UNIVERSITY OF BATH

Citation for published version:
Shepherd, P, Edum-Fotwe, K, Brown, M, Harper, D & Dinnis, R 2016, 'QUALM: Quick, Unconstrained Approximate L-Shape Method' SIGGRAPH 2016, Anaheim, USA United States, 24/07/16 - 28/07/16, . https://doi.org/10.1145/2945078.2945163

DOI:
10.1145/2945078.2945163

Publication date:
2016

Document Version
Peer reviewed version

Link to publication

University of Bath

Quick, Unconstrained, Approximate L-Shape Method

K. Edum-Fotwe[1,2], P. Shepherd[1], M. Brown[1], D. Harper[2], R. Dinnis[2]
[1]University of Bath [2]Cityscape Digital

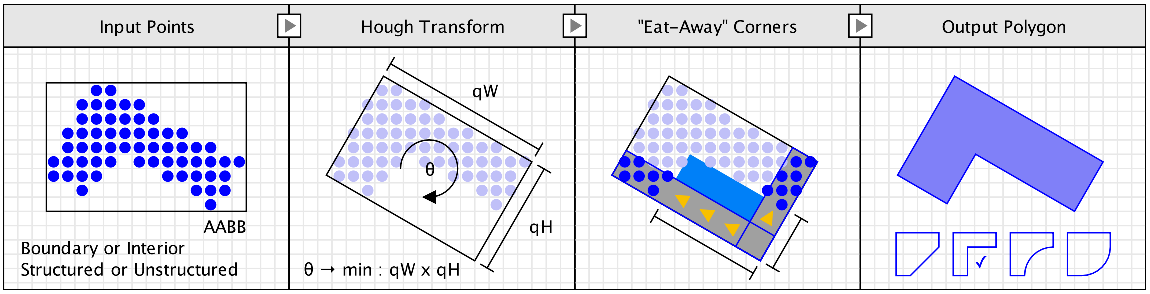

Figure 1: *Overview of the simple 2D shape approximation function - QUALM : (from left to right) the input points, the minimal area bounding box, then reducing error by 'eating-away' corners, and finally the output polygon (with alternative eat-away corner types illustrated below).*

Abstract

This simple paper describes an intuitive data-driven approach to reconstructing architectural building-footprints from structured or unstructured 2D pointsets. The function is fast, accurate and unconstrained. Further unlike the prevalent L-Shape detectors predicated on a shape's skeletal descriptor [Szeliski 2010], the method is robust to sensing noise at the boundary of a 2D pointset.

Keywords: Shape Detection, Hough Transform, *Eat-Away* Hull

Concepts: •**Computing methodologies** → **Shape modeling;**

1 Introduction and Motivation

The context of this work is the automatic recovery of clean (sparse) architectural geometry from various types of laser scan. In particular this operator aims to recover compact building footprints - that can be used for updating 2D-maps and for 3D urban modelling.

The method applies a simple observation about the nature of common rectilinear forms, in order to 'eat-away' at a minimal-area bounding box of a cluster of 2D points. One of the key benefits is determinism. Each 'eat-away' hull represents a repeatable product of the input-points. Another key benefit is resolution independence, since the method does not constrain the point-spacing of the input.

The approach executes in two stages (illustrated in fig.1). First it computes the minimal area bounding box (MABB) of the input 2D points. It then refactors each corner of the MABB by approximating the maximal inset edge-lengths, and injecting a corresponding 'eaten-away' right-angled corner in place of the MABB vertex. The appendix contains the implementation of the technique.

Measuring Geometric Error - Since this is a heuristic shape approximation method, it is vital to be able to measure the accuracy of each generated polygon relative to the input-points. For this two measures are considered. A discrete maximum point-to-edge distance and a continuous normalised shape-to-shape-overlap ratio. They enable an automatic algorithm to quantify the geometric fit.

The Discrete Hausdorff-Distance Error Measure

$$f(A,B) = max(||A_i - (B_j, B_{j+1})||) \quad \forall i \in A : \forall j \in B$$

The Continuous Intersect-over-Union Error Measure

$$(A \cap B)/(A \cup B) > \omega : \omega \in [0:1]$$

2 Results

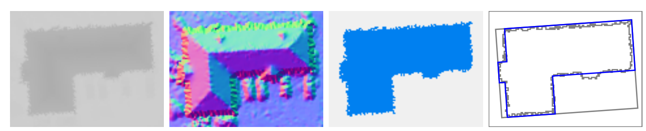

Figure 2: *an example from the 50cm point-spacing London dataset illustrating (from left to right) input-range-points, normals, difference of elevation building segment, resulting automatic l-shape footprint (scan-converted boundary in gray, eat-away hull in blue)*

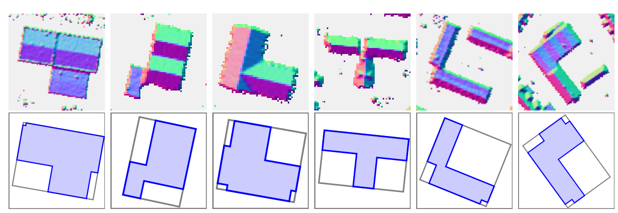

Figure 3: *Building footprints automatically recovered from 1m point-spacing airborne range scans of the city of Bath, UK*

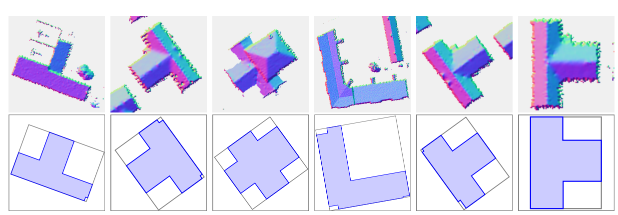

Figure 4: *Building footprints automatically recovered from 25cm point-spacing airborne range scans of the city of Manchester, UK*

SIGGRAPH '16, July 24-28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4371-8/16/07
DOI: http://dx.doi.org/10.1145/2945078.2945163

References

SZELISKI, R. 2010. *Computer vision: algorithms and applications*. Springer.

Appendix

This page presents the implemented 'eat-away' function - used to automatically recover the building footprints illustrated in the results section.

```
function QUALM ( points, hull, min_dist ) → Quick Unconstrained Approximate L-Shape Method

    points - a set of unstructured or structured 2D points
    hull - an optional dense extremal boundary hull for the input pointset (to speed up the hough-transform)
    min_dist - the minimum length of an edge in an eat-away-corner (a positive scalar to control the minimum inset size)
    return value - a 2D polygon : a sequence of vertices representing the detected L-Shape, T-Shape or S-Shape (0-4 refactored corners)

    ret ← {}
    quad ← hough_transform_minimal_area_quad(hull ? hull : points)

    for i ← 0 : i < 4 do
        min_distance ← minimum_distance_between_point_and_polygon(quad[i], hull ? hull : points)
        if min_distance > min_dist then
            prev ← quad[i > 0 ? i − 1 : 3]
            pos ← quad[i]
            next ← quad[i < 3 ? i + 1 : 0]

            prev_dx ← pos_x − prev_x
            prev_dy ← pos_y − prev_y
            next_dx ← next_x − pos_x
            next_dy ← next_y − pos_y
            prev_len ← sqrt(prev_dx × prev_dx + prev_dy × prev_dy)
            next_len ← sqrt(next_dx × next_dx + next_dy × next_dy)
            prev_ext ← (prev_len − min_distance)/prev_len
            next_ext ← min_distance/next_len

            prev_half_quad ← {
                prev,
                pos,
                vec2D(pos_x + next_dx × next_ext × 0.5, pos_y + next_dy × next_ext × 0.5),
                vec2D(prev_x + next_dx × next_ext × 0.5, prev_y + next_dy × next_ext × 0.5)
            }
            next_half_quad ← {
                pos,
                next,
                vec2D(next_x − prev_dx × (1 − prev_ext) × 0.5, next_y − prev_dy × (1 − prev_ext) × 0.5),
                vec2D(pos_x − prev_dx × (1 − prev_ext) × 0.5, pos_y − prev_dy × (1 − prev_ext) × 0.5)
            }

            prev_points_in_half ← points_inside_polygon(points, prev_half_quad)
            next_points_in_half ← points_inside_polygon(points, next_half_quad)

            prev_min_distance ← distance_to_closest_neighbour(pos, prev_points_in_half)
            next_min_distance ← distance_to_closest_neighbour(pos, next_points_in_half)

            if prev_min_distance > next_min_distance then
                prev_ext ← (prev_len − prev_min_distance)/prev_len
            else next_ext ← next_min_distance/next_len
            end if

            new_prev ← vec2D(prev_x + prev_dx × prev_ext, prev_y + prev_dy × prev_ext)
            add(new_prev, ret)                                                              ▷ new prev
            add(vec2D(new_prev_x + next_dx × next_ext, new_prev_y + next_dy × next_ext), ret)   ▷ new pos
            add(vec2D(pos_x + next_dx × next_ext, pos_y + next_dy × next_ext), ret)          ▷ new next

        else add(quad[i], ret)
        end if
        i++
    end for

    return ret
end function
```