Lecture 16: Geometric Preliminaries

Today's material is not discussed in any of our readings.

Geometric Information: A large number of modern data structures and algorithms problems involve geometric data. The reason is that rapidly growing fields such as computer graphics, robotics, computer vision, computer-aided design, visualization, human-computer interaction, virtual reality, and others deal primarily with geometric data. Geometric applications give rise to new data structures problems, which we will be studying for a number of lectures.

Before discussing geometric data structures, we need to provide some background on what geometric data is, and how we compute with it. With nongeometric data we stated that we are storing records, and each record is associated with an identifying *key* value. These key values served a number of functions for us. They served a function of *identification* the the objects of the data structure. Because of the underlying ordering relationship, they also provided a means for searching for objects in the data structure, by giving us a way to direct the search by *subdividing* the space of keys into subsets that are greater than or less than the key.

In geometric data structures we will need to generalize the notion of a key. A geometric point object in the plane can be identified by its (x, y) coordinates, which can be thought of as a type of key value. However, if we are storing more complex objects, such as rectangles, line segments, spheres, and polygons, the notion of a identifying key is not as appropriate. As with one-dimensional data, we also have associated application data. For example, in a graphics system, the geometric description of an object is augmented with information about the object's color, texture, and its surface reflectivity properties. Since our interest will be in storing and retrieving objects based on their geometric properties, we will not discuss these associated data values.

Primitive Objects: Before we start discussing geometric data structures, we will digress to discuss a bit about geometric objects, their representation, and manipulation. Here is a list of common geometric objects and possible representations. The list is far from complete. Let R^d denote d-dimensional space with real coordinates.

Scalar: This is a single (1-dimensional) real number. It is represented as float or double.

Point: Points are locations in space. A typical representation is as a d-tuple of scalars, e.g. $P = (p_0, p_1, \ldots, p_{d-1}) \in R^d$. Is it better to represent a point as an array of scalars, or as an object with data members labeled x, y, and z? The array representation is more general and often more convenient, since it is easier to generalized to higher dimensions and coordinates can be parameterized. You can define "names" for the coordinates. If the dimension might vary then the array representation is necessary.

For example, in Java we might represent a point in 3-space using something like the following. (Note that "static final" is Java's way of saying "const".)

```
class Point {
    public static final int DIM = 3;
    protected float coord[DIM];
    ...
}
```

Vector: Vectors are used to denote direction and magnitude in space. Vectors and points are represented in essentially the same way, as a d-tuple of scalars, $\vec{v} = (v_0, v_1, \ldots, v_{d-1})$.

[1]Copyright, David M. Mount, 2001

It is often convenient to think of vectors as *free vectors*, meaning that they are not tied down to a particular origin, but instead are free to roam around space. The reason for distinguishing vectors from points is that they often serve significantly different functions. For example, velocities are frequently described as vectors, but locations are usually described as points. This provides the reader of your program a bit more insight into your intent.

Line Segment: A line segment can be represented by giving its two endpoints (p_1, p_2). In some applications it is important to distinguish between the line segments $\overrightarrow{p_1p_2}$ and $\overrightarrow{p_2p_1}$. In this case they would be called *directed line segments*.

Ray: Directed lines in 3- and higher dimensions are not usually represented by equations but as rays. A ray can be represented by storing an origin point P and a nonzero directional vector $\vec{v}$.

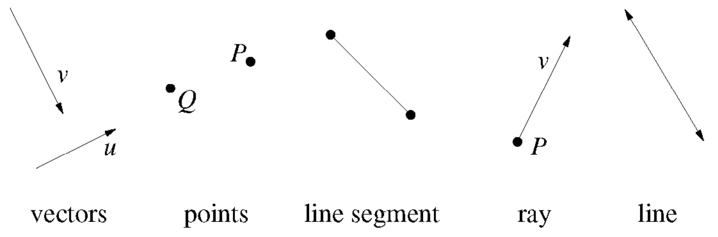


Figure 45: Basic geometric objects.

Line: A line in the plane can be represented by a line equation

$$y = ax + b \qquad \text{or} \qquad ax + by = c.$$

The former definition is the familiar *slope-intercept* representation, and the latter takes an extra coefficient but is more general since it can easily represent a vertical line. The representation consists of just storing the pair (a, b) or (a, b, c).

Another way to represent a line is by giving two points through which the line passes, or by giving a point and a directional vector. These latter two methods have the advantage that they generalize to higher dimensions.

Hyperplanes and halfspaces: In general, in dimension d, a linear equation of the form

$$a_0p_0 + a_1p_1 + \ldots + a_{d-1}p_{d-1} = c$$

defines a $d-1$ dimensional hyperplane. It can be represented by the d-tuple $(a_0, a_1, \ldots, a_{d-1})$ together with c. Note that the vector $(a_0, a_1, \ldots, a_{d-1})$ is orthogonal to the hyperplane.

The set of points that lie on one side or the other of a hyperplane is called a *halfspace*. The formula is the same as above, but the "$=$" is replaced with an inequality such as "$<$" or "$\geq$".

Orthogonal Hyperplane: In many data structures it is common to use hyperplanes that are orthogonal to one of the coordinate axes, called *orthogonal hyperplanes*. In this case it is much easier to store such a hyperplane by storing (1) an integer `cutDim` from the set $\{0, 1, \ldots, d-1\}$, which indicates which axis the plane is perpendicular to and (2) a scalar `cutVal`, which indicates where the plane cuts this axis.

Simple Polygon: Solid objects can be represented as polygons (in dimension 2) and polyhedra (in higher dimensions). A polygon is a cycle of line segments joined end-to-end. It is

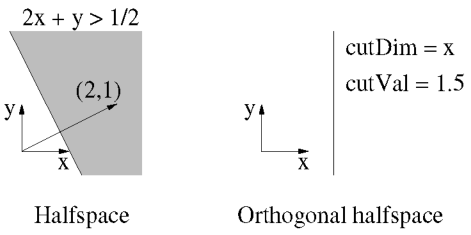


Figure 46: Planes and Halfspaces.

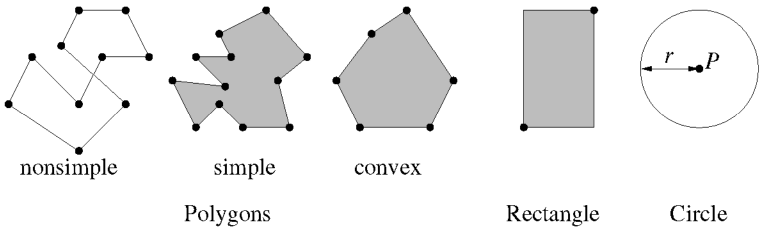


Figure 47: Polygons, rectangles and spheres.

said to be *simple* if its boundary does not self-intersect. It is *convex* if it is simple and no internal angle is greater than π.

The standard representation of a simple polygon is just a list of points (stored either in an array or a circularly linked list). In general representing solids in higher dimensions involves more complex structures, which we will discuss later.

Orthogonal Rectangle: Rectangles can be represented as polygons, but rectangles whose sides are parallel to the coordinate axes are common. A couple of representations are often used. One is to store the two points of opposite corners (e.g. lower left and upper right).

Circle/Sphere: A d-dimensional sphere, can be represented by *point* P indicating the center of the sphere, and a positive *scalar* r representing its radius. A points X lies within the sphere if

$$(x_0 - p_0)^2 + (x_1 - p_1)^2 + \ldots + (x_{d-1} - p_{d-1})^2 \leq r^2.$$

Topological Notions: When discussing geometric objects such as circles and polygons, it is often important to distinguish between what is inside, what is outside and what is on the boundary. For example, given a triangle T in the plane we can talk about the points that are in the *interior* ($int(T)$) of the triangle, the points that lie on the *boundary* ($bnd(T)$) of the triangle, and the points that lie on the *exterior* ($ext(T)$) of the triangle. A set is *closed* if it includes its boundary, and *open* if it does not. Sometimes it is convenient to define a set as being *semi-open*, meaning that some parts of the boundary are included and some are not. Making these notions precise is tricky, so we will just leave this on an intuitive level.

Operations on Primitive Objects: When dealing with simple numeric objects in 1-dimensional data structures, the set of possible operations needed to be performed on each primitive object was quite simple, e.g. compare one key with another, add or subtract keys, print keys, etc. With geometric objects, there are many more operations that one can imagine.

Basic point/vector operators: Let α be any scalar, let P and Q be points, and $\vec{u}, \vec{v}, \vec{w}$ be vectors. We think of vectors as being free to roam about the space whereas points are

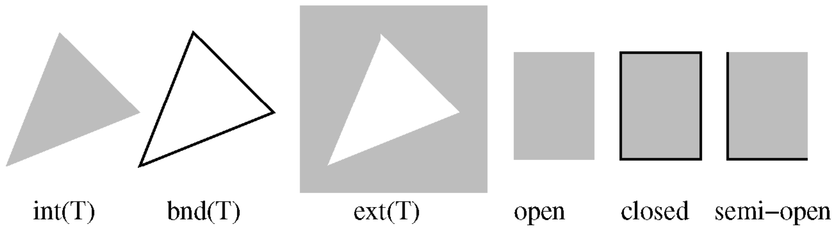


Figure 48: Topological notation.

tied down to a particular location. We can do all the standard operations on vectors you learned in linear algebra (adding, subtracting, etc.) The difference of two points $P - Q$ results in the vector directed from Q to P. The sum of a point P and vector $\vec{u}$ is the point lying on the head of the vector when its tail is placed on P.

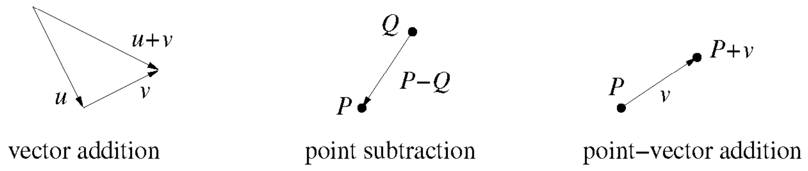


Figure 49: Point-vector operators.

As a programming note, observe that C++ has a very nice mechanism for handling these operations on Points and Vectors using operator overloading. Java does not allow overloading of operators.

Affine combinations: Given two points P and Q, the point $(1 - \alpha)P + \alpha Q$ is point on the line joining P and Q. We can think of this as a weighted average, so that as α approaches 0 the point is closer to P and as α approaches 1 the point is closer to Q. This is called an *affine combination* of P and Q. If $0 \leq \alpha \leq 1$, then the point lies on the line segment $\overline{PQ}$.

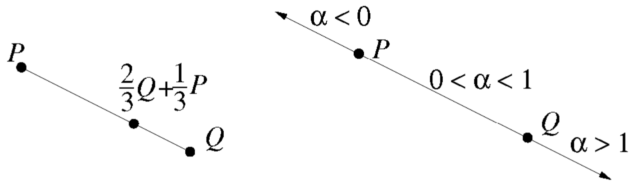


Figure 50: Affine combinations.

Length and distance: The length of a vector v is defined to be

$$||\vec{v}|| = \sqrt{v_0^2 + v_1^2 + \ldots + v_{d-1}^2}.$$

The distance between two points P and Q is the length of the vector between them, that is

$$dist(P, Q) = ||P - Q||.$$

Computing distances between other geometric objects is also important. For example, what is the distance between two triangles? When discussing complex objects, distance usually means the closest distance between objects.

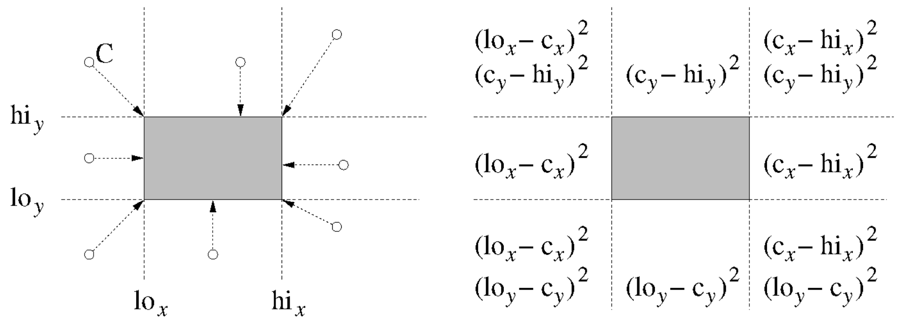
Orientation and Membership: There are a number of geometric operations that deal with the relationship between geometric objects. Some are easy to solve. (E.g., does a point P lie within a rectangle R?) Some are harder. (E.g., given points A, B, C, and D, does D lie within the unique circle defined by the other three points?) We will discuss these as the need arises.

Intersections: The other sort of question that is important is whether two geometric objects intersect each other. Again, some of these are easy to answer, and others can be quite complex.

Example: Circle/Rectangle Intersection: As an example of a typical problem involving geometric primitives, let us consider the question of whether a circle in the plane intersects a rectangle. Let us represent the circle by its center point C and radius r and represent the rectangle R by its lower left and upper right corner points, R_{lo} and R_{hi} (for low and high).

Problems involving circles are often more easily recast as problems involving distances. So, instead, let us consider the problem of determining the distance d from C to its closest point on the rectangle R. If $d > r$ then the circle does not intersect the rectangle, and otherwise it does.

In order to determine the distance from C to the rectangle, we first observe that if C lies inside R, then the distance is 0. Otherwise we need to determine which point of the boundary of R is closest to C. Observe that we can subdivide the exterior of the rectangle into 8 regions, as shown in the figure below. If C lies in one of the four corner regions, then C is closest to the corresponding vertex of R and otherwise C is closest to one of the four sides of R. Thus, all we need to do is to classify which region C lies is, and compute the corresponding distance. This would usually lead a lengthy collection of if-then-else statements, involving 9 different cases.



C
hi_y
lo_y
lo_x
hi_x
$(\mathrm{lo}_x - c_x)^2$ $(c_y - \mathrm{hi}_y)^2$
$(c_y - \mathrm{hi}_y)^2$
$(c_x - \mathrm{hi}_x)^2$ $(c_y - \mathrm{hi}_y)^2$
$(\mathrm{lo}_x - c_x)^2$
$(c_x - \mathrm{hi}_x)^2$
$(\mathrm{lo}_x - c_x)^2$ $(\mathrm{lo}_y - c_y)^2$
$(\mathrm{lo}_y - c_y)^2$
$(c_x - \mathrm{hi}_x)^2$ $(\mathrm{lo}_y - c_y)^2$

Figure 51: Distance from a point to a rectangle, and squared distance contributions for each region.

We will take a different approach. Sometimes things are actually to code if you consider the problem in its general d-dimensional form. Rather than computing the distance, let us first concentrate on computing the squared distance instead. The distance is the sum of the squares of the distance along the x-axis and distance along the y-axis. Consider just the x-coordinates, if C lies to are to the left of the rectangle then the contribution is $(R_{lo,x} - c_x)^2$, if C lies to the right then the contribution is $(c_x - R_{hi,x})^2$, and otherwise there is no x-contribution to the distance. A similar analysis applies for y. This suggests the following code, which works in all dimensions.

Distance from Point C to Rectangle R

```
float distance(Point C, Rectangle R) {
    sumSq = 0                                   // sum of squares
    for (int i = 0; i < Point.DIM; i++) {
        if (C[i] < R.lo[i])                     // left of rectangle
            sumSq += square(R.lo[i] - C[i])
        else if (C[i] > R.hi[i])                // right of rectangle
            sumSq += square(C[i] - R.hi[i])
    }
    return sqrt(sumSq)
}
```

Finally, once the distance has been computed, we test whether it is less than the radius r. If so the circle intersects the rectangle and otherwise it does not.