

# Computational Geometry Lab: SEARCHING A TRIANGULATION

John Burkardt  
Information Technology Department  
Virginia Tech

[http://people.sc.fsu.edu/~burkardt/presentations/cg\\_lab\\_search\\_triangulation.pdf](http://people.sc.fsu.edu/~burkardt/presentations/cg_lab_search_triangulation.pdf)

August 21, 2009

## 1 Introduction

This lab continues the topic of *Computational Geometry*. Having studied triangles and triangulations, we will now turn to the question of *searching*. In particular, we will suppose that we have a computational region  $\mathcal{R}$ , and a set of points  $\mathbf{P}$  which are contained in  $\mathcal{R}$ . (Actually, we will also allow the possibility that some points are outside the region.)

We further suppose that we have constructed a triangulation  $\tau$  of  $\mathcal{R}$ , consisting of **t\_num** triangles, with a typical triangle indicated by  $T_i$ . We now wish to know, for each point  $p_i$ , the identity of the triangle  $T_i$  that contains the point. If the point is shared by several triangles, it is sufficient to give just one triangle. If the point does not lie in any of the triangles, then we may be satisfied with a flag value of “-1”, or perhaps we would be interested in the index of a triangle that is near to the point.

We already know, from a previous lab, how to determine whether a point lies inside a triangle. So if we have a triangulation, and ask which triangle contains a point, we could simply check each triangle, stopping as soon as we find one. This would mean that on average we’d have to check about  $\frac{t\_num}{2}$  triangles.

It’s hard to imagine, but it’s possible to carry out this task much faster. There is a method called the *Delaunay search*, which makes roughly only  $\sqrt{t\_num}$  checks to locate a point in a triangulation. It is not unusual for a triangulation to have a million triangles, in which case the Delaunay method is about 500 times faster than the straightforward searching method. And the advantage becomes even greater when we go to a 3D mesh.

This faster method only works on special triangulations, namely, those that have the *Delaunay property*, which is explained and discussed in another lab. Moreover, a certain amount of preprocessing must be done in order to record which triangles are neighbors of each other. However, once that is done, the algorithm works beautifully, starting with a random tetrahedron as a first guess, and then “walking” from one triangle to the next, more or less directly towards the correct one.

### 1.1 Where is a Point Relative to a Triangle?

Let us recall information from the lab on barycentric coordinates for triangles. We suppose we have a triangle  $\mathbf{T}=\{\mathbf{a},\mathbf{b},\mathbf{c}\}$  and a point  $\mathbf{p}$ . We presume we have a function **Area**(**a,b,c**) for the area of the triangle formed

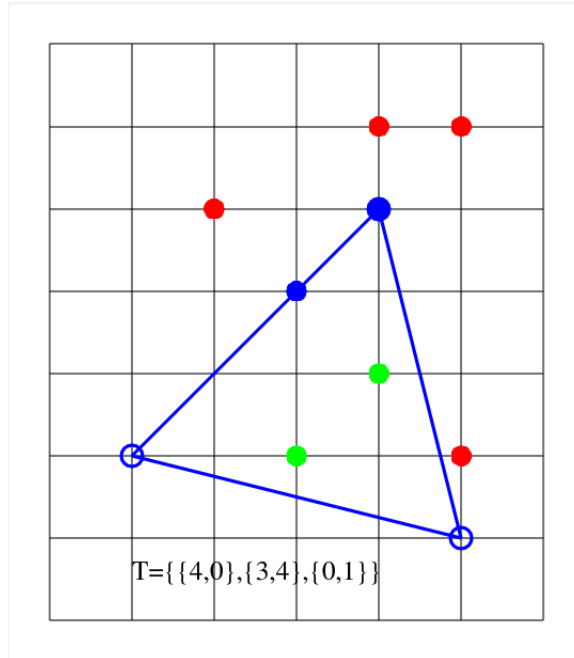


Figure 1: The Point-Triangle Orientation Test for Triangle Tex4.

by any three points. The barycentric coordinates of  $\mathbf{p}$  are defined as:

$$\xi_a = \frac{\text{Area}(p, b, c)}{\text{Area}(a, b, c)}$$

$$\xi_b = \frac{\text{Area}(a, p, c)}{\text{Area}(a, b, c)}$$

$$\xi_c = \frac{\text{Area}(a, b, p)}{\text{Area}(a, b, c)}$$

It should be clear that the sum of the barycentric coordinates is 1. Moreover, the values of the coordinates tell us about the position of  $\mathbf{p}$ . If  $\mathbf{p}$  is

- **inside** the triangle, then all the coordinates are positive;
- **on the boundary**, one coordinate is zero;
- **a vertex**, two coordinates are 0, and one is 1;
- **outside** the triangle, then one or perhaps two of the coordinates will be negative.

In particular, if  $\xi_a$  is negative, then relative to the edge  $\{\mathbf{b}, \mathbf{c}\}$ , the point  $\mathbf{p}$  lies on one side and vertex  $\mathbf{a}$  on the other. Similar statements hold for the other two barycentric coordinates.

## 1.2 Program #1: Where is a Point Relative to a Triangle?

Write a program which

- reads the definition of a triangle  $\mathbf{T}=\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ ;
- reads a point  $\mathbf{p}$ ;

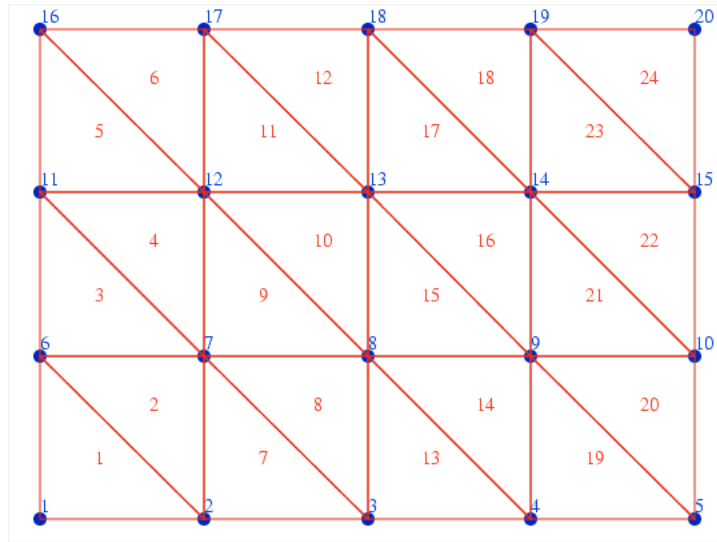


Figure 2: The **Box3** triangulation (element indices in red, node indices in blue ).

- computes the barycentric coordinates  $(\xi_a, \xi_b, \xi_c)$  of  $\mathbf{p}$  with respect to  $\mathbf{T}$ ;
- prints  $\mathbf{p}$ , the barycentric coordinates, and the word “inside” or “outside”, depending on whether  $\mathbf{p}$  is inside  $\mathbf{T}$  or not.

Use the triangle **Tex4** defined by  $\{\{4,0\},\{3,4\},\{0,1\}\}$ . For  $\mathbf{p}$ , use the points (1,4), (2,1), (2,3), (3,2), (3,4), (3,5), (4,1) and (4,5) to see if your program can detect inside/outside/on.

### 1.3 Box3, an example triangulation

For some of the coming exercises, we will want to have a simple triangulation to work with. We will use the **Box3** triangulation, which is illustrated and defined here.

The node coordinate array for **Box3** is

```

0.0  0.0
1.0  0.0
2.0  0.0
3.0  0.0
4.0  0.0
0.0  1.0
1.0  1.0
2.0  1.0
3.0  1.0
4.0  1.0
0.0  2.0
1.0  2.0
2.0  2.0
3.0  2.0
4.0  2.0
0.0  3.0
1.0  3.0
2.0  3.0

```

```
3.0 3.0
4.0 3.0
```

The element node array for **Box3** is:

```
6  1  2
7  6  2
7 11  6
12 11  7
16 11 12
16 12 17
7  2  3
8  7  3
8 12  7
13 12  8
17 12 13
17 13 18
8  3  4
9  8  4
9 13  8
14 13  9
18 13 14
18 14 19
9  4  5
10 9  5
10 14  9
15 14 10
19 14 15
19 15 20
```

#### 1.4 Where is a Point Relative to a Triangulation?

Now suppose we have a triangulation  $\tau$  and a point  $\mathbf{p}$ . How do we determine the index of the triangle  $\mathbf{T}_i$  that contains the point? How quickly can we do this?

The obvious approach, known as the *naive search*, simply checks each triangle. Such a search can take up to  $\mathbf{t\_num}$  steps. If the point is guaranteed to be in at least one of the triangles, and if we stop as soon as we find that triangle, the average number of steps is actually  $\frac{\mathbf{t\_num}}{2}$ .

When we “check” a triangle  $\mathbf{T}_i$  to see if it contains  $\mathbf{p}$ , we actually simply compute the barycentric coordinates of  $\mathbf{p}$  with respect to  $\mathbf{T}_i$ . If all 3 coordinates are nonnegative, we have found our triangle. Otherwise, we prepare to check the next triangle in the list.

#### 1.5 Program #2: Where is a Point Relative to a Triangulation?

Write a program which

- Reads the element node and node coordinate arrays of a triangulation  $\tau$ ;
- Reads a point  $\mathbf{p}$ .
- Performs the naive search algorithm to find the triangle containing  $\mathbf{p}$ ;
- Prints the triangle index, and the number of steps required.

For your triangulation, use the **Box3** example. For your point  $\mathbf{p}$ , try each of the following:  $\{2,5,1.8\}$ ,  $\{3.8,2.9\}$ ,  $\{0.0,1.6\}$ ,  $\{4.0,0.0\}$ ,  $\{5.0,2.5\}$ .

It's possible that a point is not contained in *any* triangle. In that case, let's print out the value “-1” in place of the triangle index.

## 1.6 The Neighbor Triangle Array

In order to apply a faster method for finding points in a triangulation, we must define the *neighbor triangle array*, a crucial piece of information that will help us find our way to the answer.

The neighbor triangle array is easily defined. If we think about the edges in a triangulation, most of them are shared by two triangles. The exceptions occur if an edge lies on the exterior boundary, or an internal hole of the region. So, given any triangle in the triangulation, we can ask, for each of its edges whether there is another triangle that shares that edge, and if so what the index of that triangle is. It will be convenient to call this array **neighbor**, and to assume it has the shape  $3 \times \mathbf{t\_num}$ . We will also order the neighbors of triangle  $\mathbf{t}$  so that **neighbor(1,t)** is the neighbor across the edge  $\{\mathbf{b},\mathbf{c}\}$ , (and hence opposite the first node,  $\mathbf{a}$ ). Similarly, **neighbor(2,t)** is the neighbor that shares edge  $\{\mathbf{c},\mathbf{a}\}$ , and **neighbor(3,t)** shares  $\{\mathbf{a},\mathbf{b}\}$ . It is of course possible that there is *no* neighbor that shares a given edge. In that case, we can give that array entry the special value of -1.

Consider the simple **Box3** triangulation shown in the figure. If the nodes of element 9 are listed as  $\{8, 12, 7\}$ , then the 9th row of the neighbor array will be  $\{4,8, 10\}$ . Similarly, if the nodes of element 18 are listed as  $\{18,14,19\}$ , then the 18th row of the neighbor array will be  $\{23,-1,17\}$ .

If we have a picture of the triangulation, we can see the neighbor values easily. But could we *compute* this information, having only the element node array to work with?

Imagine that we needed to know the neighbors of element 18. Triangle 18 has the edges  $\{14,19\}$ ,  $\{19,18\}$  and  $\{18,14\}$ . To find the neighbors of triangle 18, we have only to search the element node array for the occurrence of these edges. Of course, when looking for a neighbor along the edge  $\{14,19\}$ , we first have to reverse the order of the nodes, since the edge has the opposite orientation in the neighbor triangle. Then we have to realize that the nodes 19 and 14 can show up in the element node array in any one of three ways:

```
{ 19, 14,  ? }
{  ? , 19, 14 }
{ 14,  ? , 19 }
```

So we can look for occurrences of the node 19 anywhere in the element node array, and if we find one, ask whether the “next” entry is 14, remembering that if 19 occurs in the 3rd column, then the “next” element is in column 1.

By considering each triangle, and each edge of that triangle, and searching the element node array for occurrences of the reverse of that edge, we can construct the entire neighbor array.

## 1.7 Program #3: The Neighbor Triangle Array

Write a program which

- Reads the element node array of a triangulation  $\tau$ ;
- Constructs the triangle neighbor array **neighbor**;
- Prints the triangle neighbor array.

For input to your program, use the **Box3** triangulation.

## 1.8 Where is a Point Relative to a Delaunay Triangulation?

Now let us see whether we can come up with a faster answer to our question about finding the triangle in a triangulation that contains a given point. It turns out that we can only promise a faster answer if we know that our triangulated region is convex, and that the triangulation is a Delaunay triangulation. These matters are discussed in the lab on Delaunay triangulation; for this discussion, we will assume that these conditions are met. Note, however, that the convexity condition means, in particular, that the triangulation is guaranteed not to have any internal holes, or indentations in the outer boundary.

The Delaunay search algorithm can be thought of as “walking” through the triangulation, from one triangle to a neighboring one, until the correct one is found. We start at a random triangle. Our movement strategy rests on two key points:

- the barycentric coordinates tell us which direction we should move;
- the neighbor array tells us what triangle lies in each direction.

Let’s make this more clear by imagining that our triangulation is the **Box3** example, that the point  $\mathbf{p}$  is the centroid of triangle **t16**, and that as our random starting point, we choose triangle **t8**.

Our walk begins by asking whether  $\mathbf{p}$  actually lies in triangle **t8**. We answer this question by computing the barycentric coordinates of  $\mathbf{p}$  relative to **t8**. It turns out that two of the coordinates are negative, corresponding to the edges  $\{3,8\}$  and  $\{8,7\}$ . That means that we are on the wrong side of both of those edges. We may choose to move either north or east. Let’s say we choose to go north. The neighbor array tells us that we have now moved to triangle **t9**. Again we compute the barycentric coordinates of  $\mathbf{p}$ . This time, only the coordinate corresponding to edge  $\{8,12\}$  is negative, so our only choice is to move east. The neighbor array tells us we’ve moved to triangle **t10**. We continue in this way until we reach **t8**, when the barycentric coordinates of  $\mathbf{p}$  are all positive, so that we know we’ve reached our goal.

If we ever try to move in a particular direction, and discover that the neighbor array value is -1, then our search has terminated, and the point  $\mathbf{p}$  actually lies outside the triangulation. That is only true because we are assuming the triangulation is convex.

The walking procedure described here can fail if the triangulation does not have the Delaunay property. In that case, the walk may end up going in a cycle instead of reaching the goal.

To estimate the efficiency of the algorithm, we can start by supposing that we have  $t\_num$  triangles, that a typical triangulation has a “width” and “height” of about  $\sqrt{t\_num}$ , and a random starting triangle might therefore be on average about  $\frac{\sqrt{t\_num}}{2}$  triangles distant from the correct one. Thus, the Delaunay walking algorithm might take about  $\frac{\sqrt{t\_num}}{2}$  steps, whereas the naive search will typically search  $\frac{t\_num}{2}$  triangles.

## 1.9 Program #4: Where is a Point Relative to a Delaunay Triangulation?

Write a program which

- Reads the element node and node coordinate arrays of a triangulation  $\tau$ ;
- Constructs the triangle neighbor array **neighbor**;
- Reads a point  $\mathbf{p}$ .
- Performs the Delaunay search algorithm to find the triangle containing  $\mathbf{p}$ ;
- Prints the number of steps required.

For your triangulation, again use the **Box3** example. For your point  $\mathbf{p}$ , try each of the following:  $\{2,5,1.8\}$ ,  $\{3.8,2.9\}$ ,  $\{0.0,1.6\}$ ,  $\{4.0,0.0\}$ ,  $\{5.0,2.5\}$ .