sumário

1. Geometria
   - Retas
   - Ângulos
2. Trigonometria
   - Ângulos retos
   - Resolução de triângulos
   - Círculos
3. Geometria Computacional
   - Segmentos de reta e interseção
   - Polígonos e computações angulares
   - Triangulação: algoritmos e problemas relacionados

Introdução

- Computação geométrica tem se tornado muito importante em aplicações como computação gráfica, robótica e projeto auxiliado por computador, porque a forma é uma propriedade inerente de objetos reais.
- Mas, a maioria dos objetos do mundo real não são feitos de retas que vão ao infinito.
- Na verdade, os programas de computador representam a geometria como arranjos de segmentos de reta.
- Curvas ou formas fechadas podem ser representadas por coleções de segmentos de reta ou *polígonos*.
- Portanto, geometria computacional pode ser definida como a geometria de segmentos de reta e polígonos.

Segmentos de reta e interseção

- Um *segmento de reta s* é a porção de uma reta *l* que se situa entre dois pontos dados, inclusive.
- Portanto, segmentos de reta são naturalmente representados por pares de pontos finais:

```
typedef struct {
    point p1,p2;    /* endpoints of line segment */
} segment;
```

- A mais importante primitiva geométrica para segmentos que testa se um dado par se interseciona provou-se complicada por causa de casos especiais.
- Dois segmentos podem se situar em retas paralelas, significando que eles não se intersecionam.
- Um segmento pode intersecionar o ponto final do outro, ou os dois segmentos podem se situar em cima um do outro, de forma que eles intersecionam um segmento, em vez de um único ponto.

Segmentos de reta e interseção

- Este problema de casos especiais geométricos, ou *degeneração*, complica muito o problema de escrever implementações robustas de algoritmos de geometria computacional.
- Leia com atenção a especificação do problema para checar se não haverá retas paralelas ou segmentos sobrepostos.
- Sem essa garantia, deve-se programar defensivamente e tratar cada caso.
- A maneira correta de tratar a degeneração é basear toda a computação em um pequeno número de primitivas geométricas construídas cuidadosamente.

Segmentos de reta e interseção

```
bool segments_intersect (segment s1, segment s2)
{
   line l1,l2;    /* lines containing the input segments */
   point p;       /* intersection point */

   points_to_line(s1.p1,s1.p2,&l1);
   points_to_line(s2.p1,s2.p2,&l2);

   if (same_lineQ(l1,l2))    /* overlapping or disjoint segments */
     return( point_in_box(s1.p1,s2.p1,s2.p2) ||
       point_in_box(s1.p2,s2.p1,s2.p2) ||
       point_in_box(s2.p1,s1.p1,s1.p2) ||
       point_in_box(s2.p2,s1.p1,s1.p2) );

   if (parallelQ(l1,l2))    return(FALSE);

   intersection_point(l1,l2,p);

   return( point_in_box(p,s1.p1,s1.p2) && point_in_box(p,s2.p1,s2.p2) );
}
```

Segmentos de reta e interseção

- Usa-se as rotinas de interseção de linha para achar um ponto de interseção se existir.
- Caso positivo, a questão é se este ponto fica na região definida pelos segmentos de linha.
- Isto é testado verificando se o ponto de interseção se situa na caixa que envolve cada segmento de linha, que é definida pelos pontos finais de cada segmento:

```
bool point_in_box(point p, point b1, point b2)
{
   return( (p[X] >= min(b1[X],b2[X])) && (p[X] <= max(b1[X],b2[X]))
     && (p[Y] >= min(b1[Y],b2[Y])) && (p[Y] <= max(b1[Y],b2[Y])) );
}
```

- Interseção de segmentos também pode ser testada usando uma primitiva para checar se três pontos ordenados estão situados em uma direção anti-horária.

Sumário

1 Geometria
- Retas
- Ângulos

2 Trigonometria
- Ângulos retos
- Resolução de triângulos
- Círculos

3 Geometria Computacional
- Segmentos de reta e interseção
- Polígonos e computações angulares
- Triangulação: algoritmos e problemas relacionados

Introdução

- *Polígonos* são cadeias fechadas de segmentos de retas que não se intersecionam.
- Fechadas significa que o primeiro vértice da cadeia é igual ao último.
- A não interseção significa que pares de segmentos encontram-se apenas nos pontos finais.
- São as estruturas básicas para descrever formas no plano.
- Ao invés de listar explicitamente os segmentos (ou lados) do polígono, pode-se implicitamente representá-los listando os n vértices em ordem em volta do polígono.
- Portanto um segmento existe entre o i-ésimo e o $(i+1)$-ésimo pontos na cadeia para $0 \leq i \leq n-1$.

Polígono convexo

- Estes índices são calculados *mod n* para assegurar que há uma aresta entre o primeiro e o último ponto:

```
#define MAXPOLY 200 /* maximum number of points in a polygon */

typedef struct {
    int n;    /* number of points in polygon */
    point p[MAXPOLY]; /* array of points in polygon */
} polygon;
```

- Um polígono *P* é *convexo* se qualquer segmento de reta definido por dois pontos dentro de *P* se situa inteiramente dentro de *P*, isto é, se não houver "buracos" ou "caroços" tal que o segmento possa sair e re-entrar em *P*.
- Isto implica que todos os ângulos internos em um polígono convexo devem ser agudos; isto é, no máximo 180° ou π radianos.

Computação de ângulos

- Computar o ângulo definido entre três pontos ordenados é um problema complicado.
- Pode-se evitar a necessidade de conhecer os ângulos reais na maioria dos algoritmos geométricos usando o predicado anti-horário `ccw(a,b,c)`.
- Esta rotina testa se o ponto c fica à direita da reta direcionada que vai do ponto a ao ponto b.
- Caso positivo, o ângulo formado de a a c de uma maneira anti-horária em volta de b é agudo, daí o nome do predicado.
- Caso negativo, o ponto ou fica à esquerda de $\vec{ab}$ ou os três pontos são colineares.

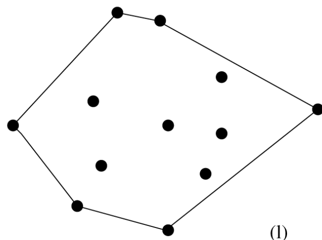
Computação de ângulos

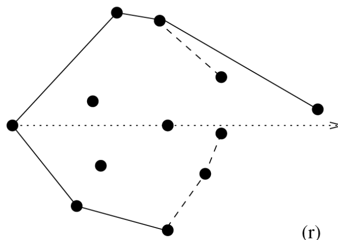
- Estes predicados podem ser computados usando a função `signed_triangle_area()`.
- Ocorre área negativa se o ponto c está a esquerda de $\vec{ab}$.
- Área zero ocorre se os três pontos são colineares.

```
bool ccw(point a, point b, point c)
{
  double signed_triangle_area();

  return (signed_triangle_area(a,b,c) > EPSILON);
}

bool cw(point a, point b, point c)
{
  double signed_triangle_area();

  return (signed_triangle_area(a,b,c) < - EPSILON);
}

bool collinear(point a, point b, point c)
{
  double signed_triangle_area();

  return (fabs(signed_triangle_area(a,b,c)) <= EPSILON);
}
```

Envoltória convexa

- A envoltória convexa é para a geometria computacional o que a ordenação é para outros problemas algorítmicos, um primeiro passo para aplicar a dados não estruturados, tal que pode-se fazer mais coisas interessantes a partir dele.
- A *envoltória convexa* $C(S)$ de um conjunto de pontos S é o menor polígono convexo que contém S (esquerda(l)):





Envoltória convexa

- Há quase tantos algoritmos diferentes para envoltória convexa quanto há para ordenação.
- O algoritmo de Graham ordena os pontos em ordem angular ou da esquerda para a direita e depois insere os pontos na envoltória nesta ordem.
- Pontos da envoltória prévios tornados obsoletos pela última inserção são apagados.
- A implementação a seguir é baseada na versão de Gries e Stojmenovi'c para o algoritmo de Graham, que ordena os vértices por *ângulo* em volta do ponto mais abaixo e mais à esquerda.
- Observe que ambos os pontos mais à esquerda e mais abaixo *devem* estar na envoltória, porque eles não podem estar situados dentro de algum outro triângulo de pontos.

Envoltória convexa

- O *loop* principal do algoritmo insere os pontos em ordem angular crescente em volta deste ponto inicial.
- Por causa desta ordenação, o ponto recém inserido deve se situar na envoltória dos pontos “inseridos mais longe”.
- Esta nova inserção pode formar um triângulo contendo os pontos anteriores da envoltória que agora devem ser apagados.
- Estes pontos ficarão no final da cadeia como as inserções sobreviventes mais recentes.

Envoltória convexa

- O critério de apagamento é se a nova inserção forma um ângulo obtuso com os dois últimos pontos na cadeia (apenas ângulos agudos aparecem em polígonos convexos).
- Se o ângulo é muito grande, o último ponto na cadeia tem de sair.
- Continua-se até que um ângulo pequeno suficiente seja criado ou que acabem os pontos.
- Usa-se o predicado `ccw()` para testar se o ângulo é grande demais:

Envoltória convexa

```
point first_point;    /* first hull point */
convex_hull(point in[], int n, polygon *hull)
{
  int i;    /* input counter */
  int top;    /* current hull size */
  bool smaller_angle();
  if (n <= 3) {     /* all points on hull!  */
    for (i=0; i<n; i++)
      copy_point(in[i],hull->p[i]);
    hull->n = n;
    return;   }
  sort_and_remove_duplicates(in,&n);
  copy_point(in[0],&first_point);
  qsort(&in[1], n-1, sizeof(point), smaller_angle);
  copy_point(first_point,hull->p[0]);
  copy_point(in[1],hull->p[1]);
  copy_point(first_point,in[n]);   /* sentinel to avoid special case */
  top = 1;
  i = 2;
  while (i <= n) {
    if (!ccw(hull->p[top-1], hull->p[top], in[i]))
      top = top-1;   /* top not on hull */
    else {
      top = top+1;
      copy_point(in[i],hull->p[top]);
      i = i+1; } }
  hull->n = top;
}
```

Envoltória convexa

- Esta implementação naturalmente evita a *maioria* dos problemas de degeneração.
- Um problema é quando três ou mais pontos de entrada são colineares, principalmente quando um destes pontos é o ponto mais à esquerda e mais abaixo da envoltória (início).
- Se houver descuido, pode-se incluir três vértices colineares em um lado da envoltória.
- Este problema é resolvido ordenando-se por ângulo de acordo com a distância do ponto inicial da envoltória.
- O mais distante destes pontos colineares é inserido por último: assegura-se que ele permanece na envoltória final em vez de seu grupo angular.

Envoltória convexa

```
bool smaller_angle(point *p1, point *p2)
{
  if (collinear(first_point,*p1,*p2)) {
  if (distance(first_point,*p1) <= distance(first_point,*p2))
    return(-1);
  else
    return(1);
  }

  if (ccw(first_point,*p1,*p2))
    return(-1);
  else
    return(1);
}
```

- O caso restante (degenerado) diz respeito a pontos repetidos.
- Que ângulo é definido entre três ocorrências do mesmo ponto?
- Para eliminar este problema, remove-se cópias duplicadas de pontos quando ordena-se para identificar a ponto da envoltória mais à esquerda e mais abaixo.

Envoltória convexa

```
sort_and_remove_duplicates(point in[], int *n)
{
  int i;     /* counter */
   int oldn;     /* number of points before deletion */
   int hole;     /* index marked for potential deletion */
   bool leftlower();
   qsort(in, *n, sizeof(point), leftlower);
   oldn = *n;
   hole = 1;
   for (i=1; i<oldn; i++) {
    if ((in[hole-1][X] == in[i][X]) && (in[hole-1][Y] == in[i][Y]))
        (*n)-;
    else {
        copy_point(in[i],in[hole]);
       hole = hole + 1;
     }
   }
   copy_point(in[oldn-1],in[hole]);
}

bool leftlower(point *p1, point *p2)
{
  if ((*p1)[X] < (*p2)[X]) return (-1);
  if ((*p1)[X] > (*p2)[X]) return (1);
   if ((*p1)[Y] < (*p2)[Y]) return (-1);
   if ((*p1)[Y] > (*p2)[Y]) return (1);
  return(0);
}
```

Envoltória convexa

- Sobre `convex_hull`: uso de sentinelas para simplificar o código.
- Copia-se o ponto origem no final da cadeia de inserção para evitar o teste explícito da condição de envoltória.
- Depois apaga-se implicitamente este ponto duplicado, atribuindo apropriadamente ao contador de retorno.
- Finalmente, note que a ordenação de pontos é por ângulos sem nunca ter realmente computado os ângulos.
- O predicado `ccw` faz o serviço.

Sumário

1 Geometria
- Retas
- Ângulos

2 Trigonometria
- Ângulos retos
- Resolução de triângulos
- Círculos

3 Geometria Computacional
- Segmentos de reta e interseção
- Polígonos e computações angulares
- Triangulação: algoritmos e problemas relacionados

Triangulação

- Achar o perímetro de um polígono é fácil: calcule o comprimento de cada lado usando a fórmula da distância euclidiana e some.
- Computar a área de "manchas" irregulares é mais difícil.
- A ideia é dividir o polígono em triângulos não sobrepostos e somar suas áreas.
- Esta operação é chamada de *triangulação*.
- Triangular um polígono convexo é fácil: basta conectar um dado vértice v a todos os outros $n - 1$ vértices.
- Desta forma, divide-se um polígono P em triângulos usando cordas que não se intersecionam e que se situam completamente dentro de P.

Triangulação

- Pode-se representar a triangulação ou listando as cordas ou, como é feito aqui, com uma lista explícita dos índices dos vértices em cada triângulo:

```
typedef struct {
    int n;                  /* number of triangles in triangulation */
    int t[MAXPOLY][3];     /* indicies of vertices in triangulation */
} triangulation;
```

- Muitos algoritmos de triangulação de polígonos são conhecidos e o mais eficiente “roda” em tempo linear do número de vértices.
- O algoritmo mais simples para programar é baseado no corte da orelha.

Algoritmo de Van Gogh

- Uma *orelha* de um polígono P é um triângulo definido por um vértice v e seus vizinhos da esquerda e da direita (l e r), tal que o triângulo (v, l, r) se situa completamente dentro de P.
- Como $\vec{lv}$ e $\vec{vr}$ são segmentos limítrofes de P, a corda que define a orelha é $\vec{rl}$.
- Sob quais condições esta corda pode estar na triangulação?
- Primeiro, $\vec{rl}$ deve se situar completamente no interior de P.
- Para isso acontecer, primeito lvr deve definir um ângulo agudo.
- Segundo, nenhum outro segmento do polígono pode ser cortado por esta corda, senão um pedaço será retirado do triângulo.

Algoritmo de Van Gogh

- O fato importante é que *todo* polígono sempre contém uma orelha; na verdade, duas no mínimo para $n \geq 3$.
- Algoritmo: teste cada um dos vértices até achar uma orelha.
- Adicionando a corda associada corta-se a orelha, decrementando o número de vértices.
- O polígono restante deve também ter uma orelha, tal que continua-se a cortar recorrentemente até que apenas três vértices sejam mantidos, ficando um triângulo.

Algoritmo de Van Gogh

- Testar se um vértice define uma orelha tem duas partes.
- Para o teste do ângulo: novamente os predicados `ccw/cw`.
- Deve-se verificar se as expectativas estão consistentes com a ordem dos vértices do polígono.
- Assume-se que os vértices sejam rotulados em uma ordem anti-horária em volta do centro virtual (veja figura).
- A reversão da ordem do polígono requereria trocar o sinal no teste do ângulo.

Algoritmo de Van Gogh

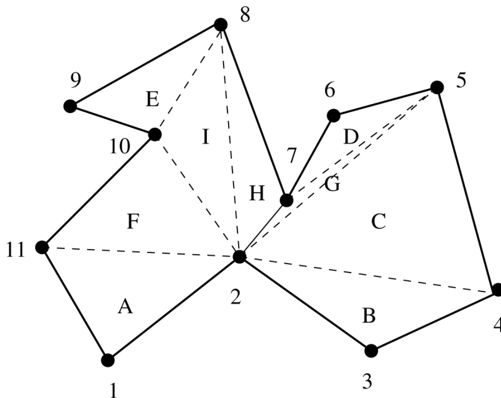


Figure: Triangulação de um polígono via algoritmo de Van Gogh (corte da orelha), com triângulos rotulados na ordem da inserção (*A*-*I*).

Algoritmo de Van Gogh

```
bool ear_Q(int i, int j, int k, polygon *p)
{
  triangle t;    /* coordinates for points i,j,k */
  int m;     /* counter */
  bool cw();

  copy_point(p->p[i],t[0]);
  copy_point(p->p[j],t[1]);
  copy_point(p->p[k],t[2]);

  if (cw(t[0],t[1],t[2])) return(FALSE);

  for (m=0; m<p->n; m++) {
    if ((m!=i) && (m!=j) && (m!=k))
      if (point_in_triangle(p->p[m],t)) return(FALSE);
  }

  return(TRUE);
}
```

Algoritmo de Van Gogh

- Para o teste de corte de segmento, é suficiente testar se existe um vértice que se situa dentro do triângulo induzido.
- Se o triângulo estiver vazio de pontos, o polígono deve estar vazio de segmentos porque P não se auto-interseciona.
- A principal rotina de triangulação limita-se a testar a "orelhidade" dos vértices e subtraí-los quando encontrados.
- Uma propriedade da representação do polígono como vetor de pontos é que os dois vizinhos imediatos do vértice i são facilmente encontrados nas posições $(i - 1)$ e $(i + 1)$ do vetor.

Algoritmo de Van Gogh

- Porém esta estrutura não suporta apagamento de vértice: deve-se definir vetores auxiliares *l* e *r* que apontem para os vizinhos esquerdo e direito correntes de todo ponto remanescente no polígono:

```
triangulate(polygon *p, triangulation *t)
{
  int l[MAXPOLY], r[MAXPOLY];  /* left/right neighbor indices */
  int i;       /* counter */

  for (i=0; i<p->n; i++) {    /* initialization */
    l[i] = ((i-1) + p->n) % p->n;
    r[i] = ((i+1) + p->n) % p->n;
  }

  t->n = 0;
  i = p->n-1;
  while (t->n < (p->n-2)) {
    i = r[i];
    if (ear_Q(l[i],i,r[i],p)) {
      add_triangle(t,l[i],i,r[i],p);
      l[ r[i] ] = l[i];
      r[ l[i] ] = r[i];
    }
  }
}
```

Computações de áreas

- Pode-se computar a área de qualquer polígono triangulado somando as áreas de todos os triângulos.
- Entretanto, existe um algoritmo mais atraente baseado na noção de áreas com sinal para triângulos, usadas como base para a rotina `ccw`.
- Somando as áreas com sinal dos triângulos definidos por um ponto arbitrário p com cada segmento do polígono P, chega-se na área de P, porque os triângulos com sinal negativo cancelam a área fora do polígono:

$$A(P) = \frac{1}{2} \sum_{i=0}^{n-1} (x_i \cdot y_{i+1} - x_{i+1} \cdot y_i) \tag{20}$$

  - onde todos os índices são calculados como o módulo do número de vértices.

Localização de pontos

- O algoritmo de triangulação define um vértice como uma orelha apenas quando o triângulo associado não contém outros pontos.
- Portanto, testar orelha requer o teste se um dado ponto p se situa dentro de um triângulo t.
- Triângulos são sempre polígonos convexos.
- Um ponto se situa dentro de um polígono convexo se estiver a esquerda de cada uma das retas direcionadas $p_i\vec{p}_{i+1}$, onde os vértices do polígono são representados em ordem anti-horária.
- O predicado `ccw` permite tomar as decisões:

Localização de pontos

```
bool point_in_triangle(point p, triangle t)
{
  int i;   /* counter */
  bool cw();

  for (i=0; i<3; i++)
    if (cw(t[i],t[(i+1)%3],p)) return(FALSE);

  return(TRUE);
}
```

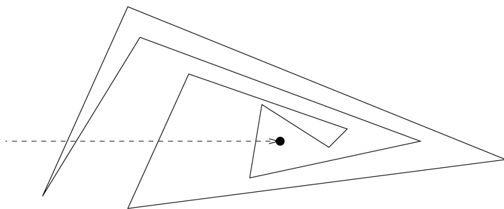
- Este algoritmo decide a *localização do ponto* (em P ou fora?) para polígonos convexos.
- Para polígonos gerais há uma solução que usa o código já apresentado.
- Corte da orelha necessita testar se um dado ponto se situa dentro de um dado triângulo.
- Pode-se usar `triangulate` para dividir o polígono em células triangulares para verificar se contêm o ponto.
- Se uma delas contiver, o ponto está no polígono.

Localização de pontos

- A triangulação é uma solução pesada para este problema: há um algoritmo mais simples baseado no *teorema da curva de Jordan*, que estabelece que todo polígono ou figura próxima tem um lado de dentro e um lado de fora.
- Não se pode ir de um para outro sem cruzar a fronteira.
- Isto fornece o seguinte algoritmo ilustrado na figura:
  - Suponha que se desenhe uma reta l que começa de fora do polígono P e vai até um ponto q.
  - Se esta reta cruza a fronteira do polígono um número par de vezes antes de chegar a q, ela deve se situar fora de P. Por que?
  - Se começarmos fora do polígono, então todo par de cruzamentos de fronteira deixa-nos do lado de fora.
  - Portanto, um número ímpar de cruzamentos coloca-nos dentro de P.

Localização de pontos

Figure: A paridade par/ímpar do número de cruzamentos de fronteira determina se um dado ponto está dentro ou fora de um dado polígono.



Referências I

[1] Skiena, Steven S.; Revilla, Miguel A.
Programming Challenges - The Programming Contest Training Manual - chapters 13 and 14.
Springer, 2003.

[2] The TikZ and PGF manual
Example: A picture for Karl's students.
http://www.texample.net/tikz/examples/tutorial/