Polygon Triangulation
Simple Polygons

- **Definition**

1. A polygon is the region of a plane bounded by a finite collection of line segments forming a simple closed curve.

2. “Simple closed curve” means a certain deformation of a circle.
Simple Polygons

Definition

1. Let \( v_0, v_1, v_2, \ldots, v_{n-1} \) be \( n \) points in the plane in the cyclic ordering.

2. Let \( e_0 = v_0v_1, e_1 = v_1v_2, \ldots, e_i = v_iv_{i+1}, \ldots, \)

\[ e_{n-1} = v_{n-1}v_0 \] be \( n \) segments connecting the points.

3. These segments bound a polygon \textit{iff} \n
   1. The intersection of each pair of segments adjacent in the cyclic ordering is the single point shared between them.

   2. Nonadjacent segments do not intersect.
Simple Polygons

1. The reason these segments define a **curve** is that they are connected end to end
2. The reason the curve is **closed** is that they form a cycle
3. The reason the closed curve is **simple** is that nonadjacent segments do not intersect.

**Jordan Curve Theorem**
Every simple closed plane curve divides the plane into two components.
Nonsimple polygons

1. For both objects in the figure, the segments satisfy condition (1) (adjacent segments share a common point)
2. but not condition (2): nonadjacent segments intersect.
Visibility

1. Point $x$ can see point $y$ (or $y$ is visible to $x$) iff the closed segment $xy$ is nowhere exterior to the polygon $P$ ($xy \subseteq P$)
Visibility

1. $x$ has clear visibility to $y$ if $xy \subseteq P$ and $xy \cap \partial P \subseteq \{x,y\}$

2. $\partial P$ means the boundary of a polygon $P$

3. By definition, $\partial P \subseteq P$

4. A guard is a point.

5. A set of guards is said to cover a polygon if every point in the polygon is visible to some guard.
Triangulation

- Diagonals and Triangulation
  1. A diagonal of a polygon $P$ is a line segment between two of its vertices $a$ and $b$ that are clearly visible to one another.
  2. The open segment from $a$ to $b$ does not intersect $\partial P$; thus a diagonal cannot make grazing contact with the boundary.
  3. Two diagonals are noncrossing if their intersection is a subset of their endpoints. They share no interior points.
  4. A triangulation of a polygon $P$
     - Add as many noncrossing diagonals to a polygon as possible so that the interior can be partitioned into triangles.
FIGURE 1.6 Two triangulations of a polygon of $n = 14$ vertices.
The Art Gallery Theorem

- Problem Definition
  1. A problem posed by Klee [1973]
  2. Imagine an art gallery room whose floor plan can be modeled by a polygon of n vertices. How many stationary guards are needed to guard the room?
  3. Each guard is considered a fixed point that can see in every direction
  4. Of course a guard cannot see through a wall of the room.
The Art Gallery Theorem

- **Max over Min Formulation**
  1. For any given fixed polygon
     - there is some minimum number of guards that are necessary for complete coverage
  2. Consider all polygons of \( n \) vertices
     - Find the maximum (worst case) of the fewest guards needed to cover the polygon
1. What is the largest number of guards that any polygon of 12 vertices needs?

2. Express as a function of $n$, the smallest number of guards that suffice to cover any polygon of $n$ vertices.
The Art Gallery Theorem

- **Max over Min Formulation**
  1. For any given fixed polygon
     - Let $g(P)$ be the smallest number of guards needed to cover polygon $P$
  2. Consider all polygons of $n$ vertices
     - $G(n)$ is the maximum of $g(P_n)$ over all polygons of $n$ vertices: $G(n) = \max_{P_n} g(P_n)$
The Art Gallery Theorem

- Empirical Exploration
  1. Certainly at least one guard is always necessary. A lower bound $1 \leq G(n)$
     - example?
  2. $n$ guards suffice for any polygon. An upper bound $G(n) \leq n$
     - why?
  3. For small $n$, we can guess $G(n)$
     - For example, $G(3) = 1$. Clearly every triangle requires just one guard.
The Art Gallery Theorem

- **Empirical Exploration**
  1. $n=4$ (convex and reflex)
     - Intuitively a polygon is convex if it has no dents.
     - A vertex is called reflex if its internal angle is strictly greater than $\pi$. Otherwise convex.
     - A quadrilateral can have at most one reflex vertex
     - Even quadrilaterals with a reflex vertex can be covered by a single guard placed near that vertex (Figure 1.4(a))

2. Thus $G(4) = 1$
The Art Gallery Theorem

- **Empirical Exploration**
  1. $n=5$ (convex, 1 reflex, and 2 reflex)
    - A convex pentagon needs just one
    - A pentagon with one reflex vertex needs only one guard for the same reason as in a quadrilateral
    - Two reflex vertices may be either adjacent or separated by a convex vertex; in each case one guard suffices. (Figure 1.4(c) and (d))
  2. Thus $G(5) = 1$
  3. $G(6) = 2$ (Figure 1.4(e) and (f))
FIGURE 1.4 Polygons of $n = 4, 5, 6$ vertices.
The Art Gallery Theorem

- Necessity $O\left\lceil \frac{n}{3} \right\rceil$

- This “comb” shape consists of $k$ teeth, with each tooth composed of two edges, and adjacent teeth separated by an edge.
The Art Gallery Theorem

- **Necessity**: $O\lfloor n/3 \rfloor$
  1. A comb of $k$ teeth has $n = 3k$ edges
  2. Because each tooth requires its own guard, $n/3 \leq G(n)$ for $n = 3k$
  3. Notice $G(3) = G(4) = G(5)$ might lead one to conjecture that $G(n) = \lfloor n/3 \rfloor$
Fisk’s Proof

- Three Coloring
  1. To prove sufficiency of \( \lceil n/3 \rceil \) guards, three coloring is used.
  2. A \( k \)-coloring of a graph is an assignment of \( k \) colors to the nodes of the graph, such that no two nodes connected by an arc are assigned the same color.
  3. Three-colorings of the triangulations in Figure 1.6 are shown in Figure 1.7
  4. Starting at the vertex indicated by the arrow, and coloring its triangle arbitrarily with three colors, the remainder of the coloring is completely forced.
FIGURE 1.7 Two 3-colorings of a polygon of \( n = 14 \) vertices, based on the triangulations shown in Figure 1.6.
Fisk’s Proof

- Three Coloring
  1. Each triangle must have each of the three colors (red, green, blue) at its three corners.
  2. Thus every triangle has a red (green or blue) node at one corner.
  3. Suppose guards are placed at every red (green or blue) node. Then every triangle has a guard in one corner.
  4. Thus, the entire polygon is covered.
Polygon Triangulation

Triangulation Theory
Existence of a Diagonal

- **Lemma 1.2.1**
  1. Every polygon must have at least one strictly convex vertex.
  2. Proof
Existence of a Diagonal

FIGURE 1.11 The rightmost lowest vertex must be strictly convex.
Existence of a Diagonal

- Lemma 1.2.2 (Meisters)
  1. Every polygon of $n \geq 4$ vertices has a diagonal.
  2. Proof
Existence of a Diagonal

**FIGURE 1.12** $vx$ must be a diagonal.
Existence of Triangulation

- Lemma 1.2.3(Triangulation)
  1. Every polygon $P$ of $n$ vertices may be partitioned into triangles by the addition of (zero or more) diagonals.
  2. Proof (by induction)
Existence of Triangulation

**FIGURE 1.13** A diagonal partitions a polygon into two smaller polygons.
Properties of Triangulations

- Lemma 1.2.4 (Number of Diagonals)
  1. Every triangulation of a polygon $P$ of $n$ vertices uses $n - 3$ diagonals and consists of $n - 2$ triangles.
  2. Proof (by induction)
1. The \textit{dual} $T$ of triangulation of a polygon is a graph with a node associated with each triangle and an arc between two nodes iff

\begin{center}
\textbf{FIGURE 1.14} Triangulation dual.
\end{center}
Triangulation Dual

- **Lemma 1.2.6**
  1. The dual $T$ of a triangulation is a **tree**, with each node of degree at most **three**.
  2. Proof
1. Three consecutive vertices of a polygon $a, b, c$ form an ear of the polygon if $ac$ is a diagonal; $b$ is the ear tip.

$P_i$ is not an ear

$P_i$ is an ear

Triangulation Dual
Triangulation Dual

- **Meisters’s Two Ears Thm [1975]**
  1. Every polygon of \( n \geq 4 \) vertices has at least two non-overlapping ears.
  2. Proof
     - A leaf node in a triangulation dual corresponds to an ear. A tree of two or more nodes (by Lemma 1.2.4) must have at least two leaves.

Base Case: A quadrilateral has only 2 ears.

A simple polygon with only 2 ears.
Polygon Triangulation

Area of polygon
Area of a Triangle

Let us denote this area as $A(T)$.
The area is one half the base times the altitude.
The base is easy: $|a-b|$ (the length of the vector $a-b$).
What about the altitude?
– not easy.
Cross Product

Recall the magnitude of the cross product of two vectors is the area of the parallelogram. A triangle is half of a parallelogram. Thus, the area of triangle whose three vertices are arbitrary points $a, b, c$ is half the length of $A \times B$. 

$A = b - a$ and $B = c - a$
Area of a Convex Polygon

Find the area of any polygon by using an expression for the area of a triangle.

First triangulation

Every convex polygon may be triangulated as a “fan,” with all diagonals incident to a common vertex.

The area of a polygon with vertices $v_0, v_1, \ldots, v_{n-1}$ labeled counterclockwise (Figure 1.16) can be calculated as

$$A(Q) = A(a, b, c) + A(a, c, d) = A(d, a, b) + A(d, b, c).$$
Area of a Convex Polygon

FIGURE 1.16 Triangulation of a convex polygon. The fan center is at 0.
Area of a Convex Quadrilateral

Two different triangulations of a convex quadrilateral $Q = (a, b, c, d)$

- The area may be written in two ways.

$$A(Q) = A(a, b, c) + A(a, c, d) = A(d, a, b) + A(d, b, c).$$
Area of a Convex Quadrilateral

Applying Equation (1.2) to

\[ A(Q) = A(a, b, c) + A(a, c, d) \]

• We get

\[ 2A(Q) = a_0b_1 - a_1b_0 + a_1c_0 - a_0c_1 + b_0c_1 - c_0b_1 \]

\[ + a_0c_1 - a_1c_0 + a_1d_0 - a_0d_1 + c_0d_1 - d_0c_1. \]

• Notice \( ac \) and \( db \) cancel
Area of a Convex Quadrilateral

Generalizing, we get two terms per polygon edge
None for internal diagonals.
So if the coordinates of vertex $v_i$ are $x_i$ and $y_i$, twice the area of a convex polygon is given by

$$2A(P) = \sum_{i=0}^{n-1} (x_i y_{i+1} - y_i x_{i+1}).$$
Can we use the same equation to calculate the area of a nonconvex quadrilateral?
Area of a Nonconvex Quadrilateral

Suppose we have a nonconvex quadrilateral \( Q = (a, b, c, d) \) (Figure 1.18)
Only one triangulation, using the diagonal \( db \).
But the algebraic expression obtained is independent of the diagonal chosen

\[
A(Q) = A(a, b, c) + A(a, c, d)
\]

- The equation is still true, even though the diagonal \( ac \) is external to \( Q \).
Area of a Nonconvex Quadrilateral

Notice $A(a, c, d)$ is negative and the area of a nonconvex quadrilateral is $\triangle abc$ minus $A(a, c, d)$. Indeed, $A(a, c, d)$ is a clockwise path, so the cross product formulation shows that the area will be negative. The phenomenon observe with a nonconvex quadrilateral is general.
Area from an Arbitrary Center

Let $T = \triangle abc$ be a triangle, with the vertices oriented counterclockwise, and let $p$ be any external point in the plane.

Then, we claim

$$A(T) = A(p, a, b) + A(p, b, c) + A(p, c, a).$$
Area from an Arbitrary Center

**FIGURE 1.19** Area of $T$ based on various external points $p_1, p_2$. 

Diagram showing a triangle $T$ with points $a$, $b$, and $c$ and external points $p_1$ and $p_2$. The diagram illustrates how the area of $T$ can be calculated from different external perspectives.
Area from an Arbitrary Center

Case 1: \( p = p_1 \)
1. \( A(p_1, a, b) \) is negative (clockwise)
2. \( A(p_1, b, c) \) is positive (counterclockwise)
3. \( A(p_1, c, a) \) is positive (counterclockwise)

Case 2: \( p = p_2 \)
1. \( A(p_2, a, b) \) is negative (clockwise)
2. \( A(p_2, b, c) \) is negative (clockwise)
3. \( A(p_2, c, a) \) is positive (counterclockwise)

All other positions for external \( p \) in the plane are equivalent to either \( p_1 \) or \( p_2 \) by symmetry
Area from an Arbitrary Center

- **Lemma 1.3.2**
  1. If $T = \triangle abc$ is a triangle, with vertices oriented counterclockwise, and $p$ is any point in the plane, then

  $$A(T) = A(p, a, b) + A(p, b, c) + A(p, c, a).$$
Area from an Arbitrary Center

- Lemma 1.3.3 (Area of Polygon)
  1. Let a polygon (convex or nonconvex) \( P \) have vertices \( v_0, v_1, \ldots, v_{n-1} \) labeled counterclockwise, and let \( P \) be any point in the plane. Then

\[
A(P) = A(p, v_0, v_1) + A(p, v_1, v_2) + A(p, v_2, v_3) + \cdots \\
+ A(p, v_{n-2}, v_{n-1}) + A(p, v_{n-1}, v_0). \tag{1.12}
\]

- If \( v_i = (x_i, y_i) \), this expression is equivalent to the equations

\[
2A(P) = \sum_{i=0}^{n-1} (x_i y_{i+1} - y_i x_{i+1}) \tag{1.13}
\]

\[
= \sum_{i=0}^{n-1} (x_i + x_{i+1})(y_{i+1} - y_i). \tag{1.14}
\]
Area from an Arbitrary Center

**FIGURE 1.20** Computation of the area of a nonconvex polygon from point \( p \). The darker triangles are oriented clockwise, and thus they have negative area.
Volume in Three and Higher Dimensions

FIGURE 1.21 Tetrahedron at the origin.
Volume in Three and Higher Dimensions

This volume is signed
It is positive if \((a, b, c)\) form a counterclockwise viewed from the side away from \(d\),
So that the face normal determined by the right-hand rule points toward the outside. (Figure 1.21)
The volume of a polyhedron may be computed by summing the (signed) volumes of tetrahedra formed by an arbitrary point and each triangular face of the polyhedron
Polygon Triangulation

Data representation
Representation of a point

Array versus Records
1. Arrays of the appropriate number of coordinates to represent all points
2. Easier to understand
3. Generates to higher dimensions more easily

Integers versus Reals
1. Integer coordinates rather than floating-point number coordinates
2. No issue of floating-point round off error
3. Verifiably correct within a range of coordinate values
Representation of a point

Point Type Definition
1. All type identifiers begin with lowercase t
2. All defined constants entirely uppercase.
3. The suffixes i and d indicate integer and double types respectively

```
#define X 0
#define Y 1
typedef enum {FALSE, TRUE} bool;

#define DIM 2                           /* Dimension of points */
typedef int tPointi[DIM];             /* Type integer point */
```

**Code 1.1** Point type.
Representation of a Polygon

Main options
1. Array or list
2. Singly or doubly linked list
3. Linear or circular

Arrays
1. Attractive for code clarity
2. Clumsy with insertion and deletion of points.

We sacrifice simplicity to gain ease of deletion.

Use a doubly linked circular list to represent a polygon
Representation of a Polygon

The data structure for a single vertex
1. tVertexStructure
2. next and prev for links to adjacent vertices
3. vnum is integer index for printout
4. vertices is global for “head” of the list

```c
typedef struct tVertexStructure tsVertex; /* Used only in NEW(). */
typedef tsVertex *tVertex;
struct tVertexStructure {
    int     vnum;  /* Index */
tPointi     v;  /* Coordinates */
bool      ear;  /* TRUE iff an ear */
tVertex      next,prev;
};
tVertex      vertices = NULL;  /* "Head" of circular list. */
```

**Code 1.2** Vertex structure.
Representation of a Polygon

Loop to process all vertices

```
tVertex v;
v = vertices;
do {
    /* Process vertex v */
    v = v->next;
} while ( v != vertices );
```

**Code 1.3** Loop to process all vertices.
Representation of a Polygon

Two basic list processing routines
1. Allocating a new element (**NEW**)
2. Adding a new element to the list (**ADD**)

As macros, with **NEW** taking the type as one parameter.
1. Macros are text based and oblivious to types.

**ADD** first checks to see if **head** is non-**NULL**
1. If so, insert the cell prior to **head**
2. If not, **head** points to the added cell

See Code 1.4
```c
#define EXIT_FAILURE 1
char *malloc();

#define NEW(p, type)  
  if ((p=(type *) malloc (sizeof(type))) == NULL) {
    printf("NEW: Out of Memory!
");
    exit(EXIT_FAILURE);
  }

#define ADD( head, p )  if ( head )  { 
  p->next = head;
  p->prev = head->prev;
  head->prev = p;
  p->prev->next = p; 
} 
else { 
  head = p;
  head->next = head->prev = p; 
}
#define FREE(p)       if (p)  {free ((char *) p); p = NULL; } 

Code 1.4 New and ADD macros. (The backslashes continue the lines so that the preprocessor does not treat those as command lines.) FREE is used in Chapter 3 and 4.
```
Code for Area

Computing the area of a polygon is now a straightforward implementation of Equations (1.12) or (1.13)

Potential problem with Area2
1. If the coordinates are large, the multiplications of coordinates could cause integer word overflow
int Area2( tPointi a, tPointi b, tPointi c )
{
    return
        (b[X] - a[X]) * (c[Y] - a[Y]) -
        (c[X] - a[X]) * (b[Y] - a[Y]);
}

int AreaPoly2( void )
{
    int sum = 0;
    tVertex p, a;

    p = vertices;  /* Fixed. */
    a = p->next;   /* Moving. */
    do {
        sum += Area2( p->v, a->v, a->next->v );
        a = a->next;
    } while ( a->next != vertices );
    return sum;
}

Code 1.5 Area2 and AreaPoly2
Polygon Triangulation

Segment Intersection
Diagonals

Our goal is to develop code to triangulate a polygon.
The key step is finding a diagonal

- **Lemma 1.5.1**
  1. for all edges $e$ of $P$ that are not incident to either $v_i$ or $v_j$, $s$ and $e$ do not intersect: $s \cap e = \emptyset$
  2. $s$ is internal to $P$ in a neighborhood of $v_i$ and $v_j$. 
Diagonals

Condition (1) of Lemma 1.5.1
1. The “diagonalhood” of a segment can be determined without finding the actual point of intersection between $s$ and each $e$
2. Only a Boolean segment intersection predicate

Condition (2) of Lemma 1.5.1
1. Distinguish internal from external diagonals
2. Rule out collinear overlap with an incident edge
Problems with Slopes

The point of intersection
1. By solving two linear equations in slope–intercept form
2. Clear method
3. Not all that difficult to code

Problem
1. Code is messy and error prone
2. Need to deal with two special cases
3. Vertical segments whose slope is infinite
4. Parallel segments whose containing lines do not intersect
5. Both case lead to division by zero
Left turn

The point of intersection

1. Can be decided by using a Left predicate

A directed line is determined by two points given \((a, b)\)

If \(c\) is to the left of the line, then the triple \((a,b,c)\) forms counterclockwise circuit. (see figure 1.22)

\(c\) is to the left of \((a,b)\) iff \(A(a,b,c)\) is positive

- This can be implemented by a single call to Area2 (See code 1.6)

Straightforward but subject to the special case objections raised earlier.
FIGURE 1.22 $c$ is left of $ab$ iff $\triangle abc$ has positive area; $\triangle abc'$ also has positive area.
Left turn

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

bool LeftOn( tPointi a, tPointi b, tPointi c )
{
    return Area2( a, b, c ) >= 0;
}

bool Collinear( tPointi a, tPointi b, tPointi c )
{
    return Area2( a, b, c ) == 0;
}

Code 1.6 Left

- If c is collinear with \( ab \), then the determined triangle has zero area.
Boolean Intersection

If $ab$ and $cd$ intersect in their interiors,

1. $c$ and $d$ are split by the line $L_1$ containing $ab$
2. Likewise $a$ and $b$ are split by line $L_2$ containing $cd$

Neither one of these conditions is alone sufficient to guarantee intersection

When two segments intersect at a point interior to both, if it is known that no three of the four endpoints are collinear
**Boolean Intersection**

**FIGURE 1.23**  Two segments intersect (a) iff their endpoints are split by their determined lines; both pair of endpoints must be split (b).
bool IntersectProp( tPointi a, tPointi b, tPointi c, tPointi d )
{
    /* Eliminate improper cases. */
    if (Collinear(a,b,c) ||
        Collinear(a,b,d) ||
        Collinear(c,d,a) ||
        Collinear(c,d,b))
    return FALSE;

    return
    Xor( Left(a,b,c), Left(a,b,d) )
    && Xor( Left(c,d,a), Left(c,d,b) );
}

/*Exclusive or: T iff exactly one argument is true. */
bool Xor( bool x, bool y )
{
    /* The arguments are negated to ensure that they are 0/1 values. */
    return !x ^ !y;
}

Code 1.7 IntersectProp
Boolean Intersection

Redundancy in this code
  1. Four relevant triangle areas are being computed twice each.

Two ways to remove redundancy
  1. Storing computed areas in local variables
  2. Designing other primitives that fit the problem better.

The first if-statement may be removed entirely for the purposes of triangulation
But sacrifice efficiency for clarity
It might be tempting to implement the exclusive-or by

\[ \text{Area2}(a, b, c) \times \text{Area2}(a, b, c) < 0 \]
\[ \&\& \text{Area2}(c, d, a) \times \text{Area2}(c, d, b) < 0; \]

• But the product of the areas might cause integer word overflow!
• The problem can be avoided by having \text{Area2} return +1, 0, or −1
Improper Intersection

FIGURE 1.24 Improper intersection between two segments (a); collinearity is not sufficient (b).
Improper Intersection

Lemma 1.5.1 requires that the intersection be completely empty for a segment to be a diagonal

Special case of improper intersection

1. An endpoint of one segment (say c) lies somewhere on the other segment $ab$ (Figure 1.24(a))
2. This only happens if $a, b, c$ are collinear
3. But collinearity is not a sufficient condition for intersection (Figure 1.24)
Betweenness

Only check betweenness of $c$ when we know it lies on the line containing $ab$

If $ab$ is not vertical

1. $c$ lies on $ab$ iff the $x$ coordinate of $c$ falls in the interval of the $x$ coordinates of $a$ and $b$

If $ab$ is vertical

1. Similarly check on $y$ coordinates

See code 1.8
bool Between( tPointi a, tPointi b, tPointi c )
{
    tPointi ba, ca;

    if ( ! Collinear( a, b, c ) )
        return FALSE;

    /* If ab not vertical, check betweenness on x; else on y. */
    if ( a[X] != b[X] )
        return ((a[X] <= c[X]) && (c[X] <= b[X])) ||
                ((a[X] >= c[X]) && (c[X] >= b[X]));
    else
        return ((a[Y] <= c[Y]) && (c[Y] <= b[Y])) ||
                ((a[Y] >= c[Y]) && (c[Y] >= b[Y]));
}

Code 1.8 Between
bool Intersect( tPointi a, tPointi b, tPointi c, tPointi 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;
}
bool Diagonalie( tVertex a, tVertex b )
{
    tVertex c, c1;

    /* For each edge (c,c1) of P */
    c = vertices;
    do {
        c1 = c->next;
        /* Skip edges incident to a or b */
        if ( ( c != a ) && ( c1 != a )
            && ( c != b ) && ( c1 != b )
            && Intersect( a->v, b->v, c->v, c1->v )
        )
            return FALSE;
        c = c->next;
    } while ( c != vertices );
    return TRUE;
}
Polygon Triangulation

Internal or External
InCone

Goal: distinguish the internal from the external diagonals
One vector B (along the diagonal) lies strictly in the open cone counterclockwise between two other vectors A and C (along two consecutive edges)
Need to consider convex and reflex angles
InCone

Convex case (Figure 1.25 a)
1. $s$ is internal to $P$ iff it is internal to the cone whose apex is $a$, and whose sides pass through $a_-$ and $a_+$
2. Easily determined by our Left function
3. $a_-$ must be left of $ab$ and $a_+$ must be left of $ba$

Reflex case (Figure 1.25 b)
1. reverse of the convex case
**FIGURE 1.25** Diagonal $s = ab$ is in the cone determined by $a_-, a, a_+$: (a) convex; (b) reflex. In (b), both $a_-$ and $a_+$ are right of $ab$. 
InCone

Distinguishing between the convex and reflex cases is accomplished with one invocation of Left

\( \alpha \) is convex iff \( \alpha_- \) is left or on \( aa_+ \)

Note that if \( (\alpha_-, \alpha, \alpha_+) \) are collinear, the internal angle at \( \alpha \) is \( \pi \), which we define as convex
InCone

```c
bool InCone( tVertex a, tVertex b )
{
  tVertex a0,a1;    /* a0,a,a1 are consecutive vertices. */
  a1 = a->next;
  a0 = a->prev;

  /* If a is a convex vertex ... */
  if( LeftOn( a->v, a1->v, a0->v ) )
    return  Left( a->v, b->v, a0->v )
             && Left( b->v, a->v, a1->v );

  /* Else a is reflex: */
  return !(    LeftOn( a->v, b->v, a1->v )
             && LeftOn( b->v, a->v, a0->v ) );
}
```

Code 1.11 InCone
**Diagonal**

\(ab\) is a diagonal iff \(\text{Diagonalie}(a, b), \text{InCone}(a, b),\) and \(\text{InCone}(b, a)\) are true

**How to order function calls**

1. InCones should be first
2. They are each constant-time calculation
3. Each performs in the neighborhood \(a\) and \(b\) without regard to the remainder of the polygon, whereas Diagonalie includes a loop over all \(n\) polygon edges.
bool Diagonal( tVertex a, tVertex b )
{
    return InCone( a, b ) && InCone( b, a ) && Diagonalie( a, b );
}

Code 1.12 Diagonal
Triangulation

- **Diagonal-Based Algorithm**

  It is an $O(n^4)$ algorithm
  1. $(n \text{ choose } 2)$ diagonal candidates = $O(n^2)$
  2. Testing each for diagonalhood = $O(n)$
  3. Reapting this $O(n^3)$ computation for each of the $n-3$ diagonals = $O(n^4)$

  Use the **two ears theorem** to speed up
  Only $O(n)$ “ear diagonal” candidates
  We can achieve a worst-case complexity of $O(n^3)$ this way
Ear Removal

- Ear Removal
  Improve the above algorithm to $O(n^2)$
  1. Because one call to Diagonal costs $O(n)$, Diagonal may only be called $O(n)$ times

Key idea
  1. Removal of one ear does not change the polygon very much
  2. Not change whether or not many of its vertices are potential ear tips.

Determination for potential ear tip of each vertex already uses $O(n^2)$, but is not repeated.
Ear Removal

- Ear Removal
  Let \((v_0, v_1, v_2, v_3, v_4)\) be five consecutive vertices of \(P\)
  Suppose \(v_2\) is an ear tip and the ear \(E_2 = \triangle(v_1, v_2, v_3)\) is deleted (see Figure 1.26)
  Only \(v_1\) and \(v_3\) change
  Neighbor vertices remain unchanged
Ear Removal

FIGURE 1.26 Clipping an ear $E_2 = \Delta(v_1, v_2, v_3)$. Here the ear status of $v_1$ changes from TRUE to FALSE.
Ear Removal

After the expensive initialization step, the ear tip status information can be updated with two calls to Diagonal per iteration.

**Algorithm 1.1** Triangulation algorithm

**Algorithm**: TRIANGULATION

Initialize the ear tip status of each vertex.

while $n > 3$ do

    Locate an ear tip $v_2$.
    Output diagonal $v_1v_3$.
    Delete $v_2$.
    Update the ear tip status of $v_1$ and $v_3$. 


Ear Removal

- Triangulation Code
  The first task is to initialize the Boolean flag \( v \rightarrow \text{ear} \)
  Need one call to Diagonal per vertex.
void EarInit( void )
{
    tVertex v0, v1, v2; /* three consecutive vertices */

    /* Initialize v1->ear for all vertices. */
    v1 = vertices;
    printf("newpath
");
    do {
        v2 = v1->next;
        v0 = v1->prev;
        v1->ear = Diagonal( v0, v2 );
        v1 = v1->next;
    } while ( v1 != vertices );
}
Ear Removal

- Triangulation Code
  The main Triangulate code consists of a double loop.
  The outer loop removes one ear per iteration.
  The inner loop searches for an ear by checking the precomputed \( v_2 \)-ear flag, where \( v_2 \) is the potential ear tip.
  Status of \( v_1 \) and \( v_3 \) are updated by calls to Diagonal and the ear is removed by rewiring the next and prev pointers for \( v_1 \) and \( v_3 \).
  Then, the head point, vertices, is moved to point to \( v_3 \).
void Triangulate( void )
{
    tVertex v0, v1, v2, v3, v4;  /* five consecutive vertices */
    int n = nvertices;          /* number of vertices; shrinks to 3. */

    EarInit();
    /* Each step of outer loop removes one ear. */
    while ( n > 3 ) {
        /* Inner loop searches for an ear. */
        v2 = vertices;
        earfound = FALSE;
        do {
            if (v2->ear) {
                /* Ear found. Fill variables. */
                v3 = v2->next; v4 = v3->next;
                v1 = v2->prev; v0 = v1->prev;

                /* (v1,v3) is a diagonal */
                PrintDiagonal( v1, v3 );

                /* Update earity of diagonal endpoints */
                v1->ear = Diagonal( v0, v3 );
                v3->ear = Diagonal( v1, v4 );
            }
        } while (earfound == FALSE);
    }
}
/* Cut off the ear v2 */
v1->next = v3;
v3->prev = v1;
vertices = v3;       /* In case the head was v2. */
n--;
break; /* out of inner loop; resume outer loop */
} /* end if ear found */

v2 = v2->next;
} while ( v2 != vertices );
} /* end outer while loop */

Code 1.14 Triangulate
Example

Figure 1.27 shows a polygon and the triangulation produced by the simple main program (Code 1.15)

```
main()
{
    ReadVertices();
    PrintVertices();
    Triangulate();
}
```

Code 1.15 main
FIGURE 1.27  A polygon of 18 vertices and the triangulation produced by Triangulate. The dark subpolygon is the remainder after the 9th diagonal (15, 3) is output. Vertex coordinates are displayed in Table 1.1.
Example

Now walk through the output of the diagonals

$v_0$ is an ear tip, so the first diagonal output is (17,1)

$v_1$ is not an ear tip, so $v_2$ pointer moves to $v_2$

$v_2$ is a tip, so print the diagonal (1,3)

Neither $v_3$ nor $v_4$ is an ear tip

At $v_5$, the next diagonal is (4,6)

$v_3, v_8$ is collinear with $v_7$, so the next ear detected is not until $v_{10}$

... Another collinearity, $v_9$ with $(v_{11}, v_{15})$, prevents $v_9$ from being an ear

...
Example

Table 1.2. The columns show the order in which the diagonals, specified as pairs of endpoint indices, are output.

<table>
<thead>
<tr>
<th>Order</th>
<th>Diagonal Indices</th>
<th>Order</th>
<th>Diagonal Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(17, 1)</td>
<td>10</td>
<td>(3, 7)</td>
</tr>
<tr>
<td>2</td>
<td>(1, 3)</td>
<td>11</td>
<td>(11, 14)</td>
</tr>
<tr>
<td>3</td>
<td>(4, 6)</td>
<td>12</td>
<td>(15, 7)</td>
</tr>
<tr>
<td>4</td>
<td>(4, 7)</td>
<td>13</td>
<td>(15, 8)</td>
</tr>
<tr>
<td>5</td>
<td>(9, 11)</td>
<td>14</td>
<td>(15, 9)</td>
</tr>
<tr>
<td>6</td>
<td>(12, 14)</td>
<td>15</td>
<td>(9, 14)</td>
</tr>
<tr>
<td>7</td>
<td>(15, 17)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(15, 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>(15, 3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis

The time complexity of the algorithm.

1. EarInit costs $O(n^2)$
2. The outer loop of Triangulate iterates $n-3$ times = $O(n)$
3. The inner search-for-an-ear loop is also $O(n)$
4. The work inside the inner loop is $O(n)$

Naively we have then a time complexity of $O(n^3)$, falling short of the promised $O(n^2)$
Analysis

Figure 1.28
After $v_0$ is deleted, the inner loop searches past $v_1, \ldots, v_6$ before reaching the next ear tip $v_7$.
Then must search past $v_8, \ldots, v_{12}$ before finding the ear tip $v_{13}$
Notice the two $O(n)$ Diagonal calls within the loop are only invoked once an ear is found
Thus although the superficial structure of the code suggest $n \times n \times n = O(n^3)$
It is actually $n \times (n + n) = O(n^2)$
Analysis

Figure 1.28 An example that forces the inner ear loop to search extensively for the next ear.
Conclusion

We learned:

1. what is a polygon, diagonal, triangulation
2. how to determine
   – the area of polygons
   – if 3 points, collinear, turn–left, in–between
   – if two segment intersect
   – if a segment is internal or external a polygon
   – ears in a polygon

Homework assignment: 1.1.4–1, 1.3.9–4, 1.6.8–2, 1.6.8–3 (due 9/16 before the class)

Programming assignment: posted online by Friday midnight (due 9/22 midnight)