Thursday, February 16, 2006 (Ming C. Lin)
Review on Computational Geometry &
Collision Detection for Convex Polytopes

COMPUTATIONAL GEOMETRY

(Refer to O'Rourke's and "Dutch" textbook)

1. Extreme Points & Convex Hulls

⇨ Providing a bounding volume

2. Voronoi Diagram&Delaunay Triangle

⇨ For tracking closest points

3. Tetrahedralization

⇨ Use in CD method & convex decomp

4. Convex Decomposition

⇨ For CD btw non-convex polyhedra

5. Linear Programming

⇨ Check if a pt lies w/in a convex polytope

Extreme Points and Convex Hull

Suppose we have a database of people with their heights and weights.

person	weights (lbs)	heights (feet)
A	150	7.0
B	100	5.0
C	160	5.5
D	200	6.6
E	250	4.8

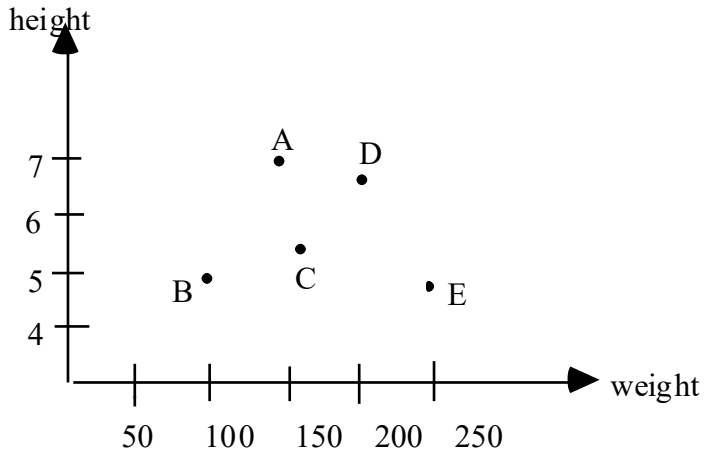
Which person(s) is(are) "extreme" in their measurements?

1. Certainly A, who is the tallest
2. B, who is the lightest
3. E, who is the heaviest & shortest.
4. D "extreme"? Both heavy and tall.
   D maximizes sum of height & weight?
   No. But, D maximizes the following:
   (weight in lbs) + 100 *(height in feet)

CONCLUSION: A data point of (weight, height) "extreme", iff there are some magic numbers *a* and *b* so that within the group of data points, it maximizes the function:

$$a * (weight) + b * (height)$$

According to this definition, C is the only person in our example that is not "extreme".



Since the set of all points *(x,y)* that satisfy $ax + by = c$ forms a straight line, which translates when *c* changes. A point in a finite point set *S* in the plane is an extreme point of *S* if one can draw a straight line through the point and have all the other points of *S* lie strictly on one side of the line.

Definition: Let S be a set of n points in R^2. A point $p = (p_x, p_y)$ in S is an ***extreme point*** for S iff there exists a, b in R such that for all $q = (q_x, q_y)$ in S with $q \mathrel{!}= p$ we have

$$a\,p_x + b\,p_y > a\,q_x + b\,q_y$$

Geometric interpretation: There is a line with the normal vector (a,b) through $\boldsymbol{p}$ so that all other points of S lies strictly on one side of this line. Intuitively, $\boldsymbol{p}$ is the most extreme point of S in the direction of the vector $\boldsymbol{v} = (a,b)$.

Definition: The ***convex hull*** of a set S is the intersection of all *convex sets* that contains S. It should be easy to see that the convex hull of S is the smallest convex polygon that contains S and that the extreme points of S are just the corners of that polygon. Solving the convex hull problem implicitly solves the extreme point problem.

Constructing Convex Hulls

- Graham's Scan
- Marriage before Conquest
  (similar to Divide-and-Conquer)
- Gift-Wrapping
- Incremental

And, many others ……

Lower bound: $O(n \log H)$, where n is the input size (No. of points in the given set) and H is the No. of the extreme points.

Voronoi Diagrams

Proximity Problem: Given a set S of n points in R^2, for each point p_i in S what is the set of points (x, y) in the plane that are closer to p_i than any other point in S?

Answer: *Voronoi diagram* of the set S

Intuition: To partition the plane into regions, each of these is the set of points that are closer to a point p_i in S than any other. The partition is based on the set of closest points, e.g. bisectors that have 2 or 3 closest points.

Given two points $\boldsymbol{p_i}$ and $\mathbf{p_j}$, the set of points closest to $\boldsymbol{p_i}$ than $\mathbf{p_j}$ is just the half-plane $Hi(\boldsymbol{p_i}, \mathbf{p_j})$ containing $\boldsymbol{p_i}$. $Hi(\boldsymbol{p_i}, \mathbf{p_j})$ is defined by the perpendicular bisector to the line segment of $\mathbf{p_i p_j}$. The set of points closer to $\mathbf{p_i}$ than any other point is formed by the intersection of at most $n-1$ half-planes, where n is the number of points in the set S. This set of points, $V_i(S)$ is called the "*Voronoi polygon*" associated with $\mathbf{p_i}$. Formally, $V_i(S)$ can be defined as the following:

$$V_i(S)=\{x \text{ in } R^2 \mid d(x, p_i) \leq d(x, q); \text{ all } q \text{ in } S\}$$

The collection of n Voronoi polygons given the n points in the set S is the ***"Voronoi diagram",*** $Vor(S)$, of the point set S. Every point ***(x, y)*** in the plane lies within a Voronoi polygon. If a point ***(x,y)*** in $Vi(S)$, then $\mathbf{p_i}$ is the point nearest to the point ***(x,y)***. The similar idea applies to the same problem in 3D or higher dimensional space.

Generalized Voronoi Diagram

Generalized Voronoi Diagram: the extension of the Voronoi diagram to higher dimensional features (such as edges and facets, instead of points)**;** i.e. the set of points closest to a *feature*, e.g. that of a polyhedron.

FACTS:

- In general, the generalized Voronoi diagram has quadratic surface boundaries in it.

- If the polyhedron is convex, then its generalized Voronoi diagram has planar boundaries.

Voronoi Regions

Definition: A "***Voronoi region***" associated with a ***feature*** is a set of points that are closer to that feature than any other.

FACTS:

- The Voronoi regions form a partition of space outside of the polyhedron according to the closest feature.

- The collection of Voronoi regions of each polyhedron is the generalized Voronoi diagram of the polyhedron.

- The generalized Voronoi diagram of a convex polyhedron has linear size and consists of polyhedral regions. And, all Voronoi regions are convex.

Delaunay Triangulations

The Delaunay triangulation is the *topological dual (graph theoretical dual or combinatorial theoretic dual)* of the Voronoi diagram.

Duality: A point in the plane has two parameters: (x,y). A (non-vertical) line in the plane also has two parameters: its slope and y-intercept. The mapping between a set of point in "primal plane" to its "dual plane" (and vise versa) has a one-to-one correspondence.

Let $p := (p_x, p_y)$. The dual of p is the line defined as $p^* := (y = p_x x - p_y)$. The dual of a line $l := (y = mx + b)$ is the point p such that $p^* = l$. That is, $l^* := (m, -b)$.

The same idea extends to 3D……

FACTS:

- The vertices of the Delaunay triangulation are the sites or Voronoi vertices (corresponding to the regions of the Voronoi diagram).

- We connect two sites with an edge in the Delaunay triangulation if and only if their Voronoi regions share an edge (thus, the two diagrams have the same number of edges, though the Delaunay triangulation has no unbounded edges).

- A point/vertex in the Voronoi diagram corresponds to a Delaunay triangle, a Voronoi edge to a Delaunay edge, and a Voronoi polygon to a Delaunay vertex (site). Note, however, that dual edges of the two graphs do not necessarily intersect.

- We may recover the Voronoi diagram by reversing the process of constructing the Delaunay triangulation.

Tetrahedralization

Tetrahedralization: *Triangulation* of *n* points in 3D. The union of resulting tetrahedra is the original solid formed by the *n* points in 3D. Tetrahedralization is an important process of *convex decomposition*.

NOTE: Held, Klosowski and Mitechell (1995) uses a tetrahedral mesh for checking interference between fly-by objects in VR environments.

Convex Decomposition

Convex decomposition: the process to divide up a non-convex polyhedron into pieces of convex polyhedra.

FACTS:

- Optimal convex decomposition of general non-convex polyhedra can be NP-hard.

- To partition a non-degenerate simple polyhedron takes $O((n + r^2) \log r)$ time, where n is the number of vertices and r is the number of reflex edges of the original non-convex object.

- In general, a non-convex polyhedron of n vertices can always be partitioned into $O(n^2)$ convex pieces.

Linear Programming

In general, a d-dimensional linear programming (or linear optimization) problem may be posed as follows:

Given a finite set A in R^d
For each $\boldsymbol{a}$ in A, a constant $\boldsymbol{K_a}$ in R, $\boldsymbol{c}$ in R^d
Find $\boldsymbol{x}$ in R^d which minimize $<\boldsymbol{x}, \boldsymbol{c}>$
Subject to $<\boldsymbol{a}, \boldsymbol{x}> \geq K_a$, for all $\boldsymbol{a}$ in A .

where $<*, *>$: standard inner product in R^d.

Lower Bound: This problem can be solved in linear time, as proposed by Seidel.

The function to be minimized is the ***objective function*** and the set of constraints together with the objective function is a ***linear program***.

- ⇨ The number of variable, d, is the dimension of linear program.

- ⇨ Linear constraints can be viewed as half-spaces in R^d. The intersection of these half-spaces, which is the set of points satisfying all constraints, is called the ***feasible region*** of the linear program. Points (solutions) in this region are called ***feasible***, points outside are ***infeasible***.

- ⇨ The objective function can be viewed as a direction in R^d, minimizing $\langle \boldsymbol{c},\boldsymbol{x} \rangle$. Therefore, the solution to a linear program is a point in the feasible region that is ***extreme*** in the direction of the vector $\boldsymbol{c}$.

Linear programming can be used to solve the problem of collision detection between two convex polytopes. Each convex polygon (polyhedron) can be viewed as a set of constraints in 2D (3D). We want to find a feasible solution (point) to the following:

Given two finite sets A, B in R^d
For each $\boldsymbol{a}$ in A and $\boldsymbol{b}$ in B,
Find $\boldsymbol{x}$ in R^d which minimize *whatever*
Subject to $\langle \boldsymbol{a}, \boldsymbol{x} \rangle > 0$, for all $\boldsymbol{a}$ in A
And $\langle \boldsymbol{b}, \boldsymbol{x} \rangle < 0$, for all $\boldsymbol{b}$ in B

Where d = 2 (or 3).

Without loss of generality, we can assume that $\boldsymbol{x} = (x_1, \ldots, x_d)$ in R^d with minimum x_d, which would be the case if $c = (0, \ldots, 0, 1)$ in R^d. This can be easily achieved with a coordinate transformation to change the basis. Still retaining the full generality, we can modify the form of constraints by isolating x_d in the expansion of $\langle \boldsymbol{x},\boldsymbol{a} \rangle$.

We obtain three possible types of constraints:

$$x_d \geq a_1 x_1 + \ldots + a_{d-1} x_{d-1} + a_d$$
$$x_d \leq a_1 x_1 + \ldots + a_{d-1} x_{d-1} + a_d$$
$$0 \leq a_1 x_1 + \ldots + a_{d-1} x_{d-1} + a_d$$

The three types of constraints depending on if a_0 is positive, negative or zero, respectively. We partition the set A accordingly, that is $A^+ = \{\boldsymbol{a} \text{ in } A : a_d > 0\}$, $A^- = \{\boldsymbol{a} \text{ in } A : a_d < 0\}$, $A^0 = \{\boldsymbol{a} \text{ in } A : a_d = 0\}$.

Renaming the variables, we arrive at out canonical form for the 2-dimensional linear programming:

Given a finite set A in R^2
Find (x, y) in R^2 which minimize y
Subject To $y \geq a_1x + a_0$, for all $\boldsymbol{a}$ in A^+
$y \leq a_1x + a_0$, for all $\boldsymbol{a}$ in A^-
$0 \leq a_1x + a_0$, for all $\boldsymbol{a}$ in A^0

Note that the three types of constraints correspond to upper half-plane, lower-half plane and "side-way" half-plane constraints on (x, y) respectively.

For 3-dimensional linear programming, instead of looking for a feasible planar point (x, y) with smallest y-coordinate, we search for a feasible point (x, y, z) in R^3 with the smallest z-coordinate. Our constraints are now half-spaces bounded by planes in R^3; we require the solution point to lie in the intersection of a set of such half-spaces.

Two-Dimensional Collision Detection

• Simple Clipping:
Cohen-Sutherland Line-Clipping Algorithm

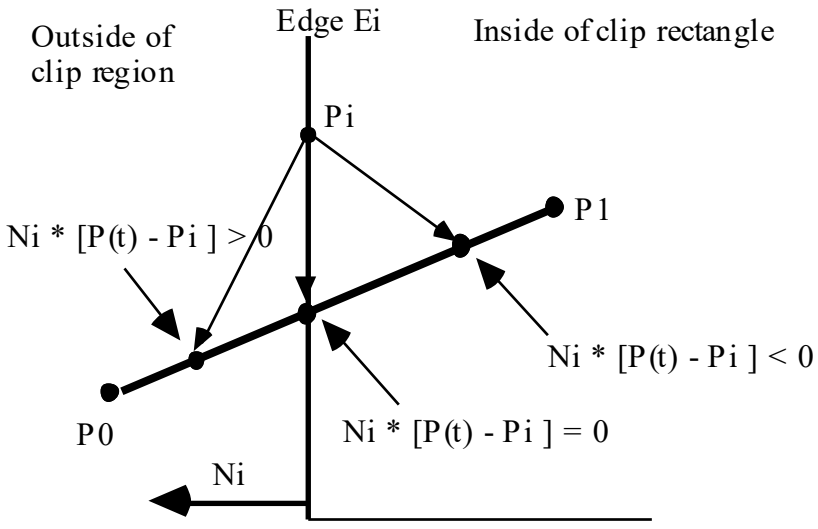
- **Clipping -- check if a given image/object lies within the boundaries of the window screen coordinates**

- $X_{min} \leq X \leq X_{max}$ **and** $Y_{min} \leq Y \leq Y_{max}$

- **Trivial Reject or Accept**

• Cyrus-Beck Techniques (1978):
A Parametric Line-Clipping Algorithm

- **Calculate the value of the parameter *t* , where lines intersect**



Dot product for 3 points outside, inside, and on the boundary of the clip region. The line: *P(t) = P0 + t (P1 - P0)*

```
%%  Pseudo Code for Cyrus Beck Parametric Line-Clipping Algorithm
{
        precalculate Ni and select a Pi for each edge Ei

        for (each line segment to be clipped) {
                if (P1 = P0)
                        line is degenerate so clip as a point;
                else {
                        D = P1 - P0;
                        te = 0;
                        tl  = 1;
                        for (each candidate intersection with a clip edge) {
                            if (Ni * D # 0)  {
                                t =  - { Ni * [P0 - Pi] }  /  (Ni * D)
                                if (Ni * D > 0)
                                    tl = min (tl, t);
                                else
                                    te = max (te, t);
                            }
                        }
                        if (te > tl)
                           return nil;
                        else
                           return P(te) and P(tl) as true clip intersection points;
                }
        }
}
```