Straight Skeleton of Polygon

Anand Padmanabhan

Different types of Polygon

- Convex
- Concave
- PSLG
- Polygon with holes

Definition

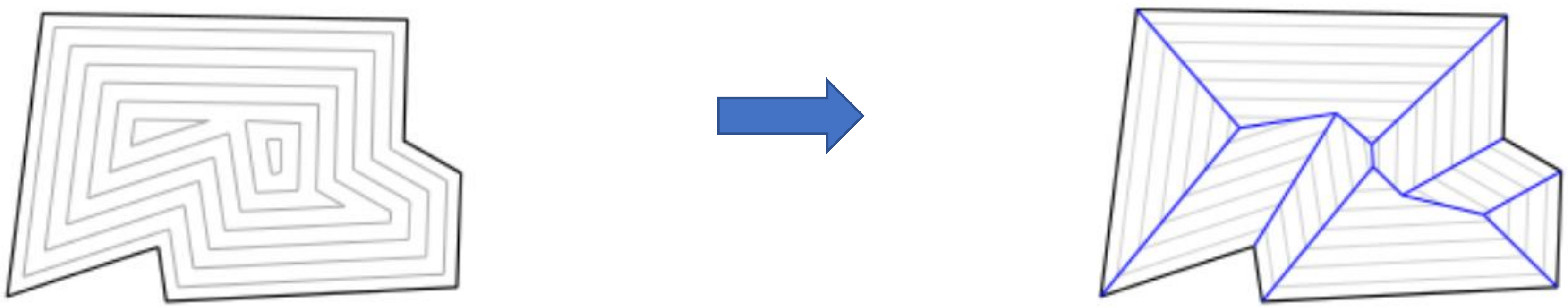
- The straight skeleton S(P) of P is defined by a wavefront propagation process where the edges of P move inwards at unit speed

- Two kind of structural changes occur to the wavefront:
  - (i) edges may collapse and vanish (edge event)
  - (ii) reflex vertices may hit another part of the wavefront and split it into parts (split event)

- The line structure that is traced out by the wavefront vertices is the straight skeleton S(P) of P

Source: The Topology of Skeletons and Offsets by Stefan Huber

Example



Source: https://en.wikipedia.org/wiki/Straight_skeleton

Literature

A Novel Type of Skeleton for Polygons

- by Oswin Aichholzer and Franz Aurenhammer

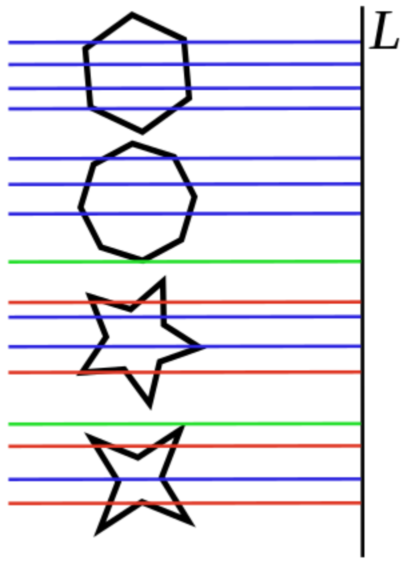
Brief description

- The new structure, called the straight skeleton, is solely made up of straight line segments which are pieces of angular bisectors of polygon edges.
- It uniquely partitions the interior of a given n-gon P into n monotone polygons, one for each edge of P.

Source: https://www.jucs.org/jucs_1_12/a_novel_type_of/Aichholzer_O.html

Monotone Polygons

- In geometry, a polygon P in the plane is called monotone with respect to a straight line L, if every line orthogonal to L intersects the boundary of P at most twice



Source: https://en.wikipedia.org/wiki/Monotone_polygon

Data Structures

- Circular doubly linked list (LAV)
- Priority Queue

Algorithm Stage 1

Source: Petr Felkel and Stepan Obdrzalek

Algorithm – Stage 1

1. Go clockwise starting from any vertex of the polygon. Organize them into a circular doubly linked list. V(1), V(2), V(3)…V(n)

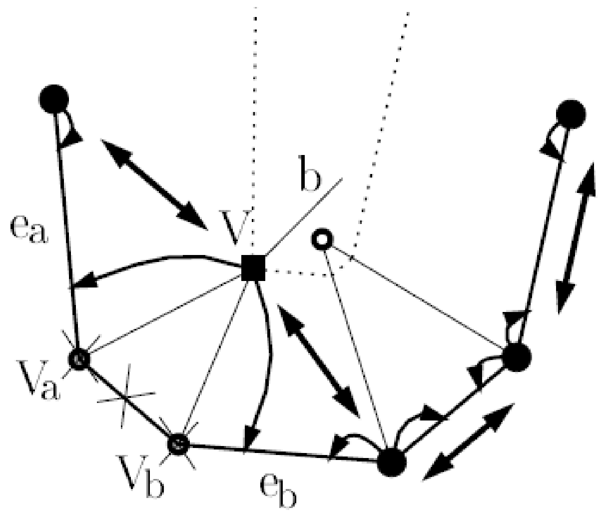
2. For each vertex Vi in the LAV, add the 2 pointers to the incident edges. Here Edge E(i-1) and Edge E(i) are incident to a vertex V(i) going counter clockwise.

   Note: Edge E(i) is also incident to vertex V(i+1)

3. Lets call the interior bisector of a vertex in a polygon as B(i). Compute the interior bisector at every vertex.

Algorithm – Stage 1

4. Find the intersection point of every adjacent bisectors. Store the intersection points in a priority queue with respect to the distance to the corresponding edge.



Source: Petr Felkel and Stepan Obdrzalek

Input

- 2D vector of points in plane – listed in Counter Clockwise fashion

```
//std::vector<std::vector<double>> F{ {2.0, 4.0}, {1.0, 1.0}, {4.0, 0.0}, {5.0, 2.0} };
//std::vector<std::vector<double>> F{ {3.5, 5}, {2.0, 4.0}, {1.0, 1.0}, {4.0, 0.0}, {5.0, 2.0} };
//std::vector<std::vector<double>> F { {0.0, 1.0}, {0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0} };
std::vector<std::vector<double>> F{ {0.5, 2.0}, {0.0, 1.0}, {0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0} };
//std::vector<std::vector<double>> F{ {0.0, 2.0}, {-3.0, 0.5}, {0.0, -1.0}, {1.0, -0.5}, {2.0, 1.0} };
//std::vector<std::vector<double>> F { {-0.1, 0.9}, {0.1, -0.1}, {1.1, 0.1}, {0.9, 1.1} };
//std::vector<std::vector<double>> F{ {0.0, 0.0}, {0.5, -1.0}, {1.0, 0.0}, {0.5, 1.0} };
```

Node and CLL

```
template <typename T>
struct CllNode {
    T* v;
    CllNode<T>* next = nullptr;
    CllNode<T>* prev = nullptr;;
    CllNode<T>(T* val) : v(val) {}
};
```

```
template <typename T>
class CLL {
private:
    T* head;
    T* tail;
public:

    CLL ()
    {
        head = nullptr;
    }

    void insert_after(T* node, T* newNode);
    void insert_before(T* Node, T* newNode);
    void push_back(T* node);
    void push_front(T* node);
    void erase(T* node);

    int Size() {
        return m_size;
    }

    T* begin() const { return head;  }

public:
    size_t m_size = 0;
};
```

Construct a LAV from this input

```
void Polygon::init_polygon(const std::vector<std::vector<double>>& F) {
    for (const std::vector<double>& f : F) {
        Vertex* v = new Vertex(f[0], f[1]);
        add_vertex(v);
    }
    init_edges();
    compute_angle_bisector();
    compute_bisector_intersections();
    compute_straight_skeleton();

    return;
}
```

Adding Vertices to LAV

Creating Edges

```
void Polygon::init_edges() {
    CllNode<Vertex>* temp = LAV.begin();
    for (int i = 0; i < LAV.Size(); i++) {
        //always go counter clockwise - just a convention
        Edge* e = new Edge(temp->v, temp->next->v);
        temp->v->e2 = e;
        temp->next->v->e1 = e;
        this->add_edge(e);
        temp = temp->next;
    }
    return;
}
```

Definition: Point and Line

```cpp
//cartesian coordinate
struct Point {
    double x = 0.0;
    double y = 0.0;
    Point(){}
    Point(double x, double y) : x(x), y(y) {};
    Point(const Point& p) {
        x = p.x;
        y = p.y;
    }
    Point& operator=(const Point& p) {
        x = p.x;
        y = p.y;
        return *this;
    }

    Point operator + (Point p) { return { x + p.x, y + p.y }; }
    Point operator - (Point p) { return { x - p.x, y - p.y }; }
    Point operator * (double d) { return { x * d, y * d }; }
    Point operator / (double d) { return { x / d, y / d }; }

    friend std::ostream& operator<<(std::ostream& os, Point p);

};
```

```cpp
// generic equation of a line
struct Line {

    //direction vector v
    Point v;
    double constant = 0;

    //slope equation y = mx + d
    double m = 0; // slope
    double d = 0; // intercept

    // standard form ax + by + c = 0
    double a = 0;
    double b = 0;
    double c = 0;

    Line() {}
```

Definition: Vertex and Edge

```
struct Vertex {
    Point pos;
    Edge* e1 = nullptr;
    Edge* e2 = nullptr;
    std::pair<double, double> bisector_p;
    Line bisector_line; // the bisector line passing through the vertex
    bool visited = false;

    Vertex (){}
    Vertex (double x, double y) : pos(x, y) { }
    Vertex(const Vertex& v) {
        pos = v.pos;
        e1 = v.e1;
        e2 = v.e2;
        bisector_line = v.bisector_line;
    }
};
```

```
struct Edge {
    Vertex* v1 = nullptr;
    Vertex* v2 = nullptr;
    Line line;
    Edge() {}
    Edge(Vertex* v1, Vertex* v2) : v1(v1), v2(v2) {
        line = Line(v1, v2);
    }
};
```

Point of Intersection of 2 lines
Distance of Point from a line

```
Point bisector_line_intersections(const Line& l1, const Line& l2) {
    Point p;
    auto a1 = l1.a; auto b1 = l1.b; auto c1 = l1.c;
    auto a2 = l2.a; auto b2 = l2.b; auto c2 = l2.c;
    p.x = (b1 * c2 - b2 * c1) / (a1 * b2 - a2 * b1);
    p.y = (a2 * c1 - a1 * c2) / (a1 * b2 - a2 * b1);
    return p;
}

double point_line_dist(const Line& l, Point p) {
    double ret;
    auto a = l.a; auto b = l.b; auto c = l.c;
    auto x0 = p.x; auto y0 = p.y;
    return abs(a * x0 + b * y0 + c) / sqrt(pow(a, 2) + pow(b, 2));
}
```

Computing equation of line from 2 edge vertices

```
// find the equation of line segment (edge)given 2 vertices of polygon
Line::Line(Vertex* v1, Vertex* v2)
{

    v = v1->pos - v2->pos;
    constant = -1 * cross(v, v1->pos);
    double x1 = v1->pos.x;
    double x2 = v2->pos.x;

    double y1 = v1->pos.y;
    double y2 = v2->pos.y;

    if (x2 - x1 == 0)
        m = std::numeric_limits<double>::max();
    else
        m = (double)(y2 - y1) / (double)(x2 - x1);

    if (m == std::numeric_limits<double>::max()) d = 0;
    else
        d = double(y2) - (double)(m * x2);
```

Contd.

```
//computing the corresponding standard form  ax + by + c = 0

if (m != std::numeric_limits<double>::max()) {
    a = y2 - y1;
    b = -1 * (x2 - x1);
    c = d * (x2 - x1);
}
else if (m == 0) {
    a = 0;
    b = 1;
    c = -y1;
}
else {
    a = 1;
    b = 0;
    c = -x1;
}
```

Compute the bisector

- Compute the bisector equation in standard form, given 2 edge equations at a Vertex V(i)

$$\frac{a_1x+b_1y+c_1}{\sqrt{a_1^2+b_1^2}} = \pm\frac{a_2x+b_2y+c_2}{\sqrt{a_2^2+b_2^2}}$$

- (a1 * a2 + b1 * b2) > 0   => ‘+’ obtuse; ‘-’ acute
- (a1 * a2 + b1 * b2) < 0   => ‘+’ acute; ‘-’ obtuse

Compute the bisector (contd.)

```
Line calc_bisector(Edge* e1, Edge* e2) {
    Line& l1 = e1->line;
    Line& l2 = e2->line;
    Line ret;
    // check if the angle between 2 edges internal to the polygon is obtuse
    bool is_obtuse = obtuse_calc(e1, e2);
    auto a1 = l1.a; auto b1 = l1.b; auto c1 = l1.c;
    auto a2 = l2.a; auto b2 = l2.b; auto c2 = l2.c;

    // If a1.a2 + b1.b2 > 0,
    // then the bisector corresponding to “ + “ symbol gives the obtuse angle bisector
    // and the bisector corresponding to “ - " is the bisector of the acute angle between the lines
    bool obtuse_check = (a1 * a2 + b1 * b2) > 0;

    double numerator = std::sqrt(pow(a1, 2.0) + pow(b1, 2.0));
    double denominator = std::sqrt(pow(a2, 2.0) + pow(b2, 2.0));
    // handle edge cases

  double k = double(numerator / denominator);
  if ((obtuse_check && is_obtuse) || (!obtuse_check && !is_obtuse)) {
      ret.a = (a1 - k * a2);
      ret.b = (b1 - k * b2);
      ret.c = (c1 - k * c2);
  }
  else {
      ret.a = (a1 + k * a2);
      ret.b = (b1 + k * b2);
      ret.c = (c1 + k * c2);
  }
  return ret;
```

Intersection point of 2 bisectors

```
struct Intersection_point {
    CllNode<Vertex>* Va;
    CllNode<Vertex>* Vb;
    Point p;
    double dist = 0; //distance of this intersection point from the corresponding line - key in priority queue

    Intersection_point(CllNode<Vertex>* Va, CllNode<Vertex>* Vb, Point p, double dist) : Va(Va), Vb(Vb), p(p), dist(dist) {}

    Intersection_point(const Intersection_point& other) {
        Va = other.Va;
        Vb = other.Vb;
        p = other.p;
        dist = other.dist;
    }

    Intersection_point& operator=(const Intersection_point& other) {
        Va = other.Va;
        Vb = other.Vb;
        p = other.p;
        dist = other.dist;
        return *this;
    }
};
```

Bisector intersections

- Compute bisector line intersections for each pair of Vertices V(i) and V(i-1)

```
void Polygon::compute_bisector_intersections() {
    CllNode<Vertex>* temp = LAV.begin();
    for (int i = 0; i < LAV.Size(); i++) {
        //calculate the point of intersection of 2 bisectors of vertices v(i-1) and v(i)
        Point I = bisector_line_intersections(temp->prev->v->bisector_line, temp->v->bisector_line);
        //find the distance of this point of intersection to the line
        double dist = point_line_dist(temp->v->e1->line, I);
        auto Int_pt = Intersection_point(temp->prev, temp, I, dist);
        pQ.push(Int_pt);
        temp = temp->next;
    }
}
```

Additional (structure of class Polygon)

```
class Polygon {

private:

    struct comp {
        bool operator () (const Intersection_point& a, const Intersection_point& b) {
            return a.dist > b.dist;
        }
    };

public:
    CLL<CllNode<Vertex>> LAV; // circular linked list of vertices (LAV)
    std::unordered_set<CllNode<Vertex>*> vertices; // total set of vertices
    std::unordered_set<Edge*> edges; // total set of edges
    std::priority_queue<Intersection_point, std::vector<Intersection_point>, comp> pQ;
    std::vector<std::vector<Point>> straight_skeleton;
```

Algorithm Stage 2

Source: Petr Felkel and Stepan Obdrzalek

Steps 1 -> 4

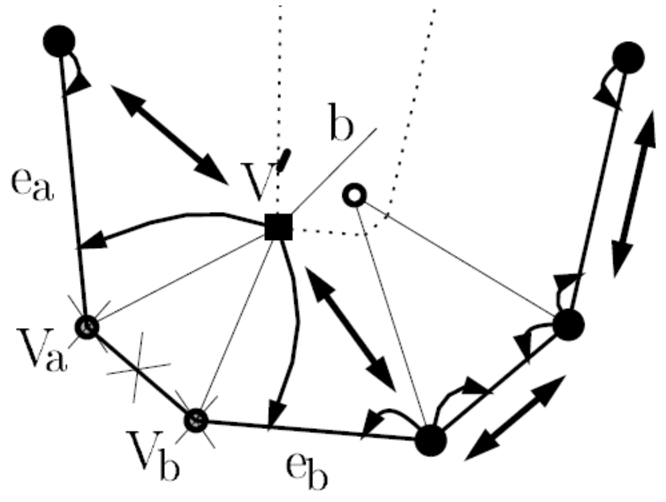
- 1. Pop the intersection point ‘I’ from top of Priority Queue
- 2. If the nodes Va and Vb pointed to by I are already processed, continue.
- 3. If not visited, mark Va and Vb visited. This demarcates the occurrence of an edge event
- 4. If there are only 3 vertices left, i.e. this is the last triangle. Then output the 3 pairs I-Va, I-Vb and I-Vc. Algorithm terminates.
- Else push the 2 arcs (lines) I-Va and I-Vb

First 4 steps in C++ code

```cpp
//Core Logic
void Polygon::compute_straight_skeleton() {

    while (!pQ.empty()) {
        //
        Intersection_point curr = pQ.top();
        pQ.pop();
        if (curr.Va->v->visited && curr.Vb->v->visited) continue;
        curr.Va->v->visited = true;
        curr.Vb->v->visited = true;

        // check if we have reached the end (last triangulation)
        if (curr.Va->prev->prev == curr.Vb) {
            straight_skeleton.push_back({ curr.p, curr.Va->v->pos });
            straight_skeleton.push_back({ curr.p, curr.Vb->v->pos });
            straight_skeleton.push_back({ curr.p, curr.Va->prev->v->pos });
            return;
        }

        //push the result
        straight_skeleton.push_back({ curr.p, curr.Va->v->pos });
        straight_skeleton.push_back({ curr.p, curr.Vb->v->pos });
```

Steps 5 -> 7

- 5. Create a new vertex V’ out of this Vertex formed by the intersection point. Create a corresponding circular linked list node out of this vertex (to be inserted)
- 6. Connect this Vertex V’ to the predecessor of Va and to the successor of Vb in the circular linked list (illustrated by the pic)
- 7. Link the edges Ea and Eb to the V’



Steps 5 -> 7 in code

```
//create a new vertex
Vertex* v = new Vertex(curr.p.x, curr.p.y);

//hook the correct edges
v->e1 = curr.Va->v->e1;
v->e2 = curr.Vb->v->e2;

curr.Va->v->e1->v2 = v;
curr.Vb->v->e2->v1 = v;

//insert it into the SLAV (LAV here) & hook up the connections;
CllNode<Vertex>* cv = new CllNode<Vertex>(v);
cv->next = curr.Vb->next;
curr.Vb->next->prev = cv;
cv->prev = curr.Va->prev;
curr.Va->prev->next = cv;
```

Steps 8 - 10

- 8. Compute the new bisector formed by line segments Ea and Eb.
- 9. Just like Step 4 from the Stage 1 of this algorithm, compute the bisector intersections of its neighboring vertices with this bisector at vertex V’.
- 10. Compute the distance of these intersection points and store them in priority queue.

Code snippet steps 8 -> 10

```
//calculator its bisector;
v->bisector_line = calc_bisector(v->e1, v->e2);

Point p1 = bisector_line_intersections(v->e1->v1->bisector_line, v->bisector_line);
double dist_p1 =  point_line_dist(v->e1->line, p1);
auto Int_pt1 = Intersection_point(cv->prev, cv, p1, dist_p1);

Point p2 = bisector_line_intersections(v->bisector_line, v->e2->v2->bisector_line);
double dist_p2 = point_line_dist(v->e2->line, p2);
auto Int_pt2 = Intersection_point(cv, cv->next, p2, dist_p2);

//Push if its greater than an epsilon
if (abs(curr.p.x - Int_pt1.p.x) > 0.01 && abs(curr.p.y - Int_pt1.p.y) > 0.01) pQ.push(Int_pt1);
if (abs(curr.p.x - Int_pt2.p.x) > 0.01 && abs(curr.p.y - Int_pt2.p.x) > 0.01) pQ.push(Int_pt2);
```

Final Step

- Output the straight skeleton

```
std::cout << std::endl << " Printing straight skeleton " << std::endl;
for (auto& pts : p.straight_skeleton) {
    for (auto& pt : pts)
        std::cout << pt << std::endl;
    std::cout << std::endl;
}
```

Thank You !