© Regeo. Geometric Reconstruction Group
www.regeo.uji.es
Technical Reports. Ref. 11/2016

Perimeter detection in sketched drawings of polyhedral shapes

P. Company[1], P.A.C. Varley[1] and R. Plumed[2]

[1]Institute of New Imaging Technology, Universitat Jaume I, Spain

[2]Department of Mechanical Engineering and Construction, Universitat Jaume I, Spain

Abstract

This paper describes a new "envelope" approach for detecting object perimeters in line-drawings vectorised from sketches of polyhedral objects.

Existing approaches for extracting contours from digital images are unsuitable for Sketch-Based Modelling, as they calculate where the contour is, but not which elements of the line-drawing belong to it.

In our approach, the perimeter is described in terms of lines and junctions (including intersections and T-junctions) of the original line drawing.

Index Terms: Sketches-based modelling. Polyhedral shapes. Contour, Perimeter.

1. Introduction

Detection of object perimeters is a fundamental cue for Sketch-Based Modelling (SBM)—the perimeter of an object "is such a fundamental cue to tri-dimensionality that it is hard for humans to suppress it" [BT81].

Digital images which capture scenes of the real world are very *rich* in content (i.e., they contain a large amount of information), but they typically store information at a low semantic level (e.g., raster or bitmap images). In contrast, SBM inputs are very *sparse* images (they contain just a reduced set of strokes), which can be converted into high semantic line-drawings containing only lines and junctions. In drawings which depict polyhedral objects, the lines and junctions in a drawing are graph-like representations that depict edges and vertices of the object.

Hence, existing approaches for identifying region boundaries in digital images—even those adapted to sketches and drawings [Sau03]—are inappropriate for detecting perimeters in plain line-drawings: semantic information would be lost in resampling lines as a raster map, which would then be processed inefficiently using algorithms designed for large amounts of data; and the output from such algorithms is a set of successive points—or sometimes an external polyline—which defines a border which envelops the region of interest.

What SBM approaches require instead is identifying the subset of lines and junctions which bound the depiction of the object. In this paper, we describe and assess our new approach for determining the object perimeter. Our approach uses the 2D line-junction connectivity of the line drawing, and works for both *wireframe* and *natural* representations (the former include hidden edges, while the latter exclude them). The output is the *circuit* (closed sequence of lines and junctions) which forms the perimeter.

Some lines only partially belong to the perimeter: the visible part of an occluded line can terminate at an intersection in a wireframe and at a T-junction in a natural line-drawing. We detect such intersections and T-junctions and include them in the sequence of corners, which are those junctions, intersections or T-junctions that are found to belong to the perimeter. Thus, the set of corners is an ordered subset of the set of junctions intersections and T-junctions.

The capability of the approach to work with intersections allows it to find the perimeter of *multigraph* line-drawings (where no path of lines allows visiting all the junctions). The perimeters of each separate subgraph are determined in addition to the global perimeter.

The rest of the paper is organised as follows. Section 2 introduces useful terminology for our work and explains the type of drawings used as input in our method. In Section 3 related work is discussed. Sections 4 explains how our algorithm works to detect the perimeter of engineering sketches. Section 5 shows some examples used to validate the method. Finally, Section 6 summarizes our conclusions.

2. Input information and terminology

The input required by our algorithm is a *line-drawing*: a list of junctions and a list of lines, where a line connects two junctions (note the similarity with the vertex-edge graphs of graph theory). Junctions are (x,y) coordinate pairs and usually correspond to vertices of the depicted object. Lines tend to correspond to edges. But a simple edge may split into a set of lines, depending on the input process. The lines highlighted in thick-red in Figure 1 right could each be one

line or two; our perimeter detection algorithm allows for either interpretation.

Some applications allow direct input of lines; others interpret *sketches* (in which lines may be overtraced for emphasis) or even images. Although vectorisation, the conversion of sketches to line-drawings as illustrated in Fig. 1, is still an open problem (as described in [JGH*08]), reasonably good solutions already exist for sketches of polyhedral objects, as considered here. Zhang et al [ZSD*06] summarise older approaches, and propose a seeded segment growing algorithm for extracting graphical primitives from a stroke.

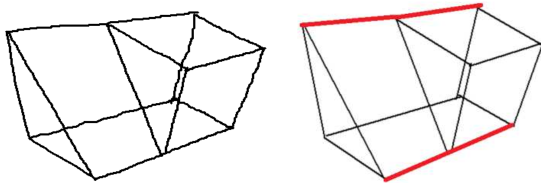


Fig. 1. Strokes (left), segmented (right, upper thick red lines) and non-segmented (right, lower thick red line) collinear edges

Vectorisation does not correct the geometrical imperfections inherent in sketching. However, as long as the topology remains unaltered, our perimeter detection algorithm is unaffected by such imperfections.

Vectorisation must however merge dangling endpoints to produce junctions which depict valid vertices (Figure 2).

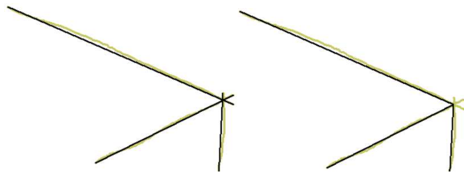


Fig. 2. Merging of three endpoints (left) to form a single junction (right)

At T-junctions in natural line-drawings (see Figure 3 left), where the endpoint of one line should meet an intermediate point of another line, vectorisation may either *split* the second line, so as to produce two new lines which meet the first line in an "ordinary" trihedral junction (Figure 3 middle), or leave the second line unaltered while ensuring that the endpoint of the first line exactly meets the second line (Figure 3 right).

Fig. 3. T-junction (left) may either split the second line (centre), or ensure that the first line touches the second (right)

T-junctions of natural line-drawings require no special code for perimeter detection if treated as real trihedral junctions. The same applies for intersections in wireframes. Since this implies that lines that represent actual edges are split into a set of two or more sub-lines, some properties of the line may be lost (e.g. collinearity) unless we enrich the line-drawing with suitable information about geometrical constraints.

In this paper, we also allow for the alternative approach: leave the lines unsplit and add the perimeter information as a complement, as our perimeter detection algorithm is designed to deal with both split and unsplit intersections and T-junctions (see Section 4.1). Perhaps, this alternative increases computational complexity, but it will hopefully preserve the design intent implicit in the strokes depicted to be seen as lines that depict edges. Thus, for the rest of the paper, "junctions" are the ordinary ones—shared tips of the lines in the 2D drawing that are assumed to be the projections of the 3D vertices of a polyhedron—while we also consider the unsplit intersections and T-junctions. We name as corners to the ordered subset of the set of junctions, T-junctions and unsplit intersections that are found to belong to the perimeter.

3. Previous work

Our long term goal is extracting as many perceived cues of information about sketched line drawings as possible. In this context, it has been stated that the number of contour edges for polyhedron projections is small and the number of intersections of contour edges appears to be even more favourable [KW96]. Thus, detecting the perimeter of sketched drawings of polyhedral shapes is an interesting goal, as far as we can get this information before we search for more high semantic level cues that help us to recover the 3D shape implicitly depicted in the 2D line drawing. Thus, on the contrary of other well-known approaches ([KW96], [PBD*01]), we do not know information about faces and their orientations while we search for the perimeter. Just on the contrary, we try to get the perimeter in order to use this information in a later search for visible and occluded faces.

Besides, we can distinguish between perimeter and silhouette, since the latter is usually defined as a set of successive points—or sometimes an external polyline—which defines a border which envelops the region of interest, and constitutes a cue for figure-to-ground distinction [IFH*03]. This implies that the output silhouette of most of the approaches applied to sketched drawings of polyhedral shapes is not a subset of the original set of lines and junctions of the polyhedron, but an overlaid entity [QJL*07].

The first work directly related with perimeter detection in drawings of polyhedral shapes is the Roberts's work on perception of three-dimensional objects from line drawings, which includes a whole section on Polygon Recognition [Rob63], [Rob65]. Roberts's approach is simple and efficient. First, at each junction, all lines connected to that junction are ordered by their orientation. The search for a polygon starts at a random line, and at each junction, the line we follow is the next in the ordered list after the one along which we arrived. The process is repeated until the initial junction is reached again, and the circuit is closed in a cycle (Figure 4).

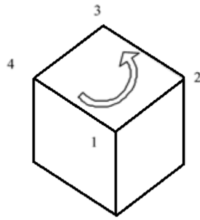


Figure 4. Robert's approach for detection of polygons in a natural line-drawing

This method can be used to identify regions in natural drawings (regions can correspond to faces of the object, or to part-faces as in Fig. 5 left), and can also find the exterior boundary polygon. It fails for wireframe drawings (Fig. 5 right).

Labelling methods (from Huffman [Huf71], Clowes [Clo71] and Waltz [Wal72], to Varley and Martin [VM00a], [VM00b]) may obtain the perimeter (Fig.5 left), but this is not their main goal, and they require catalogues of valid junction labels—so far, only trihedral and tetrahedral junctions have been catalogued fully; full catalogues of 5-hedral junction labels and beyond are not practical.

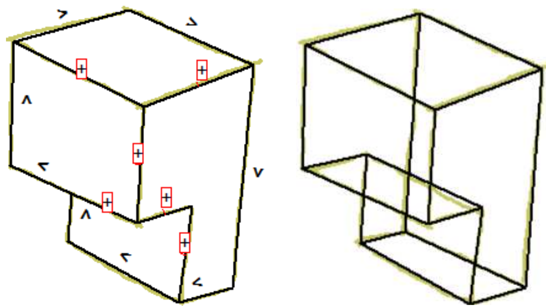


Figure 5. Labelled natural line-drawing (left) and the corresponding wireframe (right)

4. Our approach

We want to find a sequence of lines—plus the corresponding junctions—which defines the perimeter as a closed circuit. Our strategy should be tailored to the actual needs of the reconstruction approach (we advocate a "cascade" approach which first detects simple cues such as perimeter and then uses these results to further analyse the line-drawing searching for more complex cues). We only extract the information from the line-drawing that we shall need (not "as much as possible", as with labelling methods).

For finding the perimeter we firstly identify the upper junction—which together with the leftmost, rightmost and lower junctions (i.e., those with the biggest or the smallest *x*- or *y*-coordinate) must belong to the perimeter. Thus, the upper junction becomes the first corner, and at least one of the lines connected to it must also belong to the perimeter.

As we follow the perimeter clockwise around the drawing, we arrive at each junction (like junction V in Figure 6) along an *incoming* line which is already part of the perimeter (line e_0 in Figure 6), we determine an *outgoing* line to add to the perimeter (this outgoing line will then be the incoming line at the next junction, and so on). The outgoing line is always the leftmost line as viewed from the incoming line (line e_3 in Figure 6). In determining the leftmost line, angles between lines must be normalised to the range (0°, 360°)—this has the additional benefit of ensuring that the interpretation is independent of the orientation of the line-drawing.

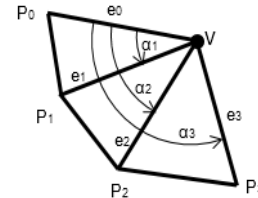


Figure 6. Inner angles between lines sharing the junction V

To find the initial outgoing line at our starting junction, we have no previous incoming line. Instead, we use an artificial incoming line which arrives vertically at our start upper junction (i.e. parallel with the y-axis) (see (P_{-1}-P_0) in Figure 7).

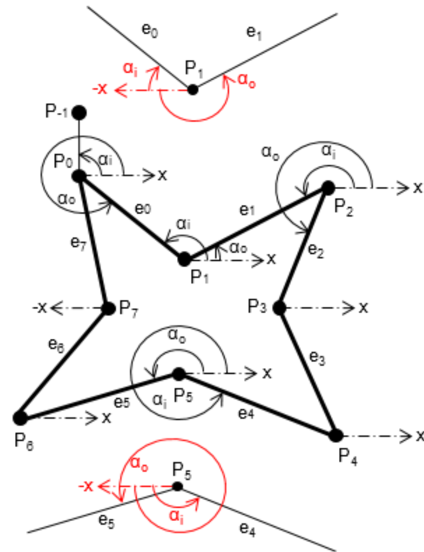


Figure 7. Inner angles relative to the x axis

The procedure ends when we return to the first corner. Dangling lines are defined as those lines with one endpoint not connected to other lines (line 0-4 in Figure 8). In case that current corner is not connected to other lines than the one most recently added to the perimeter (dangling line), this line is added again to the perimeter. Then its endpoints are re-added as corners in reverse order.

In pathological cases where the starting corner is the end of *n* dangling lines (or chains of dangling lines), we allow the algorithm to go through the initial corner *n+1* times.

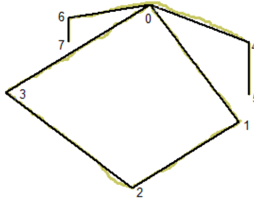


Figure 8. Simple perimeter with dangling lines

Our algorithm also copes with lines which belong only partially to the perimeter, as can happen in wireframe drawings of non-convex polyhedra. There are two possibilities: one junction belongs to the perimeter, but the other does not (as with lines 0-1 and 10-11 in Figure 9 left); or part of a line belongs to the perimeter, while its endpoints do not (as with line 2-15 in Figure 9 right).

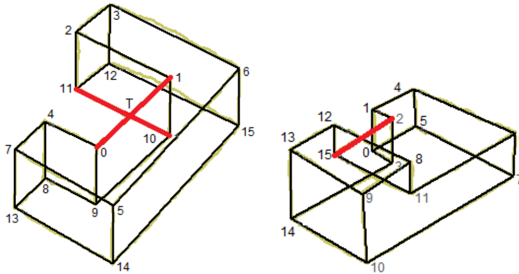


Figure 9. Partial lines in the perimeter

To consider intersections or T-junctions, each time a new line is added, we test whether any other lines cross—or end at—the current perimeter line. If so, we inspect the intersection points. The closest intersection point to the previous junction will be added to the perimeter as if it were a corner, and the left turn along the intersecting line is also added to the perimeter as its outgoing line. Since a single line may include more than one T-junction (e.g. edge 0-6 in the test-drawing of Figure 10), the crossing test must be reapplied to the new outgoing line. If no line intersects the current line, the far endpoint will become the next corner.

Figure 10. Two segments of the same line, delimited by pairs of T-junctions, belong to the perimeter

4.1. The algorithm

The input to the algorithm are lists of the (x,y)-coordinates of each junction and the head and tail junctions of each line. The algorithm first calculates derived information: subgraphs information and a list of lines connected to each junction. Note that intersections and T-junctions can be calculated only once—in advance—and used as required. The procedure to detect subgraphs is a breadth-first search to visit all junctions connected to an arbitrarily selected first junction. This results in the first subgraph. Repeating the procedure for any not-yet-visited junction results in a second subgraph. The procedure is complete when no more junctions remain unvisited.

The output of the algorithm is a list of ordered lines and corners that belong to the perimeter (PerimeterLines and PerimeterCorners). Positive numbers in the list of corner refer to junctions of the original line drawing, while negative numbers are pointers to a list of intersections or T-junctions that belong to the perimeter. The coordinates of the intersections and T-junctions that belong to the perimeter are saved (list TX), as much as the lines that produce each intersection (list TEdges). Note that the distinction between intersections and T-junctions is simple: the intersection is a T-junction if it is close to one vertex of the outgoing edge.

The complete flow of our perimeter detection function is as follows:

```
PerimeterByEnvelope()
{
  FirstCorner= GetUpperJunction()

  PreviousCorner= FirstCorner
  PreviousCorner.y += 1

  CurrentCorner= FirstCorner
  NumTJ= 0
  CurrentLine= -1
  NextCorner= FirstCorner
  TJ= false

  do
  {
    if (NextCorner >= 0) //The current corner is a junction
    {
      CurrentLine= GetOutgoingLine(CurrentCorner,
                                   PreviousCorner)
      PerimeterLines.push_back(CurrentLine)

      PreviousCorner= CurrentCorner

      if (AnyIntersection (CurrentLine))
      {
        XPoint= GetCloserIntersection(CurrentLine)
        XLine= GetInterceptingLine(CurrentLine)
        TJ= is_T-Junction()
        NextCorner= -1 //The next corner is an intersection
      }
      else
      {
        NextCorner= GetFarJunction(CurrentLine)
      }
    }

    else  //The current corner is an intersection
    {
      PreviousLine= CurrentLine
      CurrentLine= XLine
      PerimeterLines.push_back(CurrentLine)

      if (TJ)
        CurrentLine= XLine
```

```
        PreviousCorner= XPoint

        if (AnyIntersection (CurrentLine))
        {
          XPoint= GetCloserIntersection(CurrentLine)
          XLine= GetInterceptingLine(CurrentLine)
          TJ= is_T-Junction()
          NextCorner= -1 //The next corner is an intersection
        }
        else
        {
          NextCorner= GetFarJunction(CurrentLine)
        }
      }

      if (NextCorner >= 0)
      {
        PerimeterCorners.push_back(NextCorner)
        CurrentCorner= NextCorner
      }

      else
      {
        TX.push_back(XPoint)
        NumTJ - -

        TEdges[-NumTJ-1].push_back(CurrentLine)
        TEdges[-NumTJ-1].push_back(XLine)

        PerimeterCorners.push_back(NumTJ)
      }
    }
    while (CurrentCorner != FirstCorner)
}
```

For the sake of simplicity, we have omitted the test for dangling lines that occur at the beginning of the search (see Section 3) and a trap for infinite loops (stop after visiting as many junctions as the figure contains).

The perimeter finder is successively called after loading the corresponding subgraph in the *database*.

The function to determine the closest intersection point excludes both the current edge and the edges connected to it, so as to prevent detecting false intersections between the current edge and edges that share its endpoints. The function also, when required, identifies a junction that belongs to the outgoing line of a T-junction.

The full source code of the algorithm is freely available at [CVP16].

5. Validation

To test the validity of the approach, we used four types of sketch:

1. Sketches whose line-drawings are bounded by simple perimeters of full lines connected at junctions (see Figure 11).
2. Sketches whose line-drawings are bounded by perimeters which include partial lines crossing at intersections or T-junctions (see Figure 12).
3. Sketches whose line-drawings are bounded by perimeters which include non-trivial combinations of intersections or T-junctions (see Figure 13).
4. Sketches of complex “flat” drawings to validate the algorithm in the presence of complex intersections and singular points (Figure 14).

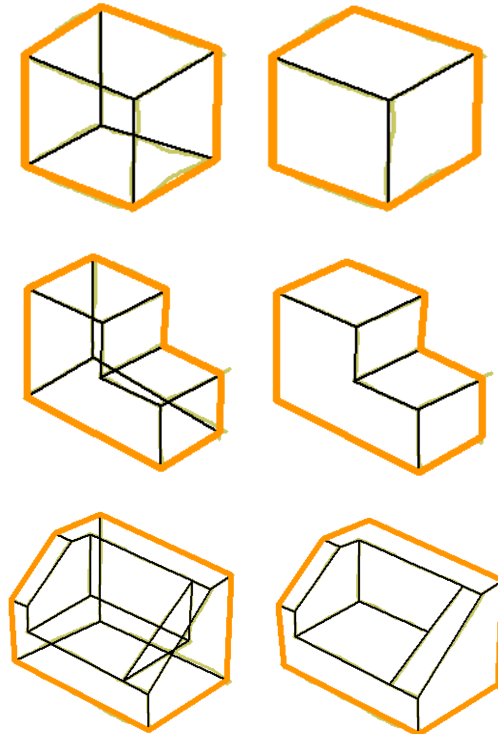
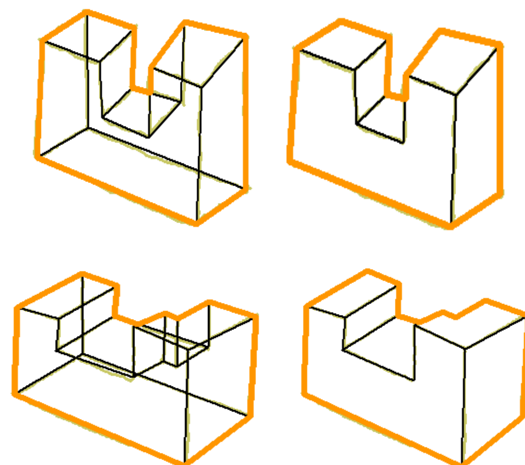


Figure 11. Perimeters of full lines



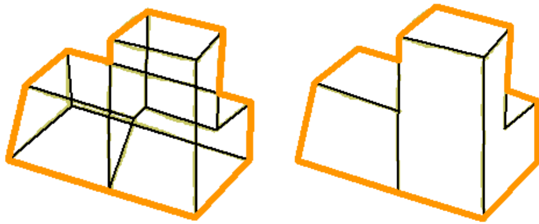
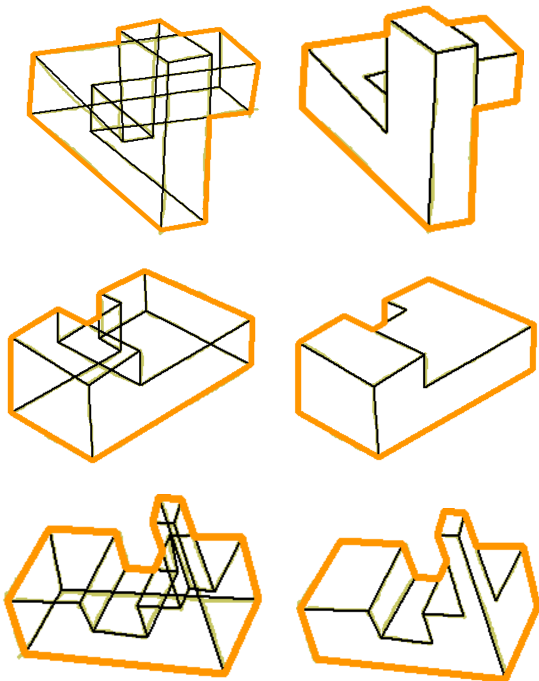
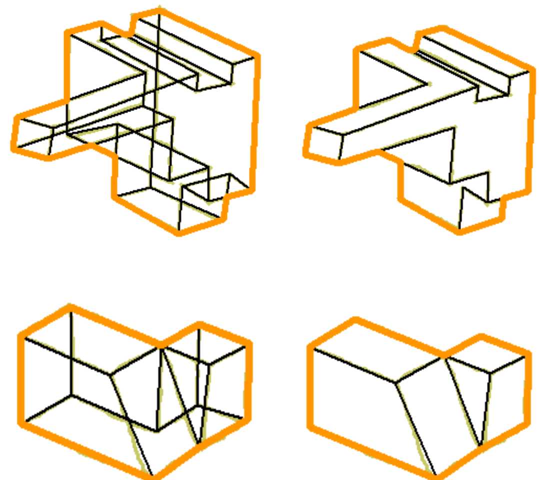


Figure 12. Perimeters including isolated T-junctions





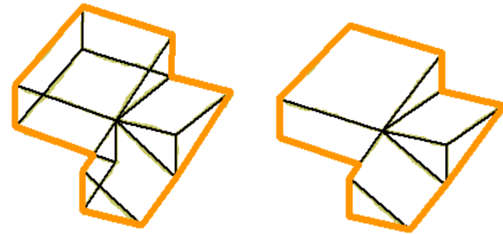


Figure 13. Perimeters including complex combinations of junctions and intersecting lines

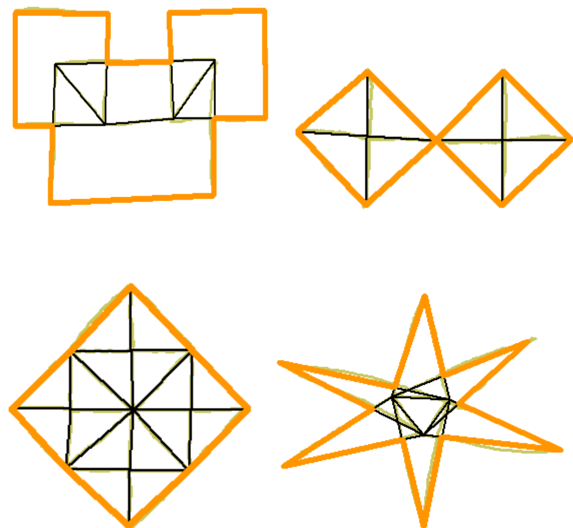


Figure 14. Drawings of flat shapes including intersections and other singular points

Finally, we tested that our approach successfully detects the perimeters of subgraphs, as well as the perimeter of the full line-drawing (Figure 15).

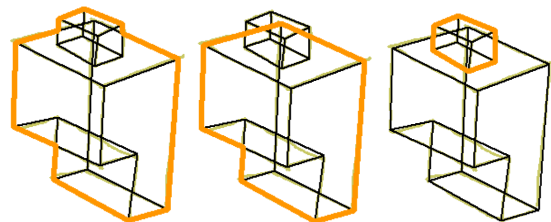


Figure 15. Perimeters of the full line-drawing and the subgraphs

We note that natural drawings with T-junctions may easily become decomposed into subgraphs if we do not split T-junctions as explained in Figure 3. Thus, a line drawing such as the first example in Figure 13 produces three sub-

perimeters in addition to the main perimeter shown in the figure.

Our approach is simple and its computational cost is small. One example with 117 edges and 71 vertices takes less than one millisecond (Figure 16). The algorithm (Section 4.1), including pairwise tests for crossing lines, is $O(n^2)$.

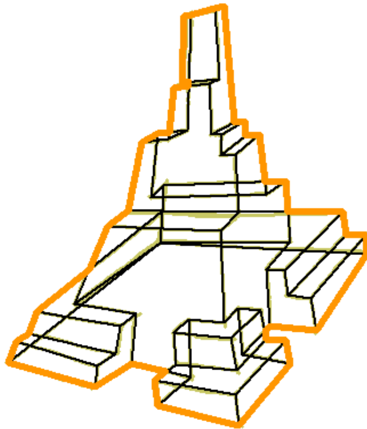


Figure 16. Example of perimeter calculation for a populated line-drawing

6. Conclusions

Perimeter detection is a basic and useful stage in Sketch-Based Modelling.

Existing approaches for perimeter detection in digital images are inappropriate, as neither the input nor the output fit the needs of Sketch-Based Modelling.

Published SBM approaches are inappropriate for wireframe representations and multigraph line-drawings.

We have developed a new approach for determining the perimeter of a 2D line-drawing which works for both natural line-drawings and wireframes. It does not require any other information than lines and junctions. It is not limited to particular types of polyhedron.

Our new approach quickly and correctly detects perimeters of line-drawings vectorised from sketches of polyhedral objects, and defines the perimeter as a subset of lines and junctions of the original line-drawing.

7. Acknowledgements

We gratefully acknowledge the helpful comments of the anonymous reviewers that greatly helped us to improve the final version of this paper.

8. References

[BT81] Barrow H.G. and Tenenbaum J.M. (1981) "Interpreting line drawings as three-dimensional surfaces", Artificial Intelligence, 17(1-3), 75-116, 1981.

[Sau03] Saund E. (2003) Finding Perceptually Closed Paths in Sketches and Drawings, IEEE Transactions on Pattern Analysis and Machine Intelligence, 25(4), pp. 475-491.

[JGH*08] Johnson G., Gross M.D., Hong J., Do E.Y.L. Computational support for sketching in design: a review. Foundations and Trends in Human–Computer Interaction, Vol. 2, No. 1 (2008) 1–93

[ZSD*06] Zhang X., Song J., Dai G., Lyu M.R.: Extraction of line segments and circular arcs from freehand strokes based on segmental homogeneity features. IEEE Trans. Systems, Man, and Cybernetics, 36 (2), (April 2006).

[KW96] Kettner L., Welzl E.: Contour edge analysis for polyhedron projections, in: W. Straßer, R. Klein, R. Rau (Eds.), Geometric Modeling: Theory and Practice, Springer, 1997. (Proc. Internat. Conf. Theory and Practice of Geometric Modeling in Blaubeuren, Germany, October 1996.)

[PBD*01] Pop M., Barequet G., Duncan C.A., Goodrich M.T., Huang W., and Kumar S.: Efficient Perspective-Accurate Silhouette Computation. In Proc. 17th Ann. ACM Symp. on Computational Geometry, pp. 60–68, Medford, MA, 2001

[IFH*03] Isenberg T., Freudenberg B., Halper N., Schlechtweg S., Strothotte T.: A Developer's Guide to Silhouette Algorithms for Polygonal Models, IEEE Computer Graphics and Applications, v.23 n.4, p.28-37, July 2003 [doi>10.1109/MCG.2003.1210862]

[QJL*07] Qin Z., Jia J., Li T.T. and Lu J.: Extracting 2D Projection Contour from 3D Model Using Ring-Relationship-Based Method. Information Technology Journal, 6: 914-918, 2007.

[Rob63] Roberts L.G. Machine Perception of Three-Dimensional Solids. PhD Thesis. MIT, Certified by Peter Elias (Thesis Supervisor). 1963.

[Rob65] Roberts L.G. Chapter 9: Machine Perception of three-dimensional solids. Optical and Electro-Optical Information processing, The MIT Press, Cambridge, Massachusetts and London, England. 1965.

[Huf71] Huffman D.A., 1971. “Impossible objects as nonsense sentences”. *Machine Intelligence*. Edinburgh University Press, pp. 295-323.

[Clo71] Clowes M.B. On Seeing Things. Artificial Intelligence, Vol. 2 (1971), pp. 79-116

[Wal72] Waltz D.M. Generating Semantic Descriptions from Drawings of Scenes with Shadows. Tech Rept AI-TR-271, M.I.T., Cambridge USA, 1972.

[VM00a] Varley P.A.C. and Martin R., 2000 (a). “A system for constructing boundary representation solid models from a two-dimensional sketch. Frontal geometry

and sketch categorisation". *1st Korea_UK Joint Workshop on Geometric Modeling and Computer Graphics*.

[VM00b] Varley P.A.C. and Martin R., 2000 (b). "A system for constructing boundary representation solid models from a two-dimensional sketch. Topology of hidden parts". *1st Korea_UK Joint Workshop on Geometric Modeling and Computer Graphics*.

[CVP16] Company P., Varley P.A.C. and Plumed R, 2016. Source code for finding perimeters in 2D line-drawings of polyhedral shapes. <http://www.regeo.uji.es/FindingPerimeter.htm>, Regeo (2016), Geometric Reconstruction Group, http://www.regeo.uji.es. Accessed October 2016.