

Path-planning by Tessellation of Obstacles

Tane Pendragon and Lyndon While
School of Computer Science & Software Engineering,
The University of Western Australia,
Western Australia 6009
email: {pendrt01, lyndon}@csse.uwa.edu.au

Abstract

We describe two algorithms for static path-planning, with the aim of deriving the trajectory that always maximises the distance of the path from the nearest obstacle. The *bubble algorithm* allocates individual pixels to generating points, growing the area owned by each point at a globally constant rate to ensure that boundaries are equidistant between neighbouring points. The *geometric algorithm* derives the same “perfect path”, but without examining pixels individually, and thus with good performance. It compares geometric elements pairwise to generate segments of the boundaries, then intersects these segments to derive the perfect path. The geometric algorithm is the lowest-complexity algorithm yet described that derives this perfect path.

Keywords: path-planning, tessellation, geometry

1 Introduction

Path-planning(Choset, 1998) can be described as the task of navigating a mobile robot around a space in which lie a number of obstacles that have to be avoided. Path-planning can be *static*, where the positions of the obstacles are fixed and a map for the whole space is produced before motion begins; or *dynamic*, where obstacles can move, appear, or disappear, and the map must be updated “on-the-fly” accordingly. Path-planning algorithms sometimes assume that the robot is a point, and sometimes assume that the robot has a non-zero size in space. In this paper, we shall assume static path-planning and we shall pre-empt the size question by defining the *perfect path* between two obstacles to be the trajectory that always maximises the robot’s distance from the two obstacles, thus permitting the biggest possible circular robot to follow the path.

Tessellation(Ogniewicz and Kubler, 1995) is the process of dividing up a space between a number of generating points that lie in the space. Each discrete point (or *pixel*) P in the space is allocated to the generating point nearest to P . Two generating points A and B are said to be *neighbours* if there exists a pixel P in the space such that P is equidistant from A and B and P is closer to A and B than to any other generating point. The *Voronoi diagram* of the space displays the boundaries between generating points which are neighbours (see, for example, Figure 1(d)).

If the generating points in the space represent all

points on the edges¹ of obstacles, there is a clear correspondence between the definition of perfect path given above and the Voronoi diagram of the space. Tessellation has been used previously as the basis of 2D path planning for nonholonomic robots in environments with obstacles(Mirtich and Canny, 1992), and for efficient path planning(Nawawi et al., 1999; Wager, 2000).

In this paper we describe two algorithms that use tessellation to derive the perfect path through a space.

Bubble tessellation The bubble algorithm works by “growing” the areas “owned” by the generating points at a globally constant rate until every pixel in the space has been allocated. Each pixel P is allocated to the first generating point to “visit” P . Pixels on the boundary between two generating points A and B are visited simultaneously by A and B , so the Voronoi diagram is easy to generate and clearly correct. Equally clear is that the bubble algorithm is horrendously inefficient: its performance is at least linear in the number of pixels in the space. However, the bubble algorithm has two virtues which are useful to us: it is obviously a correct tessellation algorithm, and it extends easily to tessellate between more-complex features such as lines and obstacles.

Geometric line tessellation The geometric algorithm builds the boundaries between geometric elements constructively, comparing the coordinates of elements pairwise to build up a set of “candidate boundaries”, then choosing amongst these candidates to finalise the Voronoi diagram. This leads to an algorithm whose asymptotic behaviour is much better than the bubble algorithm, but which achieves the same end, i.e. a perfect path.

This geometric algorithm is significant because it is the lowest-complexity algorithm yet described that derives the perfect path. Previous algorithms with comparable complexity have either placed some constraint on the space (for example, they have assumed convex obstacles), or they have derived an approximation to the perfect path. McAllister et al. (1993) present an extension of the Sweepline Algorithm (Fortune (1987)) which generates a piecewise-linear approximation to the tessellation between convex polygonal sites. This runs in $O(n \log(n))$ time, but cannot handle concave sites and produces only an approximation to the perfect path. Mayya and Rajan (1994) describe an algorithm for generating the Voronoi diagrams of polygons. By representing the reachable space of the world as a polygon (possibly containing holes), their algorithm may be used to generate

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at the Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

¹Note edges, not just vertices.

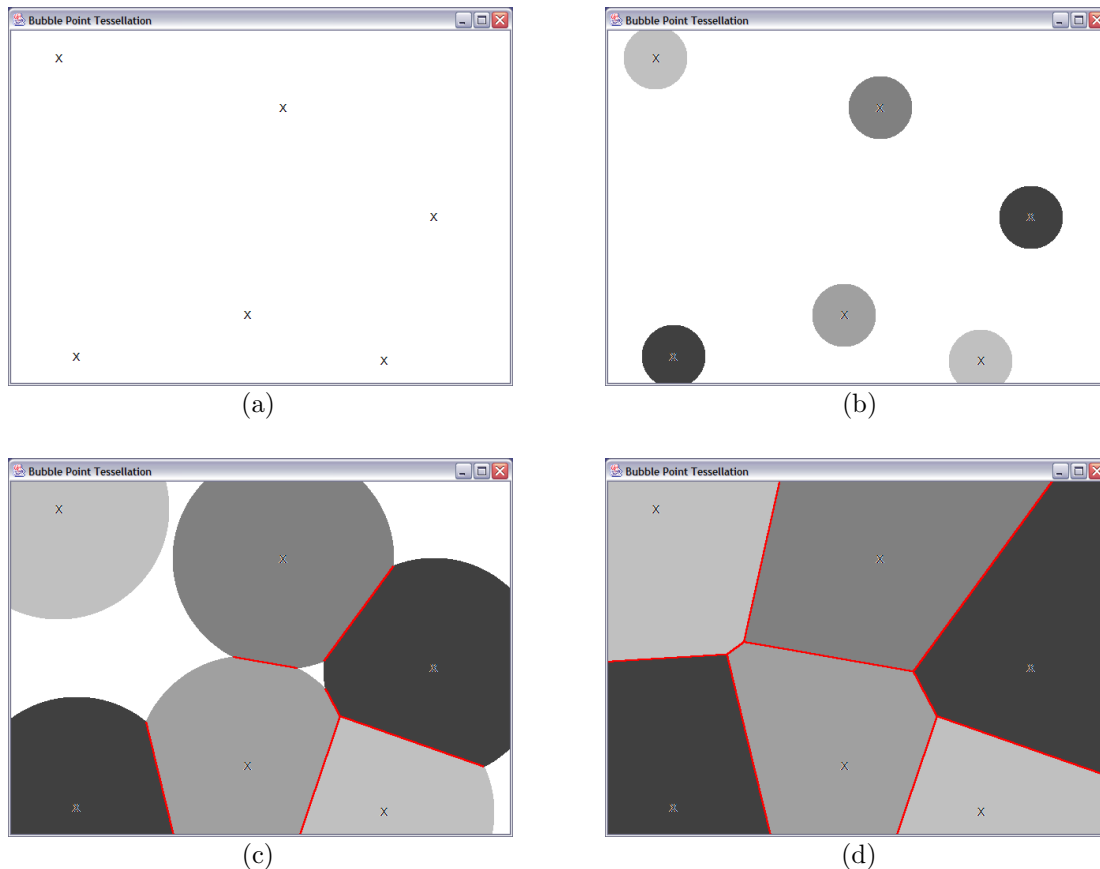


Figure 1: Point bubble tessellation in action. The crosses denote the generating points. The areas owned by the points grow at a globally constant rate.

the tessellation of the world. This algorithm runs in $O(B^2)$ time, where B is the average number of elements in the bounding contour of a site. The algorithm generates perfect paths, but it is not suitable for worlds where the reachable space is unbounded.

The remainder of the paper is organised as follows. Section 2 introduces some terminology that we use in the paper. Section 3 describes the bubble algorithm for point tessellation and discusses its complexity. Section 4 extends the bubble algorithm to tessellate between obstacles, giving the perfect path through a space. Section 5 extends the bubble algorithm to tessellate between lines, again giving the perfect path. Section 6 describes a more efficient geometric algorithm for line tessellation, shows that this algorithm too generates the perfect path, and discusses its complexity. Section 7 concludes the paper and discusses some possible improvements to the geometric algorithm.

2 Terminology

Throughout the paper, we use the following terms.

pixel A pixel is a set of co-ordinates in space.

point A point is a generating point for a tessellation. For the purposes of path-planning, a point will be a vertex of an obstacle.

line A line joins together two points. Lines are also used as “generating points” for a tessellation. For the purposes of path-planning, a line will be an edge of an obstacle.

obstacle An obstacle is a polygon in space that should be avoided by the robot. Obstacles are also used as “generating points” for a tessellation.

perfect path The perfect path between two obstacles is the trajectory that always maximises the robot’s distance from the two obstacles.

3 Point tessellation with the Bubble Algorithm

In a Voronoi diagram, each point A owns the pixels that are closer to A than to any other point. The bubble algorithm “grows” the area owned by each point outwards from the point itself until every pixel in the space has been allocated to a point. The areas owned by the points grow simultaneously at a globally constant rate to ensure that the final boundaries are equidistant from the relevant points. Initially, the area owned by each point is circular. When two areas meet, they stop growing directly towards one another, but they keep growing in other directions, and eventually they form the (straight line) boundary between the two corresponding points. The process stops when every pixel in the space is owned by some point.

This process is illustrated in Figure 1. In the figures, ownership of a pixel is denoted by the colour of the pixel. Figure 2 describes the algorithm in pseudocode. The use of distances (d) in the algorithm ensures that areas grow equally in all directions (i.e. as a circle) on a rectangular pixel array.

Bubble tessellation is horrendously inefficient: it is

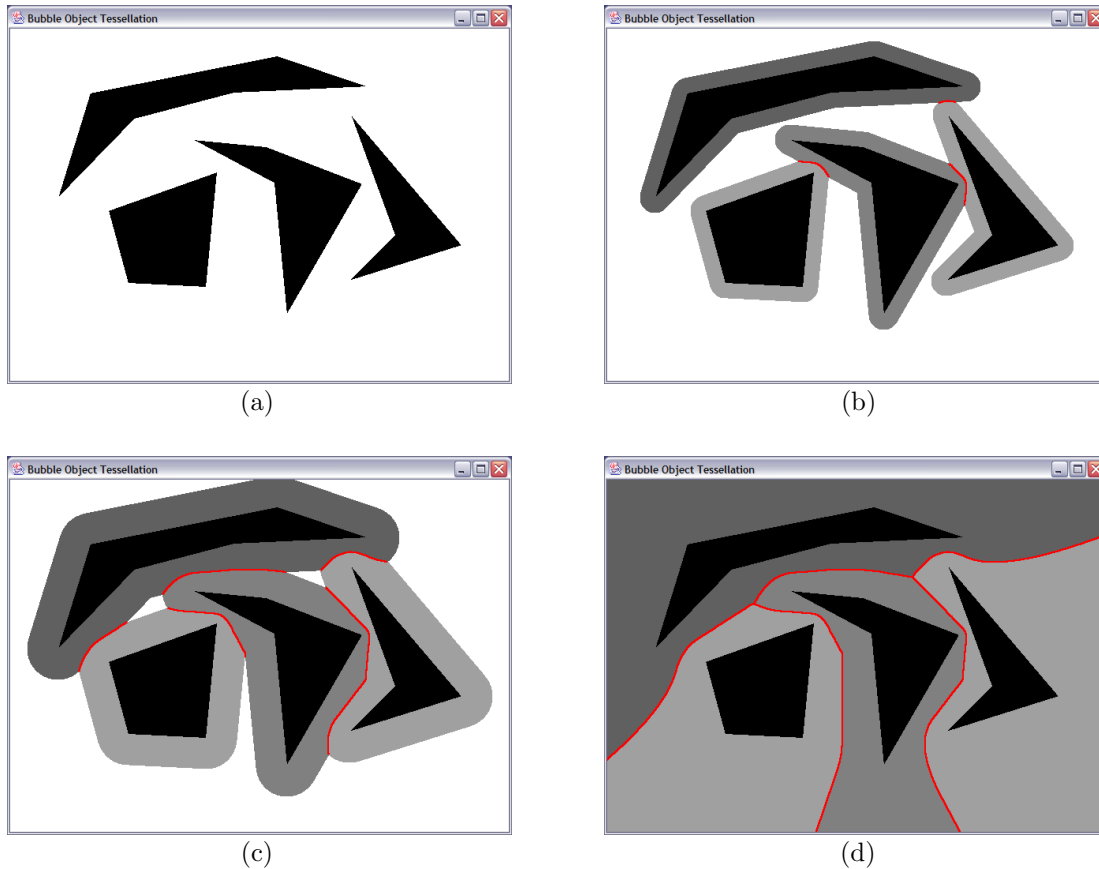


Figure 3: Obstacle bubble tessellation in action. The black areas denote the obstacles. The areas owned by the obstacles grow at a globally constant rate.

```

make each pixel uncoloured
assign a different colour to each point
d := 0
repeat
  d := d + 1
  for each uncoloured pixel p
    if p is within distance d of a point with colour c
      then give p the colour c
until all pixels are coloured

```

Figure 2: Point bubble tessellation in pseudo-code.

at least linear in the number of pixels in the space², which is usually far greater than the number of points. However, bubble tessellation has two virtues:

- it is a very good algorithm for visualising the process of tessellation: the fact that the areas owned by the points grow at the same rate means that the algorithm is “obviously” correct; and
- it extends naturally to an algorithm that is more useful for path-planning.

4 Obstacle tessellation with the Bubble Algorithm

The bubble algorithm applies almost without modification to the problem of tessellating between obstacles. The only difference is in the initialisation phase: all of the vertices and edges on the obstacles

²The version described here has an even worse complexity.

are viewed as generating points, and all of the vertices and edges on one obstacle are assigned the same colour. The area owned by each obstacle grows in the same way as before, outwards from the obstacle itself at a globally constant rate, until every pixel in the space has been allocated to an obstacle. The final area for each obstacle is the part of the space owned by that obstacle, and the edge of this area is the path to follow around the obstacle to maximise the clearance for the robot, i.e. the perfect path as defined above.

This process is illustrated in Figure 3. Again, ownership of a pixel is denoted by the colour of the pixel. Figure 4 describes the algorithm in pseudo-code.

```

make each pixel uncoloured
assign a different colour to each obstacle
d := 0
repeat
  d := d + 1
  for each uncoloured pixel p
    if p is within distance d of an obstacle with colour c
      then give p the colour c
until all pixels are coloured

```

Figure 4: Obstacle bubble tessellation in pseudo-code.

It should be clear that this process is guaranteed to give the perfect path around each obstacle: because the areas owned by the obstacles all grow at the same rate, boundaries derived between two obstacles must be equidistant from the obstacles. However, the complexity is still very poor.

We would like to be able to define an efficient (geo-

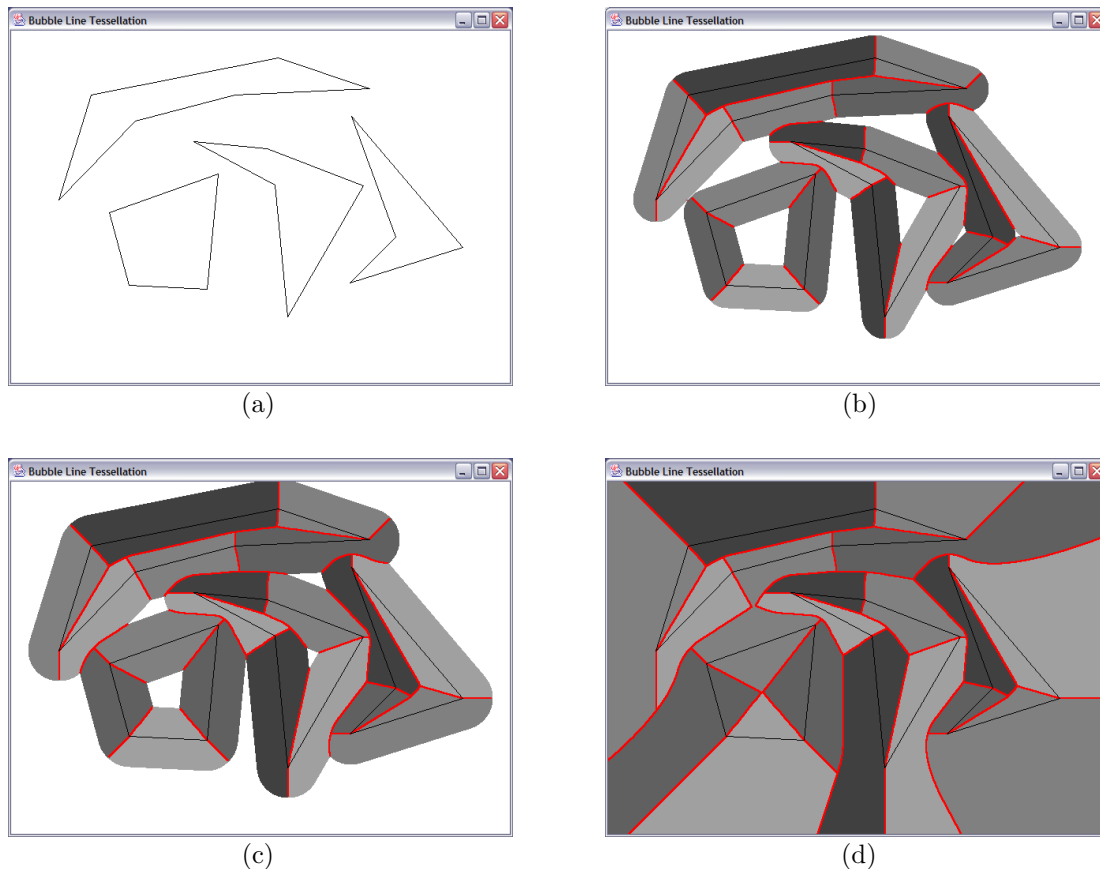


Figure 5: Line bubble tessellation in action. The black lines (most easily seen in (a)) denote the lines. The areas owned by the lines grow at a globally constant rate.

metric) tessellation algorithm between obstacles, but it is difficult to define an algorithm that derives the perfect path under our definition. An alternative approach is to tessellate between lines representing the edges of obstacles, so we first extend the bubble algorithm to deal with lines, then use that as the basis for developing an efficient geometric algorithm to tessellate between lines.

5 Line tessellation with the Bubble Algorithm

Direct efficient obstacle tessellation is hard, so we resort to line tessellation instead. In this section we present a line bubble tessellation algorithm, then in Section 6 we present an efficient geometric line tessellation algorithm.

Line bubble tessellation is again only a minor variation on point bubble tessellation. The only change from point bubble tessellation is in the initialisation phase: all of the pixels on the lines are viewed as generating points, and all of the pixels on one line are assigned the same colour. The “growing” phase of the algorithm is unchanged.

This process is illustrated in Figure 5. Again, ownership of a pixel is denoted by the colour of the pixel. Figure 6 describes the algorithm in pseudo-code.

This algorithm returns a superset of the boundaries returned by the obstacle tessellation algorithm. Simply ignoring the boundaries between lines on the same obstacle again gives the perfect path through the space. Another difference is that the path around an obstacle is returned in segments, one segment corresponding to each line on the obstacle. A simple way

```

make each pixel uncoloured
assign a different colour to each line
d := 0
repeat
  d := d + 1
  for each uncoloured pixel p
    if p is within distance d of a line with colour c
      then give p the colour c
until all pixels are coloured

```

Figure 6: Line bubble tessellation in pseudo-code.

to see this is to assign a different base colour to each obstacle and to assign the different lines on the obstacle different shades of the obstacle’s base colour. The perfect path then corresponds to the boundaries between different base colours.

We now have a bubble algorithm that gives the perfect path with poor performance, but which can be used as the basis of a geometric algorithm that gives the same path with acceptable performance. We use the idea of returning segmented paths to build the perfect path constructively without examining individual pixels.

6 Geometric line tessellation

To tessellate between lines, we break each line into three ‘elements’: the two points at the ends of the line, and the line itself, not including its endpoints.

For each element E of line segment L we first calculate the subset S_E of the space in which pixels are closer to E than to any other element of L . For example, if E is a line segment (not including endpoints), S_E contains all pixels whose projection onto the infinite extension of L are also on L . The division of space for a line AB is shown in Figure 7.

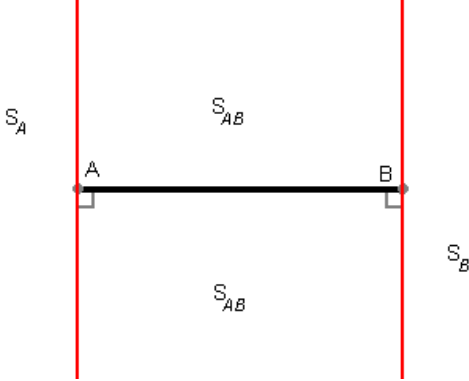


Figure 7: Space division for a line AB .

The next step is to loop through all elements E' on the other lines (i.e. not L), generating the boundaries between E and E' . This boundary is the equidistance curve between E and E' , i.e. the curve along which all pixels are equidistant from E and E' . Equidistance curves fall into four categories.

Point to Point (Figure 9(a)) The equidistance curve between two points is the perpendicular bisector of the line segment joining the two points.

Point to Line (Figure 9(b)) The equidistance curve between a point and a line is a section of parabola, with the directrix lying along the line and the focus being the point, by definition of a parabola. The start and end points of the section are perpendicular to the start and endpoints of the line.

Line to Line (Figure 9(c)) The equidistance curve between two lines is a section of the line through the intersection of the lines, which bisects the angle between the two lines. The section of the line which forms the equidistance curve is the section within which every pixel has a projection onto both defining line segments.

Line ‘pointing at’ Line (Figure 9(d)) When the intersection of the lines lies on one of the line segments, a second curve (corresponding to the perpendicular to the first curve, at the intersection of the two lines) is also required. The section of this line which forms an equidistance curve is determined in the same way as for the simple ‘Line to Line’ case.

Figure 10(a) shows the curves produced for one of the lines on the top obstacle, and Figure 10(c) shows the curves produced for all of the lines on the top obstacle.

Now let U be the union of the pixels on the equidistance curves for E . We then choose $MIN_E \subseteq U \cap S_E$

such that no pixel in U lies on the perpendicular between E and any pixel in MIN_E . Formally:

$$\forall p(p \in U \cap S_E \wedge \neg \exists q(q \in U \wedge q \text{ between } p \text{ and } E) \rightarrow p \in MIN_E)$$

This set of pixels MIN_E forms the part of the tessellation for which E is directly responsible. Figure 10(b) shows the boundary produced for one of the lines on the top obstacle, and Figure 10(d) shows the boundary produced for all of the lines on the top obstacle.

To calculate MIN_E from U , we begin with an empty set of curves C . For each curve K in U , we *merge* K with C . This operation is linear in the number of curve sections in C .

To perform the merge operation, we first find K' by removing all sections of K which are separated from the generating element E by any portion of C . If K' is non-empty, we then perform the reverse operation, and we find C' by removing all sections of C which are separated from E by K' . The resulting set of curves $C' \cup K'$ becomes the new boundary C , and when all of the curves from U have been merged, MIN_E is set to C .

The union of MIN_E for all elements E in the space is the tessellation between the lines in the space. Figure 10(e) shows the path produced for the entire space. This should be compared with Figure 3(d).

6.1 Refinements for path-planning

For the purposes of path-planning, lines are arranged into polygons representing obstacles. If E is an endpoint of a line, we can limit S_E to include only space which is closer to E than to either of the line segments neighbouring E . We also remove from S_E all points which are either inside the obstacle, or separated from E by any part of the obstacle. This removes all sections of the graph which intersect, or lie ‘behind’, portions of the obstacle. The division of space for an obstacle $ABCD$ is shown in Figure 8.

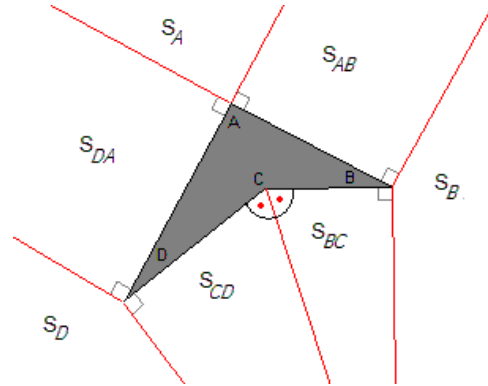


Figure 8: Space division for an obstacle $ABCD$.

Also, when generating U , we omit equidistance curves between segments on the same obstacle.

Taken together, these two refinements limit the set of curves generated by the algorithm to be exactly the tessellation between obstacles, hence providing the perfect path through the space.

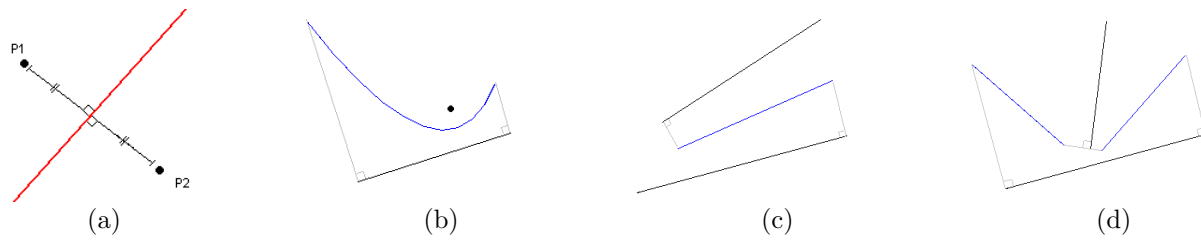


Figure 9: The four classes of equidistance curves.

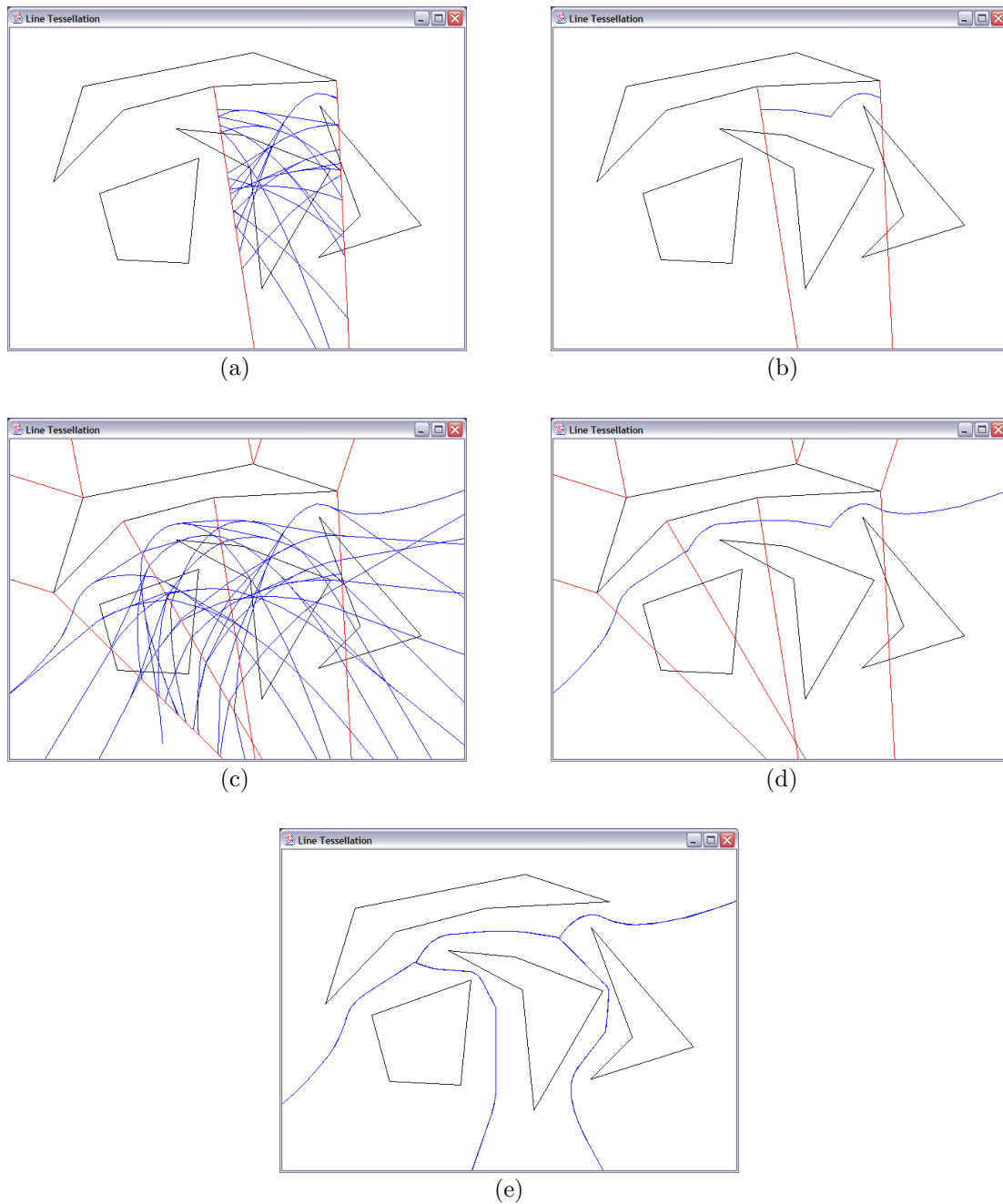


Figure 10: Geometric line tessellation in action. The hollow areas denote the obstacles and the lines radiating from the top obstacle show the space division for that obstacle.

6.2 Complexity of geometric line tessellation

Let n be the number of lines in the world. Then there are $O(n)$ elements in the world, and $O(n^2)$ pairs of elements. For each pair of elements we:

1. Generate an equidistance curve K : $O(1)$
2. Merge K with the current boundary C : $O(n)$

The merge with the current boundary is $O(n)$ in the worst case, because in principle the boundary can contain $O(n)$ curve segments. Hence the worst-case complexity of the geometric algorithm is $O(n^3)$.

However, cases in which C contains a large number of segments are rare, and for the sorts of obstacles we have investigated, C generally does not grow with n . Thus the average complexity of this algorithm is $O(n^2)$. Note that the length of C (during the merging process) for an element E depends to some extent on the order in which the equidistance curves involving E are merged. Thus the average case performance can be improved by sorting the equidistance curves before merging them.

7 Conclusions

We have described and implemented two path-planning algorithms based on tessellation that derive the perfect path through a space, in the sense of maximising the clearance for the robot at all times. The bubble algorithm is clearly correct but has a very high complexity; the geometric line tessellation algorithm is more complex but provides acceptable performance. The geometric algorithm is the lowest-complexity algorithm yet described that derives the perfect path.

We plan to modify the geometric algorithm in the near future to reduce its complexity further. We plan to use ideas from Sharp and Cripps (1991) to limit further the area in space that each geometric element must search. We also plan to explore the extension of the geometric algorithm to 3D environments, and to extend the algorithm to account for real robots with non-zero dimensions, probably by “growing” the obstacles in the intialisation phase of the algorithm.

Acknowledgements

We should like to thank David Sharp for his contribution to an earlier version of this work.

References

- H. Choset. Nonsmooth analysis, convex analysis and their applications to motion planning, 1998. URL citeseer.nj.nec.com/article/choset97nonsmooth.html.
- S. Fortune. A sweepline algorithm for voronoi diagrams., 1987.
- N. Mayya and V. Rajan. Voronoi diagrams of polygons: A framework for shape representation. In *CVPR94*, pages 638–643, 1994. URL citeseer.nj.nec.com/mayya94voronoi.html.
- M. McAllister, D. G. Kirkpatrick, and J. Snoeyink. A compact piecewise-linear voronoi diagram for convex sites in the plane. In *IEEE Symposium on Foundations of Computer Science*, pages 573–582, 1993. URL citeseer.nj.nec.com/mcallister96compact.html.
- B. Mirtich and J. Canny. Using skeletons for nonholonomic path planning among obstacles. *Proceedings of IEEE International Conference on Robotics and Automation*, pages 2533 – 2540, May 1992.
- L. Nawawi, J. R. Getta, and P. J. McKerrow. The construction of the optimal p-tree. *Australian Computer Science Communications*, 11(1):170 – 181, January 1999.
- Ogniewicz and Kubler. Voronoi tessellation of points with integer coordinates: Time-efficient implementation and online edgelist generation. *PATREC: Pattern Recognition*, Pergamon Press, 28, 1995. URL citeseer.nj.nec.com/ogniewicz93voronoi.html.
- D. W. Sharp and M. D. Cripps. Parallel algorithms that solve problems by communication. In *3rd IEEE Symposium on Parallel and Distributed Processing*, pages 87 – 94, 1991.
- M. L. Wager. Making roadmaps using voronoi diagrams. *UWA Honours Thesis*, 2000.