University of Otago

Department of Computer Science

COSC490 Project Report

Plane Detection in Point Clouds

Author:
Helena Crawford
(7562257)

Supervisor(s):
Assoc. Professor Steven Mills

October 5, 2020

Abstract

Finding geometric planes in dense point cloud models can give a useful and concise geometric representation of the object being modelled. In this project, accurate and computationally efficient methods of detecting these planes are explored. A parallel implementation of Random Sample and Consensus (RANSAC) has been developed and optimized for an 8.5x speedup, scaling up to 12 threads across large and small point clouds. Uniform space subdivision is used for an additional 6x serial speedup over plane detection on an undivided point cloud. The plane detector presented is capable of reading from file and classifying 52 million points in just under a minute, with performance benefits on smaller models and thread-limited hardware as well.

1 Introduction

Multi-view stereo modelling devices often produce a point cloud as output, where many points in space make up the object being modelled (see Figure 1). This project's main aim is to produce accurate planar models from these point clouds using a plane detection algorithm. On larger subjects of interest, the number of 3D points captured by stereo modelling methods quickly balloons to a restrictive size to store, load and compute operations on, so parallel computing and time complexity of the algorithms/libraries used are important to produce a usable plane detector.

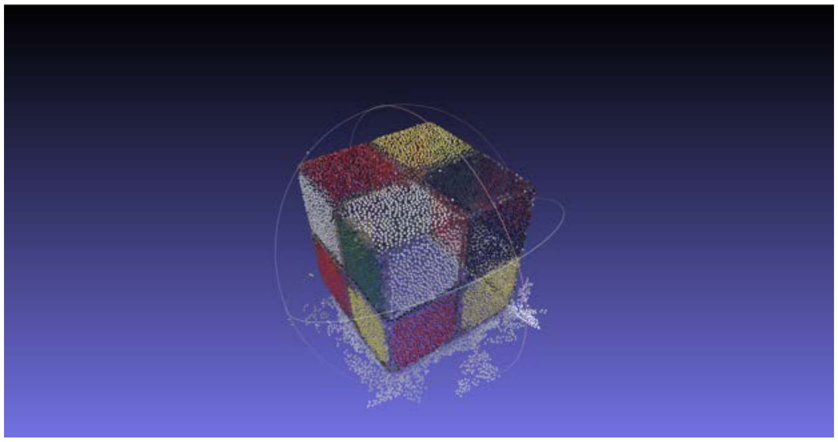


Figure 1: Point cloud of a wooden cube

1.1 Albany St Studio Model

Planar models are particularly good at efficiently representing buildings and architecture over more irregular objects (such as human beings or landscapes). Conversely, point clouds are not the ideal medium for representing large subjects in high fidelity. For example, a model of the Albany Street recording studio (Figure 2) has 52 million points, taking up approximately 1.37 gigabytes of storage. This room at the time of modelling was composed of mainly planar architecture with important smaller plane details such as the angled wall baffles, as well as poorly captured and insignificant objects (for the architectural context) cluttering the 3D model at floor level. A representative 3D model of just the planar architecture would be significantly easier to work with in terms of storage size and computation time for viewing, as well as when calculating other attributes of the building such as the reflection of sound waves in acoustics modelling. Many architectural point cloud models also have high levels of clutter and density of points, making this project viable for other subjects of interest in the field.

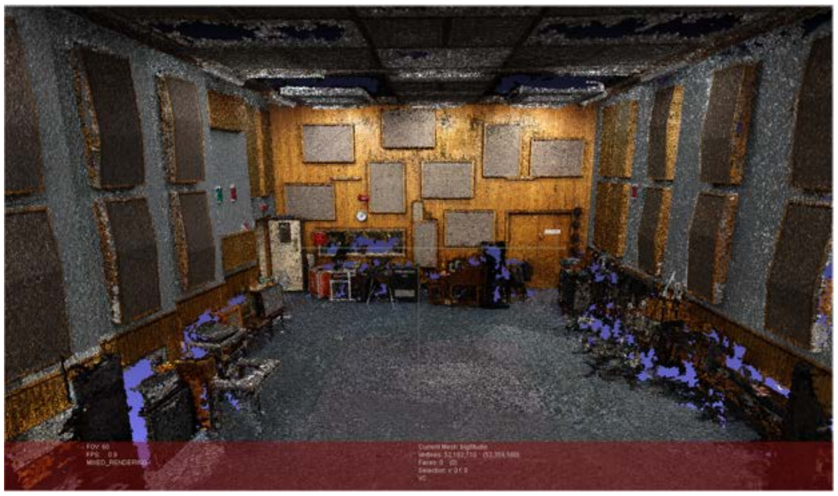


Figure 2: Point cloud of the Albany St recording studio

Smaller point clouds of the Albany St studio were created using MeshLab's Point Cloud Filtering tools on this file: 88,435, 923,190 and 7,463,361 point versions. These were used to reduce the number of points for a manageable execution time in testing while still representing the distribution of the original model.

1.2 Project Scope

In this project, I create a plane detection algorithm efficient enough to process large models like Albany St studio in reasonable time. This algorithm is robust to noise from object clutter or measurement error, and intelligent enough in its automatic parameters to run well without previous knowledge of the model in question. As the focus fields of this project are the parallelization and optimization of plane detection algorithms, creating planar models from the coefficients and point classifications outputted was not within the scope. Although reading from and writing to file for large point clouds takes a significant portion of execution time, this is ignored in the performance testing and improving the process was not an aim of this project for similar reasons.

2 Background

An overview is given of the structures that 3D models of different types are composed of, and the different algorithms and variants used to detect them from point clouds in the field. Much of this report will focus on the parallel computing analysis and implementation used to improve performance of the plane detector, so an introduction is also given to the chosen parallelization API.

2.1 3D Model Representations

The original title of this project, Modelling with Lines and Planes, offered a choice between detecting two different structures within objects in three-dimensional space (Figure 3). Lines are one-dimensional structures in a three-

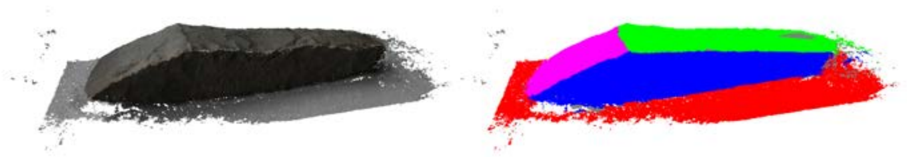


Figure 3: Model of a toki (a Māori stone adze) created using multi-view stereo methods, and the main planes identified in it [6]

dimensional space with two points defining their direction, while planes are two-dimensional structures defined by three points. Both structures require algorithms to detect and verify their quality, i.e. the number of points in the cloud that can be assigned to them (the extremes of which define their length within the model), and how well they represent the desired geometric qualities of the scene. I chose planes for this project with an interest in how they can more fully represent a scene on their own than lines (an example from 2D image reconstruction, Figure 5), as Hofer et al note that the complete 3D mesh reconstruction of their objects comes from a combination of lines and points in Figure 4.

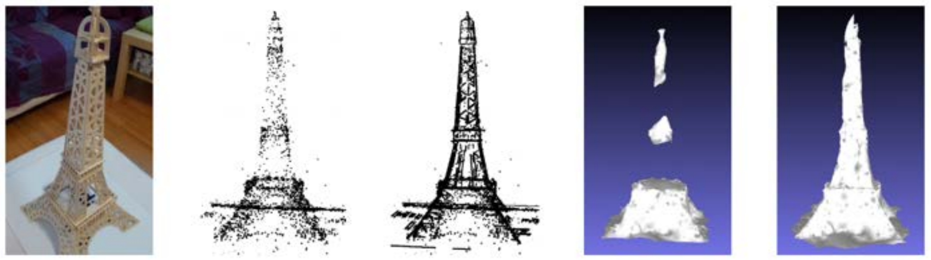


Figure 4: The original object, points, lines detected and 3D meshes generated in Hofer et al [5]. The last 3D mesh is generated from both lines detected and points from the point cloud input.

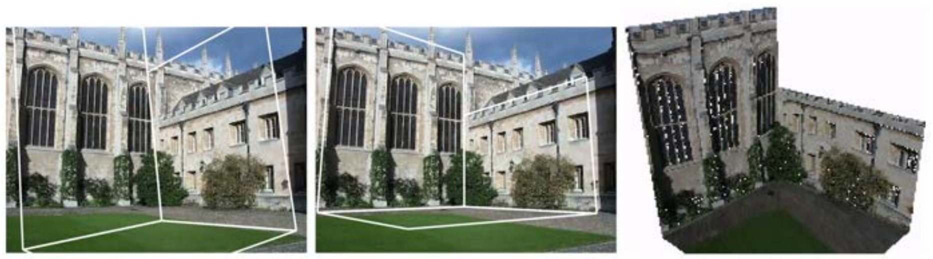


Figure 5: Texture mapped planar model, Dick et al. [11]

2.2 Eigen Library

The Eigen C++ library is chosen to easily and efficiently represent planes and lines in 3D space. `Hyperplane` is the type used for the former, a general term referring to an affine subspace of dimension `n-1` in a space of dimension `n`. In this program 2D hyperplanes are constructed in 3D space from three defining points, and the `absDistance` method is used to calculate the distance from a 3D point to it's closest point (projection) on the plane. `ParametizedLine` was another Eigen type explored to represent 1D constructs in a 3D world for its useful intersection method with `Hyperplane`, but this type was eventually optimized out in favor of manually calculating the required intersections in only the one or two necessary dimensions. The Eigen library also has the advantage of being highly optimized in its matrix operations, with Single Instruction Multiple Data parallelism built in [7].

2.3 RANSAC and Variants

Two prominent algorithms for fitting models to noisy data are Random Sample And Consensus [1] (RANSAC), and the Hough Transform. The Hough transform finds instances of shapes by a voting procedure in a parameter space, which becomes unwieldy in dimension when more than three parameters are required. However as planes only require 3 points to define, it is suitable for plane detection and has been used to great success in Limburger et al [2]. Since the PLY file format can encode point clouds with optional properties beyond location such as colour, normal vector direction, texture co-ordinates and faces for 3D meshes, Limburger et al. use normal vectors of the input points to quickly detect planes in the scene. Albany St studio and other models used for this project only consist of point locations and colours with no geometric relationships between individual points, so although this variant of the Hough transform is highly optimized and accurate, their work is not suitable for use in this project.

RANSAC finds planes by randomly generating hypotheses (sets of three points), fitting to those hypotheses and comparing them over a large number of trials. It was chosen for this project over the Hough transform due to its familiarity and simplicity (algorithm on page 6). Many variations on RANSAC exist for different applications such as BaySAC [3] (using previous trial outcomes to propose more accurate hypotheses) and Preemptive RANSAC [4] (generating a fixed number of hypotheses and verifying them in parallel). While these were not used in the plane detector, the latter informed my analysis of parallelism at the plane and trials levels of RANSAC.

Algorithm Basic Random Sample And Consensus for Plane Detection

```
Input: $X = \{\vec{x}_1, \vec{x}_2, ..., \vec{x}_n\}$, a set of 3D points
       $P$ the number of planes to find
       $T$ the point-plane distance threshold
       $R$ the number of RANSAC trials

1: procedure RANSAC($X$, $P$, $T$, $R$)
2:     for $p \leftarrow 1, P$ do
3:         $bestPlane \leftarrow \{0, 0\}$
4:         $bestPoints \leftarrow \{\}$
5:         for $r \leftarrow 1, R$ do
6:             $S \leftarrow \{x_1, x_2, x_3\}$        ▷ 3 points at random from X
7:             $thisPlane \leftarrow fitPlane(|S|)$
8:             $thisPoints \leftarrow \{\}$
9:             for all $x_i \in X$ do
10:                if $distance(thisPlane, x_i) < T$ then
11:                    $thisPoints \leftarrow thisPoints + x_i$
12:                end if
13:            end for
14:            if $|thisPoints| > |bestPoints|$ then
15:                $bestPlane \leftarrow thisPlane$
16:                $bestPoints \leftarrow thisPoints$
17:            end if
18:        end for
19:        output $bestPoints$
20:        $X \leftarrow X - bestPoints$
21:    end for
22: end procedure
```

2.4 Multiprocessing

Parallel computing is a powerful tool in the Computer Vision and Graphics fields due to the vast number of per-point comparisons and other independent operations that can be done concurrently. A popular multiprocessing API is OpenMP, chosen in this project for its ease of implementation and portability. Limburger et al. [2] use OpenMP for parallel octree generation, and comment on the potential for concurrent voting schemes if access to the accu-

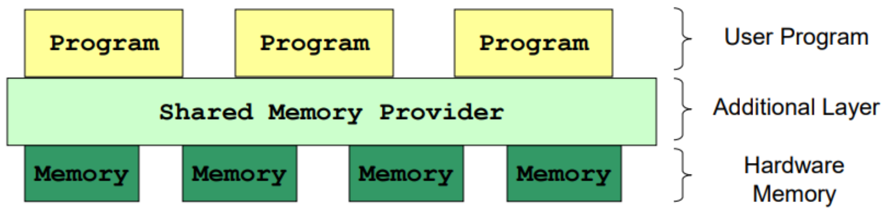


Figure 6: Shared memory diagram, National University of Singapore CS3210 - AY1920S1 - L3

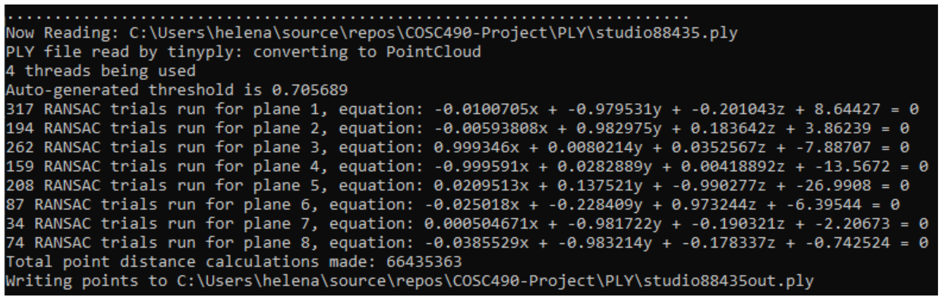
mulator cells (for summing those votes) could be controlled. OpenMP creates a virtual shared-memory architecture, where all threads running in parallel have equal access time to a shared block of memory (Figure 6). OpenMP does lack in efficient and flexible synchronization constructs to control read-/writes to shared variables, which limits the use of more complex parallel algorithms in both Limburger et al and this project. However, it is still useful in this project due to the speedup gained on even simple parallel sections of code running over thousands or millions of 3D points, and its independence from the CPU architectures used allowing easy testing in multiple hardware environments.

3 RANSAC Implementation

The planes detected by RANSAC and efficiency to run it depend on a number of input parameters and implementation details. RANSAC's number of planes to find and number of trials to run can be automatically generated with more general parameters. The threshold distance to classify a point as being on a plane can also be generated from the scene using a scale parameter independent of model dimensions.

3.1 Output

The plane detector produces a PLY file as output with inlier points recoloured to represent the found plane they lie on. It also returns the equations of the planes found in the model co-ordinate space (Figure 7). Passing the outermost point co-ordinates of each plane and a texture generated from the incoming point cloud colours along with the plane coefficients to an OpenGL pipeline for rendering a purely planar model was proposed for future work. However, while this would allow an extremely memory-efficient representation of the original model, this project's aims focus more on optimizing the plane detection process than a complete output pipeline.



```
......................................................................
Now Reading: C:\Users\helena\source\repos\COSC490-Project\PLY\studio88435.ply
PLY file read by tinyply: converting to PointCloud
4 threads being used
Auto-generated threshold is 0.705689
317 RANSAC trials run for plane 1, equation: -0.0100705x + -0.979531y + -0.201043z + 8.64427 = 0
194 RANSAC trials run for plane 2, equation: -0.00593808x + 0.982975y + 0.183642z + 3.86239 = 0
262 RANSAC trials run for plane 3, equation: 0.999346x + 0.0080214y + 0.0352567z + -7.88707 = 0
159 RANSAC trials run for plane 4, equation: -0.999591x + 0.0282889y + 0.00418892z + -13.5672 = 0
208 RANSAC trials run for plane 5, equation: 0.0209513x + 0.137521y + -0.990277z + -26.9908 = 0
87 RANSAC trials run for plane 6, equation: -0.025018x + -0.228409y + 0.973244z + -6.39544 = 0
34 RANSAC trials run for plane 7, equation: 0.000504671x + -0.981722y + -0.190321z + -2.20673 = 0
74 RANSAC trials run for plane 8, equation: -0.0385529x + -0.983214y + -0.178337z + -0.742524 = 0
Total point distance calculations made: 66435363
Writing points to C:\Users\helena\source\repos\COSC490-Project\PLY\studio88435out.ply
```

Figure 7: Example of console output for the plane detector

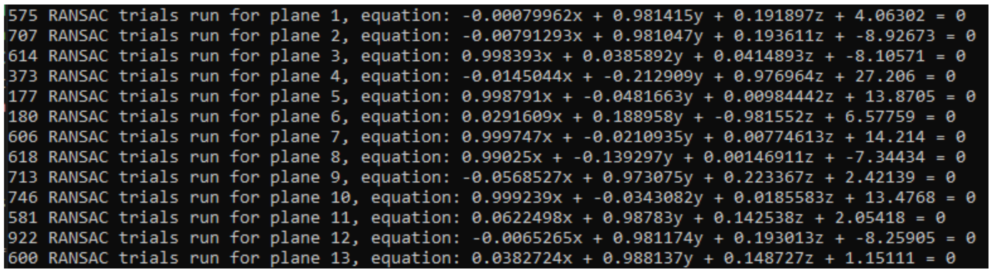
3.2 Trials

Ideally the number of random model trials to run, R, should be determined from the probability of finding the largest remaining plane in the scene. This success probability of finding the largest plane in the remaining points informs the number of trials for each plane search as such:

Algorithm Basic Random Sample And Consensus for Plane Detection

```
$inlierRatio \leftarrow 0.01$
$R \leftarrow log(1 - successProbability)/log(1 - pow(inlierRatio, 3))$
$r \leftarrow 0$
while $r < R$ and $r < maxTrials$ do
    ...
    if $thisSize > bestSize$ then
        $bestPlane \leftarrow thisPlane$
        ...
        $inlierRatio \leftarrow bestPlane.size/remainingPoints.size$
        $R \leftarrow log(1 - successProb)/log(1 - pow(inlierRatio, 3))$
    end if
    $trial++$
end while
```

Examples of the number of trials run for 0.99 and 0.5 success probability are in Figure 8 and Figure 10, and the effect on the planes detected can be seen in the output figures below. The standard success probability in Figure 9 detects the smaller planes parallel to the walls of the studio, and classifies remaining noisy points with small vertically stacked planes that do not go beyond the smaller objects scattered around the model. While not all wall baffles are distinguished from the larger planes behind them equally, this relaxed-parameter attempt at plane detection is reasonably accurate.



```
575 RANSAC trials run for plane 1, equation: -0.00079962x + 0.981415y + 0.191897z + 4.06302 = 0
707 RANSAC trials run for plane 2, equation: -0.00791293x + 0.981047y + 0.193611z + -8.92673 = 0
614 RANSAC trials run for plane 3, equation: 0.998393x + 0.0385892y + 0.0414893z + -8.10571 = 0
373 RANSAC trials run for plane 4, equation: -0.0145044x + -0.212909y + 0.976964z + 27.206 = 0
177 RANSAC trials run for plane 5, equation: 0.998791x + -0.0481663y + 0.00984442z + 13.8705 = 0
180 RANSAC trials run for plane 6, equation: 0.0291609x + 0.188958y + -0.981552z + 6.57759 = 0
606 RANSAC trials run for plane 7, equation: 0.999747x + -0.0210935y + 0.00774613z + 14.214 = 0
618 RANSAC trials run for plane 8, equation: 0.99025x + -0.139297y + 0.00146911z + -7.34434 = 0
713 RANSAC trials run for plane 9, equation: -0.0568527x + 0.973075y + 0.223367z + 2.42139 = 0
746 RANSAC trials run for plane 10, equation: 0.999239x + -0.0343082y + 0.0185583z + 13.4768 = 0
581 RANSAC trials run for plane 11, equation: 0.0622498x + 0.98783y + 0.142538z + 2.05418 = 0
922 RANSAC trials run for plane 12, equation: -0.0065265x + 0.981174y + 0.193013z + -8.25905 = 0
600 RANSAC trials run for plane 13, equation: 0.0382724x + 0.988137y + 0.148727z + 1.15111 = 0
```

Figure 8: RANSAC trials run for 99% likelihood of finding the largest plane in 52 million points

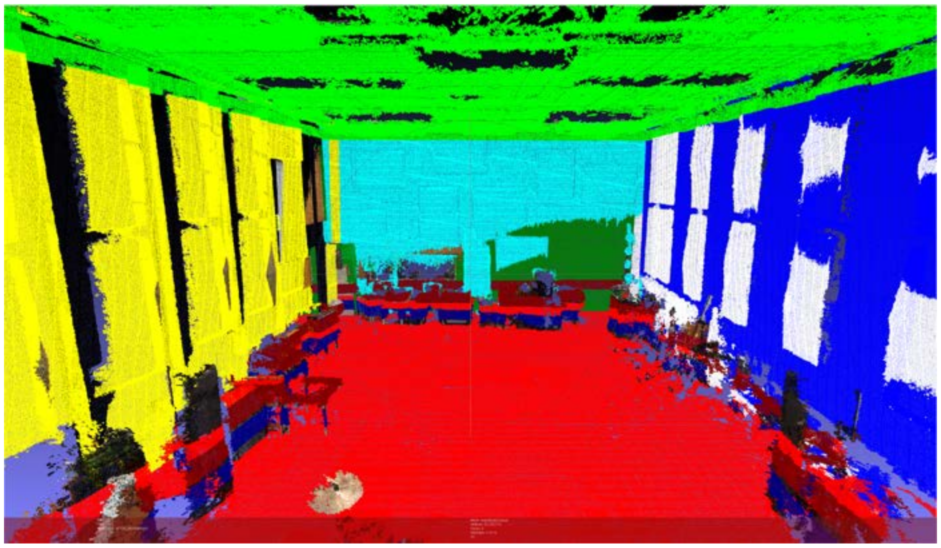
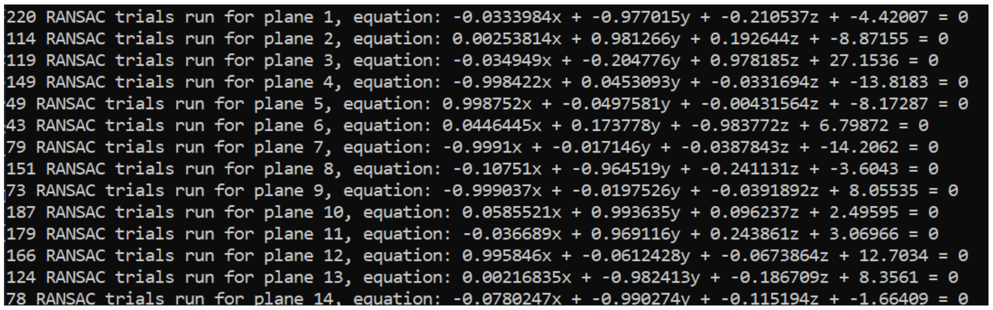


Figure 9: 0.99 success probability, full model, 1% distance threshold, 95% classified

The low probability plane detector in Figure 11 slices the right wall and floor into multiple unnecessary planes that do not reflect the repeating nature of the wall baffles. Sections of flat wall and floor are left out of their accurate planes from the 0.99 classification, and are instead claimed by the later planes covering the noisy points on the model.



```
220 RANSAC trials run for plane 1, equation: -0.0333984x + -0.977015y + -0.210537z + -4.42007 = 0
114 RANSAC trials run for plane 2, equation: 0.00253814x + 0.981266y + 0.192644z + -8.87155 = 0
119 RANSAC trials run for plane 3, equation: -0.034949x + -0.204776y + 0.978185z + 27.1536 = 0
149 RANSAC trials run for plane 4, equation: -0.998422x + 0.0453093y + -0.0331694z + -13.8183 = 0
49 RANSAC trials run for plane 5, equation: 0.998752x + -0.0497581y + -0.00431564z + -8.17287 = 0
43 RANSAC trials run for plane 6, equation: 0.0446445x + 0.173778y + -0.983772z + 6.79872 = 0
79 RANSAC trials run for plane 7, equation: -0.9991x + -0.017146y + -0.0387843z + -14.2062 = 0
151 RANSAC trials run for plane 8, equation: -0.10751x + -0.964519y + -0.241131z + -3.6043 = 0
73 RANSAC trials run for plane 9, equation: -0.999037x + -0.0197526y + -0.0391892z + 8.05535 = 0
187 RANSAC trials run for plane 10, equation: 0.0585521x + 0.993635y + 0.096237z + 2.49595 = 0
179 RANSAC trials run for plane 11, equation: -0.036689x + 0.969116y + 0.243861z + 3.06966 = 0
166 RANSAC trials run for plane 12, equation: 0.995846x + -0.0612428y + -0.0673864z + 12.7034 = 0
124 RANSAC trials run for plane 13, equation: 0.00216835x + -0.982413y + -0.186709z + 8.3561 = 0
78 RANSAC trials run for plane 14, equation: -0.0780247x + -0.990274y + -0.115194z + -1.66409 = 0
```

Figure 10: RANSAC trials run for 50% likelihood of finding the largest plane in 52 million points

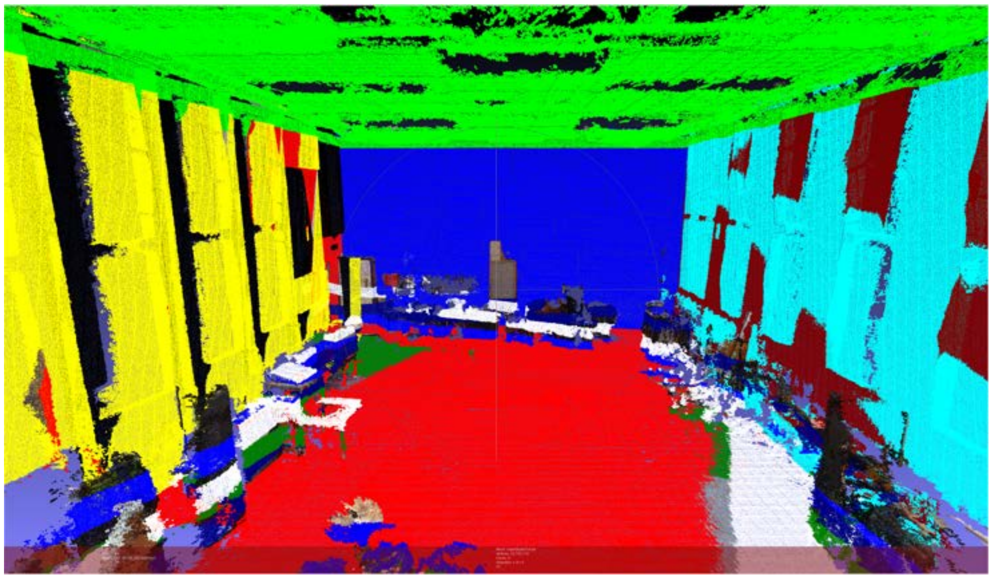


Figure 11: 0.5 success probability, full model, 1% distance threshold, 95% classified

3.3 Planes

The number of planes, P, can be set dynamically by simply continuing to find planes until a user-defined percentage of the scene is explained. The inverse of this percentage is taken as input to the plane detector, and can be thought of as the noise level: the fraction of points to ignore and leave unassigned to any plane.

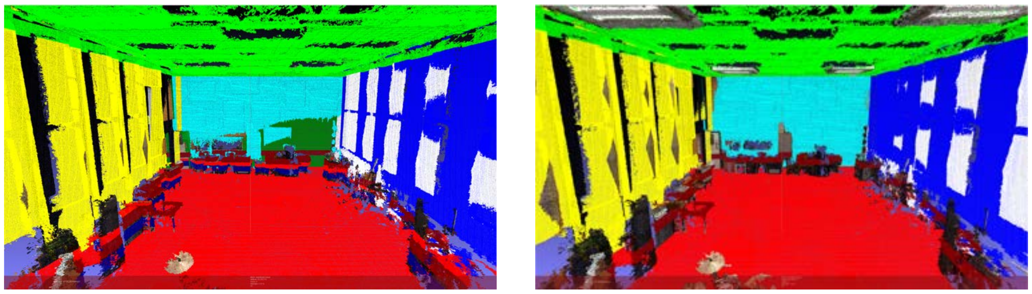

(a) 95% of points classified (b) 90% of points classified

Figure 12: The impact of unclassified points on plane detection, full model, 0.99 success probability, 1% distance threshold

More planes are detected when this parameter is set low, splitting the non-planar objects around the model into small vertically stacked planes (Figure 12a) instead of leaving them unclassified as clutter (Figure 12b). Whether to set it high or low depends on the preferred method of dealing with extraneous objects (to classify them in small planes or leave uncoloured), and the desired speed of the plane detector.

3.4 Distance Threshold

The distance threshold, T, required for points to be on a plane is calculated from the average of the model's bounding box size over the three dimensions. This simple metric produces reasonable thresholds when multiplied by an input scalar factor, around 1-2%.
Like the noise levels, more planes are detected when this threshold is set low. The wall baffles, for example, are partially distinguished at 1% threshold (Figure 13a). However, the value of this parameter holds a trade-off between more accurate planes at lower levels, and thicker planes that effectively filter

out unwanted objects around the room and encompass measurement errors in points adjacent in real life (Figure 13b).

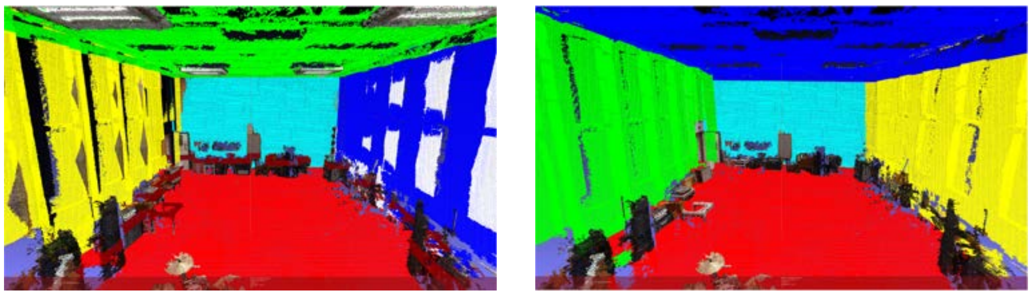

(a) Threshold of 1% of the model scale (b) Threshold of 2% of the model scale

Figure 13: The impact of distance threshold on plane detection, full model, 0.99 success probability, 90% classified

4 Parallelization

Parallel computing is used in this project to improve performance of the program by capitalizing on the number of repeated, independent tasks in the RANSAC algorithm and the multi-processor hardware available. Initially, three possible levels of loops in the RANSAC algorithm on page 6 were proposed to parallelize upon: trials, planes and points. As the former of these loops encase the latter, only one level of parallelism was able to be implemented without significant restructuring of the RANSAC algorithm.

4.1 Synchronization

Before the parallel speedup of any algorithm can be evaluated, its safety and correctness must be ensured. In a shared-memory architecture, this relies on synchronization of access to shared resources to avoid simultaneous write operations and maintain the intended order of operations. OpenMP provides numerous synchronization constructs that, while easy to use, can create significant overhead cost in memory and execution time.

4.2 Planes in Parallel

Finding planes in parallel (parallelizing the `for` loop on line 2 of the RANSAC algorithm on page 6) was proposed via a primary/secondary thread scheme, where each secondary thread runs its own set of RANSAC trials to find a plane, while the original primary thread maintains and updates a list of each point's assignment to the largest plane found thus far. This sharing of work has a few immediate issues. Firstly, every thread is searching in the same field without the removal of points from previous planes, so will come to similar conclusions that give no performance gains. Secondly, the synchronization required would be complex and inefficient in the chosen medium of OpenMP (in fact, this primary/secondary structure would better suit a distributed-memory system where separate processes message each other such as the Message Passing Interface [9]). Lastly, the performance gains depend entirely on the number of planes in the model rather than the size, where in reality the object's planar complexity is a far smaller consideration to the aims of this project than its point cloud size.

4.3 Trials in Parallel

Looking at the next level of `for` loops on line 5 of the RANSAC algorithm page 6, trials mostly operate independently until a final plane size is obtained, at which point a trial must compare its size to the largest plane found thus far in the set of trials, and update the largest plane if its size is exceeded. Synchronizing both the read of the shared largest plane and the write to update the largest plane would be a costly exercise. However, this approach still held promise due to the length and complexity of code preceding the operation. The selection of points, creation of new vectors and planes, and calculations required before comparing planes could save significant run-time when done in parallel, possibly enough to justify the overhead and synchronization costs.

Parallel RANSAC trials have been used in Pre-emptive RANSAC [4], where a fixed number of hypotheses are generated and verified in parallel using a breadth-first scheme. This highlights the benefit in the number of RANSAC trials being known in advance, as set-length parallel `for` loops are the original and most efficient use of OpenMP [8]. An upper limit on RANSAC trials can be computed from the first plane found, and this number has an indirect correspondence to the number of points in the point cloud via the inlier ratio.

This made parallel RANSAC trials more promising for parallel speedup relative to point cloud size than parallel plane-finding, and, while more complex, this method had been considered for implementation if parallelizing distance calculations returned disappointing results.

4.4 Distance calculations in Parallel

The distance calculations are fairly independent, with no iteration of the `for` loop relying on previous results. A simple parallel distance calculation for the algorithm on page 6 is given below:

Algorithm Basic Random Sample And Consensus for Plane Detection

```
 9: for $i \in X$ do                ▷ parallel for shared(thisPoints), private(i)
10:     if $distance(thisPlane, X[i]) < T$ then
11:         critical section:
12:         $thisPoints \leftarrow thisPoints + X[i]$
13:     end if
14: end for
```

The critical section directive used in the centre of the loop ensures thread safety by allowing only one thread at a time to execute the code: adding the indexes of points on the plane to a vector shared between all threads. This is the only piece of synchronization needed in this parallel block; in terms of performance, the distance calculation and comparisons are far more computation-intensive over millions of points than the vector add operation and overhead required to complete it serially, as is reflected in the interim profiling results. Of the three levels proposed to parallelize on, the distance calculations for each point in the cloud was chosen to minimize synchronization needed while promising direct performance gains for increasing model sizes.

4.5 Interim Results

The interim performance testing was done on a 4-core hyperthreaded CPU machine with 12GB RAM and 4 threads available for OpenMP, on a debug build. Profiling the serial execution of the plane detector gave a clear indication of the most computationally intensive methods (Table 1). RANSAC took up 93% of the CPU time, and specifically the distance calculations

Method	CPU %
RANSAC	93.44
- Distance comparison	91.59
- - Distance calculation from point to plane	(89.73)
- - Comparison of distance to threshold	(1.86)
Reading from PLY	5.40
Writing to PLY	0.27

Table 1: Profiling results of key methods

from points to planes within it. In the program this is done using the `Eigen::Hyperplane absDistance()` method, an efficient calculation mainly taking up CPU time with dot products and other Matrix operations. The sheer number of repetitions of this method is the main factor in its large CPU time, even with the program executing on a relatively small 88,435 point model in profiling. To find just the largest plane in this model with a reasonable 400 trials, the distance comparison is called 35 million times. Given there are multiple planes left with a decreasing but still significant number of trials run for each, and the number of trials to achieve the success probability also increases with the number of points in the scene, the calls to this function can be expected to increase exponentially with larger models.

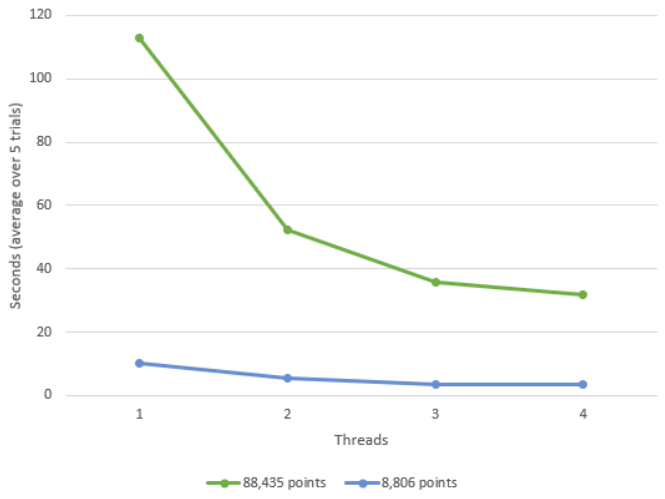


Figure 14: Plotting execution time of RANSAC methods against number of threads used

As predicted, the performance gains in Figure 14 from parallelizing the distance calculation and comparisons were significant. Due to the random nature of the algorithm giving large variations in execution time, the average of 5 run-times (of the RANSAC method only) were taken for the 8026-point and 88435-point models. The hardware and OpenMP environment were able to provide a maximum of 4 threads, with speedup of 2.16, 3.17 and 3.55 time for 2, 3, and 4 threads respectively.

This proved the expected trend of parallelization having increased benefits with more points in the model. However, there were a few quirks to these results, such as 4 threads having minimal speedup over 3 even with 88,435 points, and the parallel efficiencies (speedup per thread) for 2 and 3 threads exceeding the theoretical limit of 1. The latter could be explained by the random nature of the program giving varied execution times, with a practical goal for later work done on this project being large-scale testing with bigger models and more threads.

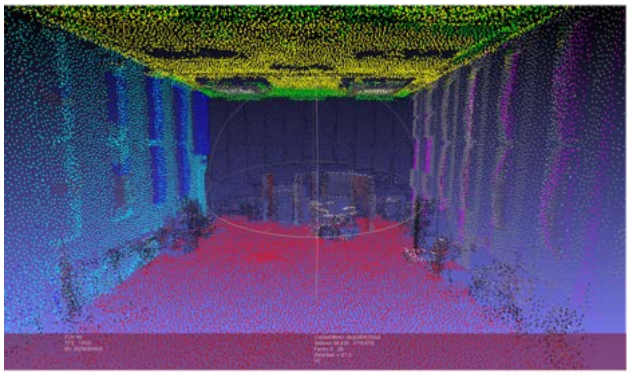


Figure 15: Output results at the interim stage of the project, 88,435 points coloured

Section 7 gives results of tests on different hardware with more threads to expand on this performance plateau observed at 4 threads. These tests demonstrated parallel overhead costs closing in on potential gains at the limit of available threads rather than larger models bringing the parallel speedup closer to four-fold. The analysis of potential areas of parallel speedup and synchronization costs in section 4 indicated diminishing returns for further parallelization in the RANSAC method, and the project focused on reducing existing synchronization overhead, uniform space subdivision and optimization from this point forwards, in order to classify more detailed models than Figure 15 in reasonable time.

5 Uniform Space Subdivision

As the interim results show, the key time complexity factor in RANSAC for plane detection is the model size. While the number of RANSAC trials and planes detected can be adjusted (with trade-off to the accuracy of those planes) through parameters, 52 million repetitions of the distance calculations per trial and per plane is the key serial factor in a fairly efficient calculation and comparison taking up 91% of the CPU time. A common

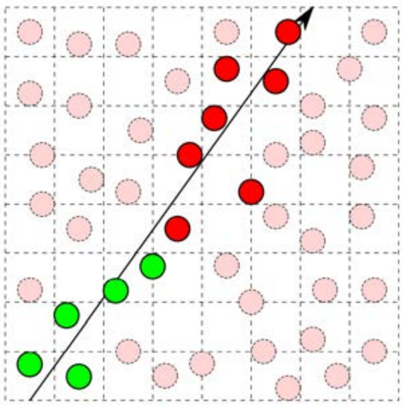


Figure 16: Raytracing in 2D uniform space subdivision, University of Otago, COSC342 - 2017 - L20

method to reduce the number of comparisons required in 3D ray-tracing is uniform space subdivision of the model into 3D cubes known as volumetric pixels, or voxels (Figure 16). While the methods required to traverse planes in 3D are more complicated than rays, this technique once optimized greatly improved the performance of the plane detector.

5.1 Construction

To create the voxel structure, the dimensions of the bounding box are divided by the given integer voxel size. In order to fit the voxels exactly to the edges and to reduce floating-point precision errors, the bounding box corners are first expanded to the nearest integer divisible by the voxel size. As the Uniform Point Cloud class is created from a copy of an existing Point Cloud, the flat point vector of its parent class is traversed in parallel to assign the points to voxels. These are stored in a 4D vector, where the first three dimensions represent the voxels and the fourth is a list of indexes of contained points. For each point, their location is hashed into 3D voxel coordinates, and their index in the flat Point Cloud vector is pushed to the point list for that voxel. Finally each voxel list is sorted to aid in removing points later on, and a copy of the 4D structure is made for this purpose:

Algorithm Uniformly Subdivided Point Cloud Construction

```
Input: $X_S, X_L, Y_S, Y_L, Z_S, Z_L$ the limits of the bounding box
       $voxelSize$ the integer size of voxels desired

1:  procedure VOXEL CONSTRUCTION($X_S, X_L, Y_S, Y_L, Z_S, Z_L$)
2:      expand bounding box edges to fit voxels of $voxelSize$ cleanly
3:      $xVoxels \leftarrow (X_L - X_S)/voxelSize$
4:      $yVoxels \leftarrow (Y_L - Y_S)/voxelSize$
5:      $zVoxels \leftarrow (Z_L - Z_S)/voxelSize$
6:      $cells \leftarrow$ 4D vector of $xVoxels * yVoxels * zVoxels$ empty int vectors
7:      for $p \leftarrow 0, pointCloud.size$ do                 ▷ parallel for loop
8:          $c[0] \leftarrow (pointCloud[p][0] - X_S)/voxelSize$
9:          $c[1] \leftarrow (pointCloud[p][1] - Y_S)/voxelSize$
10:         $c[2] \leftarrow (pointCloud[p][2] - Z_S)/voxelSize$
11:         $cells[c].append(p)$                                  ▷ critical section
12:     end for
13:     sort each voxel list in parallel when done
14:     $remainingCells \leftarrow cells$
15: end procedure
```

5.2 Plane Spanning

A number of different plane spanning algorithms to detect voxels within the threshold of the plane were investigated, including casting equidistant rays across the plane from its intersection line with a bounding box side and tracing their path with Cleary's algorithm [10]. However the most reliable algorithm found for visiting all nearby voxels follows on page 20, where the voxels on the side of the bounding box most parallel to the plane are traversed in 2D, and their distance to the plane in the third dimension is calculated. Note that z, y and x (and their respective limits) are replaced with the normal direction of the bounding box side and the two remaining dimensions in the implementation in order to use the bounding box side most parallel to the plane, but they are set here to a flat xy plane for readability.

This algorithm is optimized to take advantage of the overlapping calculations without sacrificing thread independence. While the `Eigen::ParametizedLine` type was used in the proof of concept to easily intersect four corner-based lines in the Z direction with the plane, this was changed to direct z value calculations to reduce unnecessary construction and calculation work while taking advantage of the iterative loop structure. The corner z intercept values are calculated in full only once for each multi-threaded x loop, then incremented between y loops with the pre-calculated value of moving up one voxel height.

Algorithm Voxel Plane Spanning

```
Input: $planePoints = \{\}$ an empty vector
         (to be returned with matching point cloud indexes)
       $thisPlane$ the plane object to be fitted to
         (includes the below coefficients)
       $voxel\_size$ the voxel size
       $X, Y, Z$ the number of voxels in each direction
       $t$ the threshold distance to be on the plane

    procedure PLANE SPANNING($planePoints, thisPlane, voxel\_size, X, Y, Z, t$)
        for $x \leftarrow 0, X$ do                                ▷ Outer-most loop is parallelized
3:          $cell \leftarrow \{x, 0, 0\}$                                ▷ A 3D voxel coordinate
            calculate Z intercepts for four corners of the first XY voxel
            for $y \leftarrow 0, Y$ do
6:              $cell[1] \leftarrow y$
                calculate minimum and maximum Z values of the four corners
                $lower \leftarrow$ cell of (minimum plane intercept - t)
9:              $upper \leftarrow$ cell of (maximum plane intercept + t)
                $cell[2] \leftarrow lower$, or cell 0 if lower is below the box
                while $cell[2] < Z, cell[2] <= upper$ do
12:                 for all $index \in remainingVoxels[cell]$ do
                        if $|thisPlane.distance(pointCloud[index])| < t$ then
                            add point to $planePoints$
15:                     end if
                    end for
                    $cell[2]++$
18:             end while
                shift the four corners along by $voxel\_size$ in the $y$ direction
            end for
21:     end for
        return $planePoints$
    end procedure
```

6 Optimization

After implementing Uniform Space Subdivision, testing of the two modes of the plane detector was undertaken. The slowdown at higher thread usage from the interim results was observed once more, and using new hardware available with a higher number of cores, performance was shown to significantly decrease above 4 threads. Initial testing of the uniform space subdivision methods also showed poorer performance on larger point cloud models than the undivided point cloud, prompting optimization of both the serial algorithm and parallelization structure of the plane detector.

6.1 Removed and Remaining Points

An important operation in multi-plane RANSAC is removing the points on the best plane after every set of trials, before another set commence. This has three uses within the RANSAC algorithm: correctly adjusting the number of RANSAC trials for the points remaining (page 8, line 9), avoiding taking new plane's characteristic points from claimed ones (page 6, line 6), and saving only remaining points to those planes (page 6, line 11).

While point removal at the conclusion of each set of RANSAC trials was initially implemented with a `removedPoints` vector that was checked against using a binary search when defining and adding points to the plane, it was very inefficient to check all points and terminate when the point had been removed. Therefore the vector was changed to one of remaining point indexes in the point cloud, only using the costly binary searches on the outermost loop of RANSAC to remove the successful planes' (sorted) point indexes at the conclusion of each set of trials:
This vector could then be directly accessed in parallel to distance test points for a plane.

For the uniformly subdivided point cloud, the `remainingVoxels` vector iterated through for the distance testing is a copy of the 4D vector created at construction. Removing points in place is preferred to new vector assignment so the greater 4D voxel structure is not disturbed or re-allocated. This copy is iterated though in three dimensions after each round of RANSAC trials finishes, with the algorithm on page 23 for finding and removing matching point indexes from the (sorted at construction) voxel point list in place.

Algorithm Vector Point Removal

```
1: procedure POINT REMOVAL($planeID$, $planePts$, $remainingPts$)
2:     for all $p \in remainingPts$ do
3:         if binary search finds $point \in planePts$ then
4:             $colourPointAtIndex(point, planeID)$
5:         else
6:             $remainingPts.remove(point)$
7:         end if
8:     end for
9: end procedure
```

The C++ `lower_bound` method is used as a same-complexity alternative to binary search that returns an iterator to the first point greater than or equal to the target in the range given. Because both vectors are sorted, the iterator for the start of the search range can be shifted forward for every point found and removed in the voxel list. By halving the average search range on the input plane points, this reduces the time complexity to *O(m log(n/2))* for removing points at the end of each set of trials - despite the method's relatively infrequent use, it is an important one to optimize over the naïve complexity given that every voxel must be addressed for every plane found, and that each plane can hold a significant portion of the total points in the model.

Algorithm In-place Voxel Point Removal

```
1: procedure POINT REMOVAL($planeID$, $planePts$, $voxelPts(size = m)$)
2:     $iter \leftarrow planePts.begin()$
3:     $prevIter \leftarrow iter$
4:     $d \leftarrow 0$
5:     while $d < m$ do
6:         $iter \leftarrow lowerBound(planePts.begin(), planePts.end(), voxelPts[d])$
7:         if $iter = planePts.end()$ or $valAt(iter) > voxelPts[d]$ then
8:             $iter \leftarrow prevIter$
9:             $d++$
10:        else
11:            $colourPointAtIndex(voxelPts[d], planeID)$
12:            $voxelPts.remove(voxelPts.begin() + d)$
13:            $prevIter \leftarrow iter$          ▷ d not incremented: vector shifting
14:        end if
15:    end while
16: end procedure
```

6.2 Reducing Synchronization Constructs

The motivation for the second major optimization came from some conflicting results - while two threads creating and finding planes in the uniformly subdivided point cloud took a respectable 4 and a half minutes, the same point cloud with one thread took only three minutes (Figure 21). Given the vast amount of parallelizable work, this meant the overhead of thread synchronization was surpassing the parallel speedup with increasing amounts of points. Returning to the distance calculations, a further improvement to this parallelization was made using OpenMP's reduction directive for less frequent synchronization overhead than critical statements within the per-point loop (algorithm on page 24).

The point-cloud-sized vector of boolean values used allows concurrent writing of points onto the plane (setting the corresponding value to true), and a size variable allows the number of points added to the plane to still be visible for comparisons. The reduction directive tells OpenMP to turn the incrementing of the shared size into a thread-safe batch operation, where each thread adds to its own local value across loops then writes to the shared variable in a single synchronized operation at its conclusion.

```
$thisPoints \leftarrow X$ boolean values, initialized to false
$size \leftarrow 0$
for $i \in X$ do                                  ▷ parallel for reduction(+:size)
    if $distance(thisPlane, X[i]) < T$ then
        $thisPoints[i] \leftarrow true$
        $++size$
    end if
end for
...
return $thisPoints, size$
```

7 Final Results

The execution time of the RANSAC method was profiled on a 10-core hyperthreaded CPU workstation with 64MB RAM and 20 threads available for OpenMP, using three different model sizes: approximately nine hundred thousand points, seven million points and the full fifty-two million points. Three execution times were recorded and averaged for each data point due to the random nature of RANSAC execution times. Both divided and undivided point clouds were tested on the release build, with OpenMP direction and the number of threads set between 1 and 20. The input parameters were 0.99 success probability, 10% of points left unclassified, 2% threshold, and a voxel size of 1 (27 by 23 by 59 voxels) for the uniform space subdivision.

7.1 Pre-Optimization

First, the version of the code using critical directives was tested on the two smaller models (Figure 17, Figure 18). These executions showed a similar pattern of optimal performance at 2-3 threads as the interim results, despite a significant speedup from the release build and minor optimizations. While the change from storing removed points to remaining points reduces the number of required point comparisons per plane, the key factor causing this slowdown at a higher number of threads remained the synchronization overhead.

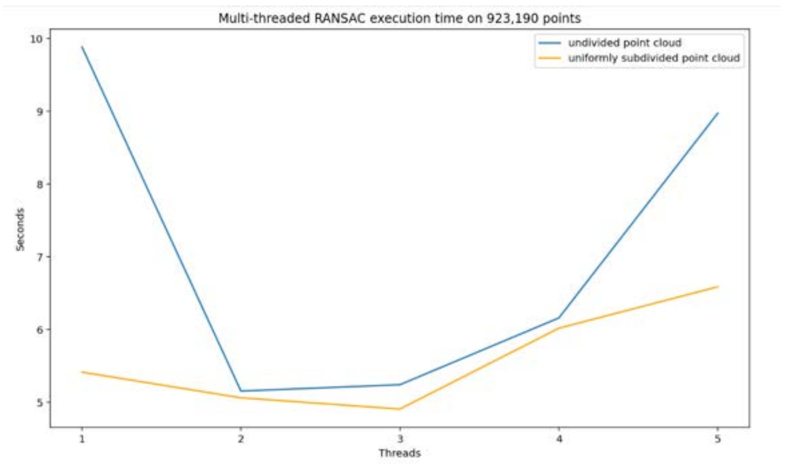


Figure 17: Unoptimized results on smaller model

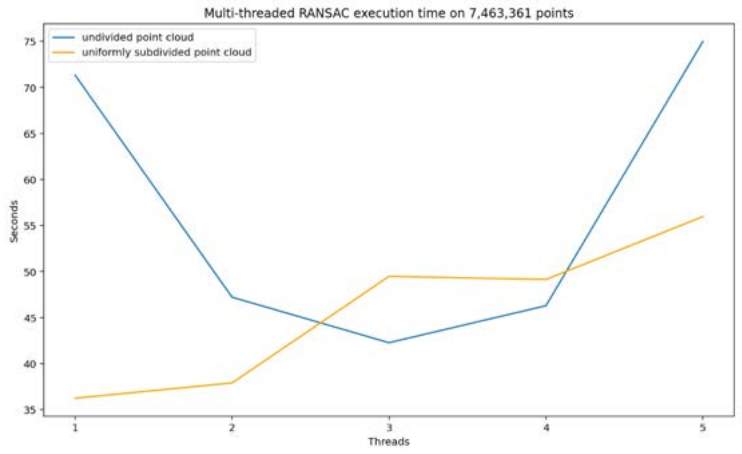


Figure 18: Unoptimized results on larger model

However, these results still formed a proof of concept for uniform space subdivision: the uniformly space subdivided point cloud ran significantly faster than the undivided one. An additional test on the full 52 million point model (up to the point of writing out to PLY, a lengthy and un-parallelizable process not in the focus of this project) showed uniform space subdivision took half the time to detect the planes, including the longer construction time (Figure 21). A point comparisons counter showed that 47 billion comparisons were made to classify the 52 million points in the undivided point cloud, while only 10 billion were made with uniform space subdivision.

7.2 Post-Optimization

After the implementation of boolean arrays and OpenMP reduction variables to reduce the frequency of synchronization needed, both point clouds were tested again. This final version of the plane detector scaled to a larger number of threads for both smaller (Figure 19) and larger (Figure 20) point clouds, and the performance plateaued past 12 threads (undivided point cloud) and 8 (divided point cloud) threads on the machine used instead of increasing.

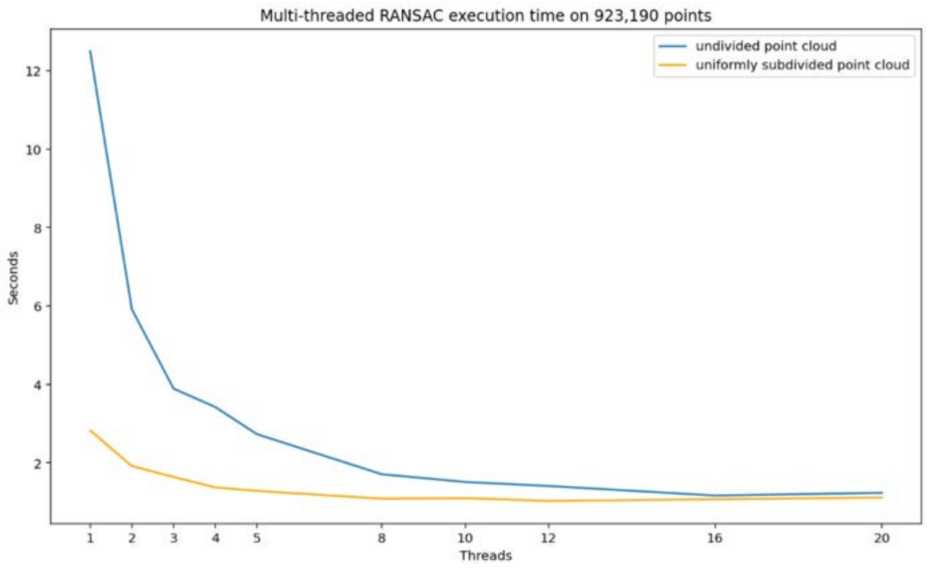


Figure 19: Optimized results on smaller model

As expected, plane detection in uniformly subdivided space has less to gain from increasing the number of threads due to the parallel `for` loop running over a smaller number of voxel rows (23 in the Albany St studio model) with limited concurrency, rather than directly on the larger set of remaining points. While the parallelization structure for uniform space subdivision was chosen to minimize thread creation and overhead for adjacent sparse voxels and was faster than the per-voxel-list alternative, its failure to significantly capitalize on thread counts above 8 raises the potential of space subdivision systems that standardize point counts within voxels for more consistent and efficient parallelization.

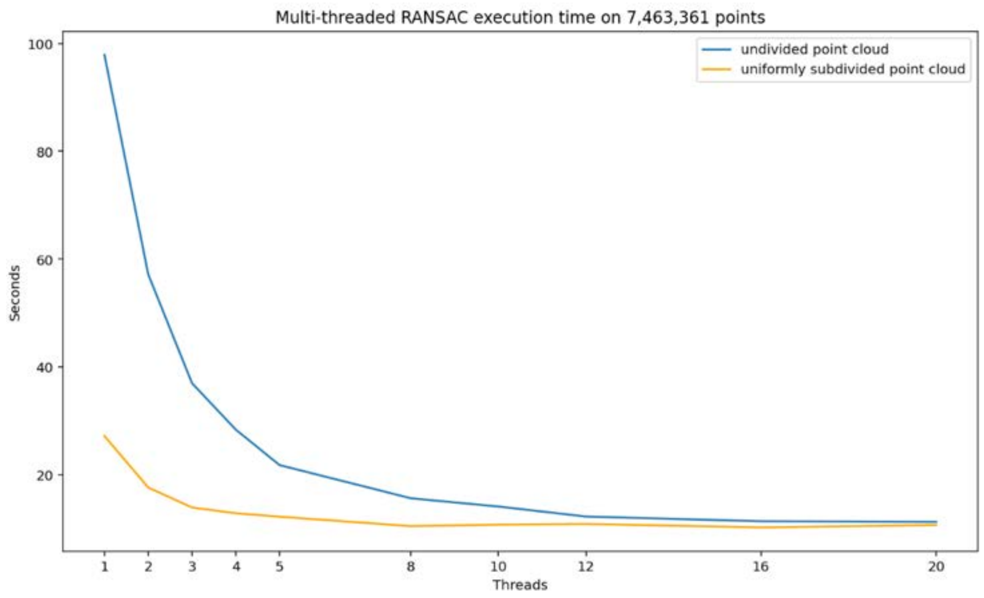


Figure 20: Optimized results on larger model

While small changes (less than 0.5 s) can be observed in the 8-20 thread range executions, no reliable conclusions can be made at that scale due to the variation in RANSAC execution times (i.e. increased or decreased accuracy of random planes) and other external factors in the testing environment. The limit to parallel speedup is also observed from 12 threads upwards in the more parallelizable undivided point cloud, suggesting the testing could have been impacted by the hardware's switch from single core processes to hyperthreading at 10 threads. Other operating system processes or CPU throttle from overheating could also have impacted the execution times at this scale, but the imperfect nature of averaging trials to control randomness means no external factor can be confirmed as significant to plane detection time across both point clouds.

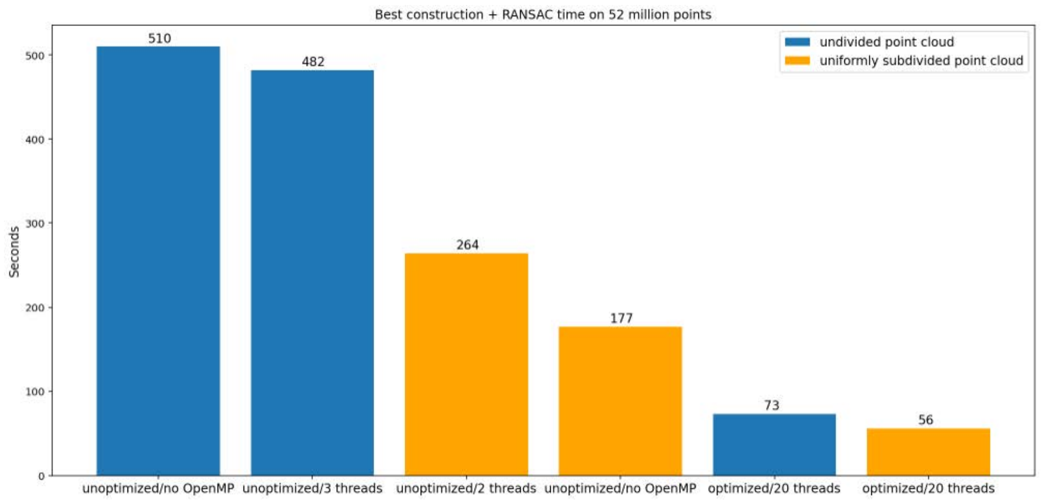


Figure 21: Point cloud construction + plane detection times on the full model

An overall look at the best times for the full 52 million point model in Figure 21 shows uniform space subdivision's 8.5x speedup with 20 threads over the serial execution on an undivided point cloud from the interim results, including the additional voxel construction time. Though the divided point cloud only has a slight advantage over the undivided one due to the limited scaling of parallel work discussed previously, it is clearly the preferred algorithm for both high-thread (with near-optimal results achieved at 8 threads), low-thread and serial plane detection (with a 12x speedup over the undivided point cloud) across the range of concurrent hardware used in this project.

8 Future Work

Octree implementation was proposed and explored for this project from the interim stage. The final results support its benefits as the logical next step for improving the plane detector's performance, however time constraints prevented operational and benchmarked results being included. Looking at more abstract ideas, multi-plane storage across trials, a hierarchical plane structure and neural network classification are proposed to improve performance, accuracy and noise classification respectively.

8.1 Octrees

As shown in Figure 22 and used in Limburger [2] et al, octrees are a particularly efficient form of space subdivision for sparse models such as the Albany St studio. Firstly, the ability to have larger voxels encompass barren areas of points could improve the efficiency of the plane detector by removing the unneeded overhead of iterating into and checking empty voxels.

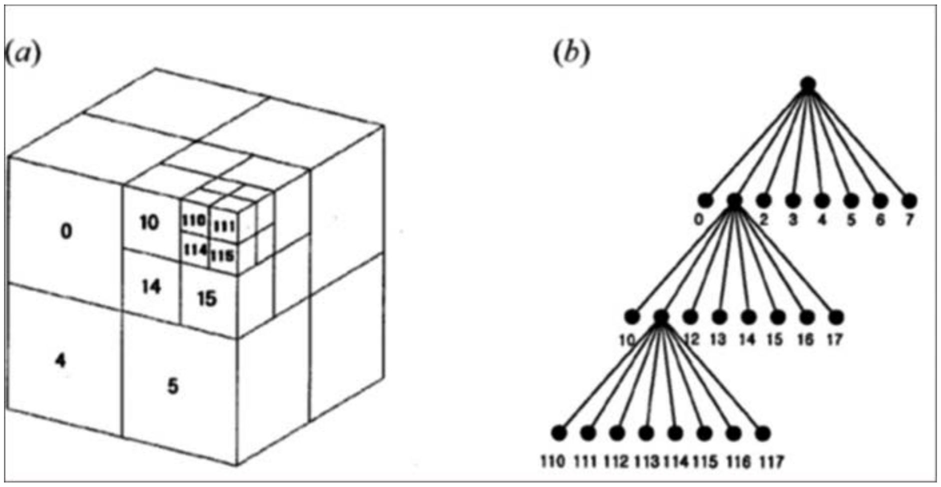


Figure 22: Octree diagram [12]

Secondly, variably sized octree voxels would standardize the capacity of their point lists for efficient parallelization, allowing the lower-thread performance of uniform space subdivision to be combined with the scalability of the undivided point cloud for high performance on both ends of the multithreading spectrum.

An Octree class was set up for the plane detector alongside the Uniform point cloud class, inheriting access methods from the flat vector PointCloud parent class. Though this class was not able to be implemented in the timeframe of this project, this structure allows the RANSAC method to take any data structure inheriting the PointCloud interface methods for easier addition in the future.

8.2 Storage of Multiple "Best" Planes

An avenue for further improvement lies in the observation that the minimum number of planes that would be acceptable to find in a scene is quickly noted by the user when viewing the model; the simple cube used to test the basic RANSAC implementation is noted to have 5 sides facing the camera, and the studio model must have at least 6 by the virtue of its closed rectangular geometry. This minimum number of planes could be inputted by the user and used to store as many "best" RANSAC trials for one plane as there are minimum planes remaining, and those planes used as a starting point for detecting subsequent planes. In the architectural models of interest for this project, the most obvious and distinct planes in a scene tend to be perpendicular [11] or parallel but a significant distance away. Applying this restriction to the planes saved, relative to the previous planes found, would ensure distinct planes are saved alongside the winner of each set of trials to speed up the next set.

8.3 Hierarchical Planes

When classifying the Albany St studio with a higher plane threshold, it generally produces six planes - the roof, the floor and the four walls.

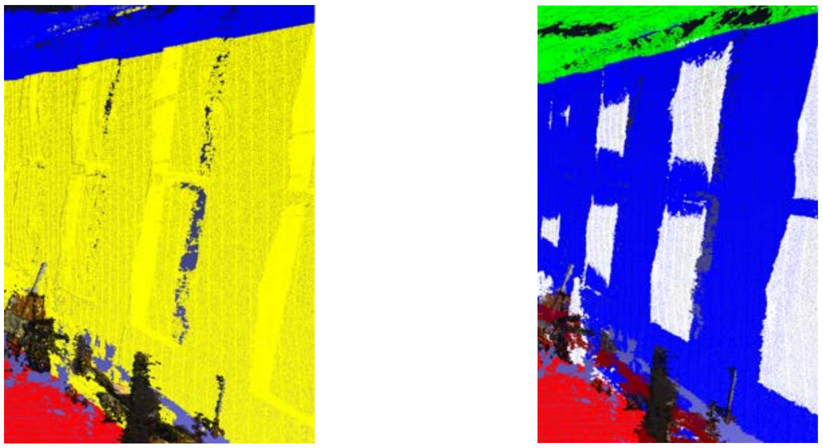

(a) Threshold of 2% of the model scale (b) Threshold of 1% of the model scale

Figure 23: A section of model that would benefit from hierarchical plane detection

The planes on a smaller scale that we are interested in, such as the roof panels and angled baffles (Figure 23), could be better detected and represented as sub-planes of the six sides. This would require a hierarchical system where key planes are detected and stored (potentially with the aid of the user estimations for the number of main planes discussed above), before being further split with a lower distance threshold into sub-planes.

8.4 Neural Network Point Removal

A stretch goal expanding on this project would be to implement more accurate classification of noise and unwanted objects with a neural network. As most of the clutter and measurement error in the point cloud is clearly recognized as not part of the subject by the human eye (for example, the music equipment on the floor of the Albany St studio and outlier points disconnected from any surfaces), a neural network mimicking the visual cortex holds promise to do the same without the explicit algorithms required of a Computer Vision approach. A pre-trained model, if enough training and testing point clouds were obtained to give reasonable accuracy, would have minimal performance impact when classifying which points to exclude from the plane detection.

9 Conclusion

In this report I have outlined the benefits of planar models over point clouds for high-fidelity architectural modelling, approaches taken to plane detection in the field and the improvements made to the basic RANSAC plane detector. Auto-generation of parameters has increased accuracy of the planes detected without prior knowledge of the point cloud, and an optimized parallel implementation has greatly improved performance on large models. Uniform space subdivision has further improved the run-time of the plane detector on 52 million points to just below one minute plus file write time, an 8.5x speedup over the basic RANSAC implementation. These results fufill the project's aims of accurate plane detection on a large-scale point cloud within reasonable time.

References

[1] Fischler, Martin A., and Robert C. Bolles. *Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography.* Communications of the ACM 24.6 (1981): 381-395.

[2] Limberger, Frederico A., and Manuel M. Oliveira. *Real-time detection of planar regions in unorganized point clouds.* Pattern Recognition 48.6 (2015): 2043-2053.

[3] Botterill, Tom, Steven Mills, and Richard D. Green. *New Conditional Sampling Strategies for Speeded-Up RANSAC.* BMVC. 2009.

[4] Nistér, David. *Preemptive RANSAC for live structure and motion estimation.* Machine Vision and Applications 16.5 (2005): 321-329.

[5] Hofer, Manuel, Michael Donoser, and Horst Bischof. *Semi-Global 3D Line Modeling for Incremental Structure-from-Motion.* BMVC. 2014.

[6] Mills, Steven. *Finding Planes In Point Clouds.* University of Otago COSC450 Assignment 2. 2017. https://www.cs.otago.ac.nz/cosc450/assignments/assignment2_2017.pdf

[7] *What happens inside Eigen, on a simple example.* Eigen Documentation, Wed Sep 30 2020, https://eigen.tuxfamily.org/dox/TopicInsideEigenExample.html

[8] Ayguadé, Eduard, et al. *A proposal for task parallelism in OpenMP.* International Workshop on OpenMP. Springer, Berlin, Heidelberg, 2007.

[9] *A High Performance Message Passing Library* OpenMPI, 29-Sep-2020, https://www.open-mpi.org/

[10] Cleary, John G., and Geoff Wyvill. *Analysis of an algorithm for fast ray tracing using uniform space subdivision.* The Visual Computer 4.2 (1988): 65-83.

[11] Dick, Anthony R., Philip HS Torr, and Roberto Cipolla. *Automatic 3D Modelling of Architecture.* BMVC. Vol. 1. 2000.

[12] Alba, Mario, et al. *Filtering vegetation from terrestrial point clouds with low-cost near infrared cameras.* Ital. J. Remote Sens 43 (2011): 55-75.