Incremental and Batch Planar Simplification of Dense Point Cloud Maps

T. Whelan[a,*], L. Ma[b], E. Bondarev[b], P. H. N. de With[b], J. McDonald[a]

[a] *Department of Computer Science, National University of Ireland Maynooth, Maynooth, Co. Kildare, Ireland*
[b] *Department of Electrical Engineering, Eindhoven University of Technology (TU/e), Eindhoven, the Netherlands*

Abstract

Dense RGB-D SLAM techniques and high-fidelity LIDAR scanners are examples from an abundant set of systems capable of providing multi-million point datasets. These datasets quickly become difficult to process due to the sheer volume of data, typically containing significant redundant information, such as the representation of planar surfaces with millions of points. In order to exploit the richness of information provided by dense methods in real-time robotics, techniques are required to reduce the inherent redundancy of the data. In this paper we present a method for incremental planar segmentation of a gradually expanding point cloud map and a method for efficient triangulation and texturing of planar surface segments. Experimental results show that our incremental segmentation method is capable of running in real-time while producing a segmentation faithful to what would be achieved using a batch segmentation method. Our results also show that the proposed planar simplification and triangulation algorithm removes more than 90% of the input planar points, leading to a triangulation with only 10% of the original quantity of triangles per planar segment. Additionally, our texture generation algorithm preserves all colour information contained within planar segments, resulting in a visually appealing and geometrically accurate simplified representation.

Keywords: planar simplification, point clouds, mapping, plane segmentation, triangulation, incremental

1. Introduction

The generation of 3D models of real-world environments is of significant interest in many applied fields including professional civil engineering, environment-based game design, 3D printing and robotics. Industrial Light Detection And Ranging (LIDAR) platforms and extended scale RGB-D mapping systems can output dense high-quality point clouds, spanning large areas that contain millions of points [2, 3]. Key issues with such large-scale multi-million point datasets include difficulties in processing the data within reasonable time and a high memory requirement. In addition to this, some features of real-world maps, such as walls and floors, end up being over-represented by thousands of points when they could be more efficiently and intelligently represented with geometric primitives. In particular the use of geometric primitives to represent a large 3D map to localise against has been demonstrated as a feasible means of real-time robot localisation [4]. In this paper, we examine the problem of planar surface simplification in large-scale point clouds with a focus on quality and computational efficiency in both online incremental and offline batch settings.

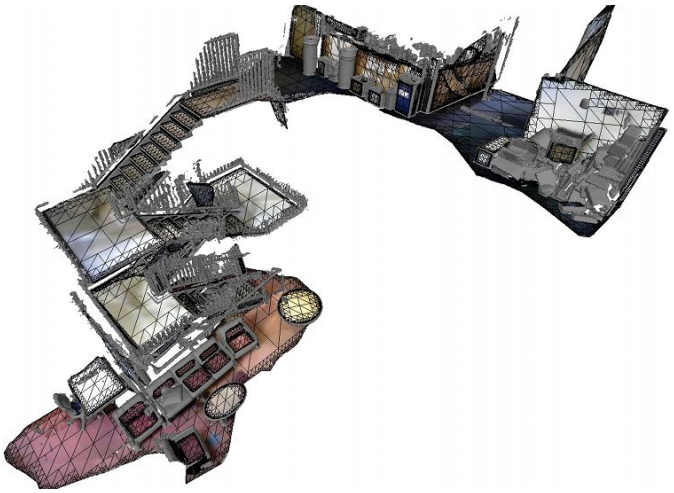


Figure 1: Scene triangulation showing a simplified mesh for planar segments with non-planar features highlighted.

2. Related Work

Existing triangulation algorithms perform poorly in removing redundancy in dense point clouds, or are not suited to the kind of data typically acquired with common robotic sensors. In addition to this, to the best of our knowledge there is no known efficient solution to incrementally grow planar segments extracted from dense point cloud data in a gradually expanding map. In this paper we address these problems with three main contributions improving on the work of [5]. Firstly we present a computationally efficient method for incrementally growing

*Corresponding author. This work was presented in part at the European Conference on Mobile Robotics (ECMR), Barcelona, September 2013 ([1]).

Email addresses: thomas.j.whelan@nuim.ie (T. Whelan), l.ma@tue.nl (L. Ma), e.bondarev@tue.nl (E. Bondarev), p.h.n.de.with@tue.nl (P. H. N. de With), johnmcd@cs.nuim.ie (J. McDonald)

existing planar segments in a dense point cloud map, which produces segmentations extremely close to those that would be achieved using a batch segmentation. Secondly we present an accurate and robust algorithm for planar segment decimation and triangulation. In comparison to the existing QuadTree-Based (QTB) algorithm [5], our algorithm guarantees geometric accuracy during simplification with fewer triangles, without duplicate points, artificial holes or overlapping faces. Thirdly we present a method to automatically generate textures for the simplified planar mesh based on dense coloured vertices. Our experimental results show that the presented solutions are efficient in processing large datasets, either incrementally online or in batch through the use of a multi-threaded parallel architecture.

In the literature triangular meshing of 3D point clouds is a well-studied problem with many existing solutions. One class of triangulation algorithms computes a mathematical model prior to triangulation to ensure a smooth mesh while being robust to noise [6, 7]. This type of algorithm assumes surfaces are continuous without holes, which is usually not the case in open scene scans or maps acquired with typical robotic sensors. Another class of algorithms connects points directly, mostly being optimized for high-quality point clouds with low noise and uniform density. While these algorithms retain fine details in objects [8, 9], they are again less applicable to noisy datasets captured with an RGB-D or LIDAR sensor, where occlusions create large discontinuities.

With real-world environment triangulation in mind, the Greedy Projection Triangulation (GPT) algorithm has been developed [10, 11]. The algorithm creates triangles in an incremental mesh-growing approach, yielding fast and accurate triangulations. However, the GPT algorithm keeps all available points to preserve geometry, which is not always necessary for point clouds containing surfaces that are easily characterised by geometric primitives. To solve this problem a hybrid triangulation method was developed in [5], where point clouds are segmented into planar and non-planar regions for separate triangulation. The QTB algorithm was developed to decimate planar segments prior to triangulation. The QTB algorithm significantly reduces the amount of redundant points, although a number of limitations degrade its performance. For example, the algorithm does not guarantee that final planar points will lie inside the original planar region, which can lead to noticeable shape distortion. The algorithm also produces duplicate vertices, overlapping triangles and artificial holes along the boundary.

There is a vast amount of literature on planar segmentation available. Oehler *et al.* adopt a multi-resolution approach that relies on using a Hough transform over co-planar clusters of surfels, ultimately relying on RANSAC for the plane fitting component [12]. Deschaud and Goulette use a region growing approach made robust to noise by growing in a voxel space over the input data rather than the raw points themselves [13]. Some algorithms extend 2D graph cut theory towards 3D point cloud data [14, 15, 16]. These algorithms are designed for general object segmentation and their complexity is in general too high for plane detection, unlike the low-complexity algorithm proposed by Rabbani *et al.* [17], which imposes a smoothness constraint

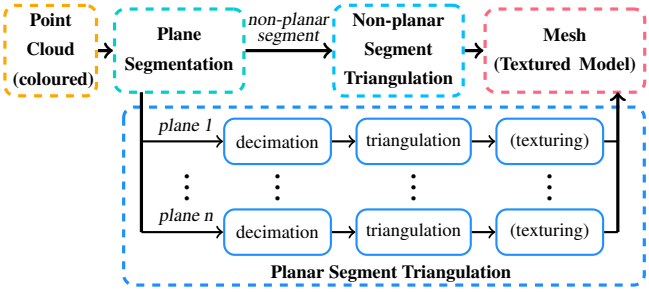


Figure 2: Parallel system architecture to process point clouds of large-scale open scene scans or maps. Here texturing is in brackets as it is an optional step.

on segmentation (discussed later in Section 4). However, like in previous work ([5]) these methods are all concerned with batch processing scenarios rather than the incremental segmentation growing scenario.

In the remainder of this paper we give an overview of our system followed by individual sections detailing batch planar segmentation, incremental planar segmentation, triangulation of planar segments and texture generation. Following this we provide results evaluating both the qualitative and quantitative performance of each of the presented techniques.

3. System Overview

3.1. Building Blocks

Our batch system architecture is shown in Figure 2. It takes a point cloud as input and generates a triangular mesh as output. If the input is a coloured point cloud, the output can also be a textured 3D model. The processing pipeline consists of three main blocks.

Plane Detection segments the input point cloud into planar and non-planar regions to enable separate triangulation and parallel processing. This design is especially beneficial for real-world environments, where multiple independent planar surfaces occur frequently. In our system we apply a local curvature-based region growing algorithm for plane segmentation, which was shown to out perform RANSAC-based approaches [5]. In the interest of completeness we include a description of this algorithm in Section 4. In the incremental scenario, the plane detection block continuously runs and only provides planar segments to be triangulated when they are marked as *finalised* (as detailed in Section 5).

Non-Planar Segment Triangulation generates a triangular mesh for non-planar segments using the GPT algorithm [11]. Given a coloured point cloud, we preserve the colour information for each vertex in the output mesh. Dense triangular meshes with coloured vertices can be rendered (with Phong interpolation) to appear similar to textured models. Additionally, as opposed to using textures, maintaining colour in vertices of non-planar segments provides easier access to appearance information for point cloud based object recognition systems.

Planar Segment Triangulation triangulates planar segments and textures the mesh afterwards, if given a coloured point cl-

oud. In our system we improve the decimation algorithm in [5] and further develop a more accurate and robust solution for triangulation. A detailed description of our algorithm is provided in Section 6. Our method for planar segment texture generation is described in Section 7.

3.2. Computationally Efficient Architecture

To improve computational performance, a multi-threaded architecture is adopted, exploiting the common availability of multi-core CPUs in modern hardware. We apply a coarse-grained parallelization strategy, following the Single Program Multiple Data (SPMD) model [18]. Parallel triangulation of planar segments is easily accomplished by dividing the set of segments into subsets that are distributed across a pool of threads. For maximum throughput of the entire pipeline, segmentation and triangulation overlap in execution. With an n-core CPU, a single thread is used for segmentation and the remaining $n-1$ threads are used for triangulation, each with a queue of planar segments to be processed. Upon segmentation of a new planar region, the segmentation thread checks all triangulation threads and assigns the latest segment to the thread with the lowest number of points to be processed. This strategy ensures an even task distribution among all threads. When plane segmentation is finished, the segmentation thread begins the non-planar triangulation in parallel to the other triangulation threads.

4. Planar Segmentation

In this section we review the curvature-based algorithm used to segment multiple planes from a large 3D point cloud.

4.1. Curvature-Based Segmentation Algorithm

Planes are characterized by their perfect flatness and can be described as sets of points that have zero curvature. In practice, open scene point cloud data can be quite noisy and points belonging to planes do not have a curvature of exactly zero. However, the curvature of points lying on planes is still low enough to distinguish them from points belonging to non-planar surfaces. This observation motivates the functionality of our algorithm, which is partially developed from the work of Rabbani *et al.* [17].

The curvature-based algorithm consists of an iterative process. Firstly, the normal of the next plane to be segmented is chosen. This is done by finding the point with the lowest curvature from the set of remaining unsegmented points. From here a *region growing* process begins using the lowest curvature point as the first seed point. In each iteration the k-nearest neighbours of the current seed point are determined and their normals are compared to the estimated plane normal. A neighbouring point is added to the current segment if its normal does not deviate from the plane normal beyond an angle threshold. A qualified neighbour is also used as a new seed point for further region growing if its curvature is sufficiently small. When no more points can be added to the current segment, a plane is considered to be fully segmented. Afterwards, the whole process restarts with the remaining set of unsegmented points, until the entire cloud has been processed. The pseudocode of the algorithm is listed in Algorithm 1.

Algorithm 1: Curvature-Based Plane Segmentation.

```
Input: 3D point cloud made of points p_i ∈ ℝ^3 with normals
       n_i and curvatures c_i
       θ_th angle threshold
       c_th curvature threshold
Output: Set of planar segments
while points remain unsegmented or the queue is not empty do
    if the queue is empty then
        pick a seed point p_s with the lowest curvature
        set the plane normal n_p to be the normal of p_s
    else
        pop out a seed point p_s from the queue
    mark p_s as segmented
    compute the k-nearest neighbours of p_s
    foreach unsegmented neighbour p_i do
        if arccos(n_p, n_i) < θ_th then
            add p_i to the current segment, mark p_i segmented
            if c_i < c_th then
                add p_i to the queue
    if the queue is empty then
        output the current segment as a plane
```

There are two major differences between this algorithm and the algorithm of Rabbani *et al.* [17]. The first difference is the integration of new points into the current segment. The original algorithm always updates the normal used for the integration of new points from the current seed point which may introduce points belonging to areas of moderate curvature in extracted segments. In this algorithm, we fix the plane normal to the normal of the first seed point and use only this normal throughout the segmentation of a single plane. Given that the first seed point has the lowest curvature available, its normal can be assumed to be a good estimation for the normal of the entire planar segment. With this modification we avoid the detection of smoothly-connected shapes, such as spheres and cylinder-like structures.

The second modification is concerned with the estimation of point curvature. The algorithm of Rabbani *et al.* [17] uses the residual of a least-squares plane fit as a substitute for curvature. In this algorithm we directly estimate the curvature using the original points. A necessary preprocessing step for this algorithm is normal estimation for point clouds. This is accomplished by local Principal Component Analysis (PCA) [19]. The PCA method for normal estimation also provides the curvature quantity using the following equation [20]

$$c = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2},\ \lambda_0 \leq \lambda_1 \leq \lambda_2, \tag{1}$$

where λ_0, λ_1 and λ_2 are the eigenvalues from the PCA process. These eigenvalues λ_0, λ_1 and λ_2 indicate the smallest, medium and largest variation along the directions specified by their corresponding eigenvectors. For points belonging to an ideal plane, we have $\lambda_0 = 0$ and hence $c = 0$. In the presence of noise, variable c becomes larger than zero.

This curvature-based algorithm works with two parameters. The first parameter θ_{th} specifies the maximum angle between the estimated plane normal and the normal of a potential point on the plane. Typically a 10° angle works well for noisy point clouds. The second parameter c_{th} is the curvature threshold, which is used to verify whether a point should be designated as a seed point for region growing. Empirically this threshold is set to a value below 0.1. Increasing the angle threshold allows planar extraction in noisier point clouds, while increasing the curvature threshold adds tolerance for surfaces which gradually become less planar.

5. Incremental Planar Segmentation

In this section we describe our method for incrementally segmenting planes from a point cloud map which is being incrementally produced in real-time by a dense mapping system, e.g. Kintinuous [3]. Our method involves maintaining a pool of unsegmented points which are either segmented as new planes, added to existing planes or deemed to not belong to any planar segment. Firstly we define a distance-based plane merging method that determines whether or not to merge two planar segments based on the distance between the points in each segment. We list this as Algorithm 2 and henceforth refer to it as the $mergePlanes$ method. In our experiments we use a distance threshold d_{th} of 0.08m.

Algorithm 2: Method for merging two planar segments.

Input: $\mathcal{A}$ planar segment with normal $\mathcal{A}_\mathbf{n}$, point cloud $\mathcal{A}_\mathcal{C}$ and timestep $\mathcal{A}_t$
$\mathcal{B}$ planar segment with normal $\mathcal{B}_\mathbf{n}$ and point cloud $\mathcal{B}_\mathcal{C}$
$\mathcal{B}_\mathcal{H}$ concave hull of $\mathcal{B}$
d_{th} distance threshold
Output: True or False if segments were merged or not
foreach *point* $\mathbf{h}_i$ *in* $\mathcal{B}_\mathcal{H}$ **do**
  if $\exists \mathcal{A}_{\mathcal{C}_k}$ *s.t.* $\|\mathbf{h}_i - \mathcal{A}_{\mathcal{C}_k}\|_2 < d_{th}$ **then**
    $\mathcal{A}_\mathbf{n} \leftarrow (\mathcal{A}_\mathbf{n}|\mathcal{A}_\mathcal{C}| + \mathcal{B}_\mathbf{n}|\mathcal{B}_\mathcal{C}|)/(|\mathcal{A}_\mathcal{C}| + |\mathcal{B}_\mathcal{C}|)$
    $\mathcal{A}_t \leftarrow 0$
    append $\mathcal{B}_\mathcal{C}$ to $\mathcal{A}_\mathcal{C}$
    compute kd-tree of $\mathcal{A}_\mathcal{C}$
    return True
return False

5.1. Segment Growing

Assuming the input to our system is a small part of a larger point cloud map that is being built up over time we must define a method for growing existing planar segments that were found in our map in the previous timestep of data acquisition. We maintain a persistent pool of unsegmented points $\mathcal{M}$ where each point $\mathcal{M}_i \in \mathbb{R}^3$ and also contains a timestep value $\mathcal{M}_{i_t}$, initially set to zero. When a new set of points are added to the map, they are added to the set $\mathcal{M}$, which is then sorted by the curvature of each point. A batch segmentation of $\mathcal{M}$ is then performed (as described in Section 4), producing a set of newly segmented planes $\mathcal{N}$. From here we perform Algorithm 3, which will grow any existing segments and also populate the set $\mathcal{S}$, that maintains a list of pairs of planes which are similar in orientation but not close in space. Algorithm 4 lists the method for merging similar planes that eventually grow close enough in space to be merged together. In our experiments we use a normal merge threshold n_{th} of 15°, a timestep threshold t_{th} of 7 and a timestep distance threshold td_{th} of 3.75m.

Algorithm 3: Method for growing planar segments.

Input: $\mathcal{N}$ set of new planar segments with normals $\mathcal{N}_{i_\mathbf{n}}$, point clouds $\mathcal{N}_{i_\mathcal{C}}$ and timesteps $\mathcal{N}_{i_t}$
$\mathcal{Q}$ set of existing planar segments with normals $\mathcal{Q}_{i_\mathbf{n}}$, point clouds $\mathcal{Q}_{i_\mathcal{C}}$ and timesteps $\mathcal{Q}_{i_t}$
$\mathcal{C}_t$ current position of sensor producing the map
n_{th} normal merge threshold
t_{th} timestep threshold
td_{th} timestep distance threshold
Output: $\mathcal{S}$ set of pairs of similar but non-merged segments
foreach *newly segmented plane* $\mathcal{N}_i$ **do**
  $\mathcal{R} \leftarrow \varnothing$
  $gotPlane \leftarrow$ False
  foreach *existing plane* $\mathcal{Q}_i$ **do**
    if $\neg \mathcal{Q}_{i_{finalised}}$ *and* $\arccos(\mathcal{N}_{i_\mathbf{n}}, \mathcal{Q}_{i_\mathbf{n}}) < n_{th}$ **then**
      compute concave hull $\mathcal{H}$ of $\mathcal{N}_i$
      $gotPlane \leftarrow mergePlanes(\mathcal{Q}_i, \mathcal{N}_i, \mathcal{H})$
      if $gotPlane$ **then**
        break
      else
        add $\mathcal{Q}_i$ to $\mathcal{R}$
  if $\neg gotPlane$ **then**
    compute kd-tree of $\mathcal{N}_{i_\mathcal{C}}$
    $\mathcal{N}_{i_t} \leftarrow 0$
    add $\mathcal{N}_i$ to $\mathcal{Q}$
    foreach *similar plane* $\mathcal{R}_i$ **do**
      add $(\mathcal{N}_i, \mathcal{R}_i)$ tuple to $\mathcal{S}$
  remove all points $\mathcal{N}_{i_\mathcal{C}}$ from $\mathcal{M}$
foreach *existing plane* $\mathcal{Q}_i$ **do**
  $\mathcal{Q}_{i_t} \leftarrow \mathcal{Q}_{i_t} + 1$
  if $\mathcal{Q}_{i_t} > t_{th}$ *and* $\forall \mathbf{q} \in \mathcal{Q}_{i_\mathcal{C}}, \|\mathcal{C}_t - \mathbf{q}\|_2 > td_{th}$ **then**
    $\mathcal{Q}_{i_{finalised}} \leftarrow True$
foreach *pair of similar planes* $\mathcal{S}_i$ **do**
  if $\mathcal{S}_{i_1 finalised}$ *or* $\mathcal{S}_{i_2 finalised}$ **then**
    delete $\mathcal{S}_i$

Each time new data is added to the map Algorithms 3 and 4 are run, after which the timesteps values of all remaining unsegmented points in $\mathcal{M}$ are incremented by 1. Points with a timestep value above t_{th} are removed from the point pool and marked as non-planar. Algorithm 3 will add new segments to the map, grow recently changed segments and ensure that similar planes which have the potential to grow into each other are kept track of. By including both spatial and temporal thresholds it ensures that the process scales well over time and space. Algorithm 4 merges segments which may not have initially been close together in space but have grown near to each other over time. Figure 3 shows an example of the incremental planar segmentation process in action.

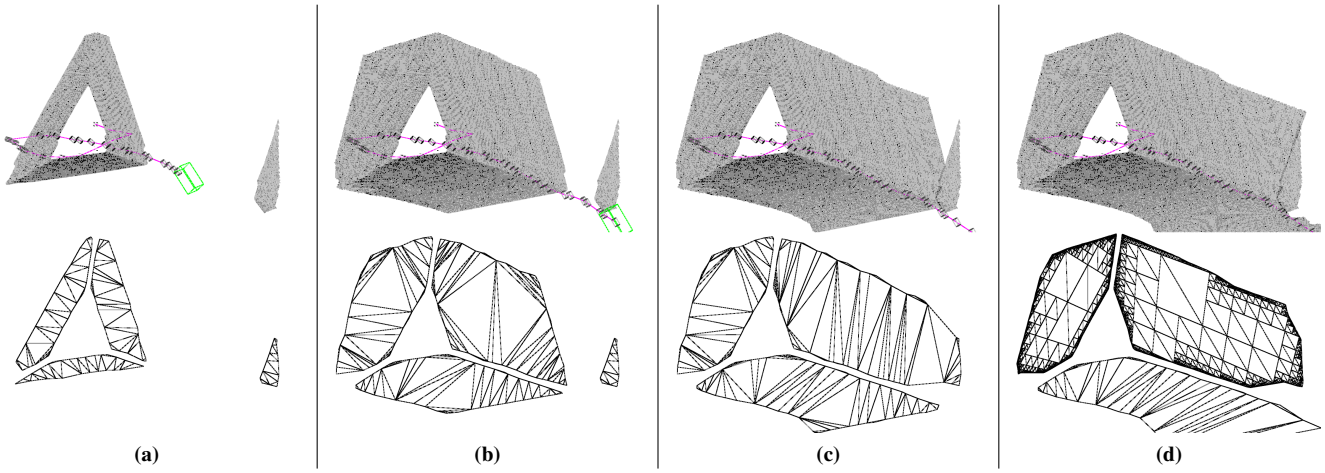


Figure 3: Incremental planar segmentation shown with point cloud, camera position (in green), camera trajectory (in pink) shown above and resulting planar segments below. From left to right: (a) initially there are four segments extracted from the point cloud; (b) as the camera moves and more points are provided, the three segments on the left are grown (as described in Algorithm 3); (c) the upper-right most segment grows large enough to be merged with the small segment on the right (as in Algorithm 4); (d) once the camera has moved far enough away from the two upper segments they are finalised.

Algorithm 4: Merging segments that have grown closer.

Input: $\mathcal{S}$ set of pairs of similar but non-merged segments with point clouds $\mathcal{S}_{i_C}$ and alpha values $\mathcal{S}_{i_\alpha}$
$\mathcal{Q}$ set of existing planar segments

foreach *pair of similar planes* $\mathcal{S}_i$ **do**
  $gotPlane \leftarrow$ False
  if $|\mathcal{S}_{i_{1C}}| > |\mathcal{S}_{i_{2C}}|$ **then**
    swap $\mathcal{S}_{i_1} and \mathcal{S}_{i_2}$
  if $\neg(\mathcal{S}_{i_{1\alpha}} == |\mathcal{S}_{i_{1C}}|)$ **then**
    compute concave hull $\mathcal{S}_{i_{1\mathcal{H}}}$ of $\mathcal{S}_{i_1}$
    $\mathcal{S}_{i_{1\alpha}} \leftarrow |\mathcal{S}_{i_{1C}}|$
  $gotPlane \leftarrow mergePlanes(\mathcal{S}_{i_2}, \mathcal{S}_{i_1}, \mathcal{S}_{i_{1\mathcal{H}}})$
  if $gotPlane$ **then**
    foreach *existing plane* $\mathcal{Q}_i$ **do**
      if $\mathcal{Q}_i == \mathcal{S}_{i_1}$ **then**
        delete $\mathcal{Q}_i$
        break
    foreach *pair of similar planes* $\mathcal{S}_j$ **do**
      if $i == j$ **then**
        continue
      if $\mathcal{S}_{j_1} == \mathcal{S}_{i_1}$ **then**
        $\mathcal{S}_{j_1} \leftarrow \mathcal{S}_{i_2}$
      else if $\mathcal{S}_{j_2} == \mathcal{S}_{i_1}$ **then**
        $\mathcal{S}_{j_2} \leftarrow \mathcal{S}_{i_2}$
      if $\mathcal{S}_{j_1} == \mathcal{S}_{j_2}$ **then**
        delete $\mathcal{S}_j$
    delete $\mathcal{S}_i$

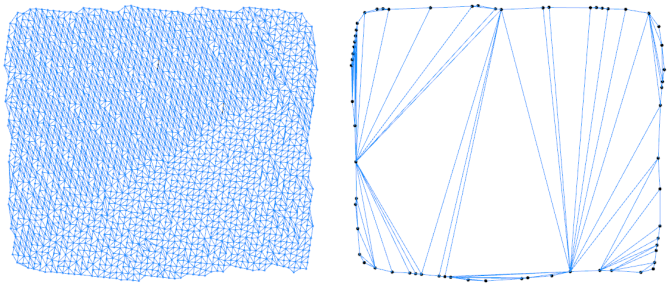


Figure 4: Undesirable planar triangulation: the left GPT mesh over-represents the shape while the right boundary-based Delaunay triangulation produces unnatural skinny triangles.

6. Triangulation of Planar Segments

In this section, our algorithm for planar segment decimation and triangulation is described. A simplified mesh of a planar segment is generated by removing redundant points that fall within the boundary of the segment. In the following text the input planar segment is denoted as $\mathcal{P}$, made up of points $\mathbf{p} \in \mathbb{R}^3$. With coloured point clouds, each point $\mathbf{p}$ also contains (R, G, B) colour components.

6.1. QuadTree-Based Decimation

Planar segments have a simple shape which can be well described by points on the boundary of the segment. Interior points only add redundancy to the surface representation and complicate the triangulation results. Figure 4 shows an example of this where the planar segment is over-represented with thousands of triangles generated with the GPT algorithm using all planar points. However, a naïve solution that removes all interior points and triangulates only with boundary points normally leads to skinny triangles (which can result in precision issues when performing computations on the mesh), again shown

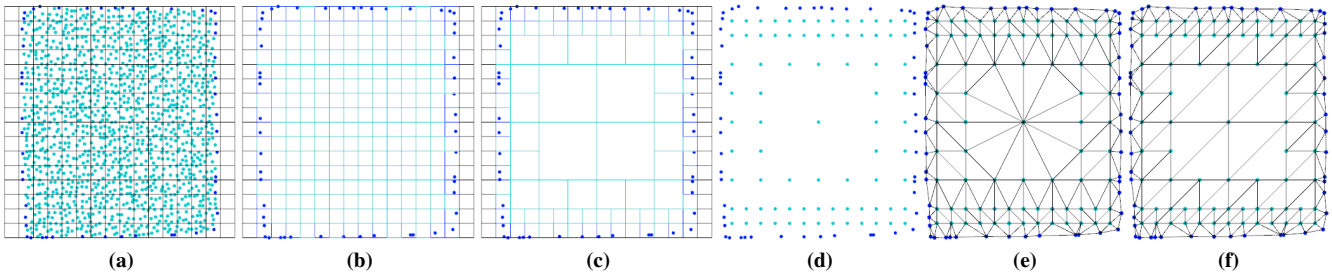


Figure 5: Planar decimation and triangulation (boundary and interior points are dark blue and teal, respectively), from left to right: (a) initialize by subdividing the quadtree bounding box; (b) classify nodes into interior (teal), boundary (dark blue) and exterior (black); (c) merge interior nodes; (d) generate vertices; (e) point-based triangulation; (f) polygon-based triangulation.

in Figure 4. With these observations in mind, the quadtree proves to be a useful structure to decimate the interior points of a segment while preserving all boundary points for shape recovery [5].

6.1.1. Preprocessing

To prepare a planar segment for decimation it is first denoised and aligned to the x-y axes. We employ PCA over the planar segment to compute a least-squares plane fit as well as an affine transformation T for x-y axes alignment, after which all points belonging to the segment are orthogonally projected onto the axis aligned best fit plane. The aligned planar segment is denoted as $\mathcal{P}_t$. Afterwards, the boundary points of $\mathcal{P}_t$ are extracted as an α-shape [21, 22]. We denote the boundary as a concave hull $\mathcal{H}$ of the planar segment, which is an ordered list of vertices describing a polygon for which $\forall \mathbf{p} \in \mathcal{P}_t$ and $\mathbf{p} \notin \mathcal{H}$, $\mathbf{p}$ is inside the polygon.

6.1.2. Decimation

Planar segment point decimation consists of four steps as shown in Figure 5. Firstly, a quadtree is constructed by subdividing the bounding box of $\mathcal{P}_t$ into a uniform grid of small cells. Typically the 2D bounding box is non-square, in which case the smallest side is extended to equalize the width and height. The resulting bounding box $\mathbf{b}$ is composed of a minimum point $\mathbf{b}_{min}$ and a maximum point $\mathbf{b}_{max}$, with a dimension $\mathbf{s} = \mathbf{b}_{max} - \mathbf{b}_{min}$. Secondly, the quadtree nodes are classified as either interior, boundary or exterior. An *interior* node is fully contained within the polygon $\mathcal{H}$, while an *exterior* node is fully outside. All others are *boundary* nodes, which intersect $\mathcal{H}$. Thirdly, the interior nodes of the quadtree are merged to create nodes of variable size, typically largest around the center and becoming increasingly fine-grained when approaching the boundary. When a parent node contains only interior children, the four child nodes are merged into one. The merged node is then also classified as interior, allowing further recursive merging with its siblings. Finally, the corner points of the remaining interior nodes are extracted as the new internal vertices $\mathcal{I}$ of $\mathcal{P}_t$, while all boundary points $\mathcal{H}$ are preserved.

6.2. Triangulation

We provide two methods for triangulation of a simplified planar segment: 1) a low-complexity Point-Based Triangulation and 2) an alternative Polygon-Based Triangulation. Both methods make use of the Constrained Delaunay Triangulation (CDT) [23].

6.2.1. Point-Based Triangulation

The point-based approach is a low-complexity triangulation method, where CDT is directly applied to the decimated segment. The ordered boundary vertices $\mathcal{H}$ serve as constraining edges and the inner vertices $\mathcal{I}$ are used as input points. An example output is shown in Figure 5. Point-based triangulation has all of the advantages of Delaunay triangulation but does produce more triangles than the polygon-based approach described next.

6.2.2. Polygon-Based Triangulation

The regular grid pattern of the inner vertices $\mathcal{I}$ immediately lends itself to a simple triangulation strategy, where two right-angled triangles are created over each interior node of the merged quadtree. To complete the triangulation, the space between the interior right-angled triangles and the boundary points $\mathcal{H}$ is triangulated using CDT. Two sets of constraining edges are input to the CDT, one being $\mathcal{H}$ and the other being a rectilinear isothetic polygon that bounds interior triangles. This two-step triangulation is similar to the QTB algorithm of [5]. However, a major difference lies in how the boundary points are connected. With our CDT-based approach, we avoid overlapping triangles and artificial holes that would normally be produced by the QTB algorithm.

Efficient computation of the polygon which exactly bounds the interior vertices $\mathcal{I}$ is non-trivial, since the interior nodes provide only sparse spatial information for geometric operations. We invoke a solution that maps the interior vertices onto a binary image, where the bounding polygon can be easily extracted using a greedy nearest-neighbour tracing algorithm normally used in image processing [24].

The binary image is represented by an $n \times n$ array, where $n = 2^{d+1} + 1$ and d is the quadtree depth. This provides a 2D grid large enough to represent the empty space between the two

1	2	**2**	2	**2**	2	**2**	2	**2**	2	**2**	2	**2**	2	**2**	2	**2**	2	**2**	2	**2**	2	**2**	2	**1**
2		4		4		4		4		4		4		4		4		4		4		4		2
2	4	**4**	4	**4**	4	**4**	4	**4**	4	**4**	4	**4**	4	**4**	4	**4**	4	**4**	4	**4**	4	**4**	4	**2**
2				4				4				4				4				4				2
2				4				4				4				4				4				2
2				4				4				4				4				4				2
2	4	4	4	**4**	4	4	4	**4**	4	4	4	**4**	4	4	4	**4**	4	4	4	**4**	4	4	4	**2**
2				4								4								4				2
2				4								4								4				2
2				4								4								4				2
2	4	4	4	**4**								4								**4**	4	4	4	**2**

Figure 6: Degree grid of the upper planar segment in Figure 5 (0-valued cells hidden). The underlined bold values are the degrees of the inner vertices $\mathcal{I}$.

vertices of any edge. To project a vertex $\mathbf{v} \in \mathcal{I}$ onto the array, a mapping function $f : \mathbb{R}^3 \rightarrow \mathbb{N}^2$ is defined by

$$f(\mathbf{v}) = \frac{n(\mathbf{v} - \mathbf{b}_{min})}{\mathbf{s}}, \tag{2}$$

where $\mathbf{b}$ is the bounding box and $\mathbf{s}$ is its dimension. The division is performed on an element-by-element basis. Given that $\mathcal{I}$ is aligned to the x-y axes, function f effectively maps from $\mathbb{R}^2$ to $\mathbb{N}^2$. We associate two elements with each array cell: a reference to the mapped vertex (effectively implementing f^{-1}) and a degree value to quantify vertex connectivity in the underlying quadtree. Initially, the degree is zero for all cells. During the triangulation of $\mathcal{I}$, the degree grid is populated. When a vertex is extracted from the merged quadtree, the reference of the corresponding cell is updated and its degree is increased by 1. This policy alone cannot fully recover the degree of a given vertex, since only the two ends of an edge are obtained from quadtree vertices. To overcome this problem, all cells between the two ends of an edge also have their degree increased by 2. Figure 6 shows a part of the degree grid of the planar segment in Figure 5. If we consider the interior triangulation to be a graph, the 2D degree grid resolves the degree of each vertex. All non-zero cells are treated as "1-valued" foreground pixels and the rest as "0-valued" background pixels in the binary image representation.

7. Texture Generation

In this section we present our texture generation algorithm for planar segments using dense coloured point clouds. Due to the significant loss of coloured vertices during decimation, the appearance of a simplified planar segment is greatly diminished. We therefore generate textures prior to decimation for the purpose of texture mapping the simplified planar mesh.

We generate textures by projecting the vertex colours of the dense planar segment onto a 2D RGB texture $\mathcal{E}(x, y) \in \mathbb{N}^3$. We define a texture resolution $\mathbf{d}$ as some resolution factor r times $\mathbf{s}$, where $\mathbf{s}$ assumes the size of the bounding box $\mathbf{b}$. In our experiments a value of $r = 100$ provides good-quality textures. The resolution factor can also be automatically computed based on point cloud density. Each pixel $\mathbf{a} \in \mathcal{E}$ is first mapped to a 3D point $\mathbf{v}$ by a mapping function $g : \mathbb{N}^2 \rightarrow \mathbb{R}^3$, defined as

(a) **(b)**

Figure 7: Texture generation, from left to right: (a) plane segment from a coloured point cloud; (b) generated texture.

$$g(\mathbf{a}) = \frac{\mathbf{as}}{\mathbf{d}} + \mathbf{b}_{min}, \tag{3}$$

with an element-by-element calculation. Since $\mathcal{P}_t$ is aligned to the x-y axes, the function g effectively maps to $\mathbb{R}^2$. A coloured point corresponding to $\mathbf{v}$ in $\mathcal{P}_t$ is found by a nearest neighbour search using a kd-tree. We have chosen this approach as it produces good-quality textures while being computationally inexpensive. However, it can be easily extended to produce even higher-quality textures by averaging a number of k-nearest neighbours. Algorithm 5 describes the texture generation process. Figure 7 shows an input planar segment and the output texture.

Algorithm 5: Vertex colour to texture.

Input: $\mathcal{P}_t$ set of transformed input vertices
$\mathcal{H}$ concave hull of $\mathcal{P}_t$
Output: $\mathcal{E}$ 2D RGB texture
foreach *pixel* $\mathbf{p}$ *in* $\mathcal{E}$ **do**
  $\mathbf{v} \leftarrow g(\mathbf{p})$
  if $\mathbf{v}$ *is inside* $\mathcal{H}$ **then**
    $\mathbf{n} \leftarrow$ nearest-neighbour of $\mathbf{v}$ in $\mathcal{P}_t$
    $\mathbf{p} \leftarrow (\mathbf{n}_R, \mathbf{n}_G, \mathbf{n}_B)$
  else
    $\mathbf{p} \leftarrow (0, 0, 0)$

When texture mapping the final planar mesh, the uv texture coordinates $\mathcal{U}$ for the vertices $\mathcal{O}$ of each face are computed with the inverse function $g^{-1} : \mathbb{R}^3 \rightarrow \mathbb{N}^2$, derived from Equation (3) as

$$g^{-1}(\mathbf{v}) = \frac{\mathbf{d}(\mathbf{v} - \mathbf{b}_{min})}{\mathbf{s}}. \tag{4}$$

With x-y axes aligned points, g^{-1} is actually mapping from $\mathbb{R}^2$. Algorithm 6 describes the uv-coordinates computation. The list $\mathcal{U}$ guarantees a 1-to-1 mapping to the set $\mathcal{O}$.

Any objects lying on a planar segment are completely excluded from the texture and not projected onto the plane. In fact, the generated texture implicitly provides the Voronoi diagram of the face of the object lying on any plane, which in turn provides position and orientation information of any object lying on a segmented plane, as shown in Figure 8.

Algorithm 6: uv texture coordinate calculation.

Input: $\mathcal{O}$ set of final face vertices
Output: $\mathcal{U}$ uv texture coordinates for $\mathcal{O}$
foreach *vertex* $\mathbf{v}$ *in* $\mathcal{O}$ **do**
  $\mathbf{a} \leftarrow g^{-1}(\mathbf{v})$
  $u \leftarrow \frac{\mathbf{a}_x}{\mathbf{d}_x}$
  $v \leftarrow 1.0 - \frac{\mathbf{a}_y}{\mathbf{d}_y}$
  Add (u, v) to $\mathcal{U}$

(a) (b)

Figure 8: Implicit object information from texture generation, from left to right: (a) input coloured point cloud; (b) generated texture with implicit Voronoi diagrams and locations of objects resting on the plane highlighted.

8. Evaluation and Results

In this section we evaluate our work with a series of experiments. We ran our C++ implementation on Ubuntu Linux 12.04 with an Intel Core i7-3930K CPU at 3.20 GHz with 16 GB of RAM. Four coloured point clouds of real-world environments were used in the experiments, as shown in Figure 9a. These datasets encompass a wide variation in the number of points, planar segments and their geometry. All four datasets have been acquired with an implementation of the Kintinuous dense RGB-D mapping system [3]. In all experiments we use the same thresholds in each algorithm, which we initially chose with preliminary experiments to provide visually and geometrically accurate results. We base a number of our comparison metrics on those originally provided by Ma *et al.* [5]. We list a number of statistics on the datasets in Table 1

8.1. Incremental Segmentation Performance

To evaluate the performance of the incremental segmentation process listed in Section 5 we compare the batch segmentations to the incremental segmentations of the four datasets both qualitatively and quantitatively. We use the open source software CloudCompare (`http://www.danielgm.net/cc/`) to

Table 1: Statistics on each of the evaluated datasets. Frame rates were subject to slight variation in practice.

Dataset	1	2	3	4
Area (m)	7×23×23	5×5×7	4×4×20	11×9×10
Capture time (s)	48	96	58	217
Frame rate (Hz)	30	30	15	15
Points per frame	619	430	3218	1729

Table 2: Incremental versus batch planar segmentation statistics. All values shown are in metres, on the distances between all vertices in the incrementally segmented model and the nearest triangles in the batch segmented model.

Dataset	1 (m)	2 (m)	3 (m)	4 (m)
Mean	0.020	0.038	0.111	0.028
Median	0.015	0.004	0.015	0.010
Std.	0.021	0.108	0.251	0.065
Min	0.000	0.000	0.000	0.000
Max	0.157	0.823	1.389	0.719

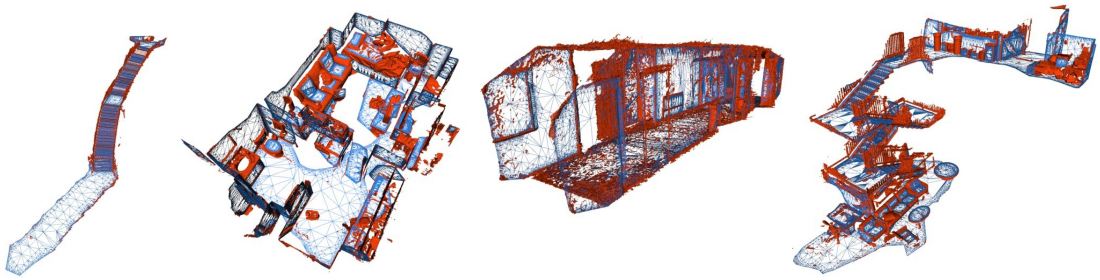
align the batch and incremental models of each dataset together to compute statistics. We quantify the quality of the incremental segmentation versus the batch segmentation by using the "cloud/mesh" distance metric provided by CloudCompare. The process involves densely sampling the batch planar model mesh to create a point cloud model which the incremental model is finely aligned to using Iterative Closest Point, an algorithm used to align two point clouds. Then, for each vertex in the incremental planar model, the closest triangle in the batch model is located and the perpendicular distance between the vertex and closest triangle is recorded. Five standard statistics are computed over the distances for all vertices in the incremental model: Mean, Median, Standard Deviation, Min and Max. These are listed for all four datasets in Table 2.

Figure 10 shows heatmap renderings of the "cloud/mesh" error of each incremental segmentation compared to the batch segmentation. Notably in each dataset there are a number of highlighted green planes, these are planes which were not detected in the incremental segmentation model but exist in the batch segmentation. In general the incremental segmentation occasionally fails to segment small planar segments whereas the batch segmentation always finds all planes that match the criterion set out in Algorithm 1. Additionally, as the incrementally grown planes use a moving average for the planar segment normal, some planes may have a slightly different orientation when compared to the batch model. This is evident in particular in Figure 10 (a) and (c). Taking this qualitative information into account as well as the statistics in Table 2 we find that the incremental segmentation algorithm produces segmentations extremely close to what would be achieved using the batch process and is suitable to use in a real-time system that must generate and use the planar model online.

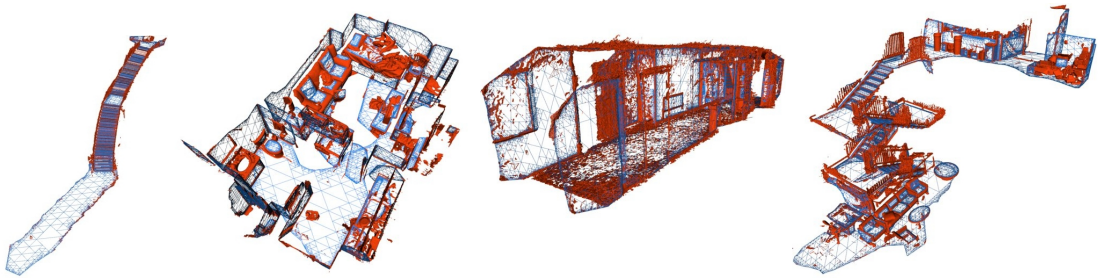
8.2. Triangulation Performance

To assess the triangulation performance, qualitative and quantitative evaluations are presented. A comparison of the triangulation algorithms is shown in Figure 9, where Figures 9b and 9c show highlighted point-based and polygon-based triangulations, respectively. Additionally we present fly-throughs in a video submission available at `http://www.youtube.com/watch?v=uF-I-xF3Rk0`. It can be seen that both algorithms produce a highly simplified triangulation, while preserving the principal geometry of the planar segments. The video also shows combined triangulations of all four datasets

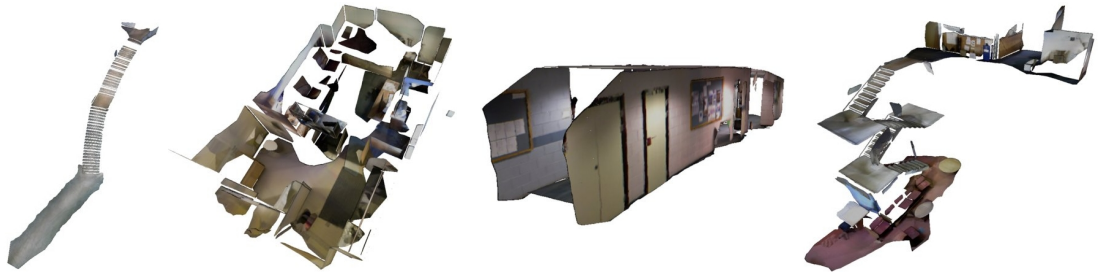
(a) Four dense coloured point cloud datasets used for evaluation, numbered 1, 2, 3 and 4 from left to right.



(b) Point-based triangulation, with planar and non-planar meshes highlighted in blue and orange, respectively.



(c) Polygon-based triangulation with planar and non-planar meshes highlighted in blue and orange, respectively.



(d) Textured simplified planar segments from each dataset.

(e) Complete 3D model with our proposed system.

Figure 9: Four evaluated datasets (numbered from 1 to 4 from left to right) with various triangulation and texturing results.

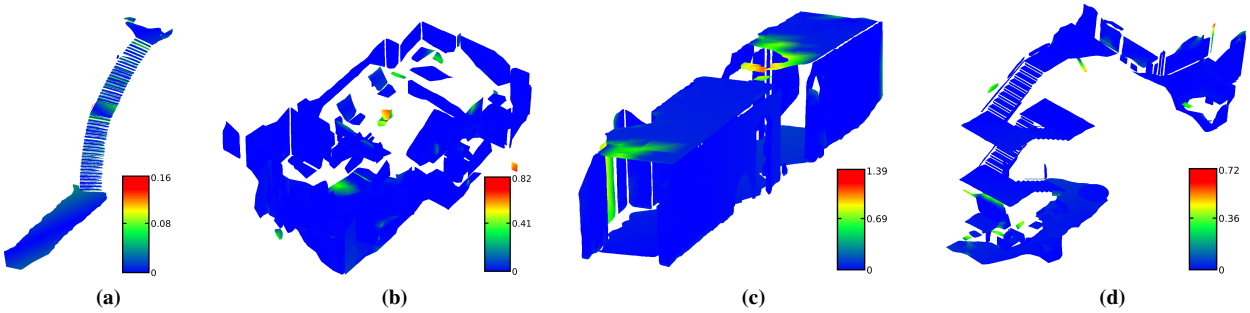


Figure 10: Heat maps based on the distances between all vertices in the incrementally segmented model and the nearest triangles in the batch segmented model for all four datasets. Colour coding is relative to the error obtained where blue is zero and tends towards green and red as the error increases. This figure is best viewed in full colour. All values shown are in metres.

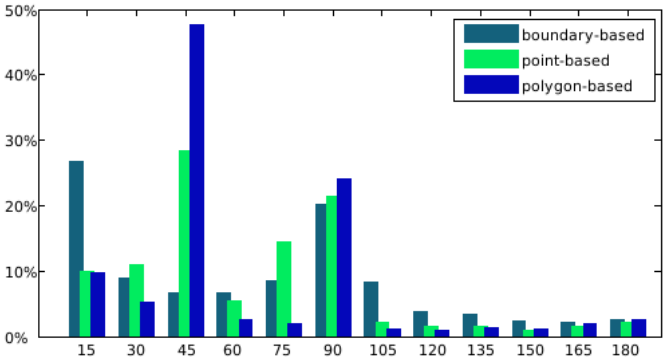


Figure 11: Triangulation quality measured with a normalised histogram of the angle distribution of planar meshes.

which use our polygon-based approach for planar segments in combination with the GPT mesh generated from non-planar segments.

Further assessment of mesh quality is done by measuring the angle distribution across meshes. A naïve simplified planar mesh is set as a baseline, which applies Delaunay triangulation to only the boundary points of a planar segment. The normalized distribution is shown in Figure 11, collected from the 400 planar segments of the four datasets. Taking this Figure into account along with the qualitative results shown in Figures 3 (d) and 5 (f) we can infer that approximately 80% of the triangles from the polygon-based triangulation are isosceles right-angle triangles, resulting from the quadtree-based triangulation. With point-based triangulation, the angles spread over 30°-90°, whereas the naïve boundary-based triangulation shows an even more random distribution. Defining a skinny triangle as one with a minimum angle $<15°$, the percentages of skinny triangles with boundary-based, point-based and polygon-based triangulation are 28%, 10% and 10%, respectively.

The effectiveness of planar segment decimation is also evaluated. Table 3 shows the point count for planar point decimation. Approximately 90% of the redundant points are removed with our algorithm, which is 15% more than the QTB algorithm, despite the fact that both algorithms are based on a quadtree. Part of this reduction gain comes from our triangulation methods, which add no extra points once decimation is completed, unlike the QTB algorithm. In Table 4, the mesh simplification statistics with triangle counts are also given. We take the triangle count of GPT for non-decimated planar segments as the baseline. In accordance with the point count reduction, both of our algorithms require no more than 10% of the amount of triangles of a non-decimated triangulation, and both perform better than the QTB algorithm.

Table 3: Planar point reduction with our decimation algorithm in comparison to the QTB algorithm.

Dataset	1	2	3	4
Total points	890,207	1,094,910	2,799,744	5,641,599
Planar points	540,230	708,653	1,303,012	2,430,743
QTB decimation	105,663	303,348	189,772	457,678
Our decimation	47,457	84,711	43,257	76,624

Table 4: Planar mesh simplification with our triangulation algorithms measured with triangle counts, in comparison to GPT and the QTB algorithm.

Dataset	1	2	3	4
GPT	1,020,241	1,350,004	2,293,599	4,665,067
QTB	90,087	288,833	182,648	433,953
Point-based	85,653	161,270	79,988	143,396
Polygon-based	76,367	130,642	66,413	118,275

8.3. Texture Generation Performance

In Figures 7 and 9d, generated textures are shown. The output textures incorporate almost all visual information contained in the original dense point cloud, enabling a photo realistic and aesthetically-pleasing textured 3D model.

8.4. Computational Performance

Lastly, we evaluate the computational efficiency of our algorithms. We firstly evaluate the incremental method for planar segmentation followed by the parallel system used for large-scale batch data processing.

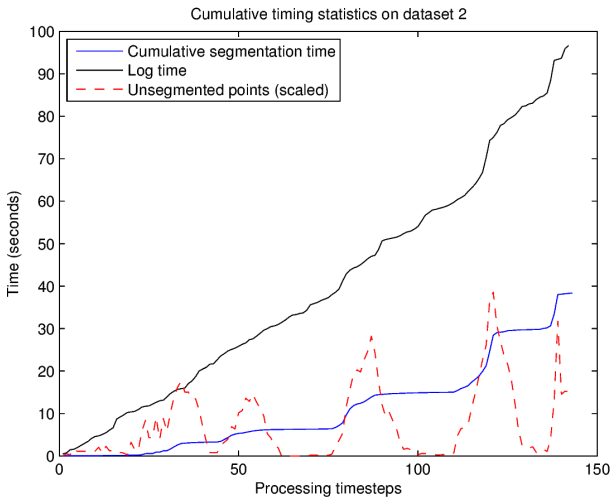


Figure 12: Graph comparing cumulative log file time (real world time) versus the time the incremental segmentation method spends processing data on dataset 2. Also shown is the size of the unsegmented point pool over time (linearly scaled).

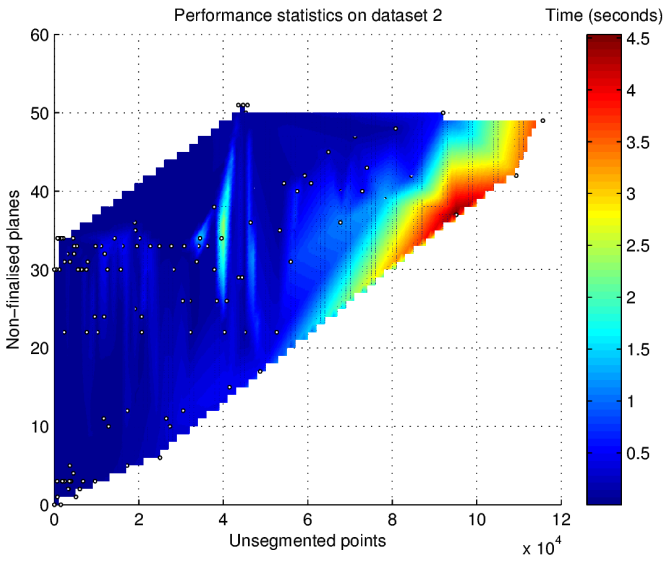


Figure 13: Heat map rendering showing the relationship between the number of unsegmented points, the number of non-finalised planes and the execution time of the incremental segmentation method. The white dots represent the samples used to generate the map (using linear interpolation).

8.4.1. Incremental Performance

In contrast to the results presented in Section 8.4.2 which include the full pipeline from batch planar segmentation to triangulation, we only evaluate the performance of the incremental planar segmentation algorithm listed in Section 5 here. We analyse the online performance of the algorithm as a component of the Kintinuous dense mapping system [3]. In this setting, small point cloud slices of the environment being mapping are provided to the incremental segmentation method gradually over time as the camera moves through the area. Figure 12 shows a plot of the cumulative log file time (real world time) versus the time spent incrementally growing planes with the process described in Section 5. It can be seen that throughout the mapping process the incremental segmentation method is processing the data it is provided with faster than it is being produced, meaning that real-time online operation is being achieved. Also shown is a linearly scaled line representing the size of the unsegmented point pool over time. The relationship between this line and the segmentation time is immediately evident in how the cumulative processing time of the segmentation method increases when the number of unsegmented points increases. Similarly, the total processing time ceases to increase as the number of unsegmented points tends to zero.

In Figure 13 the relationship between the number of non-finalised planes, the number of unsegmented points and the execution time of the segmentation method is visualised. It can be seen that as the number of unsegmented points becomes large the execution time increases quite a lot. However for a fixed number of unsegmented points, in particular above 6×10^4, an increased number of non-finalised planes improves performance. This suggests that growing and merging existing planes is computationally cheaper than adding an entirely new plane. The peak around 4×10^4 unsegmented points came from an early on influx of many new points to the map. Along with the observation of apparent plateauing of cumulative processing timing in Figure 12 we can conclude that the number of unsegmented points introduced to the segmentation method at any one time influences computational performance the most. If too many are introduced, for example if the density of the input point cloud is too high or the mapping system is producing data too quickly, real-time performance may be hindered. However this can easily by remedied by downsampling the data provided to the segmentation algorithm.

8.4.2. Batch Performance

With batch processing the baseline for performance comparison is standard serial processing with the GPT and QTB algorithms. Table 5 shows the execution times. The point-based and polygon-based triangulations are approximately of the same speed, both 2 to 3 times faster than the GPT and QTB algorithms. The results also show that the texture generation algorithm is fast in execution, processing multi-million point datasets in less than 2 seconds. Examining the bottom half of Table 5, it is clear that the parallel system architecture has a profound effect on the overall performance. The execution time decreases with an increasing number of triangulation threads. An effect of diminishing returns becomes apparent as the number of triangulation threads increases, due to the overhead associated with the parallel implementation. However, as the per-thread workload increases, such as inclusion of texture generation, the overhead of parallelization becomes amortised. Other aspects of particular datasets affect the performance gains achieved with parallelisation. In particular, in dataset 1 the abundance of small thin planes appears to have a significant effect on performance, where going from three to five threads results in slightly worse performance, again attributed to the overhead associated with starting new threads for small planes

Table 5: Efficiency of triangulation and the parallel architecture, measured in seconds. The 1:x ratio denotes 1 segmentation thread with x triangulation threads.

Dataset	1	2	3	4
Number of planar segments	101	116	66	117
Serial GPT	18.6	24.3	44.2	91.1
Serial QTB	16.7	18.7	38.3	73.1
Serial point-based	6.9	9.8	17.7	40.2
Serial polygon-based	6.9	9.5	17.8	40.0
Serial polygon-based (texture)	8.3	10.0	20.3	41.4
1:1 Polygon-based	6.4	8.1	15.1	33.8
1:1 Polygon-based (texture)	7.6	8.5	17.4	35.2
1:3 Polygon-based	3.6	4.2	8.3	19.2
1:3 Polygon-based (texture)	4.4	4.1	9.2	19.6
1:5 Polygon-based	3.7	3.5	7.9	16.1
1:5 Polygon-based (texture)	4.7	3.5	8.7	16.2

which do not perform processing for very long.

Both point-based and polygon-based triangulation yield accurate and computationally efficient planar segment triangulations with significant point and triangle count reductions, both exceeding the performance of the QTB algorithm. The point-based approach is of low complexity and maintains good triangular mesh properties that are desirable for lighting and computer graphics operations. The polygon-based approach yields higher point and triangle count reductions with a more regularized mesh pattern, capturing information about the scene in the form of principal geometric features, such as the principal orientation of a planar segment. While the polygon-based method produces less triangles, it does generate T-joints in the mesh. Such features are detrimental when employing Gouraud shading and other lighting techniques to render a mesh with coloured vertices. The polygon-based and point-based methods offer a trade-off depending on the desired number of triangles or the intended use of the final triangulation. With robot navigation in mind, the low polygon-count models achieved with our system are suitable for use in a primitives-based localization system, such as the KMCL system of Fallon *et al.* [4].

The gaps between planar and non-planar triangulations are apparent. The gap can also be closed by including the boundary vertices of the segmented planes into the non-planar segment GPT triangulation, as shown in Figure 14. The number of boundary vertices can be increased with a smaller alpha value when computing the concave hull of each segment or by linearly interpolating between boundary vertices. Extra vertices can also be extracted from the vertex degree grid used in polygon-based triangulation. In our system we chose to leave these gaps open, as this separation gives an easier visual understanding of any map, implicitly providing a separation between structural features (like walls, table tops) and "object" features, useful in automatic scene understanding, manipulation and surface classification.

Figure 14: Joining of GPT mesh with planar segment triangulations. Left shows unjoined segments and right shows segments joined with interpolated boundary vertices.

9. Conclusions

In this paper we have studied the problem of the incremental segmentation and triangulation of planar segments from dense point clouds with a focus on quality and efficiency. Three significant contributions are made. Firstly, we have introduced a computationally feasible and strong performing method for the incremental segmentation of planar segments in a gradually expanding point cloud map. Our method is suitable for real-time online operation and produces segmentations faithful to those achieved in a batch processed scenario. Secondly, we have made a strong improvement on planar segment decimation and triangulation. Both of the presented point-based and polygon-based triangulation methods produce a more accurate, simpler and robust planar triangulation than the existing QTB algorithm, while including the ability to join up planar triangulations with the dense non-planar triangulation. With these two algorithms approximately 90% of input planar points are removed, and the planar segments are triangulated with no more than 10% of the amount of triangles required without decimation. Thirdly, we have developed a computationally inexpensive algorithm to automatically generate high-quality textures for planar segments based on coloured point clouds. The results show that our system provides a computationally manageable map representation for real-world environment maps and also generates a visually appealing textured model in a format useful for real-time robotic systems.

References

[1] L. Ma, T. Whelan, E. Bondarev, P. H. N. de With, J. McDonald, Planar simplification and texturing of dense point cloud maps, in: European Conference on Mobile Robotics, ECMR, Barcelona, Spain, 2013.

[2] P. Henry, M. Krainin, E. Herbst, X. Ren, D. Fox, RGB-D mapping: Using Kinect-style depth cameras for dense 3D modeling of indoor environments, The Int. Journal of Robotics Research.

[3] T. Whelan, M. Kaess, J. Leonard, J. McDonald, Deformation-based loop closure for large scale dense RGB-D SLAM, in: IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems, IROS, Tokyo, Japan, 2013.

[4] M. F. Fallon, H. Johannsson, J. J. Leonard, Efficient scene simulation for robust Monte Carlo localization using an RGB-D camera, in: IEEE Intl. Conf. on Robotics and Automation (ICRA), 2012.

[5] L. Ma, R. Favier, L. Do, E. Bondarev, P. H. N. de With, Plane segmentation and decimation of point clouds for 3d environment reconstruction, in: Proc. the 10th Annual IEEE Consumer Communications & Networking Conference, 2013.

[6] M. Kazhdan, M. Bolitho, H. Hoppe, Poisson surface reconstruction, in: Proc. of the 4th Eurographics Symposium on Geometry Processing., 2006, pp. 61–70.

[7] A. C. Jalba, J. B. T. M. Roerdink, Efficient surface reconstruction from noisy data using regularized membrane potentials, IEEE Trans. on Image Processing 18 (5) (2009) 1119–1134. doi:10.1109/TIP.2009.2016141.

[8] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, G. Taubin, The ball-pivoting algorithm for surface reconstruction, IEEE Trans. on Visualization and Computer Graphics 5 (4) (1999) 349–359. doi:10.1109/2945.817351.

[9] C. E. Scheidegger, S. Fleishman, C. T. Silva, Triangulating point set surfaces with bounded error, in: Proc. of the 3rd Eurographics symposium on Geometry Proc., Eurographics Association, 2005.

[10] M. Gopi, S. Krishnan, A fast and efficient projection-based approach for surface reconstruction, in: Proc. Computer Graphics and Image Processing., 2002, pp. 179–186. doi:10.1109/SIBGRA.2002.1167141.

[11] Z. C. Marton, R. B. Rusu, M. Beetz, On fast surface reconstruction methods for large and noisy point clouds, in: Proc. IEEE Inter. Conf. Robotics and Automation ICRA '09, 2009, pp. 3218–3223. doi:10.1109/ROBOT.2009.5152628.

[12] B. Oehler, J. Stueckler, J. Welle, D. Schulz, S. Behnke, Efficient multi-resolution plane segmentation of 3d point clouds, in: Intelligent Robotics and Applications, Vol. 7102 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 145–156. doi:10.1007/978-3-642-25489-5_15.

[13] J. Deschaud, F. Goulette, A fast and accurate plane detection algorithm for large noisy point clouds using filtered normals and voxel growing, in: 3DPVT10, 2010.

[14] A. Golovinskiy, V. G. Kim, T. Funkhouser, Shape-based recognition of 3D point clouds in urban environments, ICCV.

[15] A. Golovinskiy, T. Funkhouser, Min-cut based segmentation of point clouds, in: IEEE Workshop on Search in 3D and Video (S3DV) at ICCV, 2009.

[16] J. Strom, A. Richardson, E. Olson, Graph-based segmentation for colored 3D laser point clouds, in: Proc. of IEEE/RSJ IROS, 2010.

[17] T. Rabbani, F. van Den Heuvel, G. Vosselmann, Segmentation of point clouds using smoothness constraint, Inter. Archives of Photo. Remote Sensing and Spatial Info. Sciences 36 (5) (2006) 1–6.

[18] F. Darema, D. George, V. Norton, G. Pfister, A single-program-multiple-data computational model for epex/fortran, Parallel Computing 7 (1) (1988) 11 – 24. doi:10.1016/0167-8191(88)90094-4.

[19] N. J. Mitra, A. Nguyen, L. Guibas, Estimating surface normals in noisy point cloud data, in: special issue of International Journal of Computational Geometry and Applications, Vol. 14, 2004, pp. 261–276.

[20] M. Pauly, M. Gross, L. P. Kobbelt, Efficient simplification of point-sampled surfaces, in: Proc. of the Conference on Visualization, VIS '02, IEEE Computer Society, 2002, pp. 163–170.

[21] M. Duckham, L. Kulik, M. Worboys, A. Galton, Efficient generation of simple polygons for characterizing the shape of a set of points in the plane, Pattern Recogn. 41 (10) (2008) 3224–3236. doi:10.1016/j.patcog.2008.03.023.

[22] B. Pateiro-López, A. Rodríguez-Casal, Generalizing the convex hull of a sample: The r package alphahull, Journal of Statistical Software 34 (i05).

[23] V. Domiter, B. Zalik, Sweep-line algorithm for constrained delaunay triangulation, Int. J. Geogr. Inf. Sci. 22 (4) (2008) 449–462. doi:10.1080/13658810701492241.

[24] J. Marquegnies, Document layout analysis in SCRIBO, Tech. Rep. CSI Seminar 1102, Research and Development Laboratory, EPITA (July 2011).