Accelerating Point Cloud Cleaning

Rickert Mulder

Supervised by

A/Prof Patrick Marais

Department of Computer Science

University of Cape Town

A thesis submitted for the degree of

MSc in Computer Science

2017

Contents

List of Figures

List of Tables

Acknowledgements

This thesis is in dedication to my family and friends for their unwavering support through the years and to Patrick for his infinite patience.

Abstract

Capturing the geometry of a large heritage site via laser scanning can produce thousands of high resolution range scans. These must be cleaned to remove unwanted artefacts. We identified three areas that can be improved upon in order to accelerate the cleaning process. Firstly the speed at which the a user can navigate to an area of interest has a direct impact on task duration. Secondly, design constraints in generalised point cloud editing software result in inefficient abstraction of layers that may extend a task duration due to memory pressure. Finally, existing semi-automated segmentation tools have difficulty targeting the diverse set of segmentation targets in heritage scans. We present a point cloud cleaning framework that attempts to improve each of these areas. First, we present a novel layering technique aimed at segmentation, rather than generic point cloud editing. This technique represents 'layers' of related points in a way that greatly reduces memory consumption and provides efficient set operations between layers. These set operations (union, difference, intersection) allow the creation of new layers which aid in the segmentation task. Next, we introduce roll-corrected 3D camera navigation that allows a user to look around freely while reducing disorientation. A user study shows that this camera mode significantly reduces a user's navigation time (29.8% to 57.8%) between locations in a large point cloud thus reducing the overhead between point selection operations. Finally, we show how Random Forests can be trained interactively, per scan, to assist users in a point cloud cleaning task. We use a set of features selected for their discriminative power on a set of challenging heritage scans. Interactivity is achieved by down-sampling training data on the fly. A simple map data structure allows us to propagate labels in the down-sampled data back to the input point set. We show that training and classification on down-sampled point clouds can be performed in under 10 seconds with little effect on accuracy. A user study shows that a user's total segmentation time decreases between 8.9% and 20.4% when our Random Forest classifier is used. Although this initial study did not indicate a significant difference in overall task performance when compared to manual segmentation, performance improvement is likely with multi-resolution features or the use of colour range images, which are now commonplace.

Chapter 1

Introduction

Laser range scanning enables detailed geometric record keeping of cultural heritage sites. Digital 3D records can help guide restorative maintenance and/or reconstruction efforts of damaged or destroyed heritage. They also allow physically accurate representations of noteworthy places to be exhibited on-line.

To create a 3D model of a heritage site, raw laser range scans need to be processed along with photographs in a 3D reconstruction pipeline. The reconstruction process follows a series of steps that require skilled operators with specialised software. Point cloud cleaning is one of the most labour intensive and time consuming parts of the process. For more complex environments, it can take an experienced operator from 30 to 120 minutes to clean a single scan. The speed at which the cleaning process proceeds is highly dependent on well designed software. Cleaning is a subjective task that involves manually removing unwanted points from laser scan data to ensure that the final reconstructed model is free of unwanted artifacts. Examples of such artifacts include tourists present during data capture or cars parked nearby. Unfortunately, for heritage scans, it is hard to quantify in advance what kinds of unwanted points will be present, which makes the automation of such a task a daunting prospect.

In this work we identify three problem areas in existing point cleaning software. Firstly, a cleaning work flow often require that intermediate results be saved in layers. Creating a large number of layers in existing systems can exert memory pressure, leading to sluggish performance and lack of interactivity. Secondly, viewpoint disorientation or restrictions whilst moving between areas of interest in a fully 3D workspace can slow down the cleaning process. Lastly, semi-automated segmentation algorithms are usually designed to isolate a small number of specific targets object types. Due to the unpredictable nature of unwanted object points in heritage scans, these algorithms are often inappropriate or they do not achieve a useful level of accuracy.

1.1 Aims

The aim of this research is to accelerate the point cloud cleaning process. In line with this aim, we have three objectives:

- Reduce the impact of inefficient layering on system resources.
- Reduce the navigation overhead of the 3D workspace.
- Speed up segmentation by introducing a segmentation tool that can learn object classes and segment them on the fly.

These aims are addressed through a carefully designed software framework, as explained below.

1.2 The CloudClean Framework

We implement our proposed solutions in the context of a new open source point cloud cleaning framework we call CloudClean. The main goal of the system is to facilitate the implementation and design of new semi-automated cleaning tools. In the design of our system, we address our first two objectives. The CloudClean framework addresses the objectives listed above as follow:

Figure 1.1: The process of using our semi-automated segmentation tool, involves creating an initial labelling, running the algorithm, and finally touching up the result

Firstly, a novel layering technique is introduced that support a large number of layers while consuming a near constant amount of memory. It also supports extremely efficient set operations. Secondly, a new roll-corrected first person camera is introduced that maximises rotational freedom while avoiding disorientating states. We show, via a user experiment, how this camera mode significantly reduces navigation overhead. Lastly, we create a semi-automated segmentation tool that harnesses a Random Forest classifier to interactively learn new object classes from examples, and assists the user in segmenting the remainder of a scene (see Figure 1.1). Preliminary findings from a user experiment show, without significance, that this method reduces the overall cleaning time.

1.3 Contributions

This principal contributions of this thesis are:

1. a new open source, cross-platform, point cloud cleaning framework designed for creating and evaluating new semi-automated segmentation methods. This framework is the first

open source point cloud software that supports and cleaning iterative work flow with undo capabilities.

2. a novel point selection layering technique that can support a large number of layers with a near constant amount of memory.

3. a roll-corrected first person camera that maximises rotational freedom while at the same time avoiding confusing camera orientation states.

4. a semi-automated segmentation technique that can learn new object classes and segment them on the fly.

1.4 Layout

The remainder of this thesis is laid out as follows. Chapter 2 provides an overview of the heritage 3D reconstruction pipeline and highlights inefficiencies in the cleaning process. Then, in Chapter 3 we introduce, CloudClean, a point cloud segmentation framework. In designing the system we address our first two objectives. An efficient layering scheme that consume near constant memory is introduced, and we show how a roll-corrected first person camera speeds up navigation by avoiding disorientating states. In Chapter 5 we develop an interactive Random Forest classification tool can be used to speed up segmentation and conclude with suggestions for future work in Chapter 6.

Chapter 2

Background

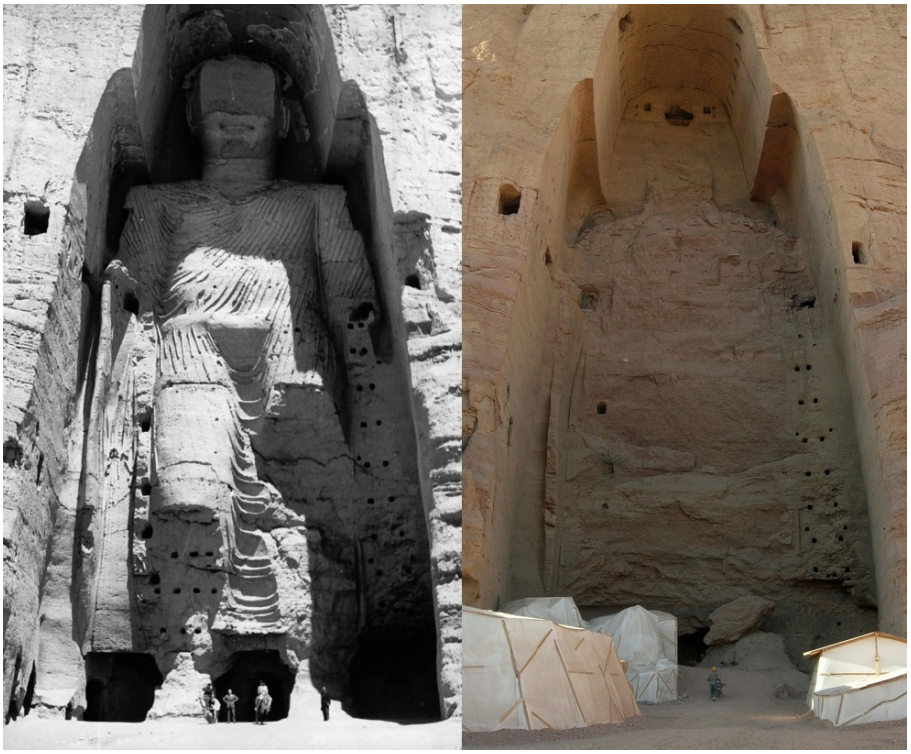


Figure 2.1: Taller Buddha of Bamiyan before and after destruction[1]

In 2001 the Buddhas of Bamiyan in Afghanistan (see Figure 2.1) were destroyed [62] by the Taliban government as the Buddhist symbols were seen as idols. In just a few weeks, centuries of history were destroyed. Heritage sites in many parts of the world face similar threats or are at risk of deterioration. Laser range scanners allow us to document the spatial characteristics of cultural heritage sites in more detail than ever before. Such digital 3D records can help guide restorative maintenance and/or reconstruction efforts. It also allows us to preserve a physically accurate record of these places and expose more people to heritage sites on the web.

[1]Source: UNESCO/A Lezine

2.1 3D scanning

3D scanners are optical devices that capture the shape, position and appearance of real world objects by recording the surface coordinates of objects. Many types 3D scanners are used in heritage preservation, each with specific strengths and weaknesses. 3D scanning technologies can be broadly classified into two categories, namely triangulation and time of flight scanners.

2.1.1 Triangulation scanners

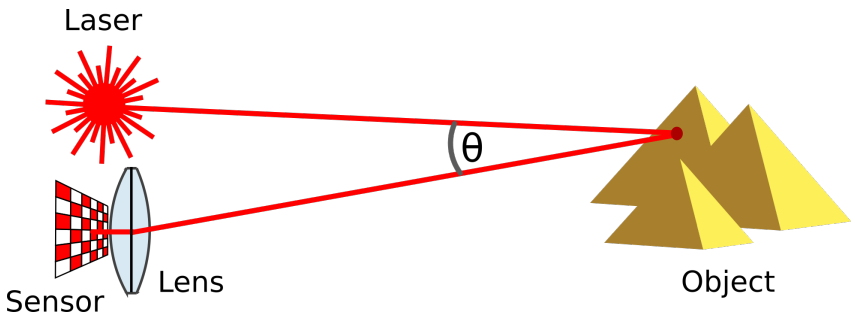


Figure 2.2: Laser triangulation scanner

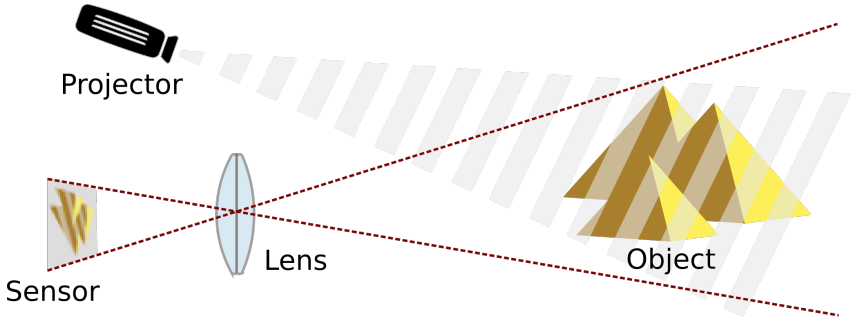


Figure 2.3: Structured light scanner

Triangulation scanners, as the name suggests, uses trigonometric triangulation to record the position of surface points. Triangulation scanners use either laser or structured light [9]. The laser version (Figure 2.2) emit laser pulses that are reflected by an object and recorded by a sensor at a known position relative to the pulse origin. The sensor directly measures the angle of the reflected laser beam, which is used to compute the position of a point on an object surface. Structured light scanners (Figure 2.3) emit a series of linear patterns. The reflected light patterns are captured by a camera sensor. Perspective distortions in the reflected light patterns are used to compute surface coordinates.

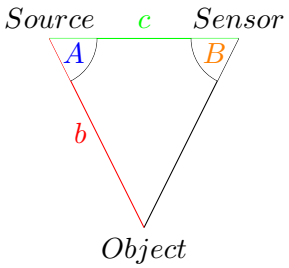


Figure 2.4: Triangulation

The position of a surface coordinate relative to the scanner can be computed using the triangle in Figure 2.4. Given the distance between the light source and sensor (c), the outgoing angle of emitted light (A), and incoming angle of reflected light (B), the distance to the object is given by $b = c\frac{sin(B)}{sin(\pi - A - B)}$.

2.1.2 Time of flight scanners (TOF)

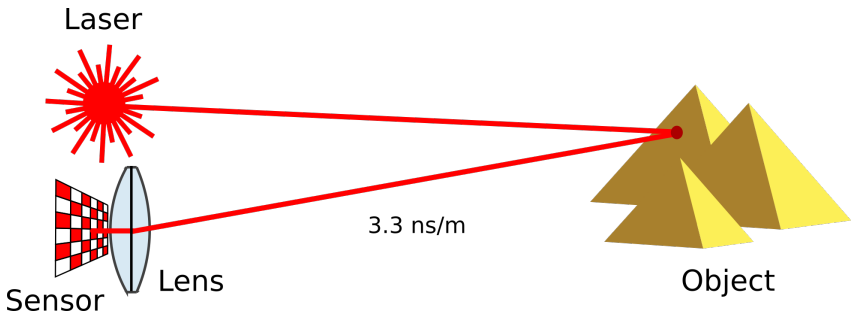


Figure 2.5: Time of flight scanner

Time of flight (TOF) scanners emit laser pulses similar to triangulation laser scanners. Unlike triangulation scanners, it uses the time it takes for a pulse to reflect off an object and return to measure position (see Figure 2.5). Given the round trip time of the pulse (t), the distance (d) is given by $d = ct/2$, where c is the speed of light. The accuracy of the distance calculation depends on how accurately time can be measured [18].

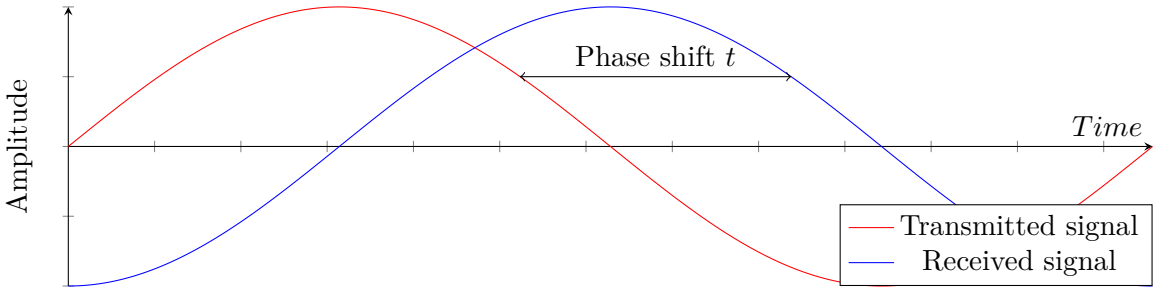


Figure 2.6: Phase shift in returned signal

Type	Range	Precision	Speed	Portability
Structured light	<1m	10 micrometer	Seconds	High
Laser triangulation	<1m	10 micrometer	Seconds	Medium
TOF	2-1000m	Medium	Minutes	Low
Phase	2-100m	Low	Seconds to Minutes	Low

Table 2.1: Comparison of scanning technology

Laser phase-shift scanners also measure the round trip time of a laser pulse [19]. What differentiates phase-shift scanners from traditional laser scanners is that the power of the laser light is modulated in a sinusoidal wave. The phase shift in the returning pulse is used to compute the round trip time (see Figure 2.6). Due to the cyclical nature of the signal, the distance computed from the phase shift can be ambiguous. This ambiguity can be resolved by taking measurements across multiple frequencies [5].

2.1.3 Comparison

Triangulation scanners can achieve 10 micrometer precision over distances less than one meter. Over longer distances, however, the triangle in figure 2.4 becomes elongated. This results in less accurate distance calculations when compared to TOF scanners [34]. Structured light triangulation scanners are typically hand-held and less prone to motion distortions when compared their laser based counterparts. Structured light scanners also tend to be faster and easier to operate compared to other scanners [9]. This makes them preferable when capturing smaller objects at short range.

TOF scanner accuracy is determined by how accurately the round trip time of the laser pulse can be measured. Compared to triangulation scanners, measurement error makes time of flight scanners less accurate over distances less than 2 meters. Over larger distances (up to 1 km), time of flight scanners have an advantage over triangulation scanners [18]. Time of flight scanners are, however, slower than triangulation scanners. The speed of TOF scanners depends on the resolution it is set to capture at. Tens of thousands of points may take seconds while resolutions with millions of points may take minutes.

Phase-shift laser scanners occupy the niche in between triangulation and traditional TOF scanners. Phase-shift scanners are effective in the 2-100m range and are much faster compared to TOF traditional scanners. A phase-shift laser scanner's range is, however, limited by the cyclic nature of their sinusoidal pulse [5]. Objects from beyond the scanner's designed range can be erroneously interpreted as being within the design range, which results in hard to remove artefacts. This happens when the scanner fails to disambiguate pulses returned from close by and far away objects. Phase-shift scanners compensate for this disadvantage by being much faster and more accurate than traditional TOF scanners. [18]

2.2 3D reconstruction pipeline

[1]Source: `http://www.rapidform.com/3d-scanners/`
[2]Adapted from Rüther and Held [51]

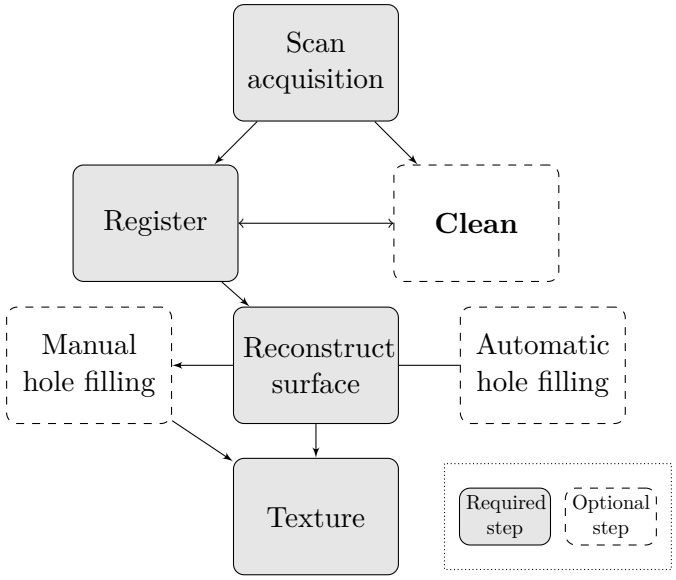


Figure 2.7: Reconstruction pipeline [2]

The 3D modelling of a heritage site starts with the acquisition of range scans. Collected scans are subject to a series of processing steps (see Figure 2.7). Unwanted objects and noise are typically removed during cleaning. Missing data, often caused by occlusions, can be synthesised during hole filling. Subsequently, scans are combined by transforming them into a common coordinate frame during registration. After registration, a surface model can be constructed from the combined point set. Finally the reconstructed mesh is textured which results in a final model [51].

2.2.1 Data acquisition

Some degree of planning is required before scanning a heritage site. Firstly, an appropriate resolution needs to be agreed on. The image resolution determines the time required to capture a scan, and later processing time is determined by the number of point samples. Furthermore, planning equipment placement ahead of time can help ensure that an optimal amount of coverage is achieved as some degree of scan overlap is required in order avoid registration problems later in the pipeline. [51]

This work focuses primarily on terrestrial TOF and Phase-shift laser scanners. TOF and Phase-shift scanners rotate around a tripod while recording angle and range measurements as the laser pivots up and down. The result is a 3D image in the shape of a dome that has a hole on the ground where the tripod stood. This image is exported as 3D coordinates organised into a 2D grid referred to as a range image (see Figure 2.8). In addition to XYZ samples, range images may also include the intensity return of the laser pulse, colour values, and other scanner specific meta-data [19].

We limit our work to range images that includes intensity values but not colour. This because colour scans were not readily available to us at the time.

[3]Rendering based on data provided by the Zamani Project

Figure 2.8: 2D scan grid of intensity values [3]

2.2.2 Registration

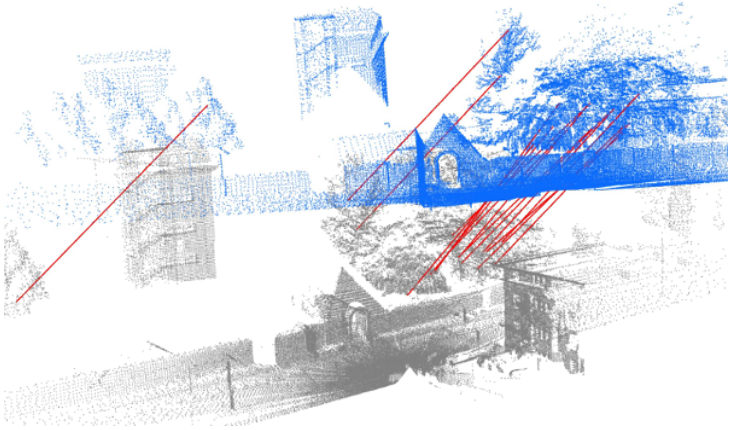


Figure 2.9: ICP correspondences [4].

Range images record object points with diminishing surface resolution as a function of a object's distance from the scanner. Scans from multiple perspectives can be combined to fill holes caused by occlusions and achieve a more uniform sample density. The process by which scans are transformed common coordinate system is called registration.

Scans can be registered with or without the use of physical reference objects called targets. Targets are be placed in and around a heritage site during data acquisition in order to find correspondence points in two or more scans. [4]. Without targets, correspondences need to be determined via distinct surface features. Algorithms such as Iterated Closest Point (ICP) can automatically align scans using such surface features [3].

The use of targets is likely to result in highly accurate registration without the need for much human intervention. However, in order to achieve this, targets need to be captured at a high resolution which may extend the duration of an expedition. Some have found targets to be impractical as they often need to be placed in hard to reach places, or are required in large numbers when dealing with intricate indoor environments [51].

[4]Source: `http://pointclouds.org/documentation/`

Time end effort to complete an scanning expedition is reduced when targets are omitted during data acquisition. An ICP based registration procedure, may however, require more human effort in order to achieve a good initial alignment, which is a prerequisite of ICP. Once an initial alignment is achieved ICP can compute correspondences between surfaces. Correspondences are used to compute incremental transformations that minimises the distance between two surfaces. Transformations are applied after each correspondence computation until convergence is reached [47]. ICP can sometimes fail. When registering surfaces with relatively few features, unique correspondences are less likely to be found. Variations of the original ICP algorithm that use different types of correspondent matching and optimisation procedures, can help reduce such instances. The procedure, however, remains labour intensive as success is never guaranteed.

2.2.3 Cleaning

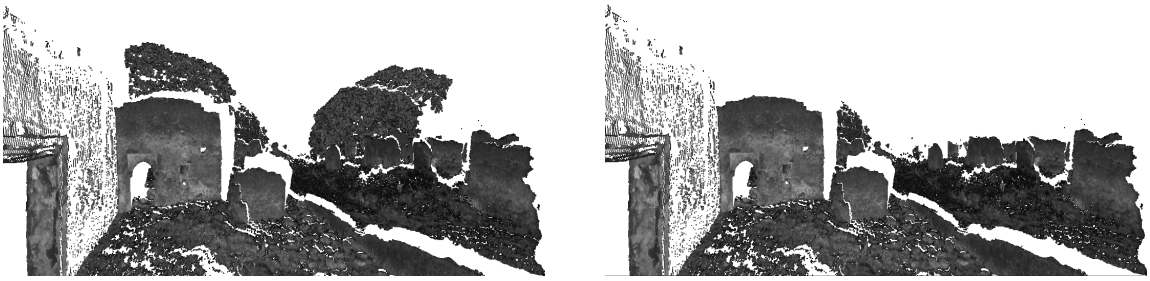


Figure 2.10: Trees cleaned from a scan

Cleaning could be considered an optional step. However, unless a scanning expedition proceeds flawlessly, unwanted artefacts will compromise the quality of final model. Unwanted artefacts include: trees, people, power lines, cars, animals, etc. as well as scanner induced artefacts (see Section 2.1.2). An object is usually removed because it is not part of the subject matter. Many meshing algorithms, however, are unable to produce coherent surface meshes from points associated with trees and shrubs. For this reason vegetation is usually also targeted.

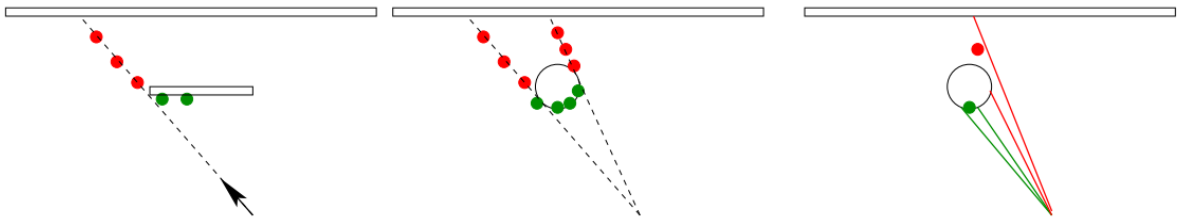


Figure 2.11: Mixed pixels. Green are valid points and red are not. [64]

Scanner induced artefacts is the result of imperfect equipment rather than physical surfaces. A common type of noise is the mixed pixel phenomenon. This occurs when a laser pulse partially strikes a nearby and distant object. The result is an interpolated data point between the near and far surface [64] (see Figure 2.11). This typically manifests as column of points behind doorways and other edges.

Cleaning is mostly a manual task that requires expert judgement. It can be performed on range images before registration or on point clouds after registration [51].

Registration helps mitigate problems associated with non-uniform sample density and incomplete data. Registration, however, also results in the creation of large monolithic files. Working on large registered point sets on under resourced machines can be problematic. It is therefore often preferable to perform cleaning on unregistered range images. A disadvantage of this approach is that one will encounter and clean the same region in more than one range image, as scans need to overlap to be registered. This disadvantage is, however, offset by access to a 2D grid view that an make it easier to identify regions that can be hard recognise in a 3D point cloud view.

2.2.4 Surface reconstruction

Surface reconstruction is the process that converts a discrete point model into a triangulated surface model. This can be achieved either via interpolation or approximation. The goal of interpolation is to connect neighbouring points by computing a triangulation whereas approximation methods aim to approximate a surface that fits the samples [53].

Interpolation methods are very sensitive to noise. Poor triangulations can be produced when scans are not properly registered or the point samples exhibits high variance. Preprocessing can mitigate these problems but is time consuming and often leads to a loss of detail [51].

Surface approximation is less susceptible to noise when compared to interpolation methods. Poisson surface reconstruction [30] is a popular method because it is both noise resistant and retains great detail. The technique uses an interpolated normal field to solve a Poisson equation. Results are dependent on the quality of the normal estimation. It is preferable to compute normals prior to registration, in order to save time, when using this approach. Poisson surface reconstruction produces "water-tight" surfaces and thus automatically fill small holes. While this is useful for producing visually pleasing 3D models, it result in a historically inaccurate model as new surface data may have been synthesised in the process.

Moving Least Squares (MLS) is another popular surface approximation method that does not over smooth or interpolate missing data [1]. Like Poisson, it also requires normal estimates. MLS computes a local surface approximation at a sample point by considering its neighbourhood. Every point in a neighbourhood is weighed according to its importance. A surface is then computed by minimising the weighed distance to the surface for each point in the neighbourhood. Poisson and MLS have both been adapted for out-of-core execution and GPU acceleration [35].

2.2.5 Hole filling

It is unlikely that a scanning expedition will achieve complete coverage of a site. Furthermore, samples are lost during cleaning and surface reconstruction. Hole filling is an optional step in the reconstruction pipeline that seeks to synthesise missing data [57]. For historical data, hole filling is not desirable as data is synthesised. Models that have been filled are more aesthetically pleasing when exhibited. It is therefore important that any synthesised data is labelled as such in the historical record.

Small holes can be filled automatically by surface reconstruction algorithms (discussed in 2.2.4). Larger patches are harder to fill convincingly with automated methods and may require manual effort to produce good results [51].

2.2.6 Texturing

The final step of the reconstruction pipeline is texturing. Coloured models are not only visually pleasing, but also an important part of the historical record. Some laser scanners are equipped with sensors that record a colour value with each vertex. Although these colour values can be a convenient way to texture a modal, using this data can produces inferior results when compared to the use of photos [51]. One problem when texturing a model is dealing with changing light conditions throughout the day. Because scanning is time consuming, vertex colours sampled by a scanner may vary between overlapping scans. Taking photos is less time consuming so it is easier to collect samples around the same time of day. When using embedded colour, one is also limited to the resolution of the geometry. Photos on the other hand, lets us use textures with a higher resolution than the geometry.

Texturing from photos is, however, a more involved process. The texture needs to be manually projected onto the geometry. To achieve this, internal and external camera parameters need to be known or estimated. This includes the position and orientation of the camera as well as the focal length of the lens. If these parameters are not known, they can be estimated via software by selecting or computing correspondences between pictures [51].

2.3 Cleaning

Capturing and reconstructing a heritage site is a very time consuming and labour intensive process. After data acquisition the majority of man hours are spent on cleaning. In large heritage scanning initiatives, a single range image can require between 30 and 120 minutes of work by an experienced individual. Very high resolution scans can take up to a day to clean. Consequently, existing techniques leave much to be desired in terms of cleaning efficiency [51].

The goal of point cloud cleaning is to separate wanted from unwanted points. In order to create this separation, one needs to recognise what physical world objects, if any, point samples represent. Despite recent advances in computer vision and machine learning, it is very difficult to fully automate point cloud segmentation since what qualifies as “unwanted points” is largely determined by context and can be highly subjective. This dependence on human judgement is a big reason why point cloud cleaning remains a human driven task.

The point cloud cleaning task requires a user to group laser point samples through the use of a 3D workspace. This requires the user to first bring an area into view, before the points can be selected via various selection tools. This process is repeated many times for a particular scan, so it is important to have the ability to reverse mistakes.

2.3.1 Navigation

To segment a region of interest, the user must first use the interface to navigate to the relevant location in the point set. The navigation time can be considered as an overhead of the core cleaning task. Employing efficient navigation techniques can thus play a large role in reducing the overall cleaning time.

The main difficulty associated with 3D navigation is controlling 3 degrees of rotational and translation freedom using 2 dimensional mouse and keyboard inputs. Most 3D software implement variations of the ArcBall [58] or Virtual Sphere [11] to solve this problem. These approaches allow the user to orbit around an object by dragging a point on the surface of a virtual ball. This is typically paired with the ability to translate the camera via "Pan" (along x-y axis) and "Zoom" (along the line of sight). Rotation around a central point is best suited for object manipulation or exploratory movement. During cleaning of large scenes, targeted movement through the 3D workspace is more common.

Navigating along a horizontal plane is natural for surface dwelling individuals [26]. Many games, facilitate such movement over planes, by employing a first person perspective (FPP) camera for 3D navigation. In this navigation mode the user typically translates the camera position along a horizontal plane using arrow keys while using the mouse to change the camera orientation. This can also be extended to support vertical movement (flying) as is done in games such as Second Life [5].

Most point cloud and range image editing systems only provide a 3D view of scan data. Humans are, however, more adept at interpreting 2D data despite living in a 3D environment [33]. This is likely the reason why reason Z+F [71] includes a 2D panoramic view of range image intensity values in their system. This provides the user with another perceptive that can help identify objects. It is especially useful when inspecting sparsely sampled objects that, when viewed in 3D, can appear to be nondescript points floating in space.

2.3.2 Selection

Once an area of interest has been brought into view, the core segmentation task can be performed. The manner in which a user approaches the task is affected by the chosen software's feature set. Point cloud software typically provide a user with some notion of a *selection* and/or *layer*. Selections allow one to temporarily label points before applying an operation, such as deletion. Layers are generally used to keep track of points that are not currently being manipulated and can be hidden from view until needed at a later time. Hidden layers allow a user to follow a more iterative cleaning work flow: instead of discarding points immediately, large areas can saved for later refinement. Using layers to keep track of unwanted points, rather than deleting them, can be preferable as preserving original scan data is often required by heritage practitioners.

Point cloud software is not usually designed specifically for cleaning. For generalised point cloud editing tasks, actions such as inserting new points into a cloud or modifying the position of points may be need to be supported. These actions are however not used in pure segmentation. General purpose point cloud editing therefore need to meet a more restrictive set of design

[5] `http://secondlife.com/`

constraints. These constraints can result in a compromised cleaning work flow in order to satisfy requirements of other work flows.

Software such as Meshlab [13] and CloudClean conceptualise the workspace as a set of mutually exclusive layers. New layers are created by duplicating or removing points from the original layer. This allows points in layers to be transformed independently. The disadvantage is that recovering memory after removing points, requires that the original data structure be resized which can incur a performance penalty. Alternatively each subsequent layer will consume additional memory if removed points are simply marked as such. When working with large datasets or on under resourced machines, memory pressure can compromise system performance and in turn increase task duration in this scenario.

Propriety software such as Bentley Pointools [44] appear to represent layers by mapping each point to a boolean value that indicates its membership status with respect to the layer. This approach incurs a cost of $O(n)$ per layer, where n is the size of the point cloud or range image referenced in the layer. Bentley, however, restricts the number of layers to 7 - possibly because of high memory overhead associated with each additional layer.

2.3.3 Segmentation tools

An efficient point cloud segmentation tool should allow one to *accurately* isolate a set of *target points* in as short a time as possible. The primary goal of a selection tool is thus to minimise time and maximise accuracy.

The accuracy of a classification is often operationally defined using an F-score. An F-score combines both precision an recall in a single metric (see Equation 2.1). Given a set of target points that we want to segment, precision is the number of correctly selected points divided by the total number of selected points (see Equation 2.2). Recall is the number of correctly selected points divided by total number of targeted points (see Equation 2.2). Precision and recall is visualised in Figure 2.12. The F-score combines precision and recall as a weighted sum.

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{2.1}$$

$$\text{precision} = \frac{|\{\text{target points}\} \cap \{\text{selected points}\}|}{|\{\text{selected points}\}|} \tag{2.2}$$

$$\text{recall} = \frac{|\{\text{target points}\} \cap \{\text{selected points}\}|}{|\{\text{target points}\}|} \tag{2.3}$$

Segmentation tools can be divided into automated, semi-automated and manual tools. Automated tools segment a scan without any user input, while semi-automated tools require a form of user input before or after performing automatic segmentation. Manual tools directly translate inputs into a segmentation action.

[6] Adapted from: `https://en.wikipedia.org/wiki/Precision_and_recall`

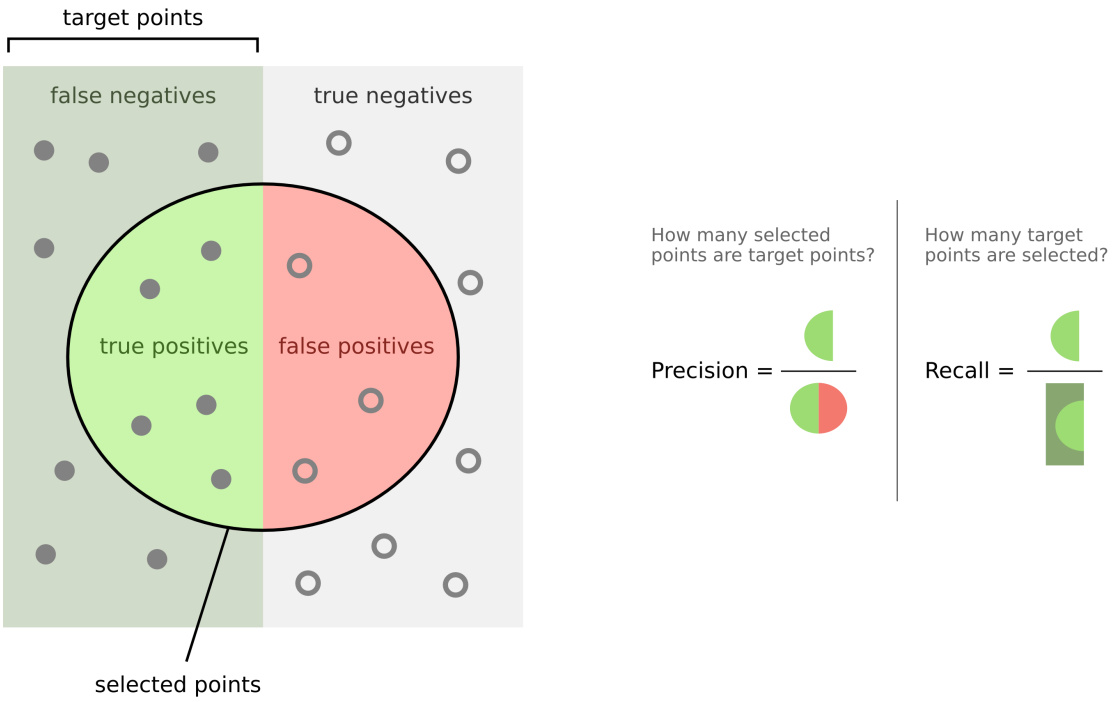


Figure 2.12: Precision and recall [6]

Due do the high accuracy requirements of heritage cleaning, the level of accuracy obtained through automated tools will rarely be sufficient. A user will thus usually have to manually relabel incorrectly segmented points. Because most cases will still require user input, such tools effectively only provide semi-automated segmentation. Semi-automated tools that require upfront user input, typically also require manual touch up work after the automation has completed. Manual segmentation can be very costly in terms of time in effort. Ideally, semi-automated tools would be used to perform the bulk of the work, and manual tools would only be used for the final touch up.

The amount of touch up required is related to the degree of accuracy that can be achieved by an algorithm. Touching up a high accuracy segmentation can however still be very time consuming if the the incorrectly labelled parts are widely distributed and require many manual actions. It is not hard to imagine a scenario where the invocation of an algorithm results in a state where it would have been better to have manually segmented the scan from scratch. The value of a semi-automated segmentation algorithm should therefore not be measured in accuracy but the degree to which it reduced the overall task time.

2.3.4 Manual selection tools

The most manual selection mode is *point picking*, which allows a user select a single point by clicking on it in the view port. This can be implemented either via ray casting or by directly reading the point index from the frame buffer. Ray casting is typically more suitable for lager objects as floating point errors become problematic when targeting small areas. This selection mode can be found in most software, including Meshlab [13].

Brush tools are similar to point picking tools but have an adjustable area of influence. Instead of selecting a single point, all the points within a radius of the clicked area is selected. One way is to compute the selected points is by reading point indices contained within the area of influence from the frame buffer. A potential problem with this approach is that occluded or culled points, that are not rendered, will not be selectable. Neighbouring points in screen space could also be at different depths, so a user may unintentionally select background points. Instances of this tool can be found in propriety packages including Bentley Pointools [44].

Polygon selection requires the user draw a 2D polygon on the view port around an area of interest. The points contained within the on screen polygon are selected when then polygon is completed. As with other tools, using the frame buffer to resolve points can be problematic due occlusions. This can be overcome by projecting all points to the view port plane and performing a point in polygon test on each vertex. For large point clouds interactivity may be compromised if the computation is performed on the CPU. A polygon selection tool is most effective when target points that can be separated from their surroundings by changing the camera perspective. If such a perspective cannot be obtained, additional work is usually required to remove background points from the selection. Systems such as *Cyclone* [32] mitigate this problem by allowing the user to restrict a tool's area of influence with a limit box.

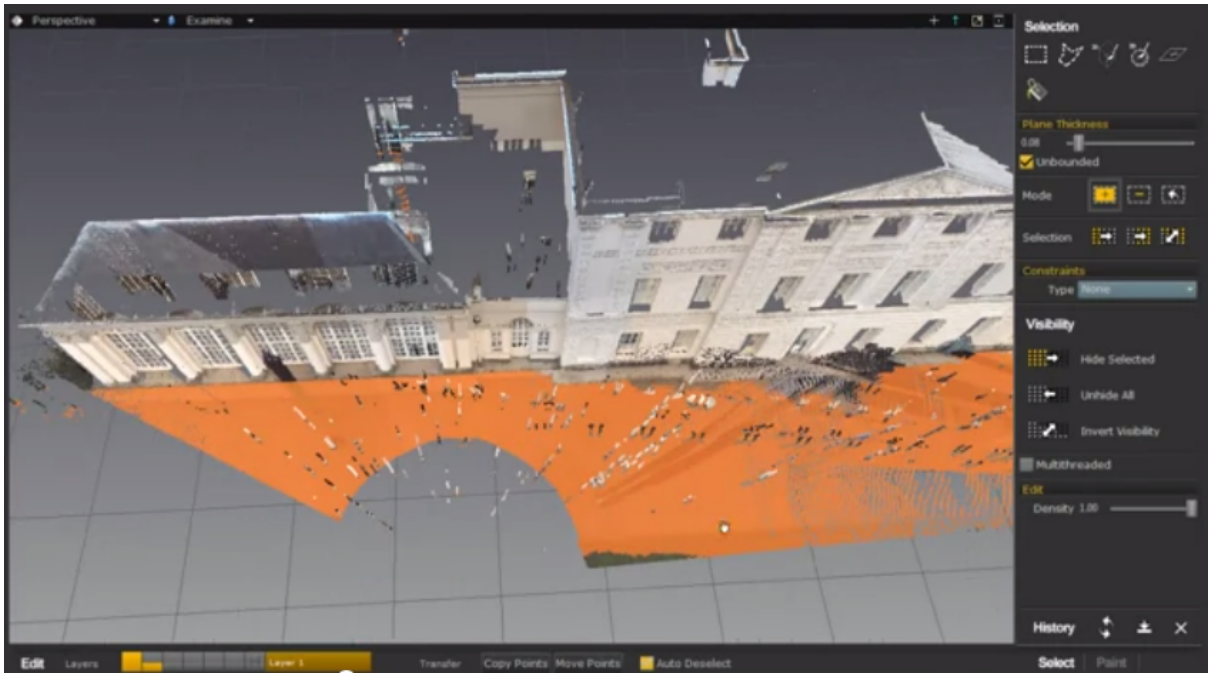


Figure 2.13: Plane selection tool in Pointools [44]

2.3.5 Semi-automated tools

Semi-automated tools range from simple filters to more complex machine learning algorithms. To speed up the cleaning task, the processing time associated with the segmentation algorithm needs to scale sensibly with what it accomplishes. Specifically, the longer the task runs, the more useful work it will need to perform to remain useful.

Flood fill tools uses a region growing algorithm usually expands the selection recursively to neighbouring points if the membership criteria is met. Complex membership computations can affect runtime of a flood fill tool. Pointools [44] lets one control the membership criteria by setting a threshold on the colour or intensity. Other variations include plane selection [32] (see Figure 2.13) and power line removal [65]. The main difficulty with flood fill tools is that it can be hard to determine an appropriate threshold value for a target region. This can result in an excessive amount of trial and error. Meshlab somewhat mitigates this problem by allowing a user to interactively control the extent of the fill via the mouse wheel.

Outlier removal is a filter, provided by Meshlab, that discards points that have less than a specified number of neighbours. Processing 5 million points using a 0.5m diameter takes 12 seconds on our test hardware. Removing isolated points manually would take much longer than 12 seconds. When used with range images the non-uniform density the scan becomes a problem. Because the sample density of a range image decreases away from the origin, samples further away are more likely to be classified as noise using this filter. Consequently, increased manual correction may be required to achieve the desired selection accuracy.

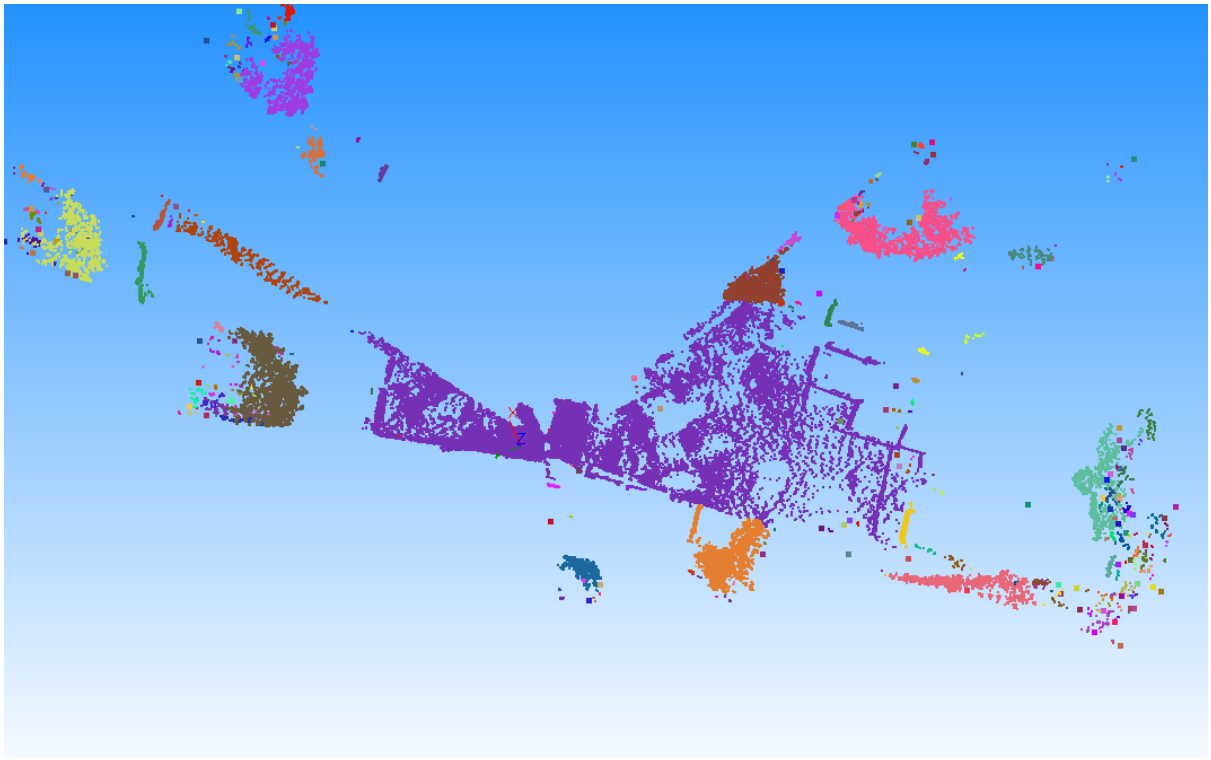


Figure 2.14: Clustering issue in 3D Reshaper [60]

Clustering tools are also affected by non-uniform density problems. *3D Reshaper's* [60] clustering tool lets one automatically group spatially related points. When used on range scans it can be seen that more clusters are detected further away from the origin (see Figure 2.14).

Other semi-automated tools include those aimed at extracting ground planes [61, 65], finding rooftops [65], walls [65], buildings [61] and vegetation [61]. One problem with most specialised semi-automated segmentation tools is that the targets that it has been designed to segment, usually manifest differently in heritage sites. Old eroded structures are harder to segment using tools designed to for modern buildings. Because of the strong emphasis on accuracy in heritage

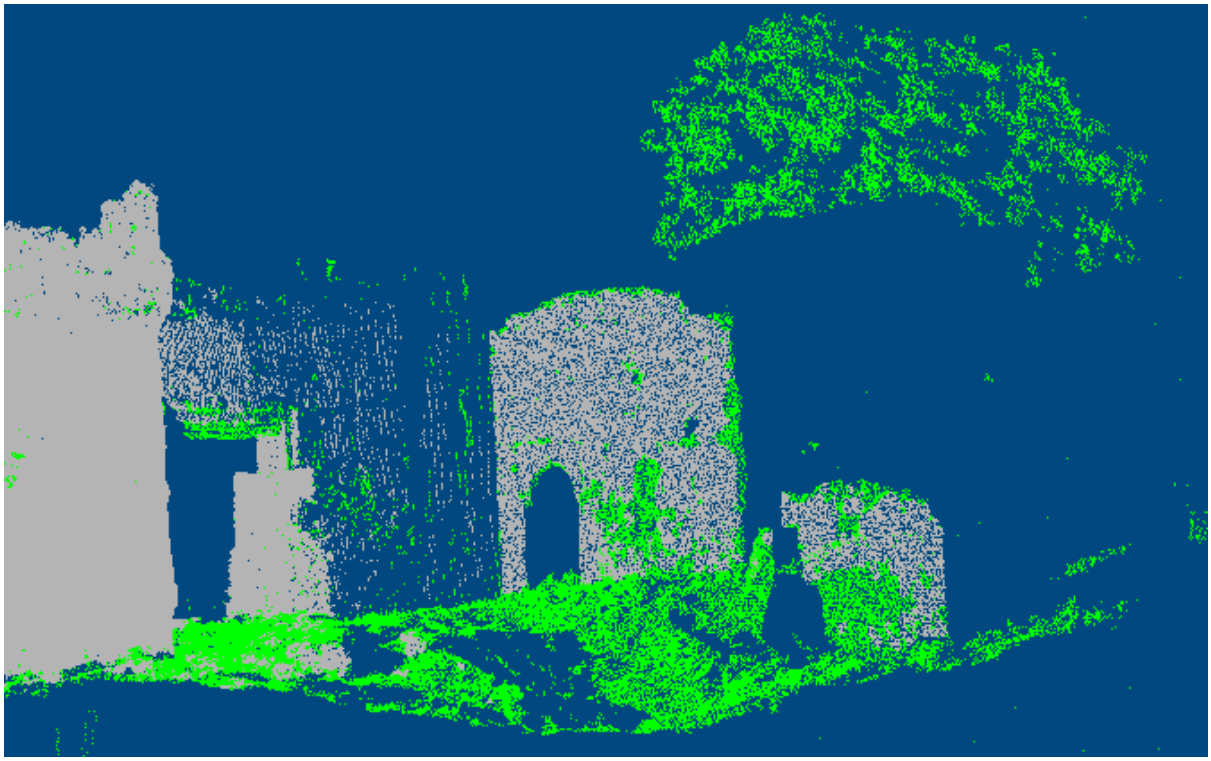


Figure 2.15: Vegetation filter in VR Mesh Studio [65]

preservation, segmentation results that may be sufficient in a non-heritage context are unlikely to be good enough for preservation purposes. The touch up time for the segmentation results may thus exceed useful levels. One such case is illustrated by the vegetation removal tool in VR Mesh Studio's [65]. This tool erroneously label points as plant growth when they are not - as shown by the mislabelling on the eroded ruin walls in Figure 2.15.

Not only can segmentation targets manifest in different ways, but there are a virtually endless number potential targets. Because it's not feasible to use hundreds of specialised tools, in the heritage context it is often more practical to revert to manual methods.

2.3.6 Undo

Besides navigation and selection, other system aspects can also affect task duration. When your work is the cumulative result of potentially hundreds of actions, the ability to undo an action is an important feature [41]. Without it, a mistake can derail hours of work. A user will also need to exercise greater caution, which can further decrease task performance [36].

Unlike most propriety software, the two most popular open source systems for point cloud editing, Meshlab and Cloud Compare, do not implement undo functionality.

2.4 Summary

Digital heritage preservation using laser scanning, is an important but also very time consuming process that requires a lot of planning and processing. One of the most time consuming phases is the removal of unwanted noise and artefacts from laser range images. Despite advances in computer vision and machine learning this task remains mostly human driven. Well designed software plays a key role in determining the speed at which the cleaning task can be accomplished.

We've identified three areas that can be improved upon in order to accelerate the cleaning process. Firstly the speed at which the user can navigate to an area of interest has a direct impact on task duration. Secondly generalised point cloud editing software has more design constraints that can lead to system architectures that are not ideal for a cleaning work flow. One such instance is the sub optimal abstraction of layers that may extend a task duration due to memory pressure. Finally, existing semi-automated segmentation tools have difficulty targeting the diverse set of segmentation targets in heritage scans. Most semi-automated segmentation tools are designed for specific targets that, when targeted in heritage scans, may not achieve a sufficient degree of accuracy. While possible to design better algorithms, the potentially unlimited number of targets that users may want to remove, makes it impractical create a specialised algorithm for each.

In the next chapter we present a new open source framework for range image segmentation that addresses the first two areas. The core system architecture features a novel system architecture and layering technique designed to reduce memory pressure and reduce computation. We also implement a first person roll-corrected camera mode and show how this reduces the navigation overhead of the cleaning task. In the final chapter we show how Random Forests can be trained interactively, per scan, to assist users in a point cleaning task.

Chapter 3

Software framework

In this chapter we cover the design and implementation *CloudClean*, an open source point cloud segmentation framework.

The aim of this system is to facilitate the design and evaluation of point cloud segmentation tools. During the design of this system we address navigation overhead through the use of a roll-corrected camera mode and introduce an efficient layering architecture that supports a large number of layers at near constant level of memory consumption.

3.1 System overview

A new segmentation technique is best evaluated by measuring its impact on an existing work flow. This allows one to attribute changes in performance or accuracy, to the introduction of the new tool. Existing open source solutions do not lend themselves to an iterative cleaning work flow due to the lack of undo support and versatile layering. We therefore implement a new extensible open source system that addresses these shortcomings. We also implement a set of the most commonly used tools from existing systems, namely *brush*, *polygon lasso* and *flood fill* tools, in order to recreate a typical manual cleaning flow.

Tools are implemented in the context of a 3D workspace (see Figure 3.1). The main view port contains a 3D view of one more range images. A tab at the top of the window lets a user switch to a 2D panoramic view. On the left side of the view port, under the undo and redo buttons, tools can be activated and deactivated via toggle buttons. Activating a tool exposes tool specific options in the right hand panel. Selection tools, such as the brush, can make use of 8 selection colours. Selections can be converted into layers that are listed in the layers panel above the tool options. Points associated with hidden layers can not be selected.

We use an extended version of PCL's (Point Cloud Library) [50] *PointCloud* data type to represent point cloud state. Using PCL in our system allows one to reuse a large number of existing algorithms.

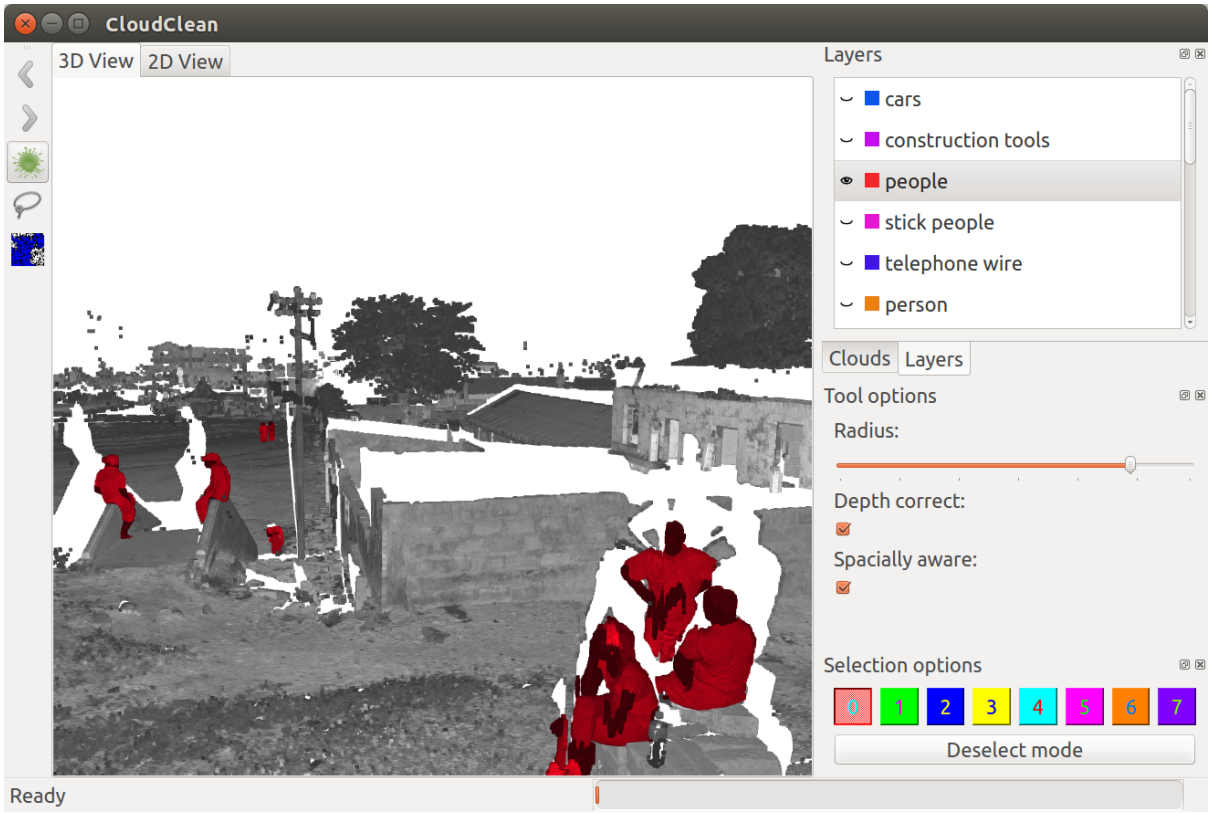


Figure 3.1: CloudClean interface

3.2 Architecture

CloudClean is designed using a plug-in architecture. The core of the system manages and renders point clouds, layers, and selections. Other functionality, including input/output and state manipulation, is implemented using run-time plug-ins (see Figure 3.2).

The system starts by setting up data structures for managing point clouds, associated layers, and a undo stack for manipulating state. These objects are then passed to the main window that initialises OpenGL and sets up the 2D and 3D view ports. The main window also creates listeners so that the GUI can be updated in response to changes in the point cloud and layer data. Once this is completed the plug-in manager is started by passing it a reference to these core system components.

The plug-in manager is built on top of QT's plug-in framework. QT provides an abstraction over operating system specific linker and loader APIs that allows one to build cross platform extensions that can be loaded and unloaded at run-time. The ability to reload plug-ins at runtime enables one to update a tool's code while maintaining state in the core system. This facilitates a quicker compile-run-test cycle as only the plug-in code needs to be reloaded.

All plug-ins implement the same interface (see Listing 3.1) regardless of functionality. The requirements of this interface are: functions that report the plug-in's name, initialise the plug-in, and a clean up its state. The QT plug-in framework lets us attach meta data to plug-ins that can be read before the plug-in is loaded. CloudClean uses a meta data file included via

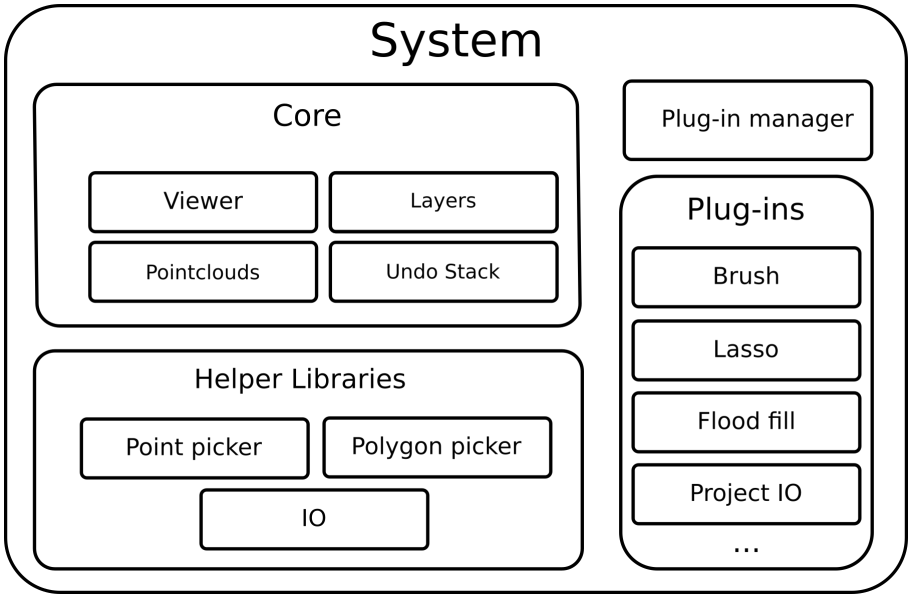


Figure 3.2: The system is implemented as a plug-in architecture. The user interface and state management is handled by the core system. Segmentation and other tools that manipulate the system state are loaded into the system as plug-ins.

this mechanism to specify the version of the plug-in, and what other plug-ins it is dependent on. When started, the plug-in manager reads the meta data of each plug-in located within a predetermined directory. In doing so, a dependency graph is created. The system ensures that each plug-in's dependencies are loaded before loading it. The loading order is important as the system will crash if symbols defined in other plug-ins cannot be resolved.

A two step plug-in initialisation procedure is used. During the first phase a reference to the core system is passed to each plug-in. This allows a plug-in to install itself by performing tasks such as creating new menu items, hooking into the render loop, and listening for system events. After the first initialisation phase, some plug-ins will have fully initialised themselves. In the second phase, plug-ins that are dependant on other plug-ins are given a reference to the plug-in manager. This allows a plug-in to look up and call other plug-ins that it may depend on.

After each plug-in is loaded and initialised, the plug-in manager starts monitoring the plug-in directory. The system will load any new plug-ins that are detected, unload plug-ins that are deleted, and reload modified plug-ins. Before unloading a plug-in the clean-up function is invoked. This gives the plug-in the opportunity to remove references to itself from the system before its destructor is called.

Aside from the core and plug-in system there are some static libraries in the main binary with commonly used functionality. The most notable are the point picker, polygon picker and PTX IO code. The functionality in these libraries does not change often and is thus included as part of the main binary.

This system architecture was based on Meshlab's design, but improved on it in two key ways. Firstly Meshlab defines 4 types of plug-in interfaces: *edit*, *filter*, *render*, and *IO*. These interfaces limit what functionality plug-ins can provide. Because our approach does not prescribe what a plug-in can do, developers are free to extend or modify the system in any way

that proves useful. Secondly, replacing statically compiled plug-ins with run-time reloadable plug-ins, reduces research and development time.

```
#include <QObject>
#include <QtPlugin>
#include "pluginsystem/export.h"

class Core;
class PluginManager;

class IPlugin: public QObject  {
    Q_OBJECT
 public:
    virtual QString getName() = 0;
    virtual void initialize(Core * core) = 0;
    virtual void initialize2(PluginManager * pm) {}
    virtual void cleanup() = 0;
};

Q_DECLARE_INTERFACE(IPlugin, "CloudClean.iplugin")
```

Listing 3.1: Plug-in interface

3.3 Core

The core system supports loading range images, rendering them in 2D or 3D, and saving range images back to file. All state manipulation is entirely implemented as plug-ins. The only file format currently supported is Leica's text based PTX files [31]. Additional formats can be supported via plug-ins. In the following sections we describe the PTX file format, how it is represented in the system, and the functionality that allows plug-ins to manipulate range images.

3.4 PTX IO

PTX is an ASCII encoded format for representing colour and monochrome range images (see Listing 3.2). While PTX files support an RGB channel, access to colour range image data was limited. The system was therefore developed with support for only XYZ coordinates and intensity values. PTX files encode coordinates and intensity values as a large grid of floats in column major order. The grid dimensions are defined in a header at the start of the file. After this, the header contains the XYZ position of the scan origin, a $3 * 3$ rotation matrix, and a $4 * 4$ transformation matrix that combines a translation to the origin and the rotation matrix. These fields can be used to keep track of registration transforms.

```
WIDTH
```

```
HEIGHT
ORIGIN_X ORIGIN_Y ORIGIN_Z
ROT_11 ROT_12 ROT_13
ROT_21 ROT_22 ROT_23
ROT_31 ROT_32 ROT_33
ROT_11 ROT_12 ROT_13 ORIGIN_X
ROT_21 ROT_22 ROT_23 ORIGIN_Y
ROT_31 ROT_32 ROT_33 ORIGIN_Z
0      0      0      1
X Y Z I
X Y Z I
X Y Z I
X Y Z I
...
```

Listing 3.2: PTX format

The a PTX file contains all scan data, including non returned points. Non returned points occur when the laser pulse is not reflected by an object. These points are recorded as $(0, 0, 0, I)$. It is not uncommon for half the contents of a PTX file to be non returned points.

Directly reading this data into the system is not memory efficient. It would also require extra processing and complexity on the part of algorithms that operate on point clouds, as the non returned points will have to be filtered ever time. Simply discarding non returned points when loading a file, will effectively destroy the grid structure. We need a point's position in the original scan grid in order to: render a 2D panoramic view, perform inexpensive normal computations, and the ability to save data back in PTX format.

When loading a PTX file in CloudClean, we therefore save the original XY coordinate of each point in the grid before discarding non returned points. The original values are saved in a subclass of the PointCloud<Point::XYZI> data structure provided by PCL [50].

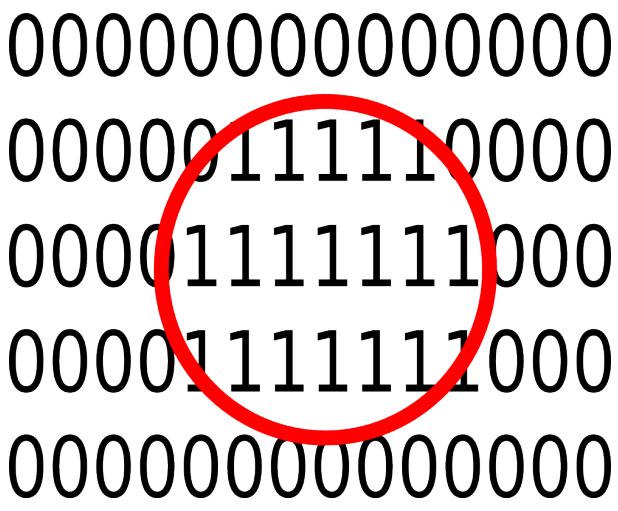
3.5 Selections and Layers

Selections are stored inside a PointCloud object as a vector of 8 bit integers. This lets us easily represent 8 overlapping selection states for each point in the cloud.

Layers need more careful consideration in order to avoid inefficiencies. Unlike selections, we would like to be able to represent more than 8 overlapping layers. When using a vector of booleans to encode layer membership, memory consumption grows linearly with each subsequent layer. Memory could be more efficiently used by representing layer membership as a linked list of point cloud indices. This would, however, compromise random access which is a problem when performing frequent lookups or removing points from layers.

Our novel implementation of layers consumes a near constant amount of memory while maintaining random access. It also allows one to perform set operations (union, difference, intersection) with very little computation. We achieve this by associating an n-bit integer label with each point in a cloud. n bits lets us create 2^n unique labels. Labelling points in this way

does not support overlapping layers. To achieve this we use separate data structure to keep track of layers that reference labels: layers are represented as sets of labels.



Layer	Label set
red	1

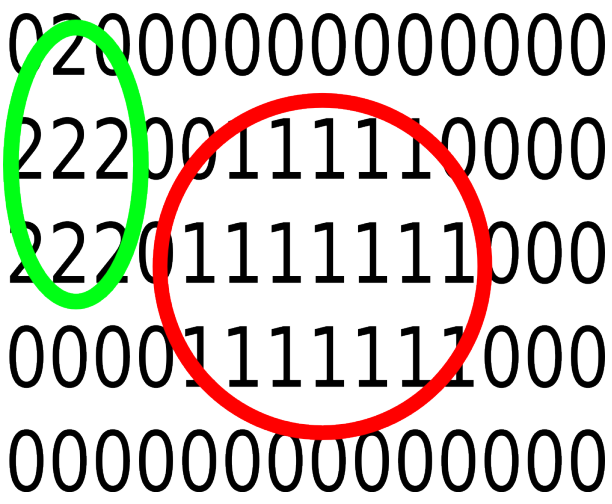
Figure 3.3: After the first layer is created the associated points are labelled with '1'. The label is associated with the red layer.

To illustrate this technique, we present a simple example which shows the labels associated with a set of points. Initially each point in the cloud is assigned a "0" label that is not associated with any layer (see Figure 3.3). To create a layer, we assign a new label to each member of the new layer, and then associate the label with the layer. In our example we assign a label of "1" to the points in the new red circle, and then associate the label "1" with the red layer (see Figure 3.3).

To add an additional non overlapping layer the same process is followed. First a new label is generated ("2" in this example), then points in the layer are assigned this label. The new label is then added to the label set of the new green layer (see Figure 3.4.

Creating overlapping layers is a little more involved. In Figure 3.5 a new blue layer is created that overlaps with the red layer. We cannot follow the same procedure as before, because relabelling already labelled points would disassociate them form their existing layer. Assigning a new integer label to the points in the blue segment would remove the overlapping points from the red layer. This problem is solved by creating two new labels instead of one. First we assign "3" to the points that do not overlap with the red layer. This label is then added to the blue layer's label set. The overlapping points are given the label "4". This label is then added to both the label set of the red and blue label. The blue and red layer now both reference the "4" label.

Set operations can be achieved by simply adding and removing labels from layers. To create a new layer containing the intersection between the red and blue layers we only need to find the intersection between the two label sets (i.e. 4), and create a new layer (see Table 3.1). Union and difference operations can be achieved similarly. Set operations with naive maps would require that each point in the point cloud be visited, and would incur a computational and memory cost of $0(n)$ where n is the point cloud size.



Layer	Label set
red	1
green	2

Figure 3.4: Creating of a second layer that does not overlap with others result in the creation of a new label which is assigned to the layer.

Layer	Label set
red	1, 4
green	2
blue	3, 4
red ∩ blue	4

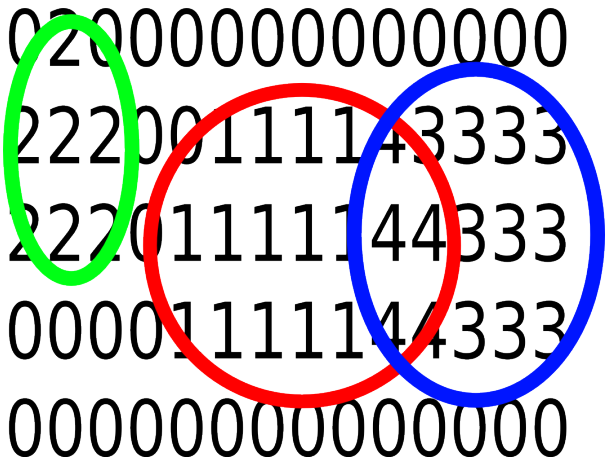
Table 3.1: Creating a new layer from the intersection of two layers is achieved by appending the common labels to a new label set.

A potential disadvantage of this technique is that number of layers is limited by the number of layer intersections. In the worst case each newly created layer overlaps with every other layer. In this case the number of bits allocated for the label will limit us to the same number of layers. A 16 bit implementation will therefore be limited to 16 layers. If no overlaps occur, a maximum of 65536 layers could be created.

3.6 Rendering

A point cloud and its associated selections and layers are stored in main memory. This data is copied to the GPU for rendering and kept in sync via dirty checking. For each execution of the render loop, the camera, point clouds, layers, and selections are synchronised with GPU data structures if they have changed.

For 3D rendering, each point cloud coordinate and its intensity value is copied to a GPU buffer. The selection state and label of each point is also copied into separate GPU buffers (see Table 3.2). The rendering pipeline uses a standard camera matrix to perform a perspective



Layer	Label set
red	1, 4
green	2
blue	3, 4

Figure 3.5: Creating a layer that overlaps with others results in overlapping points be assigned a new label which is assigned to both layers.

transformation in the vertex shader. The colour of each point is also computed in the vertex shader and used directly in the fragment shader.

During 2D rendering the XYZ coordinate are ignored and the PTX file's XY grid coordinates are uploaded and used instead. The OpenGL context for 2D and 3D rendering share the same buffers, so the same selection and label data are used. An orthogonal projection matrix is used to transform vertices onto an image plane, and vertex colours are computed in the same way as in the 3D rendering. After the vertex shader is run, a geometry shader is used to transform each point into quad on the image plane. Rendering quads instead of points ensures that there are no gaps between points. The fragment shader again simply uses the colour values passed through from earlier shaders.

In both 2D and 3D rendering pipelines the colour of each point is determined by point cloud intensity value, the selection state and the layer membership. When no selections or layers are active, the intensity value of a point is simply passed through to the fragment shader. The colour of a selected point is computed by multiplying the intensity value with the selection colour. If one or more layers are active, the average colour of all active layers is mixed with the selection colour and multiplied with the intensity value of the point. If any layer is hidden, the colour of a point set to transparent.

In order to determine the average colour of all the active layers that a point belongs to, we only need to know its label. As explained in Section 3.5, all points with the same label, have the same layer membership. A lookup table of each label's colour can thus be generated and used in our shader.

Consider Table 3.3. To generate the lookup table in Table 3.4, we determine what set of active layers each label maps to. The average colour can then be computed by averaging the

Index	X,Y,Z,I (float[4])	Label buffer (uint16)	Selection mask (uint8)
0	0.8, 1.2, 0.2, 0.9	0	10000000
1	0.7, 0.5, 0.8, 0.3	2	10000000
2	4.3, 0.5, 1.7, 0.9	2	10000000
3	0.6, 1.8, 0.1, 0.6	1	01000000
4	0.9, 0.5, 0.8, 0.5	2	01000000
5	0.1, 0.4, 3.2, 0.9	3	01000000
6	2.2, 0.5, 0.3, 0.2	5	00000000
7	1.0, 0.9, 0.1, 0.5	4	00000000
⋮	⋮	⋮	⋮

Table 3.2: Three buffers are created on the GPU to render point clouds with layers. This comprises a buffer to hold the 3D coordinates and intensity value of each point, a buffer that maps labels to points, and a buffer to represent selections.

Layer name	Colour	Active	Visible	Labels
grass	$rgba(0, 0.8, 0, 1)$	true	true	0, 2, 4
walls	$rgba(0, 0, 1, 1)$	true	true	0, 3
tree	$rgba(0, 1, 0, 1)$	false	true	2, 3
⋮	⋮	⋮	⋮	⋮

Table 3.3: Layer data

colours of layers associated with each label. Not shown in Table 3.4 is that we set the alpha channel of the label colour to 0 when a layer is hidden.

The lookup table is copied into a buffer texture that is used by the shader to find the blended label colour associated with each point. An example rendering of two intersection layers is shown in Figure 3.6. Changing layer colours or toggling a layer invisible only requires that the lookup table be recomputed and loaded to a GPU.

Using our layering scheme allows one to easily render layer intersections. Layers also allow us to visualise a reference segmentation and compare it to a selection. This is invaluable during user evaluation as it takes the guess work our of where a target objects starts and ends. We use this extensively in Section 5.5.

One caveat of our rendering implementation is that all points and associated render data are copied to the GPU. When rendering many or extremely large data sets, GPU memory may be exhausted. Larger data sets could however easily be supported by adding a pre-processing step to create a multi-resolution data structure.

3.7 Undo

To make mistakes less costly, selection and layer operations need to be reversible. The two primary design patterns used to implement undo are the *command pattern* and the *memento*

Label	Colour
0	$walls.colour$
1	$grass.colour$
3	$\frac{walls.colour+tree.colour}{2}$
4	$grass.colour$
⋮	⋮

Table 3.4: Label colour lookup table

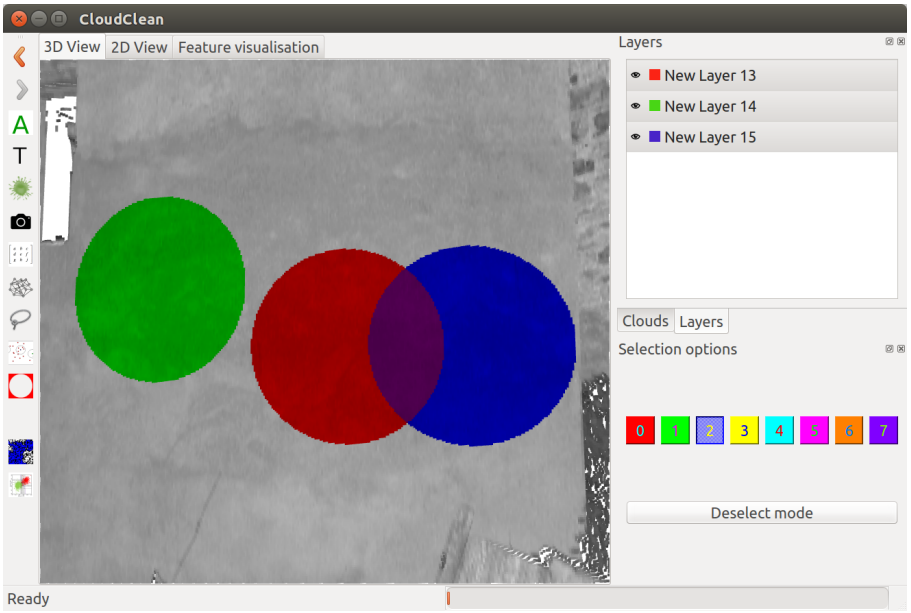


Figure 3.6: Alpha blending layers

pattern [20]. When using the memento pattern the state of the system must be saved before applying an operation. An operation can then be reverse by restoring the saved system state. When using the command pattern, only the data required for applying the new state and restoring the old state need to be saved. This data is stored on a command object that has undo and redo methods called by the system to apply and revert operations. These methods are called by a command stack when command object is pushed or popped from from it.

Using a memento pattern can quickly exhaust memory resources when dealing with large amounts of system state. Consider a brush selection tool. As the mouse cursor dragged across the screen, one would have to create a selection command for each intermediate position of the mouse in order to reflect the action in real-time. If each command object requires the selection state of all points in a range image be saved, the command history would have to be very shallow in order to avoid exhausting system memory.

The command pattern lets us keep track of changes far more efficiently. In the case of the aforementioned brush tool, we would only need to save newly selected points in a command object. Unlike the memento pattern, a series of selections could easily be kept in on the command stack without exerting memory pressure.

QT provides all the scaffolding required to implement the command pattern. The QUndoStack provides built in functionality to add undo and redo buttons that manage the command stack. The QUndoCommand class can be extended to create custom commands that are compatible with the QUndoStack. Pushing a command to the QUndoStack calls the redo method which results in the action being applied. Popping from a QUndoStack results in the undo method of the last command being invoked. QUndoCommands can also be merged. In our example of a brush tool one could create 100's of command objects in a single stroke. Undoing 100's of commands for a single stroke would be slow. Merging lets us combine multiple commands into one.

CloudClean provides 5 built in commands for manipulating selections and layers. The select command lets one select and deselect points. Two layer creation commands are available: one creates a new layer from point cloud indices and the second one creates a new layer from existing labels. Finally, layer update and layer delete command are available. Only manipulating layers and selections via commands, ensures that all actions are reversible.

3.8 Plug-ins

To support a basic point cloud cleaning work flow, four plug-ins were implemented. Firstly, a brush and lasso segmentation tool was implemented. These two manual tools are all a user needs to clean range images. In fact, some organisations only use Cyclone's [32] lasso tool and limit box to perform cleaning. Then we implemented a plane selection flood fill tool. Lastly, in order to save the system's layer and selection state, a project file plug-in was added.

3.8.1 Brush

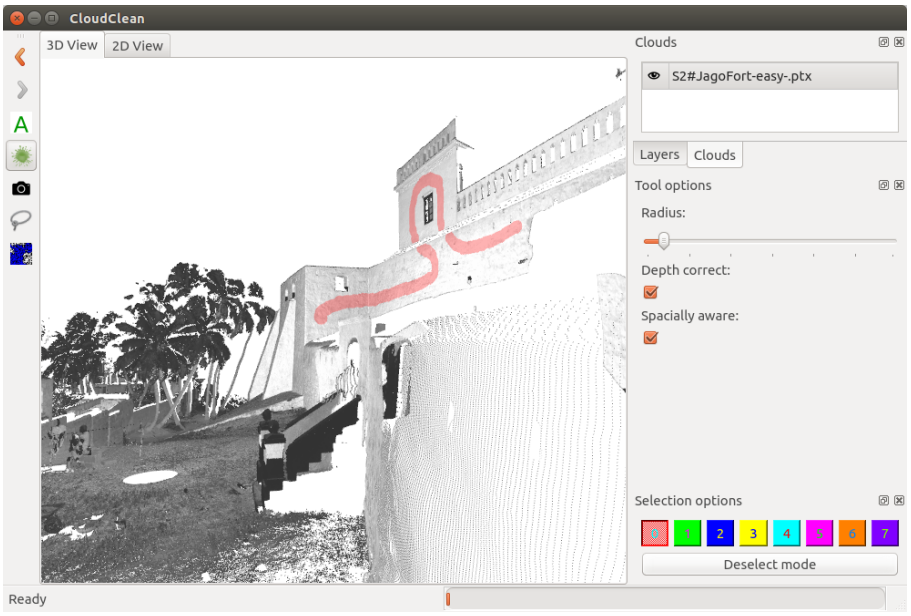


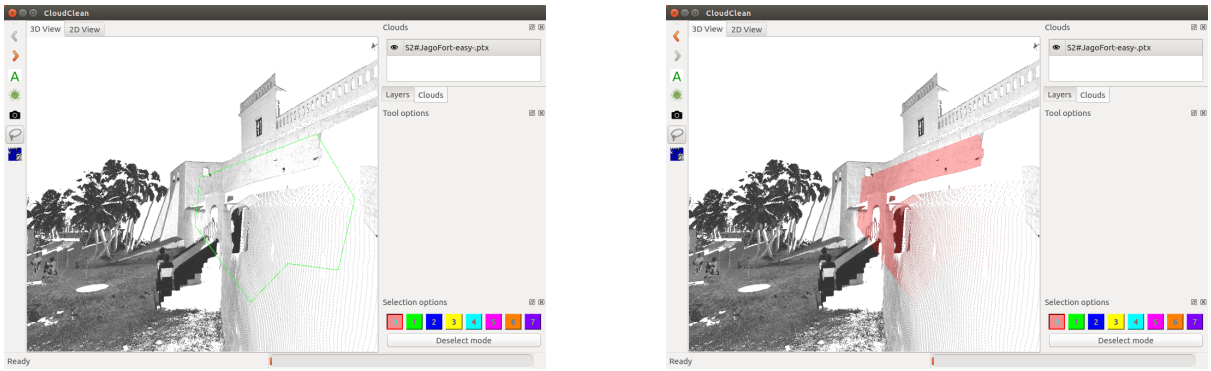
Figure 3.7: Brush tool

The brush tool allows the user to make selections by painting (see fig. 3.7). The tool can be set to deselect by holding the control key or explicitly selecting deselect mode on the side panel. The tool allows the user use any of the system's 8 selection colours, and paint with an adjustable radius.

The tool uses a point picking subroutine built into CloudClean. When activated, the point picking routine uses a special shader to render each point's primitive id instead of its colour value to an off screen frame buffer. The position of the mouse over the view port can then be used to find the primitive id in the frame buffer. Once this has been determined a kd-tree is used to perform a nearest neighbour search within the set radius of the selected point. This ensures that the selection is depth sensitive. The returned points are then marked for selection by constructing a new selection command and pushing it to the undo stack.

The painting action results in multiple overlapping points sets are selected. QT allows us to define a merge function in order to combine consecutive commands of the same type. We use this to merge all selection commands from the same stroke. This saves memory and prevents the user from having to undo each fragment of the stroke individually.

3.8.2 Lasso



(a) Lasso polygon (in green)

(b) Lasso selection

Figure 3.8: Lasso tool

The lasso tool lets a user construct a polygon on the view port, and select all points that fall within it (see Figure 3.8). The plug-in keeps track of each click position and the current position of the mouse to draw a polygon on screen. The polygon is drawn via its own shader that gets executed when CloudClean emits the plug-in draw event from the render loop. The user signals that the polygon is complete by double clicking. The points inside the polygon are selected by performing a 2D point in polygon test. This test uses the same off screen frame buffer technique as in Section 3.8.1. Points are again selected by pushing a selection command to the undo stack.

3.8.3 Plane flood fill

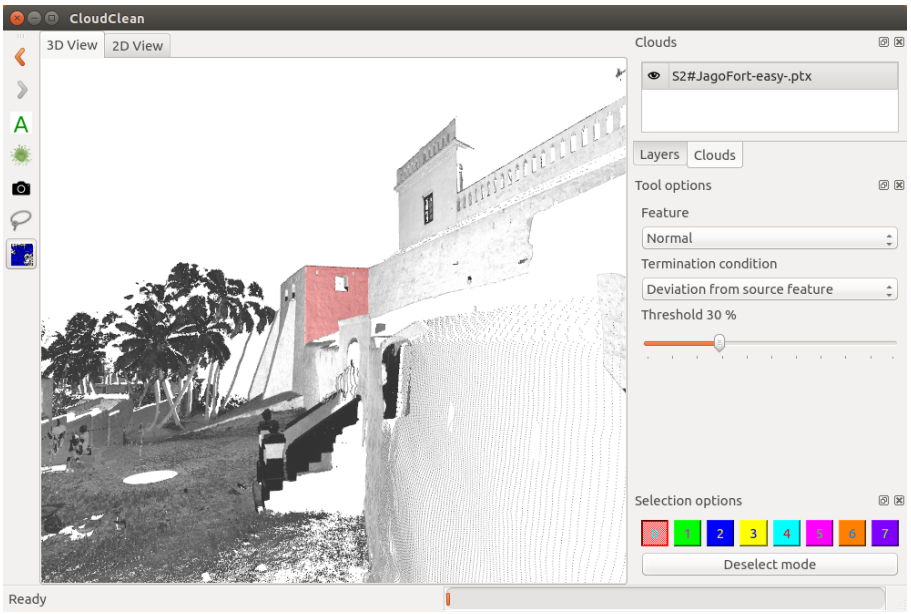


Figure 3.9: Plane flood fill tool

The plane flood fill tool allows a user to select planar regions (see Figure 3.9). As with the lasso and brush tool, the plane tool looks up the point under the clicked mouse position via the CloudClean point picker. The region growing process starts by looking up the K nearest neighbours of the initial point. Because planar regions should have points with normals in the same direction, the angular distance between the normal of the initial point and its neighbour can be used to determine whether to include it in the selection. We allow the user to set custom threshold value expressed in the angular distance between two normals. This can be be used to tolerate imperfect planes where some normals might deviate from the original direction. This process is repeated for every newly added point. The selected region is created via an undo-able command.

3.8.4 Project

An important part of running segmentation user experiments, is to have the ability to compare a participant's result with a ground truth reference segmentation. The project plug-in lets us serialise selections and layers representing the ground truth and initial selection states to file. It is also useful for saving intermediate work.

```
1 CloudCleanproject
2 VERSION
3 HAD_SELECTIONS
4 NUMBER_OF_CLOUDS
5 CLOUD_FILE_PATH_1
6 CLOUD_FILE_PATH_2
7 CLOUD_FILE_1_LABEL_COUNT
```

```
1 2 3 4 5 5 6 6 9 9 // LABELS
1 2 4 8 8 8 9 9 9 9 // SELECTION MASK
CLOUD_FILE_2_LABEL_COUNT
667 2 26237 4724 27 // LABELS
1 2 4 8 8 8 9 9 9 9 // SELECTION MASK
LAYER_COUNT
LAYER_1_NAME
LAYER_1_VISIBILITY
LAYER_1_COLOUR
LAYER_1_LABEL_COUNT
3 42 242 324 342 423 // LABEL SET
LAYER_2_NAME
LAYER_2_VISIBILITY
LAYER_2_COLOUR
LAYER_2_LABEL_COUNT
4 8 9 // LABEL SET
EOF
```

Listing 3.3: CloudClean Project format

The project plug-in serialises system state to an ASCII based .ccp (CloudClean Project) file format (see Listing 3.3). The file starts with the magic string "CloudCleanproject" followed by a version number and a boolean that indicates whether selections are present in the file. The number of cloud files referenced within the file is then stated, followed by the file paths. The next line starts by stating the number of labels to follow which is then followed by as many integer labels. If present, the same number of selection mask values follow. The same format is then repeated for each cloud file referenced. The final part of the file lists all the project layers. The number of layers is first stated where after the first layers, name, visibility state, colour and number of labels is stated. This is followed by the list of labels in the layer. The same format is repeated for the remaining layers where after the file ends.

3.9 Navigation

In Chapter 2 it was argued that a first person perspective (FPP) camera provide a navigation style that is analogous to how people navigate physical environments. As such it is arguably the most appropriate navigation mode for virtual environments. FPP navigation, however, requires one to make a trade of between rotational freedom and potential disorientation. If the camera can be rotated freely, it is easy for a person to end up in non-upright position, which can be disorientating. The reason for this is that unlike the real world, a virtual environment doesn't provide one with the vestibular information to determine an "up" direction. Recovering to an upright position can be difficult as users have trouble achieving a 3D orientation using 2D inputs.

Most 3D games, sensibly, trade rotational freedom for reduced disorientation. This trade-off is implemented by limiting the camera's pitch range to 180° and applying rotation around the world x and y axis rather than around to the camera's current orientation. This prevents the camera from going upside down and from rolling. Because games usually expect the player to remain upright, this trade-off is fine. In point cloud editing, however, it is not uncommon for

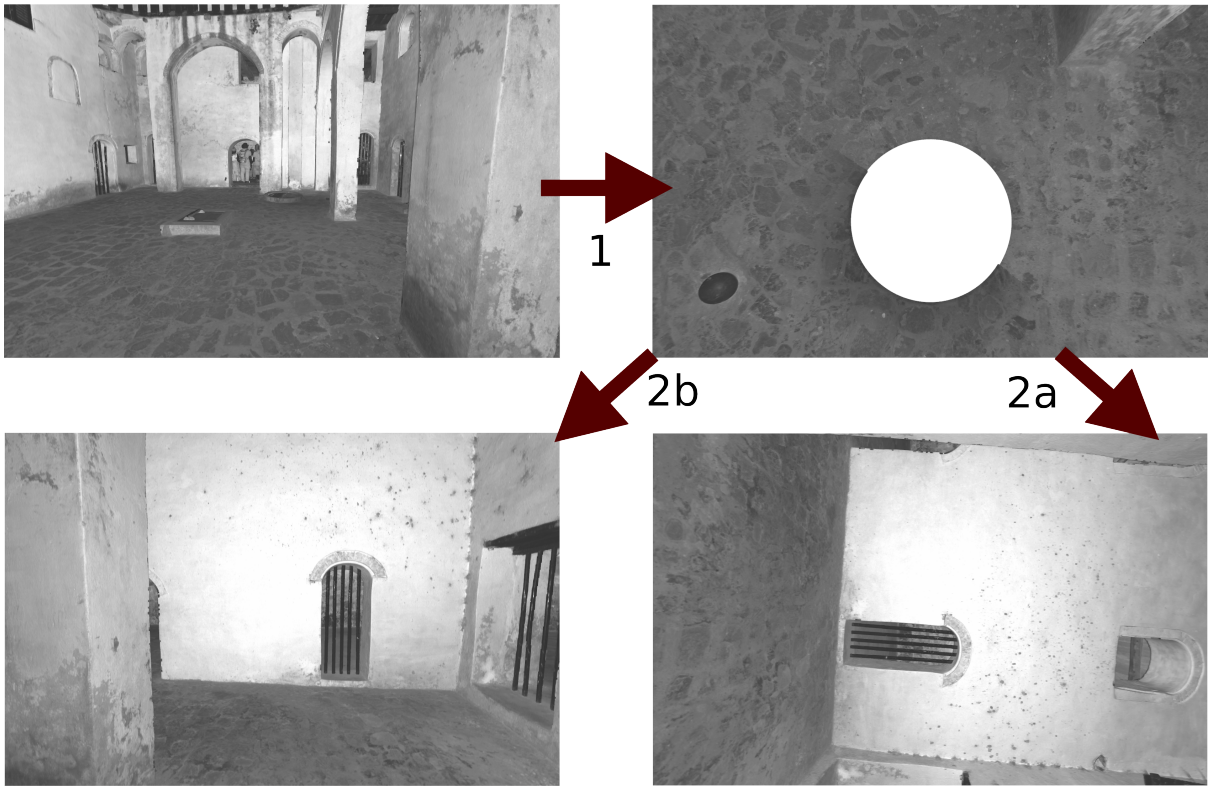


Figure 3.10: One way to produce a roll state relative to the world with with a FPP camera is by pitching down (1) and then yawing right (2a). Our roll corrected FPP camera avoids this problem (2b)

a person to manipulate points under his or her current position. Limiting the pitch prevents a person from quickly bringing an area into view when the most direct camera movement would result in an upside down orientation. Limiting the yaw around the world y axis also prevents the camera from looking left and right when pointing down.

Our variation of the FPP camera mitigates the problem of accidental disorientating reference frames without restricting camera rotations. This technique allows a person to freely rotate the camera around the yaw and pitch axis of the local reference frame. A user can therefore look around naturally without being restricted by invisible barriers. In allowing this level of freedom, the camera can become rolled relative to the world axis. If the camera is upright, pitching the camera 90° down and then yawing 90° to the right, would orient the camera on its side relative to the ground (see Figure 3.10). This undesirable state can be disorientating.

Being rolled, relative to the ground, is not always undesirable though. Some degree of roll is required in order to give a user the freedom to look down without limitations. A roll state only becomes undesirable when a user starts to look straight ahead along the horizon. When looking straight ahead along the horizon, a user is accustomed to being upright.

The camera's pitch, relative to the ground, can be used to formulate a heuristic to determine when a roll state is undesirable: as the pitch angle approaches 0, roll becomes disorientating. Roll states are less disorientating when the pitch is far from 0, as the user's intention is likely to look up or down. This heuristic can be used to determine when to unroll the camera.

In our system we apply a small correctional roll rotation to the camera with every interaction if the heuristic indicates that a roll state is undesirable. With successive interactions the effect of the correction slowly nudges the camera back upright. To control the speed of the roll-correction a damping coefficient d is used. We disable roll-correction when the pitch is not in the -45° to 45° range. (See Equation 3.1 and Figure 3.11)

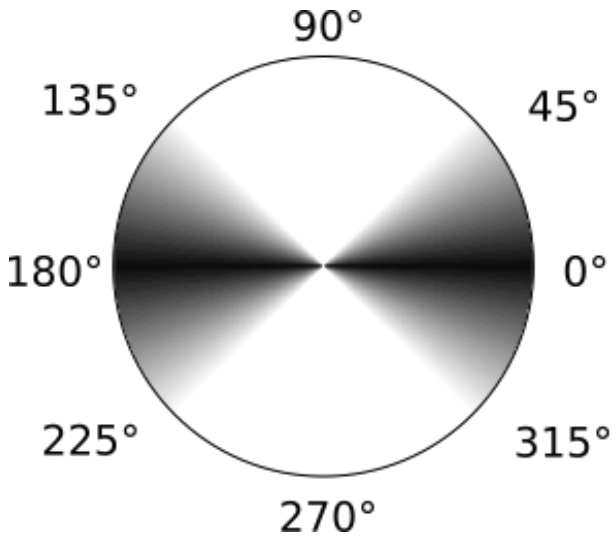


Figure 3.11: Visualisation of roll-correction factor: camera angles within darker regions have more roll-correction applied

$$correctionFactor(\theta) = \begin{cases} d(1 - |\cos(\theta)|) & : |\theta| \leq 45° \\ 0 & : |\theta| > 45° \end{cases} \tag{3.1}$$

3.9.1 User testing

To test whether roll-correction speeds up navigation, a user experiment was designed in which participants were given a navigation task. The task requires a user to navigate between two positions in a point cloud, from a disoriented state, with and without roll-correction enabled. Our hypothesis is that if roll-correction helps orientate the user, a user should be able to complete the task more quickly with roll-correction turned on.

Design

The independent variable in this experiment is whether roll-correction was on during the navigation task. The dependent variable is the time the user take to complete the task.

In this experiment we use a repeated measures design. In a repeated measures design, each participant take part in both the experimental and control conditions. The primary advantage of this design is that a participant can serve as his or her own control. This reduces the effect of individual differences in task speed. Secondly, fewer participants are needed as all partake in both conditions. A repeated measures design, is however, also exposed to order effects. The order in which a participant performs a task may affect his or her performance. A participant

Figure 3.12: Overview of the navigation environment

may get better over time (learning effect), or he/she may get tired and exhibit diminished performance.

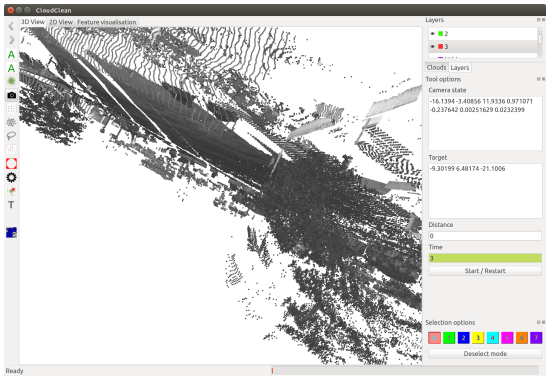
We control for order effects in two ways. Firstly, counterbalancing is used to alternate the order in which the user is exposed to each condition. Secondly, a priming task is provided to familiarise the user with the system and the task. This serves to reduce learning effects.

Participants

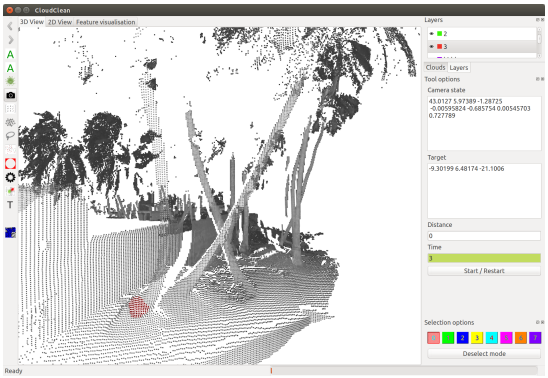
After ethics approval was obtained, 19 participants with varying levels of computer experience were recruited on University notice boards. In total 13 men and 6 women were recruited and participated in a one hour experimental session. Prior to the experiment, three other participants were recruited for a pilot study. No personally identifiable information were collected. Participants were provided with an informed consent form that outlined the procedure and purpose of the experiment. Participants were informed that they could withdraw at any time without penalty.

Materials

A range image of a fort (see Figure 3.12) was used for navigation tasks. Two target areas, as seen in Figure 3.13b and Figure 3.14b, were marked by colouring the points. Users started the navigation task from the disorientated positions pictured in Figure 3.13a, and Figure 3.14a. A plug-in was created to reproduce these starting position for each test. It was also used to input the target position, start a timer when the user started moving, and stop the timer once the user was at the target position.

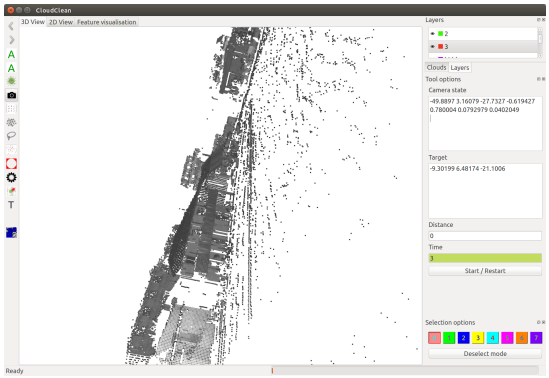


(a) Initial state

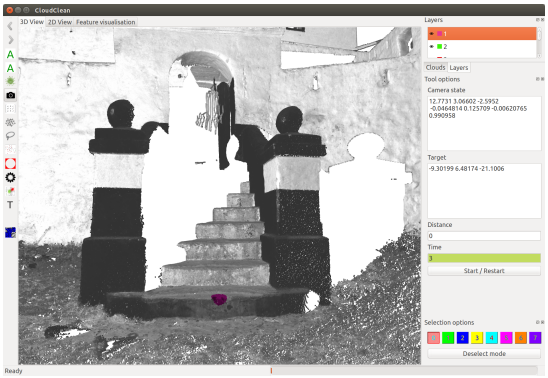


(b) Target state

Figure 3.13: Palm navigation task



(a) Initial state



(b) Target state

Figure 3.14: Stair navigation task

Procedure

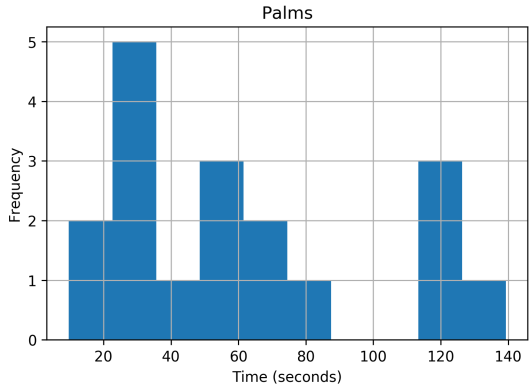
Before starting the experiment, controls were explained to participants and they were given a priming task. For this task, users were asked to navigate to each target position with and without roll-correction. Afterwards, users were allowed to move around the point-cloud environment until comfortable with the controls and familiar with the environment.

Users were asked to perform each navigation task 3 times under each condition (18 total trials). A user would thus be asked to navigate to the palm trees with roll-correction on 3 times, before performing the same navigation task 3 times with roll-correction turned off. The navigation tasks to the stairs followed the same procedure. Every second participant were subjected to the roll-corrected condition after first completing the non roll-corrected condition.

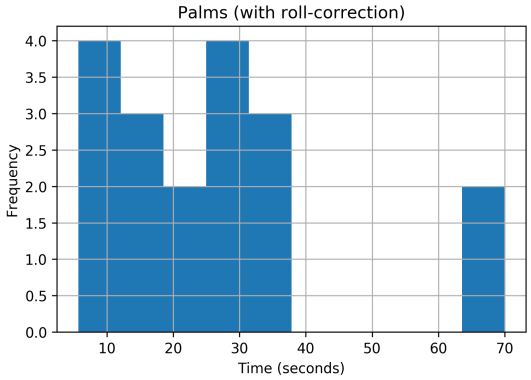
The average duration of for each task and condition was recorded using the plug-in.

Results

Users performed both tasks significantly faster with roll-correction toggled on. In the first task (navigate to palm trees) the mean time without roll-correction was 60.77 seconds with a

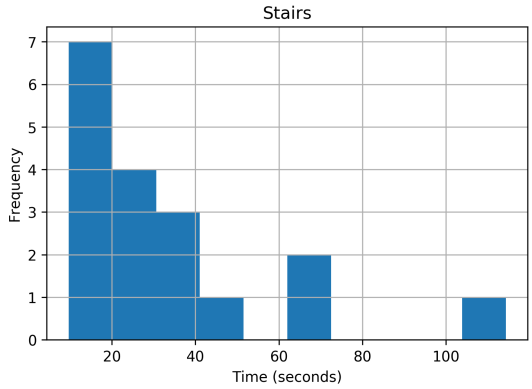


(a) Without roll-correction

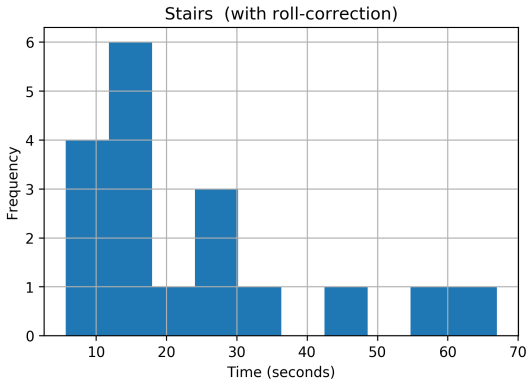


(b) With roll-correction

Figure 3.15: Distribution of palm navigation task results



(a) Without roll-correction



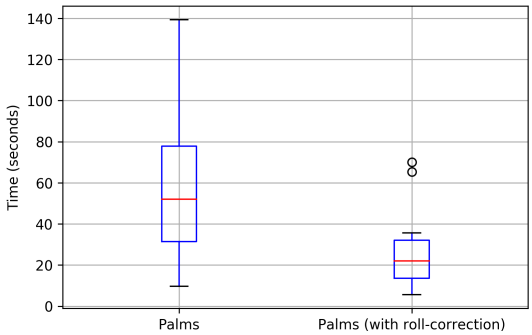
(b) With roll-correction

Figure 3.16: Distribution of stairs navigation task results

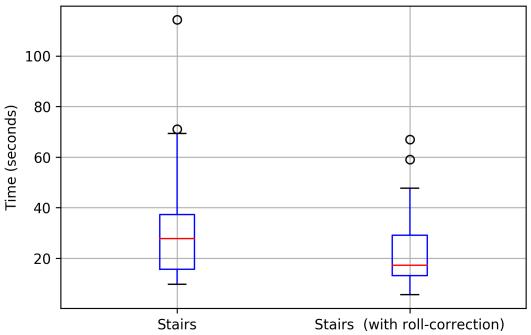
standard deviation of 39.74 seconds. With roll-correction the mean time was 25.65 seconds with a standard deviation of 18.11 seconds. This amounts to a 35.13 second (57.8%) improvement that is significant ($p < 0.05$) with p-value of 0.0002. In the second task (navigate to stairs) the mean time for the control condition was 34.24 seconds with a standard deviation of 26.60 seconds. Roll-correction reduced the mean time to 24.02 seconds with a standard deviation of 17.73 seconds. That is a 10.22 second (29.8%) reduction that is significant ($p < 0.05$) with a p-value of 0.0069. Figure 3.15 and Figure 3.16 shows (and a Shapiro-Wilk test [55] confirms) that the results of neither experiment are normally distributed. A non parametric Wilcoxon signed-rank test [70] was therefore used to compare paired samples.

The distribution of the palm navigation task (Figure 3.15) appear to be somewhat multi-modal. As our sample is from a general student population, two cohorts likely correspond users that have experienced with virtual 3D environment and those that do not. On the tail end of the distribution some outliers are found. The same pattern can be seen in the second experiment (Figure 3.16). The reduced number of outliers could be explained by practise effects, as users may have gained experience from the first experiment.

In Figure 3.17a it can be seen that the inter quartile range of the roll-corrected condition is much smaller than for the non roll-corrected condition. This can be explained by users being



(a) Palms navigation task



(b) Stairs navigation task

Figure 3.17: Box and whiskers plot of results

more likely to get stuck in disorientated states. Due to the assistance provided by the roll-correction, this is less likely to happen in the roll-corrected condition, resulting in less variance. Highly inexperienced users are however challenged by the whole experience, thus roll-correction has a less pronounced effect on their navigation times. This effect manifests as two outliers in the roll-corrected condition. In the second experiment (Figure 3.17b), inter quartile range of the non roll-corrected state is reduced. This can again be attributed to experience. The outliers however still appear to skew the results in both conditions.

Removing outliers does not diminish effect size or significance. Our results show that roll correction is effective in reducing the navigation time of experienced and inexperienced users alike.

Full results are available in Appendix A.

3.10 Conclusion

In this chapter we described the design and implementation of CloudClean, our point cloud segmentation framework. The main aim was to support the development and testing of new semi-automated segmentation tools. In line with this goal, it provides a 3D workspace that supports loading multiple range images, creating of layers and selections, project files, and a set of basic segmentation tools provided through a plug-in framework. This environment supports a basic cleaning work flow that can be enhanced though the plug-in system. This allows the impact of new segmentation algorithms (added as plug-ins) to be benchmarked against an existing set of tools via user evaluations.

During the design we also addressed three point cloud cleaning time sinks. Firstly our new layer representation supports a large number of layers while consuming a near constant amount of system memory. This addresses potential slowdowns due to memory pressure on resource constrained systems. It also provides the added benefit of extremely efficient set operations. The alpha blending of layers and selections allow segmentation targets to be clearly delimited during user experiments, which removes the need participants to have good judgement.

Secondly, all selections and layer operations were implemented via a undo stack. While this is not significant in the general point cloud software ecosystem, it is the first open source point cloud segmentation enabled framework to that allows one to undo costly mistakes.

Lastly a roll-corrected camera mode was developed to give users unrestricted rotational freedom while avoiding disoriented states. We showed that, compared to a non roll-corrected version of our camera, it speeds user navigation significantly between 29.8% and 57.8%.

In the next chapter we review the literature on semi-automated segmentation. This is followed by Chapter 5 where we use CloudClean to implement and evaluate our interactive Random Forest classification tool.

Chapter 4

Semi-automated segmentation

Cleaning of terrestrial range images for heritage preservation has not been the topic of much research. As discussed in Section 2.3.3, *semi-automated* segmentation tools in existing systems are mostly aimed at specific target classes. Due to the unpredictability of unwanted points, what a tool was designed to segment rarely matches the class of the target points. It is thus difficult to reliably achieve useful levels of accuracy with specialised semi-automated tools. Users therefore often revert back to manual tools.

In Chapter 5, our goal is to create a segmentation tool that can learn from examples, and be used interactively. In this chapter we review relevant literature on general purpose semi-automated segmentation.

4.1 2D segmentation

Figure 4.1: Intelligent scissors

Figure 4.2: Magic wand

Efficient semi-automated segmentation of images has been of great importance in 2D image editing. As such, tools for this task have improved greatly over time. Early efforts to create semi-automated image segmentation tools include Intelligent scissors, Magic wand, and more recently, Grab cut.

Intelligent scissors [37] help a user trace object boundaries by snapping a contour to areas of high contrast. While very useful, the tool often fails when the contrast between the object and background is low. Given the amount of user interaction required, this can often lead to situations where the use of this tool is more effort than it is worth.

Magic wand[43] requires less work. This intelligent flood fill uses simple colour statistics in order grow a region within a set threshold. The most common failure case occurs when the background texture of the image is not distinct from the foreground. When this happens it may require many attempts before a user finds an optimal threshold value.

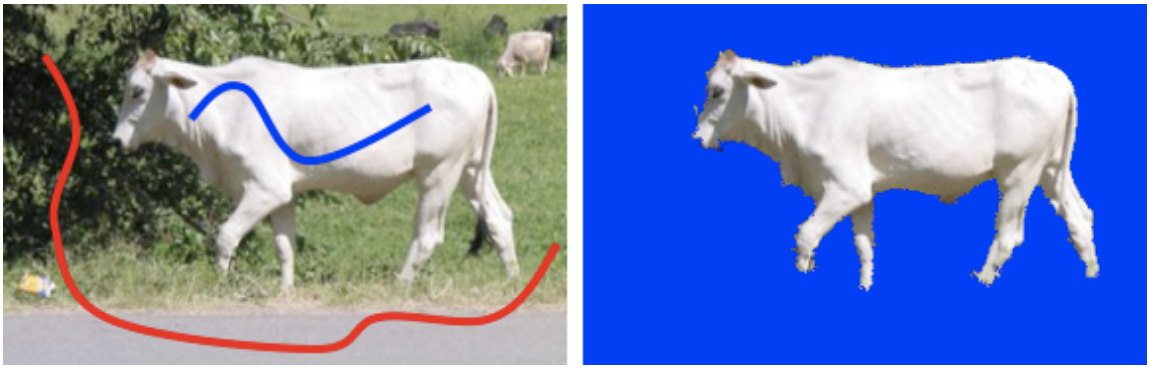


Figure 4.3: Interactive graph cut segmentation

Grab Cut [46] distinguishes itself from earlier tools in that very little user interaction is needed to produce exceptional binary segmentations. The interface requires the user to coarsely label parts of the image. The algorithm then classifies each pixel as being part of a foreground object or part of the background, based on the use provided input (see Figure 4.3).

Grab Cut exploits the observation that pixel values in images tend to vary smoothly. Neighbouring pixels are therefore likely to be part of the same object ("guilty by association"). The probability of two pixels having the same label can be modelled by a Markov Random Field (MRF). Rather than independently segmenting each pixel, contextual information can be used to improve segmentation accuracy. Grab Cut uses this probabilistic framework to encourage neighbouring pixels to have the same label. The core idea is to start by determining the likelihood of each pixel's label independently. This likelihood is obtained from a Gaussian Mixture Model (GMM) generated from user inputs. When the likelihood of a pixel and its neighbour having the same label is considered, the most probable labelling of the image can be determined. Grab cut achieves this by formulating the segmentation problem as an energy minimisation that corresponds to the maximum marginal likelihood labelling of the pixels.

The maximum marginal likelihood labelling is determined via a graph cut. A graph is constructed that connects each pixel to its 4 neighbours (see Figure 4.4). The weights of connecting edges are determined by how similar a pixel's brightness value is to its neighbour. Edges between pixels with similar intensity values are assigned high weights and edges between dissimilar intensity values are assigned low weights. Each pixel node is also connected to a virtual foreground and background node. The edge weights between each pixel node and the

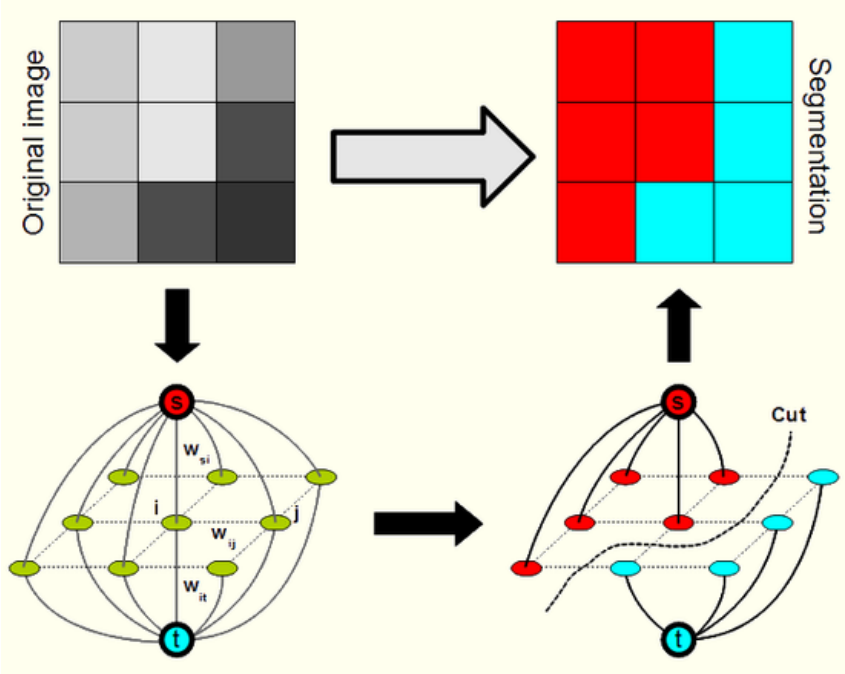


Figure 4.4: Graph cut optimisation [7]

foreground or background node, is determined by the likelihood that the pixel's colour will occur in the foreground or background colour distribution (as per GMM). If the pixel is likely to be in the foreground or background the edge weight to the background or foreground node will be high. The goal of the segmentation algorithm is to split the graph into two sub graphs by removing edges in such a way that the background and foreground nodes are in separate graphs, and the accumulated weight of the removed edges is minimised. This ensures that a good trade-off is achieved between prediction of the GMM and goal of having a smooth segmentation. The binary version of this problem can be solved exactly and efficiently via a max-flow/min-cut graph partitioning algorithm such as the Ford-Fulkerson [16] method, which has $O(VE^2)$ time complexity, given V vertices and E edges. As such the graph cut algorithm can be use interactively for typical image sizes.

4.2 Point cloud segmentation

Golovinskiy et al. [21] adapts graph cut segmentation as part of a completely automated system for segmenting point-clouds captured in city streets. The first step in this approach is to locate clusters of points, using various heuristics, that represent potential object locations in the point cloud. The segmentation step constructs a graph by connecting the 3 nearest neighbours of each point in the cluster. Spatial contiguity is encouraged by setting the edge weights to the inverse distance between points. Distant neighbours are thus more likely to be separated. Each point is also connected via an edge to a virtual foreground and background node. Edge weights are set as a function of a point's horizontal distance to the cluster origin. Foreground weights increase as points locations approach the origin while background weights decrease. The foreground and background edge weight calculations require that an approximate object radius is known. Several graph-cut iterations may be required to estimate an appropriate radius.

Because the distance between points determine pairwise edge weights, we found that it is extremely challenging to optimally parametrise this graph cut on non-uniform range images. We were unable to achieve even moderate success. The min-cut optimisation invariably led to partitions where the foreground node was attached to only a small number of nodes close to the object centre.

As evident from our experience implementing Golovinskiy et al. [21], modelling the relationship between points is difficult, especially for irregularly sampled range images. Other work that specifically focuses on segmenting non-uniform range images include Anguelov et al. [2]. Anguelov et al. [2] models relationships between neighbouring points in range images by using a combination of the distance between points and the dot product of their normals. Adjacent point normals on most surfaces are likely to point in the same direction. Using normals in addition to distance to model the degree of relatedness between points reduces the likelihood of biased segmentations. Anguelov et al. [2] determined the prior probability distribution of points labels in the range image via a Support Vector Machine (SVM) [6]. Anguelov et al. [2] demonstrates segmentation results of up to 93% accuracy using this graph cut implementation.

Munoz et al.'s [39] work completely abandons modelling point relationships via heuristics. Firstly, range images are reduced to related clusters. Functional gradient boosting is then used to learn the relationships between points instead of formulating it by hand. When targeting wires, tree-trunks, vegetation, ground planes, and facades in their urban Oakland dataset, up to 97.2% accuracy could be achieved.

Segmenting objects by modelling relationships between points has its limitations. The underlying assumption that neighbouring points are likely to have the same class does not hold in all cases. In heritage data one may want to target non-coherent points such as scanner noise or fog. Using a graphical framework in this case, not only adds overhead but is likely for lead to poor results. Weinmann et al. [68] demonstrates that graphical models are not essential to achieve high accuracy range image classification. In using SVM with point features that capture local structure, Weinmann et al. [68] was able to outperform Munoz et al. [39] on their own Oakland dataset.

4.3 Machine learning

Other popular supervised learning methods that have been used to classify range images include Naive Bayes [27], Nearest Neighbours (NN) [45, 29], K Nearest Neighbours (k-NN) [45], and Random Forest [8]. Weinmann et al. [68] found that SVM outperformed NN, k-NN, and Naive Bayes in a range image classification task. SVM is known to perform exceptional well on a wide variety of classification tasks [15], but may require prior knowledge about input features and can be difficult to tune [24]. Without proper scaling, features with wider numeric ranges can dominate those with narrower ranges [24]. An appropriate kernel also needs to be selected and kernel hyper-parameters need to be tuned for good performance [24].

Deep neural nets is another machine learning technique worth considering. One advantage of neural nets are that careful feature selection is not required as the method can automatically generate hierarchies of features during training, not unlike the human visual system. Advances in neural networks [54] have allowed researchers to solve increasingly difficult problems by utilising increasingly larger amounts of data. Unfortunately, due of the large amount computation and

labelled training data required to achieve good results, deep neural nets is not appropriate for interactive training and classification tasks.

Random Forest [8] is an classifier with compelling performance and accuracy characteristics. Santner et al. [52] reports that it takes 1 second to train on 2000 samples on recent hardware. Training and evaluation scales linearly across multiple cores which is useful when performance becomes a problem. It is also the only method to outperform SVM in Fernández-Delgado et al.'s [15] evaluation of 179 classifiers on 121 data sets, and has been demonstrated to work well on range images [56]. Other benefits of Random Forest are that the method is tolerant of noise, does not require input normalisation, and has fewer tunable parameters compared to SVM and many other classifiers. Random Forest is also inherently multi-class. This adds to its efficiency as multiple binary classifiers do not have to be trained and evaluated [52].

Given these characteristics, Random Forest appears to be the best candidate for a general purpose interactive range image classification tool.

4.4 Random Forest

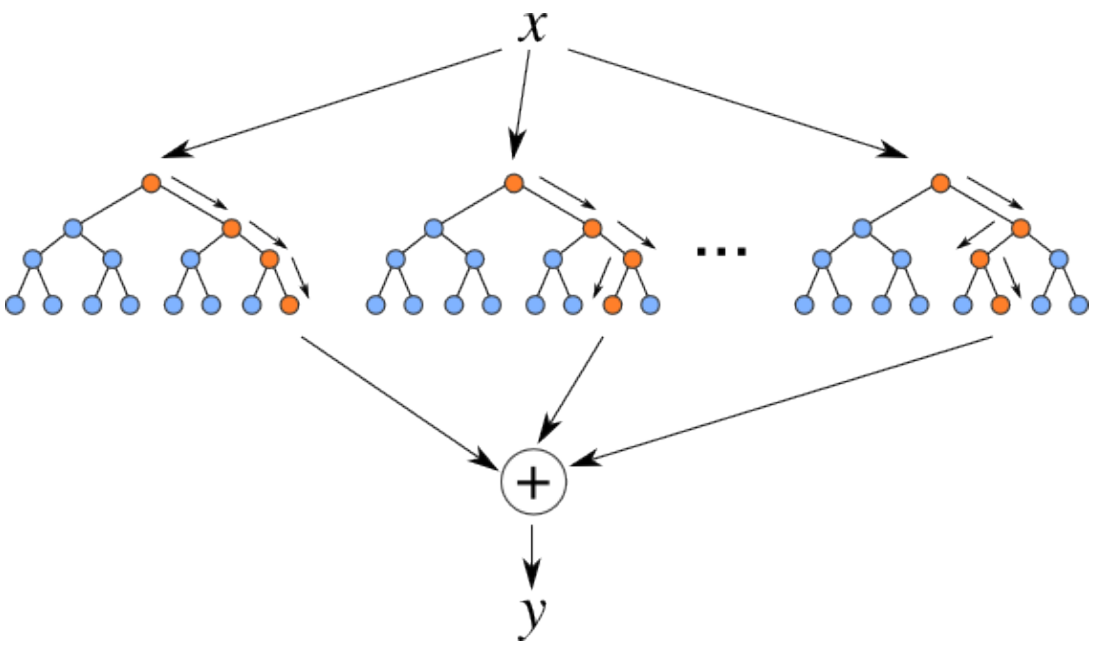


Figure 4.5: Random Forest [1]

Random Forest [8] is an ensemble classifier that uses a multiple randomised decision trees (see Figure 4.5). A decision tree is a binary tree used for classifying data. When classifying a data point represented by a feature vector, each non-leaf node runs a test to determines whether to branch left or right. A test usually compares one feature to a threshold value such as (>0.5). As a tree is traversed different features are considered until the probability of each class is given by a leaf node.

[1]Source: http://www.ee.iitkgp.ac.in/ispschool/mvlss2016/

Given a set of feature vector and class pairs, a decision tree can be trained by finding a feature and threshold that best splits the data set by class at each node in the tree (determined by information gain or gini index). A perfect set is achieved when the tree can perfectly discriminate between the different classes. When a perfect set is achieved we stop growing the tree. When a perfect set is not reached, the tree is grown until the maximum depth is reached. [8]

In a Random Forest, each tree is grown with a random training set that is sampled with replacement (samples can be selected more than once). At each node in the tree a set of features are also sampled randomly with replacement. When using Random Forest we need to decide on the number of trees we want to grow and the maximum depth of each tree. Deep decision trees tend to over-fit data. In a Random Forest over-fitting is managed by having many trees vote on the outcome of a classification. More trees typically improve classification accuracy, with diminishing returns.

We use Christian et al.'s [12] on-line Random Forest implementation. It has the ability to learn incrementally on streaming data, consumes less memory, and is more tolerant of noise than Breiman's [8] original off-line implementation. This comes at a small trade-off in accuracy for small sample sets. With larger samples Christian et al.'s [12] implementation converges to the accuracy of off-line Random Forest.

In Christian et al.'s [12] implementation, a Poisson process is used to determine whether a sample is used in a particular tree. For each node in a tree, random tests are generated. Instead of determining the test by randomly selecting a candidate features and finding an optimal threshold, as proposed by Breiman [8], features and thresholds are determined randomly. After a fixed number samples have been seen by a given node, the best test is selected and a split in the tree created. The number of samples before a split is an extra hyper-parameter that needs to be considered.

The Random Forest in Christian et al.'s [12] implementation can also be updated over time by discarding trees that have high out of bag error and replacing them with new trees. This is useful for updating the forest in real time as the user labels more data. Unfortunately there was no scope to explore this feature in this work.

4.5 Features

Because Random Forest is less prone to over-fit training data, less caution is needed when selecting an optimal number of features. However, the overall amount of computation is still affected by how many features are used. Interactivity can be compromised by excessive feature computation. We thus aim to use the minimal set of preferably low cost features that can discriminate well between object classes likely to be found in heritage scans.

Features can be grouped into geometric and non geometric types. Colour [72] and LIDAR intensity [59] returns are popular non geometric features that require no additional computation. Our data sets contain no colour information but LIDAR intensity returns have been shown to differentiate well between different surface types [59]. Geometric features encode the relationship between points. The surface normal is one such feature that can be estimated by fitting a plane to 3 or more points. Geometric features have varying computational cost depending on the

size of the neighbourhood that needs to be considered. Features used in previous work include height [63, 67, 10] (above the ground plane), surface curvature [25, 45], density [68], verticality [14], and spin images [28, 21, 2].

The most widely used features are derived from a principal component analysis on the covariance matrix for a local neighbourhood of points [42, 17, 10, 22, 68]. The resulting eigenvalues and eigenvectors describe the distribution of points along 3 principal axis. Because planar data is distributed along the two primary axis and linear data is distributed along one axis, one can use eigenvalues to classify lines, planes and other geometry. West et al. [69] formulated 6 derivative measures to describe point cloud data. These features are Linearity (Equation 4.1), Planarity (Equation 4.2), Sphericity (Equation 4.3), Anisotropy (Equation 4.4), Eigenentropy (Equation 4.5) and Omnivariance (Equation 4.6). Rusu [48] proposes an eigenvalue derived estimate of the change in curvature around a query point (Equation 4.7). PCL [50] implements another curvature estimate where the normals in a neighbourhood are projected onto plane represented by the normal at the query point. A principal components analysis on the projected normals results in a curvature estimate along the two primary axis of the plane.

$$Linearity = \frac{\lambda_1 - \lambda_2}{\lambda_1} \tag{4.1}$$

$$Planarity = \frac{\lambda_2 - \lambda_3}{\lambda_1} \tag{4.2}$$

$$Sphericity = \frac{\lambda_3}{\lambda_1} \tag{4.3}$$

$$Anisotropy = \frac{\lambda_1 - \lambda_3}{\lambda_1} \tag{4.4}$$

$$Eigenentrophy = \sum_{i=1}^{3} \lambda_i \lg \lambda_i \tag{4.5}$$

$$Ominvariance = \sqrt[3]{\prod_{i=1}^{3} \lambda_i} \tag{4.6}$$

$$Change\ in\ curvature = \frac{\lambda_3}{\lambda_1 + \lambda_2 + \lambda_3} \tag{4.7}$$

Rusu [48] recognises that features such as curvature and normals are tolerant of noise, but don't encode sufficient information for computing correspondences without excessive false positives. Point Feature Histograms (PFH) is an attempt to design a more descriptive feature suitable for such applications. It encodes the relationships between points and their normals across multiple scales and represents this in a multidimensional feature vector. The original algorithm runs in $O(nk^2)$ where k is the neighbourhood size and n is the point cloud size. Fast Point Feature Histograms (FPFH) [49] reduces this to $O(nk)$. The algorithm is shown to discriminates well between different geometric shapes in low noise indoor environments. The test datasets used in this work are, however, filtered for outliers and re-sampled to avoid the effects of noisy non-uniformly sampled data. Filtering is impractical for our use case as the filtered points still need to be assigned a class.

Spin images [28] produce 2 dimensional histograms that are also aimed at the identification of objects or correspondences. It is important that the neighbourhood over which spin images

are computed be appropriate for size of object to be targeted. The high dimensionality of spin images necessitates the use of dimensionality reduction techniques. Anguelov et al. [2] uses PCA to reduce two $5 * 10$ spin image features to 45 principal components. This is a $O(n^3)$ computation in the number of features, and makes spin images ($n = 50$) rather expensive to reduce.

4.6 Feature selection

Selecting the optimal set of features and appropriate neighbourhoods sizes can be challenging. Guyon and Elisseeff [23] identifies 3 approaches to feature selection namely filtering, wrapping, and embedding.

Filter methods involves the evaluation features in isolation without the use of a classifier. Using this approach requires one to define a function to score the performance of a feature for a given training set. This approach allows one to easily rank features and select the best ones independent of what classifier is used. A problem with filtering is that features can be weak when used in isolation but strong when combined with others.

Wrapper methods acknowledges complimentary features and thus evaluate subsets of features by measuring classifier performance. Finding the optimal subset of features for a given classifier can lead to a combinatorial explosion. To avoid this exponential increase in complexity, two greedy search strategies can be followed, namely: forward selection and backward elimination. In forward selection, features are added one by one while classification accuracy is monitored. If the addition of a feature improves the classifier accuracy, it is kept. Backward elimination follows the same process, but features are removed if their absence increases performance. These strategies can also help avoid over fitting by reducing the number of features. The disadvantage is that once a feature is removed or added, it‘s presence or absence in the feature set is not reconsidered. This may result in a locally optimal solution.

Embedded methods perform feature selection as part of the training process of some types of classifiers such as decision trees. An embedded method requires that all potential features be computed, even when they go unused. The overhead of unused feature computation is problematic in interactive applications.

4.7 Summary

In this chapter we considered the problem of interactive segmentation from user provided examples. The painting interface used grab cut is a very simple way to input examples. The exceptional results produced by graphical modelling local relationships, lead us to explore similar applications in the 3D domain

Graphically representing a segmentation problem in range images presents two problems. Firstly, the non-uniform distribution of points in range images makes it difficult to model relationships between neighbouring points. Using the distance between points as a similarity measure leads to biased segmentations. Factoring in the difference of point normals in the edge weight computation reduces this bias. The best results are however obtained by learning

the pairwise weights. The second problem is that fog and random scanner noise violates the coherence assumption of the graphical model. Targeting non-coherent artefacts is therefore likely to results in poor quality segmentations. Weinmann et al. [68] shows that using SVM and selecting features that encode local structure can outperform segmentation algorithms based on graphical models, without depend a coherence assumption.

While SVM is an excellent classifier, Random Forest has key advantages over it and other supervised learning algorithms. It is fast, scales linearly over multiple cores, exhibits best in class accuracy [15], and is easy to understand. Given these characteristics, Random Forest appears to be the most appropriate candidate for an interactive example based segmentation tool.

We also reviewed a number of popular features from the literature. While Random Forest tends not to over-fit training data when many features are used, more features does come at a computational cost, which can compromise an iterative work flow.

In the next chapter we systematically determine an optimal set of features and parameters to use in a Random Forest based segmentation tool based. This is followed by a user evaluation of the tool using the final parametrisation.

Chapter 5

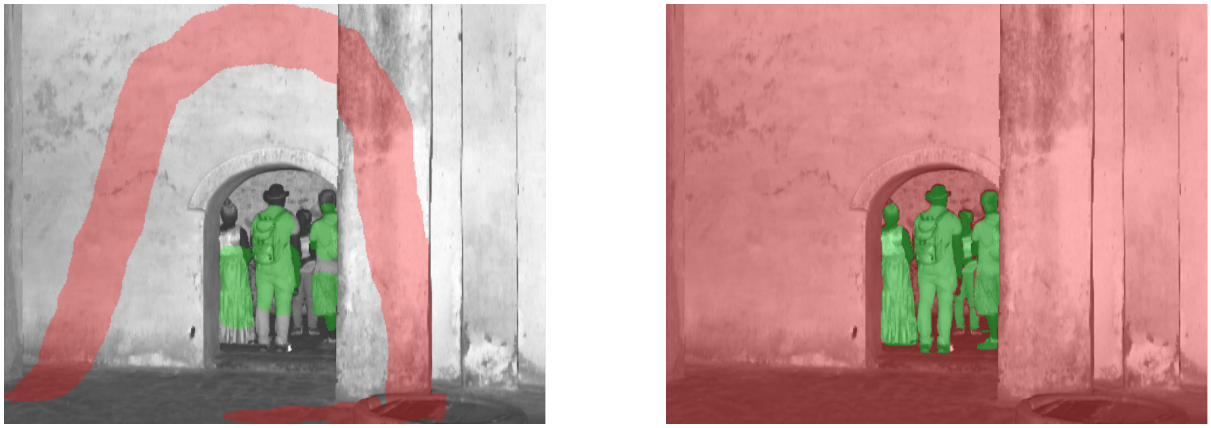
Interactive Random Forest classification

In this chapter we design a semi-automated segmentation tool that harnesses Random Forest to interactively learn new object classes from partial user labellings, provided via CloudClean selections. We aim keep training and classification time low enough such that the overall user segmentation time with the use of this tool is less than a manual work flow.

Our first goal is to find an optimal set of features for use in a Random Forest classifier. We start by preparing a challenging set of reference segmentations from heritage sites. Forward selection is then used to systematically perform feature selection using this test data. We then tune Random Forest hyper-parameters and consider the accuracy-performance trade-off when down sampling range images. Our second goal is to show that the tool reduces the overall segmentation time. We evaluate this in a user experiment at the end of the chapter.

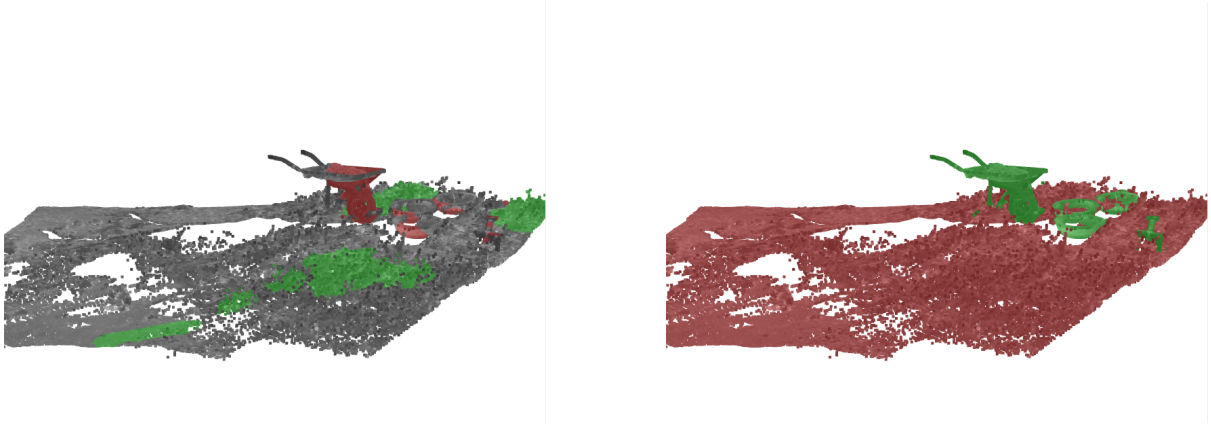
5.1 Feature selection

During feature selection, we are interested in segmentation accuracy, as well as algorithm run time. As accuracy increases we can tolerate longer processing times because more work is being performed. At some point, however, a user's attention will be lost and he or she may not notice task has completed. Nielsen [40] states that after 1 second of waiting a user's flow of thought is interrupted, and after 10 seconds his or her attention will be lost, and he/or she will want to perform other tasks while waiting. As such, we want to keep the computation time under 10 seconds. We are however aware that performance varies between systems, and that our laptop test system (Intel Core i7-6500U 2.50GHz CPU with 12GB of memory) has fairly modest compute power compared to higher end desktop hardware. As such we prioritise accuracy over performance during the feature selection process, and attempt to optimise after the fact.



(a) Training labels. (b) Reference labels.

Figure 5.1: People and Facade.

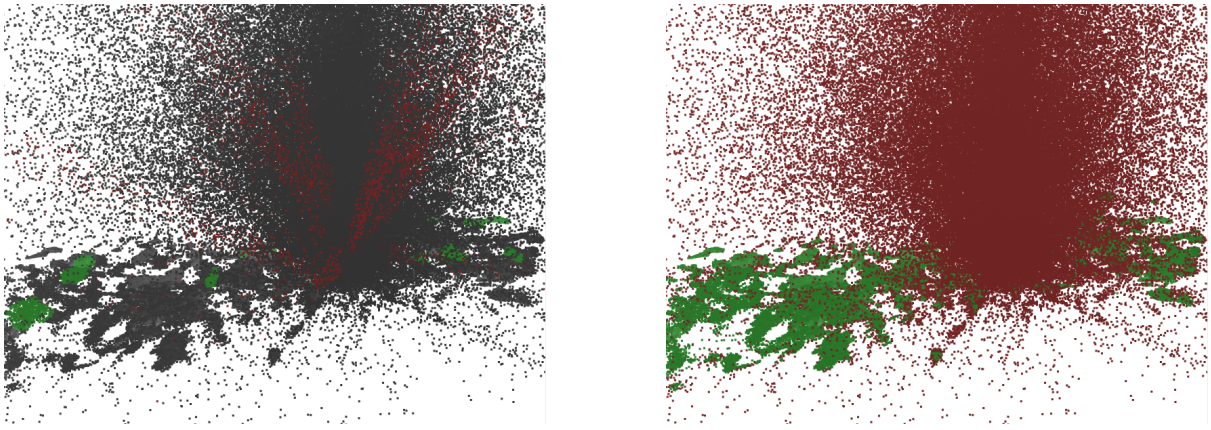


(a) Training labels. (b) Reference labels.

Figure 5.2: Tools and ground

(a) Training labels. (b) Reference labels.

Figure 5.3: Brick wall and tree



(a) Training labels.

(b) Reference labels.

Figure 5.4: Fog

5.1.1 Test data

	fog	people	tools	tree
Total points	5699065	9759091	5462705	7416177
Visible	5699065	179338	206553	581656
Target	15344	6237	25078	26854
Background	875434	38583	47432	107571
Initial F-score	0.116112	0.761875	0.776021	0.605797

Table 5.1: Scan statistics

Feature selection presents us with a dilemma. Our aim is to produce a classifier that can interactively learn previously unseen object classes. However, in order to perform feature evaluation, we need to know what kind of objects a user might need to segment. For the purpose of this evaluation we use scans of four types of unwanted points frequently found in heritage scans, namely: people (Figure 5.1), equipment (Figure 5.2), vegetation (Figure 5.3), and fog (Chapter 4). In the screen shots of our test data, we show the initial coarse user labelling on the left, and the ground truth reference segmentation on the right.

The first 3 scans also represent 3 degrees segmentation difficulty. The people in the archway (Figure 5.1), does not require much work to segment using a brush tool. The equipment in the grass requires more effort to isolate, and the vegetation (Figure 5.3) is most difficult to segment.

In the first three scans, we cropped the area of interest, as it is common for users of software packages like Cyclone [32] to isolate a subregion before using a segmentation tool. In Cyclone [32], this can be achieved with a limit box that restricts the area that is affected by the tool. In CloudClean we achieve the same effect by: selecting a region with the lasso tool, inverting the selection to create a new layer, and then hiding the newly created layer. Hiding points in this way also reduces the number of points that need to be classified, which reduces the algorithm run time.

Unwanted points are, however, not always isolated. We therefore also include an uncropped scan that contains widely dispersed points created by fog (Figure 5.4). This scan not only a

contains larger number of points compared to other scans, but represents a class of non-coherent artefacts that can not be adequately modelled using the graphical frameworks described in Chapter 4.

Table 5.1 shows the total number of points in each scan, the visible points used, the total number of points in the foreground and background labelling, as well as the F-score of these initial labellings relative to the reference segmentation. We use the accuracy measurements as a baseline for interpreting classifier performance in subsequent sections.

In the test data it can be seen that the coarse labelling of background points, sometimes have an order of magnitude more samples than the foreground. A problem with user provided labellings is that the number of samples of each class can be skewed. A classifier trained on a skewed sample, may produce a biased classification. We ensure that an equal number of samples from each class is used during training. We do this by sampling only as many points as is in the smallest class for each class.

5.1.2 Features

In our feature evaluation we consider: X, Y, Z and LIDAR intensity values, point normals, principal curvatures [50], eigenvalues, linearity (Equation 4.1), planarity (Equation 4.2), sphericity (Equation 4.3), anisotropy (Equation 4.4), eigenentrophy (Equation 4.5) omnivariance (Equation 4.6), and curvature (Equation 4.7).

The relative height of a point above the ground has been used to discriminate between objects at different heights in previous work [63, 67, 10]. Tree crowns for instance, can be expected to be above a certain elevation. Because the ground is not always a flat plane, the ground plane usually needs to first be estimated. Given enough decision trees, X and Y coordinates may provide enough context for the Z coordinate to learn such relationships without ground plane estimation. Because we do not expect our classifier to extrapolate beyond a single scan, XYZ coordinates have the potential to be more discriminative compared to other applications.

Tatoglu and Pochiraju [59] showed that LIDAR intensity values can help to discriminate between different surface types. Normals are included as planes facing the same direction have the same normal. Normals are efficiently computed by considering neighbours in the scan grid.

Principal curvatures [50] are computed using the point normals in a neighbourhood. The size of this neighbourhood is important as it affects the amount of computation required as well as how descriptive the feature is. Noisy scans may therefore need a larger number of points in order to accurately estimate curvature. However, as the neighbourhood size increases, so does the computational cost and the risk of including unrelated nearby points which may affect accuracy. Eigenvalues and derivative features require similar consideration, as the neighbourhood size similarly affects computation and discriminative power.

5.1.3 Down sampling

Many candidate features require neighbourhood lookups. Performing a nearest neighbour search for every point in a range image can be costly. For example, using FLANN (Fast Library for Approximate Nearest Neighbours) [38] to find neighbours in a 5cm radius on a range image

containing 5.6 million points, takes about 5 minutes. Over larger neighbourhoods this problem is exacerbated.

Clustering the data set in an approach similar to Golovinskiy et al. [21] is one way reducing the amount of data that needs to be processed. Unbiased clustering is, however, a potentially challenging problem when dealing with non-uniform density of range images. The non-uniform density of range images, is in fact, a major contributor to the cost a neighbourhood search. Because objects close to the scanner are sampled extremely densely, small regions may contain more points than is necessary for classification purposes. The performance of neighbourhood queries can therefore be improved by selectively down sampling such dense regions.

PCL [50] contains a grid based down sampling routine for this purpose. The routine inserts each point onto a voxel grid where after it returns the centroid of each voxel. Because PCL preallocates memory for the grid, the minimum voxel size for large regions is limited by system memory. To accommodate small voxels without running out of memory we use an octree instead of a grid (see Algorithm Algorithm 1).

Algorithm 1 Octree based down-sampling

```
1:  function octreeDownSample(points, voxelSize)
2:      outputPoints ← []
3:      bigIdxToSmallIdx ← []
4:      octree ← newOctree(points, voxelSize)
5:      for leaf in octree.leafs do
6:          centroid ← newPoint(0, 0, 0)
7:          for idx in leafs.indices do
8:              centroid ← centroid + points[idx]
9:              bigIdxToSmallIdx[idx] ← outputPoints.length
10:         end for
11:         outputPoints.insert(centroid/leafs.indices.length)
12:     end for
13:     return outputPoints, bigIdxToSmallIdx
14: end function
```

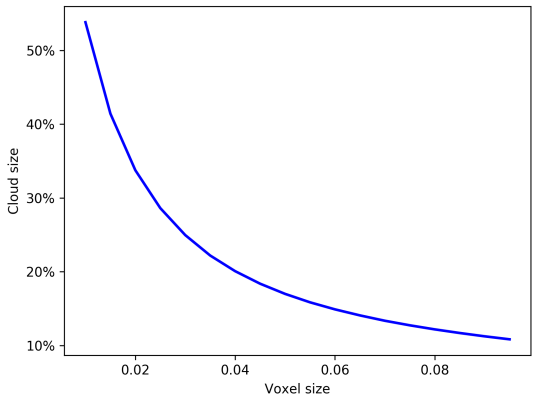


Figure 5.5: Number of points for voxel resolution

The first step in this method is to construct a new octree with a predetermined voxel resolution. After indexing the point cloud we iterate over the octree leaf nodes to compute the centroid of the points in each node. During the centroid computation, we keep track of what points, from the original cloud, were included in the centroid via a map. This map enables us to transfer labels from the down-sampled cloud back to the original cloud. Figure 5.5 shows how the voxel size of the octree affects degree of down-sampling in a range images with 5.6 million points.

To determine what a reasonable degree of down sampling might look like, it is worth considering what size neighbourhood should sufficiently characterise a surface. Intuition about what size neighbourhood is optimal can be gained by looking at Figure 5.4, Figure 5.3, Figure 5.2, and Figure 5.4. The smallest target object in these data sets is the spade in Figure 5.2. The spade has a shaft diameter of roughly 4cm. A neighbourhood size of 4cm should characterise the points on the handle surface without including nearby points associated with the ground. To ensure that the spade shaft is not reduced to a single point, a voxel resolution of less than 4cm would be needed for down-sampling. This may, however, be prohibitively expensive when attempting to maintain interactivity.

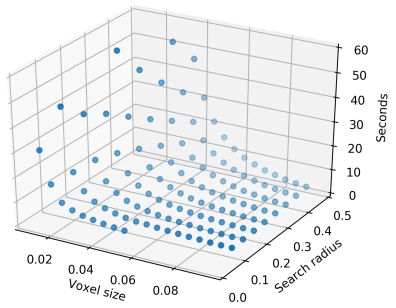


Figure 5.6: Total search time for search radius and voxel resolution

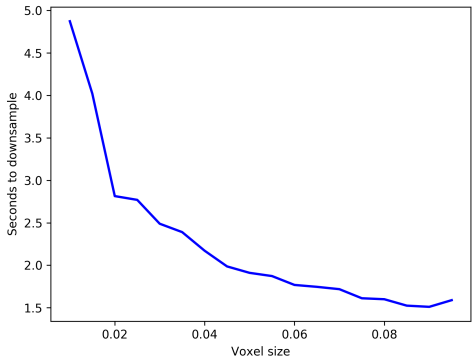


Figure 5.7: Effect of voxel resolution on downsample time

Figure 5.6 shows how the neighbourhood lookup time is reduced by down-sampling. Using a modest 0.02m voxel resolution results in reduced lookups times, but are still extremely costly for large neighbourhoods. Figure 5.5 shows how a 5.6 million point cloud is reduced by half with a 0.02m voxel resolution. Interestingly Figure 5.7 indicates that more aggressive down-sampling is computationally less expensive. This down-sampling technique should thus be very effective for reducing much higher resolution scans to manageable densities.

In Section 5.4, we further explore the relationship between classification accuracy and down sampling. Until then, we use a relatively expensive 0.02m voxel size to manage computational cost.

5.1.4 Evaluation

We employ a forward selection wrapper method [23] for feature selection. Wrapper methods use a classifier to evaluate feature performance, as opposed to an independent measure. The

disadvantage is that the selected features are not guaranteed to work well with other classifiers. When changing classifiers the feature selection process will have to be repeated. As Random Forest is the only classifier considered in this work, a wrapper method is appropriate. Our feature evaluation starts with the default Random Forest hyper parameters used by Christian et al. [12]. That is: 100 trees with a maximum depth of 10, a minimum of 100 samples seen before splitting the tree, and 20 randomly generated tests per split.

Forward selection starts by scoring and ranking individual features. We do this by classifying our test data using a Random Forest with a single feature and recording the F-score relative to the reference labelling of a scan. The highest ranked feature is then combined with the next best feature. If the combination of features improves the F-score, the feature is included for all subsequent tests. If not, the feature is discarded. Subsequent tests include features ranked in descending order of score.

5.1.5 Feature radius selection

Features parametrised over multiple neighbourhood sizes could be considered distinct features in their own right. Given the computational cost associated with performing neighbourhood lookups, including multiple parametrisations of a feature (over different radii) would result a great deal of computation that will translate into a long waiting periods for a user. In order to avoid this cost, we first find the optimal radius for principal curvatures and eigenvalue derived features.

To determine the optimal neighbourhood sizes for Eigenvalue and Curvature based features, we measure the F-score achieved by classifying our test data with each feature, over a 0.1m (Figure 5.9), 0.2m (Figure 5.10), 0.3m (Figure 5.11) and 0.4m (Figure 5.12) radius.

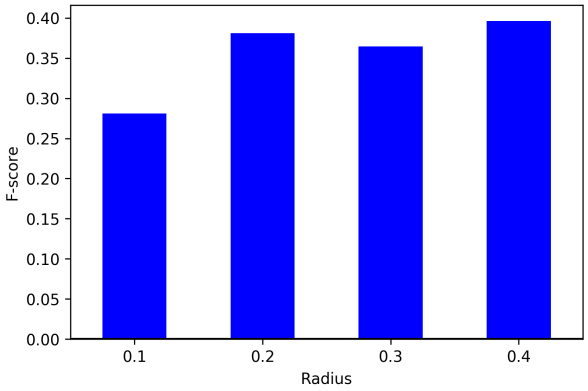


Figure 5.8: Mean feature F-score per search radius

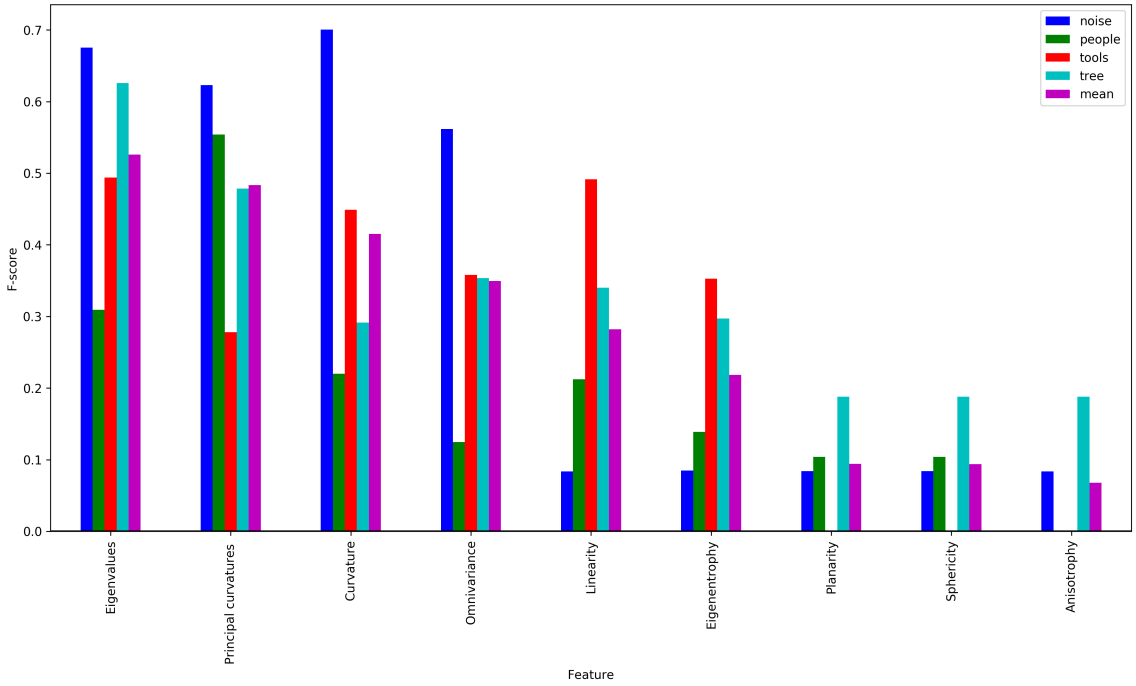


Figure 5.9: F-score per feature (radius 0.1m)

In Figure 5.9 it can be seen that, when a 0.1m radius is used, raw eigenvalues perform best, followed by principal curvatures. It is interesting to note that for the 'fog' scan, the eigenvalue derived curvature estimate performs better than principal curvatures. This could be explained by the normal estimation process. A point associated with fog is likely to have unreturned neighbouring points in the scan grid. When a point has no neighbours in the scan grid, the normal estimation fails and assigns a default value. The eigenvalue derived curvature estimate does not rely on normals and would thus not be affected by this. Omnivariance also appears to be a strong feature for classifying fog. Planarity, sphericity and anisotrophy do not work with a 0.1m radius due to numerical problems associated with a small number of neighbours.

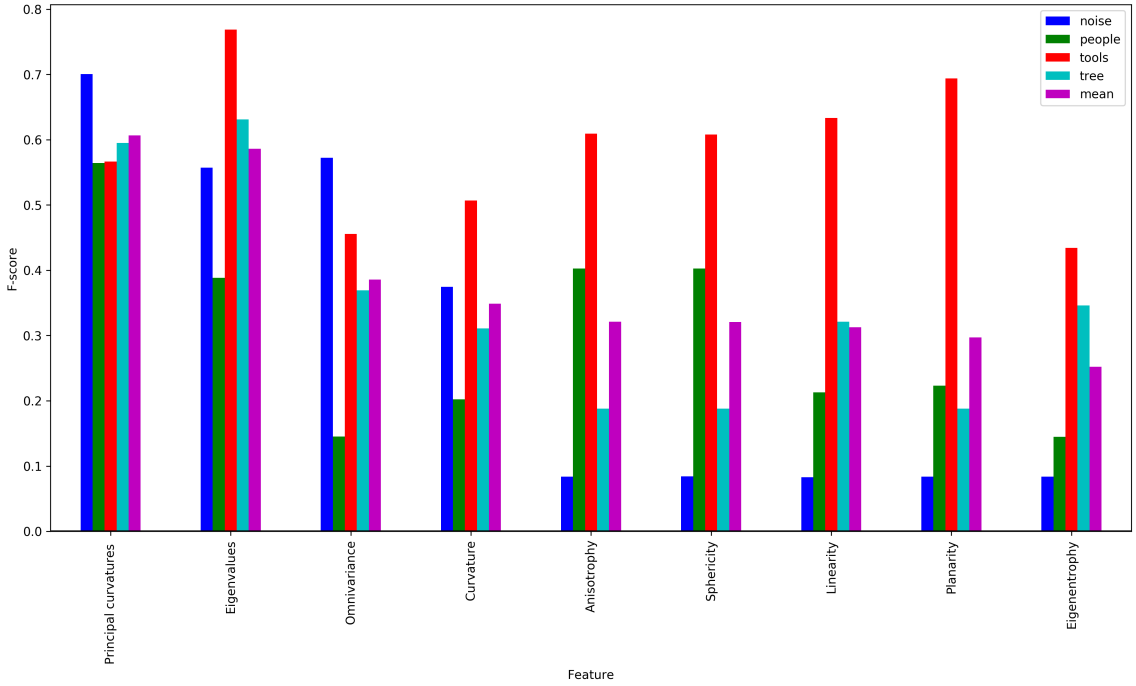


Figure 5.10: F-score per feature (radius 0.2m)

Increasing the search radius to 0.2m resolves the numerical problems experienced with planarity, sphericity and anisotrophy (see Figure 5.10). The average F-score across scans and features also improve from 0.28 to 0.38 (Figure 5.8). Principal curvatures out performs eigenvalues at a 0.2m search radius. The larger radius appears to compensate for missing normals. Principal curvature discriminates well between a target regardless of the scan being classified, while the F-score for eigenvalues and derived features vary between scans. The scan containing random tools scan achieves relatively high accuracy across all features. This could be attributed to the fact that the ground is relatively featureless at this scale and the tools are all close to the ground.

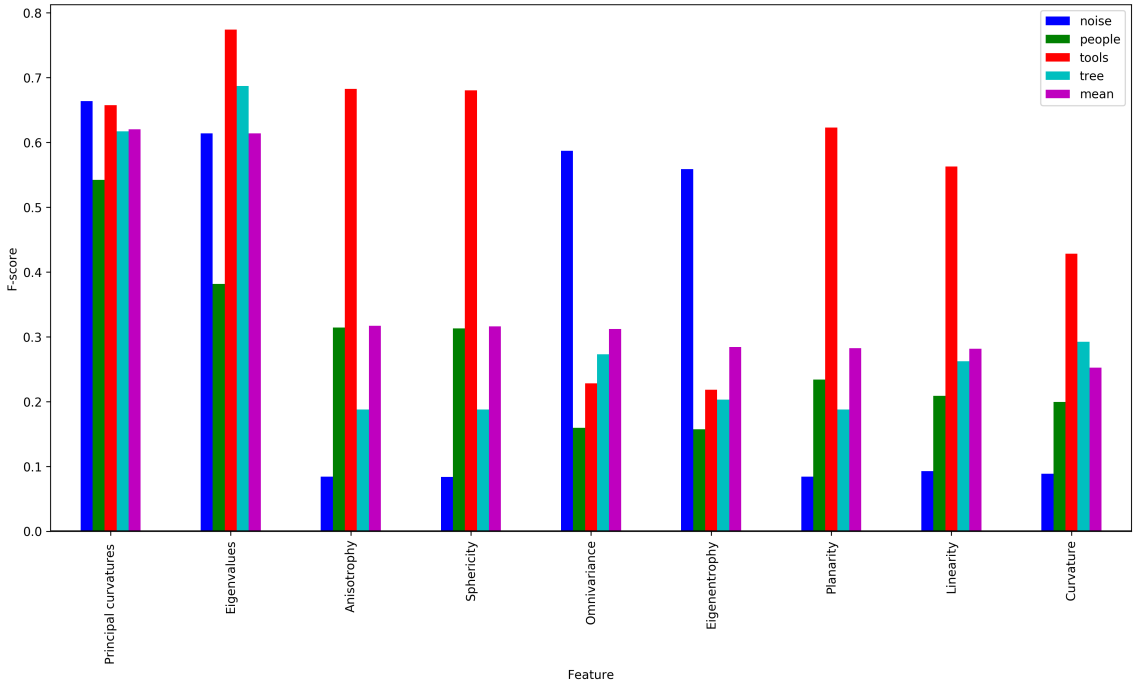


Figure 5.11: F-score per feature (radius 0.3m)

The average F-score across all features drops from 0.38 to 0.36 (see Figure 5.8) when increasing the search radius to 0.3m. This average appears to be lowered primarily by weaker features (see Figure 5.11). The average F-score for the two best features, principal components and eigenvalues, both increases by roughly 2 percent. In both cases twice the computational resources are required (see Figure 5.13). Given marginal accuracy gains in only the two best features and our focus on accelerating the cleaning process, this is not a good trade-off.

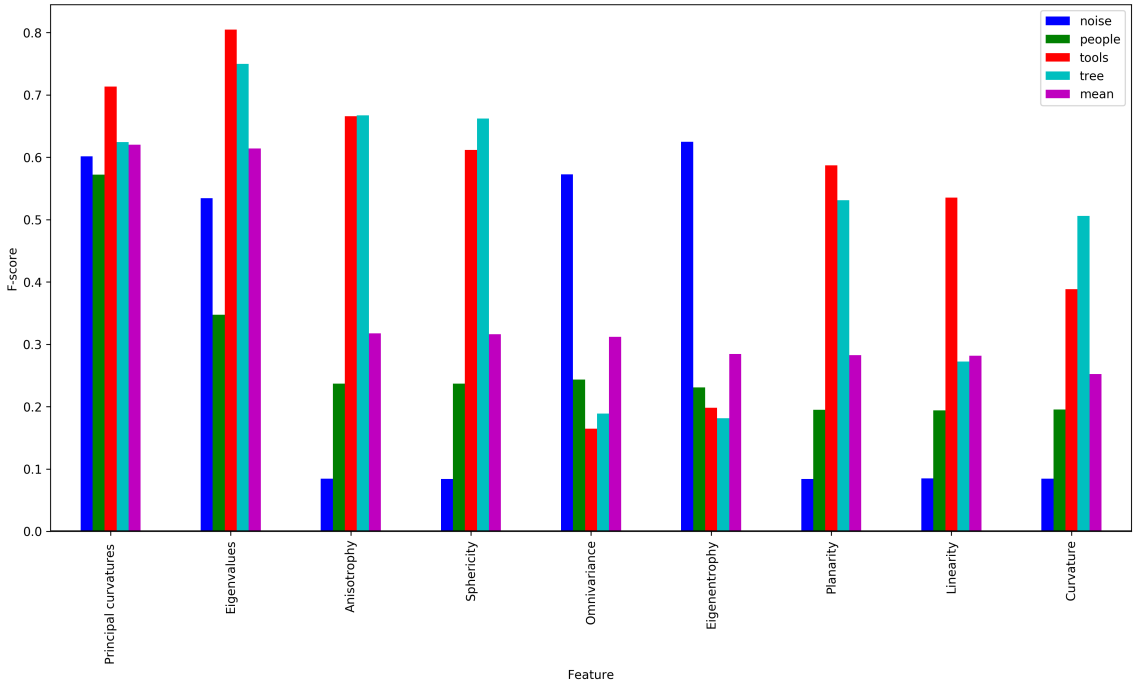


Figure 5.12: F-score per feature (radius 0.4m)

Using a 0.4m search radius improves the average F-score by just under 2% over a 0.2m search radius (see Figure 5.8). The computational cost increases 4 times for principal curvatures and doubles for eigenvalues (see Figure 5.13). As with a 0.3 radius the additional resources required outweigh the benefits.

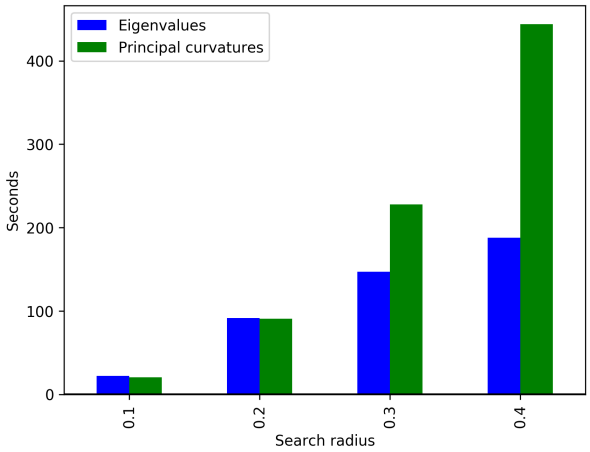


Figure 5.13: Feature computation time for search radius (5.6 million points)

5.1.6 Feature selection

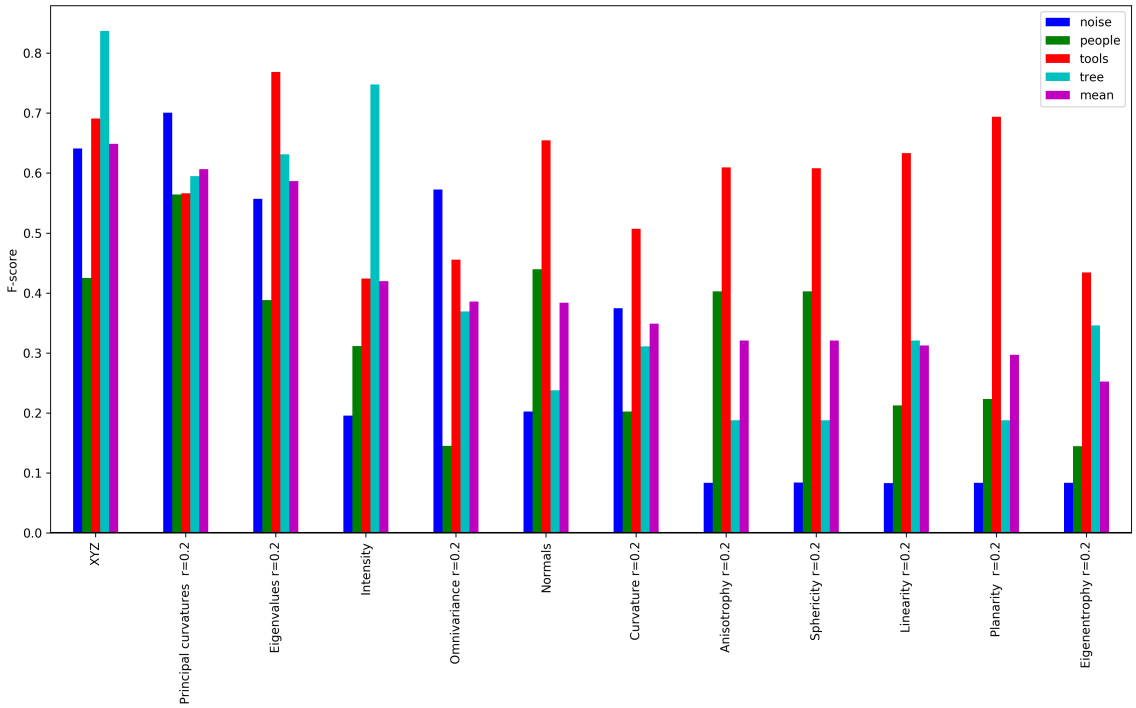


Figure 5.14: All features

After including non-parametrised features (XYZ, Intensity and Normals), our final set of ordered (mean F-score) candidate features are shown in Figure 5.14. During forward selection, each feature in Figure 5.14 is included sequentially in order of mean F-score. After including a feature, the classifier is trained and evaluated on our 4 test scans. A feature remains in feature set as long as it improves the average F-score. The result of this procedure is shown in Table 5.2.

Table 5.2 shows that the average F-score across all 4 scans is maximised when using XYZ coordinates, Principal curvatures, Eigenvalues and Intensity in our feature set. Interestingly, only a modest 1.5% increase in accuracy is gained when including Eigenvalues with XYZ coordinates and Principal curvatures. By removing eigenvalues from our final feature set, the average accuracy is 84% which represents an accuracy decrease of only 0.5%. Given the 100 seconds the eigenvalues computation requires on the largest scan (Figure 5.13), this is an acceptable trade-off.

5.2 Random Forest hyper-parameters

After feature selection, four Random Forest hyper-parameters need to be tuned: the number of decision trees used, the depth of each tree, the number of random tests generated, and the number of samples tested before selecting a test to split on. We tune only the first 3 parameters and split a tree after 100 samples, as suggested by Christian et al. [12]. We start by varying the number of trees used and keeping the depth and number of random tests the same as our initial values (10 deep, 20 tests).

	fog	people	tools	tree	mean	gain
XYZ +	0.496	0.452	0.647	0.836	0.608	0.000
XYZ + Principal curvatures	0.793	0.731	0.765	0.837	0.782	0.174
XYZ + Principal curvatures + Eigenvalues	0.796	0.758	0.812	0.821	0.797	0.015
XYZ + Principal curvatures + Eigenvalues + Intensity	0.894	0.766	0.839	0.879	0.845	0.048
XYZ + Principal curvatures + Eigenvalues + Intensity + Omnivariance	0.891	0.739	0.803	0.845	0.819	-0.025
XYZ + Principal curvatures + Eigenvalues + Intensity + Normals	0.903	0.769	0.825	0.874	0.843	-0.002
XYZ + Principal curvatures + Eigenvalues + Intensity + Curvature	0.888	0.691	0.834	0.840	0.814	-0.031
XYZ + Principal curvatures + Eigenvalues + Intensity + Anisotrophy	0.889	0.665	0.832	0.825	0.803	-0.042
XYZ + Principal curvatures + Eigenvalues + Intensity + Sphericity	0.883	0.709	0.809	0.840	0.810	-0.035
XYZ + Principal curvatures + Eigenvalues + Intensity + Linearity	0.889	0.733	0.738	0.829	0.797	-0.047
XYZ + Principal curvatures + Eigenvalues + Intensity + Planarity	0.897	0.646	0.793	0.844	0.795	-0.050
XYZ + Principal curvatures + Eigenvalues + Intensity + Eigenentrophy	0.891	0.700	0.809	0.840	0.810	-0.035

Table 5.2: Forward selection

5.2.1 Number of trees

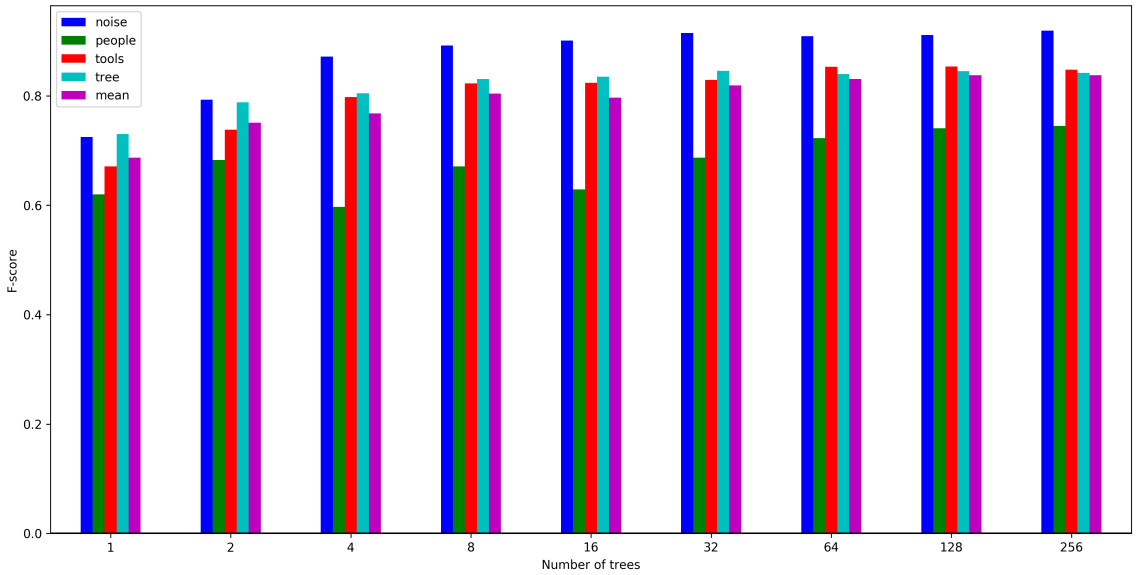


Figure 5.15: F-score for N trees of depth 10

Random Forest is designed to reduce the high variance of individual decision trees by averaging the output of each tree. Adding more trees thus has the effect of reducing over-fitting. In Figure 5.15 we see that the average F-score over the scans increases up to 83% with 64 trees, after which gains plateau. It can be seen that the F-score for the 'people' scan drops when moving from 2 to 4 trees and then again when moving from 8 to 16 trees. Results appear to stabilise after 32 trees.

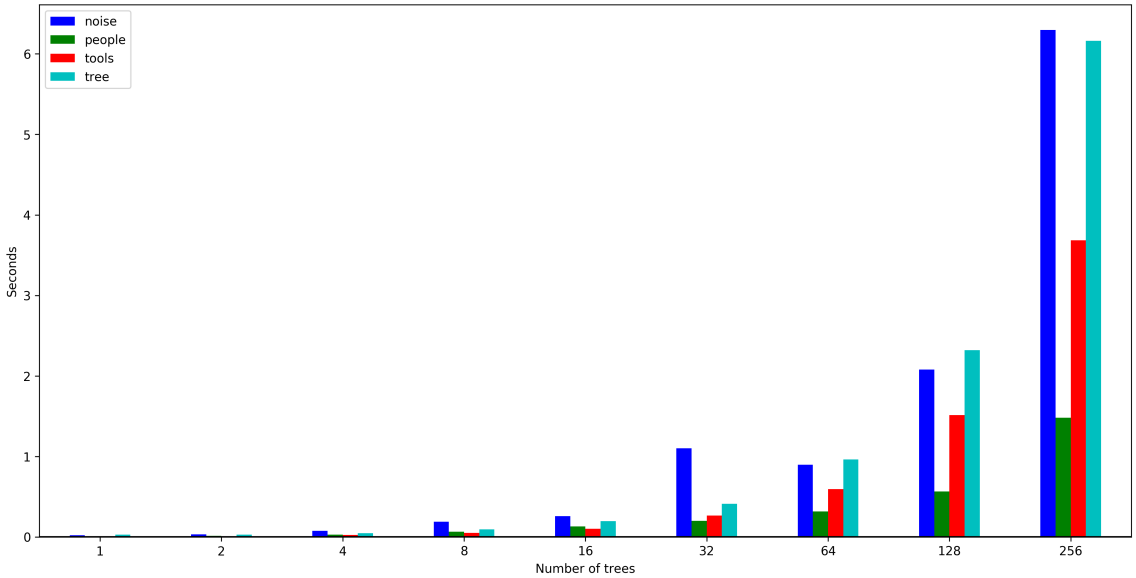


Figure 5.16: Time to train N trees of depth 10

Figure 5.16 shows that the training cost scales with the number of trees. The amount of labelled data also affects the training duration. Ideally the user should spend little time labelling data so a large amount of labelled training data is not expected. In instances where large areas are labelled, training on fewer samples could be beneficial. On our test data and hardware, training can be completed in under a second with 64 trees.

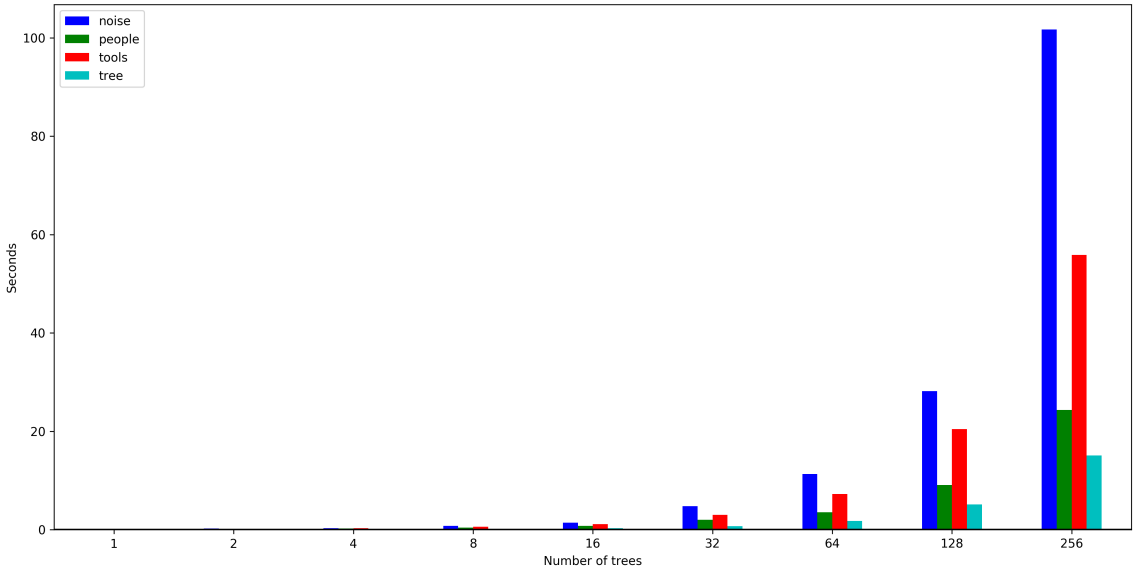


Figure 5.17: Time to classify scans with N trees of depth 10

As expected, the classification time scales with the number of trees. With 64 trees the largest scan (fog) takes around 11 seconds to classify while the largest cropped scan (tools) takes 7 seconds to classify. The smallest cropped scan takes 2 seconds to classify. These figures are cut in half when reducing the number of trees to 32. With 8 trees all scans can be classified in under a second. Given the roughly order of magnitude classification speed up associated with reducing the number of trees from 64 to 8 and the meagre 3% reduction in accuracy, 8 trees seems like a reasonable choice. The reduction in accuracy associated with moving to 16 trees, however, suggest that using 8 trees may be insufficient. The error rate of a Random Forest is increased when two trees are correlated Breiman [8]. When using only 8 trees we run a larger risk of trees being correlated by chance. We therefore use 64 trees moving forward and revisit this decision in Section 5.4.

5.2.2 Maximum tree depth

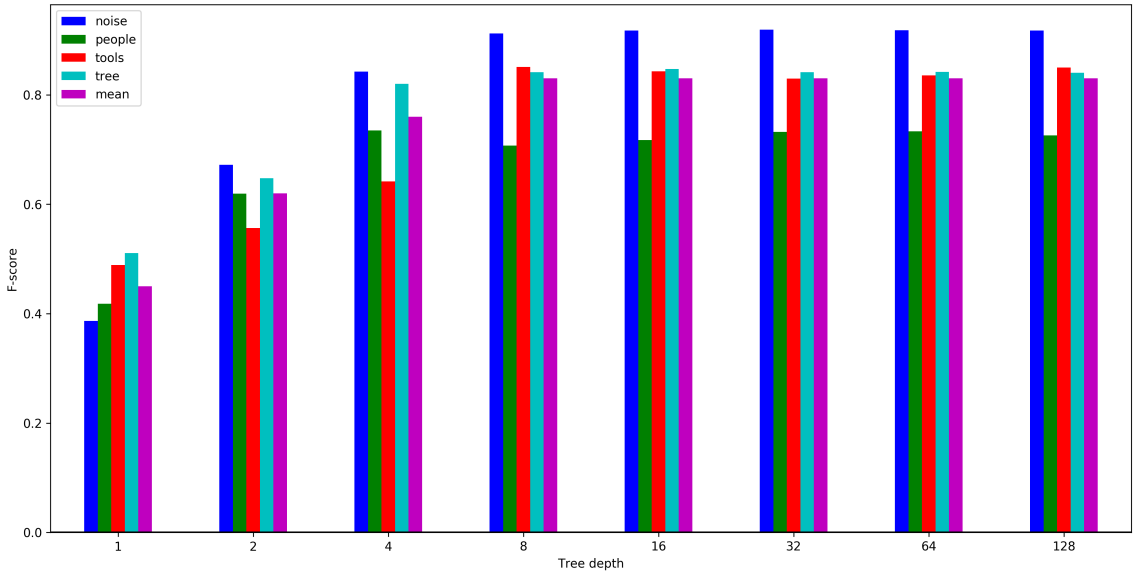


Figure 5.18: F-score for 64 trees of depth N

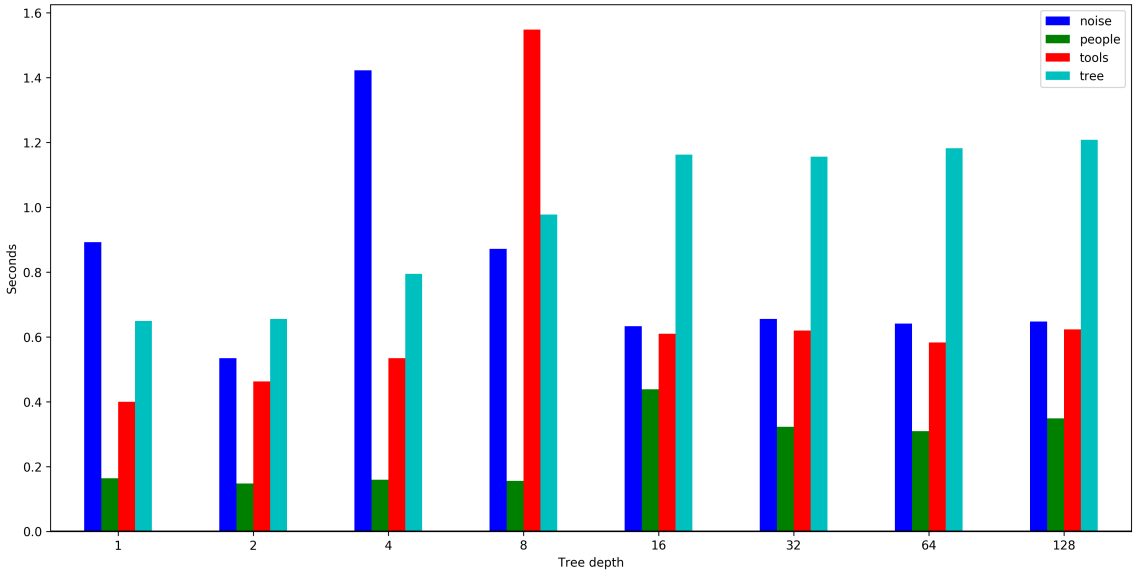


Figure 5.19: Time to train 64 trees of depth N

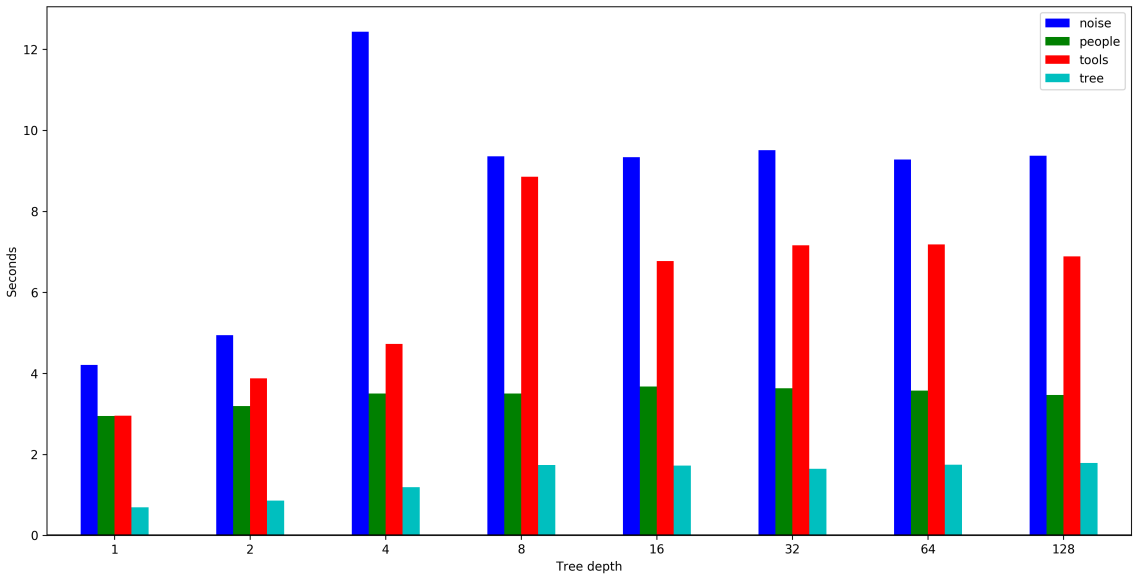


Figure 5.20: Time to classify scans with 64 trees of depth N

Increasing the maximum decision tree depth increases the likelihood of individual trees overfitting the training data. In aggregate this improves accuracy in Random Forest. Breiman [8] recommends that trees are grown to the largest extent possible. In Figure 5.18 it can be seen that accuracy stops increasing after a maximum depth of 8. Training time appears to remain constant after a maximum depth of 16 (see Figure 5.19). In our Random Forest, a tree stops growing after a pure set is achieved. This could explain why after a depth of 16 training cost plateaus. After a maximum depth of 8, the cost of classifying a trees also remains constant at around 10 seconds. The cost of evaluating an individual tree increases logarithmically as the depth increases. If the trees did grow beyond a depth of 16, we would thus not expect the cost to increase dramatically. We use a tree depth of 8 going forward.

5.2.3 Number of random tests

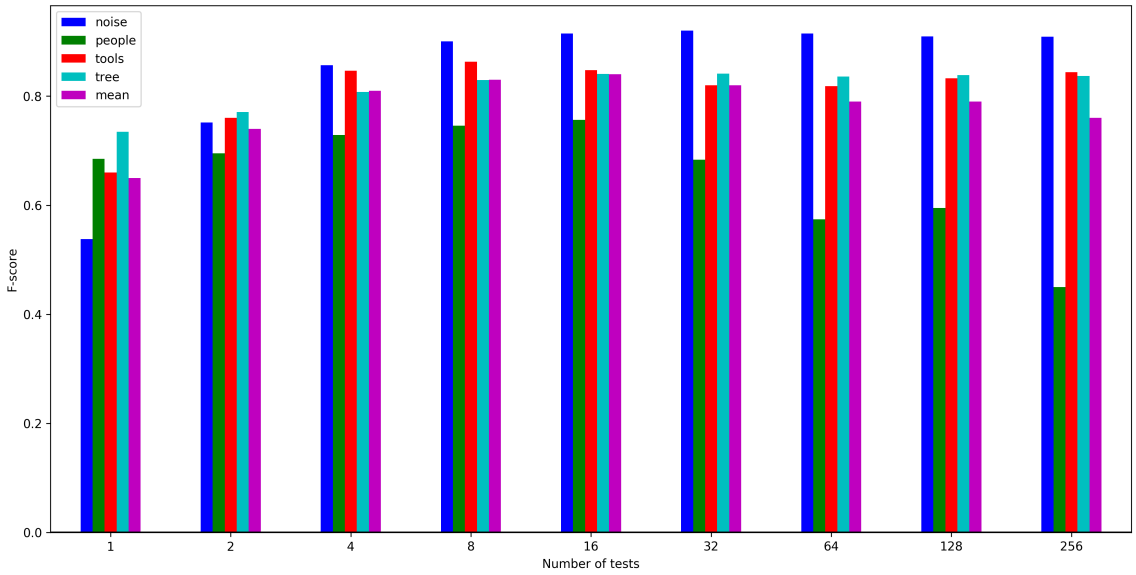


Figure 5.21: F-score using N random tests

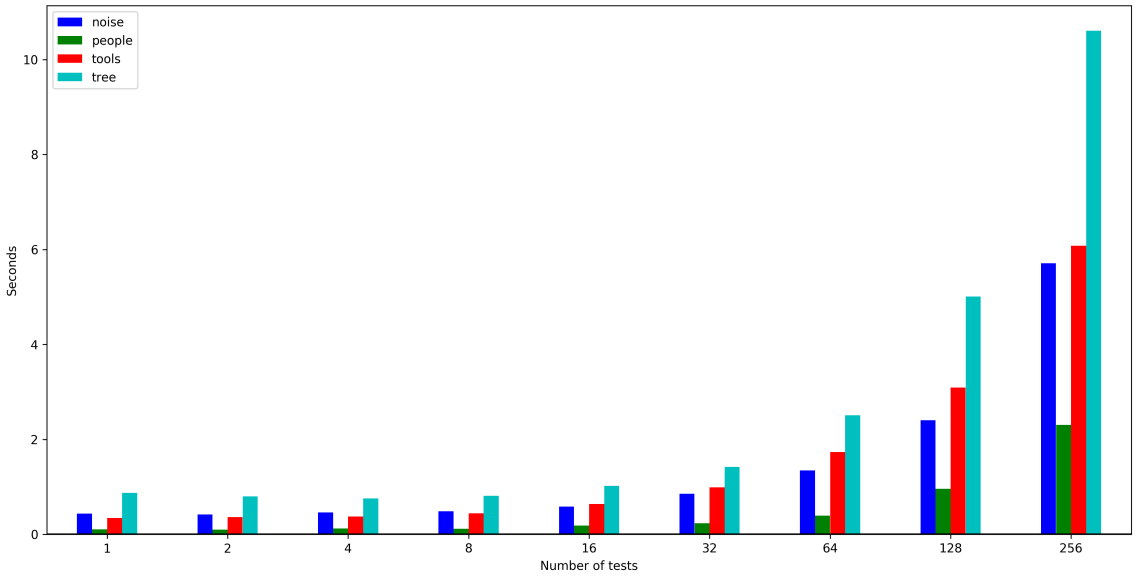


Figure 5.22: Time to train using N random tests

Generating random tests increases the likelihood of finding the optimal test for a given split in a tree. In Figure 5.21 it can be seen that the accuracy increases as more random tests are considered. After generating more than 16 random tests it can be seen that the results for the 'people' scan starts to decrease in accuracy. The reason for this is that the scan has fewer samples to train with than the other scans (6237 of each class after balancing the samples, see Table 5.1). As more random tests are generated we increase the likelihood that the best random test for a node in a in tree will be similar to the best test in parts of other trees, as they are more likely to sample the same data. This increases the correlation between trees and

reduces accuracy. Classification time remains under 1 second for 16 random tests per tree split where after cost increases substantially (see Figure 5.22). We thus use 16 random tests moving forward.

5.3 Results

In the previous sections we performed feature selection and classifier tuning with the primary focus being on accuracy instead of performance. We determined that XYZ, Intensity and Principal curvatures (radius 0.2m) were the optimal set of features for our test data. This was determined after range images were down sampled with a 0.02m voxel size. In this section we analyse the results.

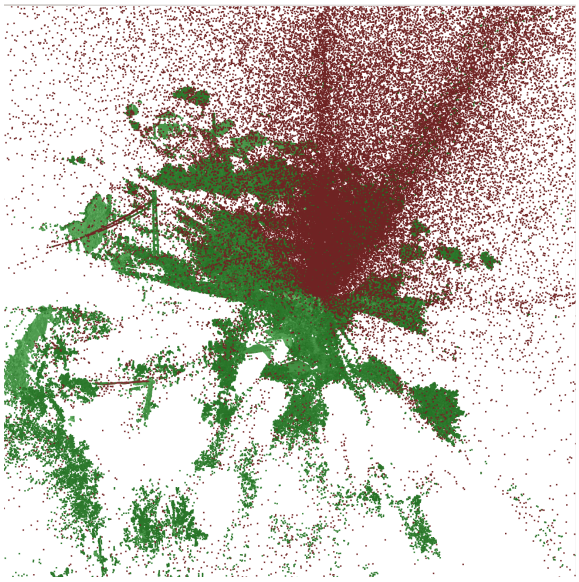
	fog	people	tools	tree
Initial accuracy	0.12	0.76	0.78	0.61
Final accuracy	0.91	0.67	0.85	0.84
Down sample duration	3.41s	2.78s	1.86s	1.35s
Curvature compute duration	90.77s	48.64s	34.12s	14.74s
Training duration	0.74s	0.20s	0.50s	0.94s
Classification duration	9.66s	3.40s	6.22s	1.49s
Total duration	104.58s	55.02s	42.70s	18.53s

Table 5.3: Accuracy and classification cost

Table 5.3 shows a summary of the accuracy and segmentation time for each test scan. It is clear than the bulk of processing time arises from feature computation (up to 90 seconds). The next biggest component is the classification time (up to 10 seconds). We know that the classification step can be optimised, without sacrificing accuracy, by reducing the number of trees. This will however not be sufficient as the feature computation dominates the cost equation.

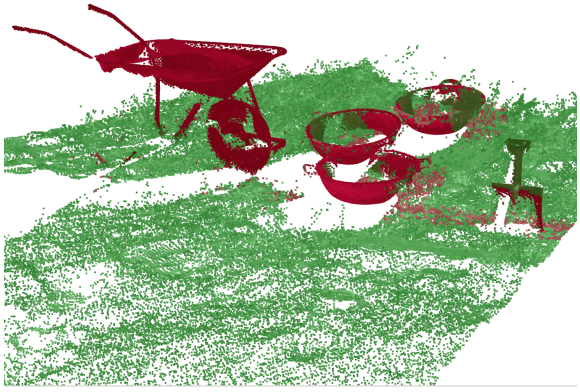
Looking at accuracy it can be seen that changes in accuracy over the baseline varies between scans. The greatest increase in accuracy is on the 'fog' scan (0.12 to 0.91). When compared to manually removing such points from a scan, a 105 second wait may be preferable. However, because non interactive filtering tools exists in packages like Meshlab [66], that is not a fair comparison. Meshlab's filter to remove isolated points with respect to diameter, would arguably be a better tool to compare against. When using this filter a user specifies the radius at which a point without neighbours is considered isolated, and is subsequently removed. Running this filter on 5 million points using a 0.5m diameter takes 12 seconds on our test hardware. The problem is, however, that a user needs to specify the correct radius to achieve the desired result. Much trial and error may thus be required to achieve the desired result. Given that undo is a missing feature in Meshlab, this may require that the scan be reloaded if the applied filtering was too aggressive. The ray of points at the centre of Figure 5.4 can also not be identified using an isolated point filter due to its density. Using Random Forest to specify filter parameters by example could allow users to avoid much trail and error, as well as achieve better results.

The worse case scenario manifests itself in the 'people' scan where a 9% decrease in accuracy is observed. Looking at Figure 5.1 one notices that the people under the archway are easy to isolate manually. A quick stroke of the brush tool is sufficient to select most of the points



(a) Fog

(b) People



(c) Tools

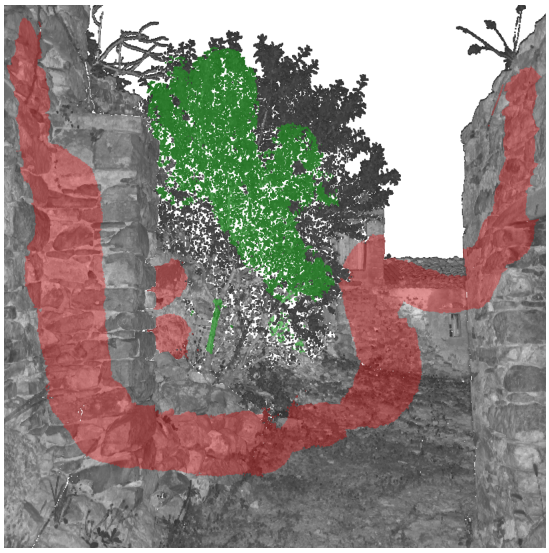
(d) Tree

Figure 5.23: Segmentation results using final test parameters

(a) Training labels.

(b) Result labels.

Figure 5.24: Refined segmentation of people and facades



(a) Training labels.

(b) Result labels.

Figure 5.25: Refined segmentation of trees

associated with them. There is thus very little to gain by using automation. This does not, however, explain the bad results. The first thing to notice in Figure 5.23b is that the wall inside the arch is misclassified as being the in same class the people. It would appear that the Random Forest relies primarily on point coordinates to localise the target. Labelling parts of the wall as shown in Figure 5.24a makes the Random Forest de-emphasize the XYZ feature. The resulting segmentation has an F-score of 0.87 which is 11% above baseline. In Figure 5.24b it can be seen that parts of the scan are still incorrectly segmented. The top of the doorway is now mislabelled, likely because it has similar curvature characteristics to the people and curvature is a more discriminative feature following additional labelling.

A similar effect can be seen in Figure 5.23c where the wall behind the tree is mislabelled. When parts of the wall behind tree are labelled as shown in Figure 5.25a, the resulting classification (Figure 5.25b) is better and has an improved F-score of 91% (+6%). These results suggest that using XYZ coordinates to enforce a degree of coherence is somewhat limiting.

The segmented tools in Figure 5.23c have a lower than expected F-score of 0.84%. The fact that wheelbarrow is high off the ground is likely big factor in why it was accurately segmented. The tools close to the ground were not completely segmented, possibly for the same reason. Down sampling may have also played a role in the misclassification of the spade's shaft.

5.4 Performance and accuracy

In the last section we saw that the computation of curvature dominates the processing time. Without parallelisation and/or more computational resources the tool can not run interactively without a trade-off in accuracy. In this section we look at what can be achieved on modest hardware by down sampling and reducing the number of trees used for classification.

Voxel size	fog F-score	people F-score	tools F-score	tree F-score	mean F-score
0.02	0.91	0.67	0.85	0.84	0.82
0.03	0.90	0.75	0.83	0.83	0.83
0.04	0.74	0.73	0.82	0.83	0.78
0.05	0.88	0.69	0.83	0.83	0.81
0.06	0.84	0.71	0.84	0.83	0.80

Table 5.4: F-score per scan for down sample voxel size

Voxel size	fog	people	tools	tree
0.02	101.17s	52.24s	40.84s	17.18s
0.03	38.84s	14.60s	15.92s	4.88s
0.04	18.20s	5.93s	8.62s	2.13s
0.05	12.15s	2.94s	5.62s	1.21s
0.06	9.19s	1.80s	4.10s	0.75s

Table 5.5: Processing time (in seconds) per scan for down sample voxel size

In Table 5.4 we list the F-score achieved in all of our tests at increasingly more aggressive levels of down sampling. In Table 5.4 we show the total processing time associated with each level. The first row of both tables shows the level of down sampling used for previous tests (0.02m). As the level of down sampling increases, accuracy levels diminish. Initially, however, increasing the down sample voxel size from 0.02m to 0.03m increases the aggregate accuracy and requires approximately 1/3 of the processing time. Apart from the 'tree' scan, the processing time is still more than 10 seconds. Increasing the voxel size to 0.04m brings the processing time for all but the 'fog' test case to under 10 seconds. The accuracy level for this test also falls by 16%. The F-score for 'people' and 'tools' tests decrease by 2% and 1% respectively while the 'tree' test result remains unchanged. Further down sampling with a 0.05m voxel size increases the overall accuracy and further reduces processing time. The 'fog' test result recovers to 88%

and can almost be computed in the same 12s as the Meshlab filtering tool. Increasing the voxel size to 0.6 brings all processing to under 10 seconds with an average of 80% accuracy.

Down-sampling can decrease processing time by an order of magnitude while only reducing accuracy by 2%. Interactions with feature computations can, however, lead to unexpected results. Feature selection after more aggressive down sampling may result in a different feature set that may be more appropriate for the degree of down sampling used. Our classification workload also scales with down sampling. Given that the classification step represents approximately only 10% of our previous workloads, it is not worth reducing the number of trees.

In the next section we report on user tests conducted using a 0.05m down sampling voxel size on cropped scans with a limited set of features, as suggested by our earlier results. As the results above show, using the down sample size should dramatically accelerate the computation required and still produce acceptable results.

5.5 User testing

A limitation of our Random Forest evaluation is that it doesn't directly translate into reductions in overall task time. What we aim to achieve is that the initial user labelling, training and running the classifier, then finally touching up the result, takes less time than segmenting the scan manually.

To test this a user experiment was designed. We selected 2 test scans not previously used in the Random Forest evaluation. Users were tasked to segment a scan from scratch or with classifier assistance in a timed experiment. Our hypothesis is that the total time for the classier assisted condition will be less than the non-assisted condition, if our classifier allows one to segment scans more quickly.

5.5.1 Design

The independent variable in this experiment is whether a user had use of the classifier during the segmentation task. The dependent variable is the time the user take to complete the task.

In this experiment we use a repeated measures design. In a repeated measures design, each participant take part in both the experimental and control conditions. The primary advantage of this design is that a participant can serve as his or her own control. This reduces the effect of individual differences in task speed. Secondly, fewer participants are needed as all partake in both conditions. A repeated measures, is however, also exposed to order effects. The order in which a participant performs a task may affects his or her performance. A participant may get better over time (learning effect), or he/she may get tired and exhibit diminished performance.

We control for order effects in two ways. Firstly, counterbalancing is used to alternate the order in which the user is exposed to each condition. Secondly, a priming task is provided to familiarise the user with the system and the task. This serves to reduce learning effects.

A source of error variance in our experiment is the initial labelling of the scan. As participants are likely to label different regions before invoking the classifier, the amount of touch

(a) Initial labelling (b) Segmentation result (c) Reference labels

Figure 5.26: Tree test data

up time will vary. This effect is compounded by inexperience. Knowing what regions to label comes with practise. As seen in Section 5.3, adding an extra stroke in the right place can greatly impact the result accuracy. To eliminate this source of variance, without recruiting a large number of participants, the initial classifier assisted task labelling was performed by a single, well primed, participant, and used for all subsequent experiments. The initial labelling was timed, and took roughly 10 seconds. This time was added to the classifier assisted task duration.

5.5.2 Participants

Point cloud cleaning usually requires expert judgement in order to determine what is unwanted and what not. However, as explained in Section 5.5.4, we use layers to provide users with a template of what needs to be selected, thus removing the need for expert users.

Nineteen participants with varying levels of computer experience were recruited on University notice boards, after ethics approval was obtained. In total 13 men and 6 women were recruited. Participants were offered R40 for participating in a one hour session. Prior to the experiment, three other participants were recruited for a pilot study. No personally identifiable information was collected. Participants were provided with an informed consent form that outlined the procedure and purpose of the experiment. It also informed participants that he or she could withdraw at any time without penalty.

5.5.3 Materials

Two test scans were used as shown in Figure 5.26 and Figure 5.27. The first scan was selected because it contains vegetation against structures which is one of the most common type of unwanted points. Covering this use case proves that the tool is useful for a large portion of work. The doors and shutters in the second scan are unlikely targets, but were selected as because it not something the feature selection was optimised for. Performing well on this case gives us more confidence in the ability of the classifier to adapt to other object classes with the given feature set.

The initial labelling in Figure 5.26a represents a 63.96% initial F-score. Our classifier raised the F-score to 91.58% in Figure 5.26b. The initial labelling for the second scan is 64.31% as

(a) Initial labelling (b) Segmentation result (c) Reference labels

Figure 5.27: Courtyard test data

shown in Figure 5.27a and our classifier increased this to 78.69% in Figure 5.27b. A down-sampling resolution of 0.05m was used.

CloudClean, was used to classify both scans in both experimental conditions. An additional plug-in was created to time the segmentation task. A reference layer was provided to the plug-in in order to compute the F-score of the current selection. The F-score is computed at 1 second intervals after the timer has started. Once the desired F-score is reached the timer would stop automatically.

5.5.4 Procedure

Before starting the experiment, participants were given a priming task. During the priming procedure users were given instructions on how to use the lasso, brush and plane selection tools. Users were then given 3 targets to that they could test each tool on. Once the user was comfortable the test would begin.

In the timed segmentation task, users were asked to recreate a reference segmentation by selecting the points. The reference segmentation was presented to them in a coloured layer on screen. They layer makes allows us to effectively paint a target on the point cloud. Layers and selections in CloudClean use alpha blending. When a point is both selected and in the reference layer, the colours of the layer and selection blend. This gives the participant feedback on his/her action. The F-score of the selection at the bottom of the screen also give the user feedback on his or her progress.

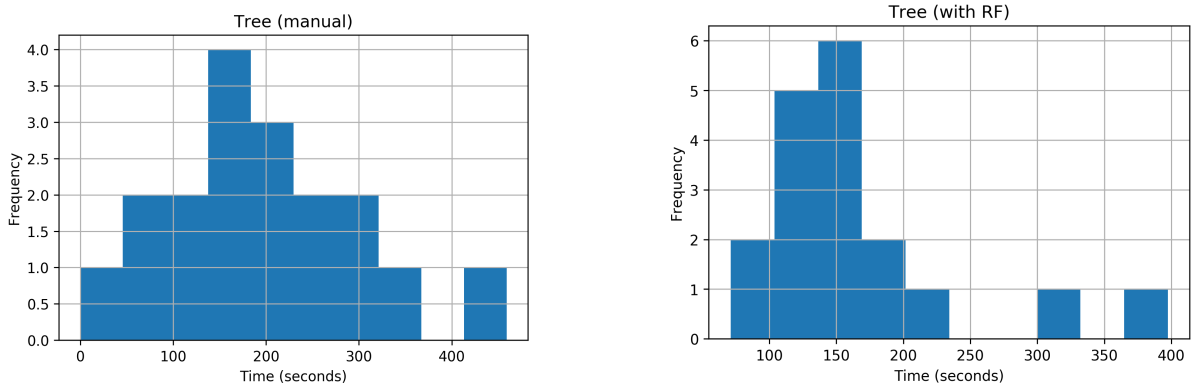
The segmentation had to be recreated with 97% accuracy the tree, and 95% accuracy for the courtyard. In the pilot study it was determined that segmenting the last 2% of the courtyard scan resulted in the experiment running over the allocated time. It was thus reduced to 95%. In the control condition, experiment users started with no selection state. In the experimental condition users were presented with the existing result, achieved through machine learning from the first user's initial labelling.

	Manual condition	Classifier assisted condition				
	Total time	Label time (estimated)	Processing time	Touch-up time	Total time	Difference
Tree segment	206.5s ±100.1s	10s	8.75s	141.7s ±73.5s	164.6 s±42.0s	-42s (-20.4%)
Wall segment	268.3s ±140.3s	10s	9.01s	206.2s ±148.3s	244.7s ±23.86s	-23s (-8.9%)

Table 5.6: Results of classifier assisted segmentation user study.

5.5.5 Results

Given a coarse user labelling, the classifier assisted segmentation reduced the time to label the remaining scene points quite dramatically (see Table 5.6). However, no significant reduction in *overall* task duration was found for either task. In the first task (tree segment) users completed the manual segmentation in an average time of 207±100 seconds. The classifier assisted segmentation was performed in 142±73 seconds with a p-value of 0.03, which is significant for $p < 0.5$. When the estimated labelling (10s) and processing (8.75s) is added, the overall time increases to 164±78 seconds representing a 42 second (20.4%) speed up. The p-value for this speed-up is however 0.23 indicating no significance at $p < 0.5$. In the second task (wall segment) users completed the manual segmentation in an average time of 268±140 seconds. The classifier assisted segmentation was performed in 206±148 seconds, which was not significant. When labelling (10s) and processing overhead (9.01s) is added, the total time is 244.46±158 seconds, representing a 23 second (8.9%) speed-up, also with no significance at $p < 0.5$.



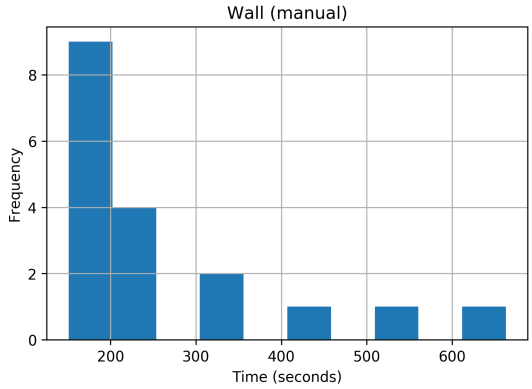
(a) Task without Random Forest

(b) Task with Random Forest

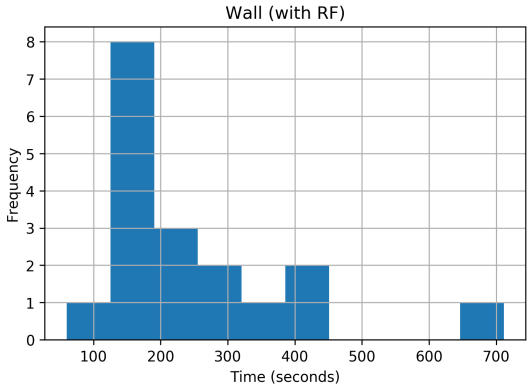
Figure 5.28: Distribution of tree segmentation task results

The results of the first task (see Figure 5.28) appear to be normally distributed. However, after testing for normality using a Shapiro-Wilk test [55], the distribution of the classifier assisted condition was found not to be normally distributed ($p < 0.01$). The same is true for both conditions of the second task (see Figure 5.29). A non parametric Wilcoxon signed-rank test [70] was therefore used to compare paired samples.

In the first task Figure 5.30 shows a much smaller spread in the classifier assisted condition. The initial assistance that the algorithm provides likely reduces the effort required up-front. The remainder of the work that requires more precision then varies depending on one's personal ability and experience. The outliers are likely due to a lack of computer experience of some participants. The second task was more difficult and required the accuracy requirement to be reduced during a pilot study. It can also be seen that it look longer than the first. During the experiment many participants found it extremely difficult to achieve the last 1% or 2% of

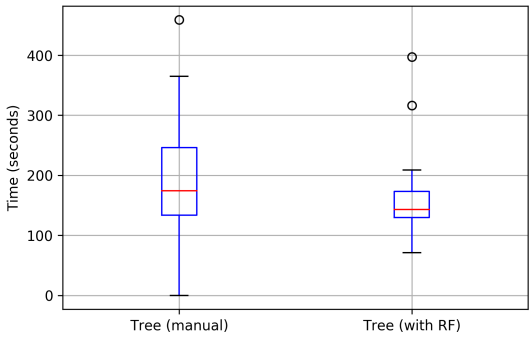


(a) Task without Random Forest

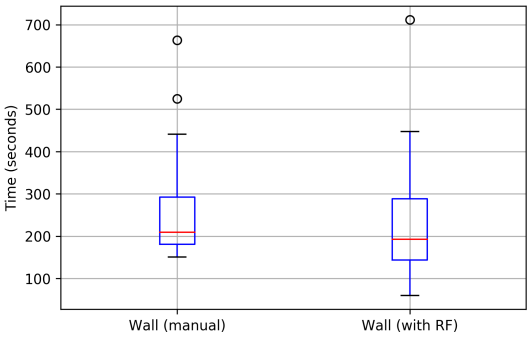


(b) Task with Random Forest

Figure 5.29: Distribution of wall segmentation task results



(a) Tree segmentation task



(b) Wall segmentation task

Figure 5.30: Box and whiskers plot of results

required accuracy. This was likely due to the shutters and doors having a low intensity value that made it hard to see the overlap between the target layer and the participants selection. In Figure 5.30 it can be seen that the spread was bigger in the classifier assisted task. This can be attributed to a participant not noticing some incorrectly selected regions that were automatically labelled.

It can also be seen that the problem of enforcing coherence using XYZ coordinate values as discussed in Section 5.3, also manifest in Figure 5.27c. Exerting more effort in ensuring that the coarse user labelling is spatially distributed may reduce the time required for touch-up. This could be aided by adapting the system to preview segmentation result as the coarse labelling proceeds. This functionality can be supported by Christian et al.'s [12] on-line Random Forest implementation. Engineering a set of features that do not rely of XYZ values is also worth investigating.

It is worth emphasising that — given the coarse labelling — the time required for user interaction was much lower than for the regular manual cleaning process. A bigger sample size, with a less diverse set of users or more priming, may have resulted in significance. The results, however, still show that the learning process produces an initial scan segmentation that requires a fair amount of "touch-up" time.

Engineering better features may lead more accurate results which could reduce the required touch up time. Computing curvature or eigenvalues at multiple resolutions is one avenue worth exploring. This is however likely to come at an increase in computational cost which will necessitate the use of a GPU and/or multi core techniques. Utilising the extra information in colour channels is also unexplored in this work. In scans that contain colour information it is likely to have a big impact.

Full results are available in Appendix B.

5.6 Summary

In this chapter our aim was to create a segmentation tool that would adapt to the unpredictable objects or noise one is likely to target in heritage scans. Such a tool needed to be suitable for interactive use as cleaning is an iterative process.

Selecting features and tuning classifier hyper-parameters presented a dilemma. In order to evaluate the features and classifier hyper-parameters used for segmenting unpredictable targets, we need to test data that represent the types of targets we expect. Four test scans were selected containing difficult targets often found in heritage scenes. The hope was that the selected features and Random Forest hyper-parameters that perform well on such difficult test data, would be able to learn other difficult targets.

Prior to feature selection it became apparent that a nearest neighbour search dominated the cost of computing many features. The observation was made that range images have very dense clusters of points close to the scan origin. The sample density in these regions were far denser that necessary. A memory efficient octree based down sampling technique was shown to thin out dense areas while leaving sparser areas unaffected. Using this method with a 0.02m down sampling resolution allowed us to manage neighbourhood lookup cost. It was however still too computationally expensive to compute features over multiple radii, thus an optimal radii was determined to be 0.2m.

A forward selection wrapper method was used to determine an optimal feature set. A wrapper method was used as no other classifiers were under consideration. The optimal feature set was determined to be XYZ, Intensity, and Curvature; eigenvalues were discarded since their large computational cost did not result in a meaningful increase in accuracy.

The optimal Random Forest hyper-parameters were determined to be 64 trees, with a maximum depth of 16, and 100 random tests to be evaluated before each split. The Random Forest does not appear to be very sensitive to the number of trees. After 8 trees the marginal benefit is negligent. The classification cost does however increase linearly with the number of trees. When using 64 trees this only represents 10% of the total run time of the tool. The feature computation clearly dominates the cost equation and by comparison the actual classification is very cheap.

In an effort to reduce the feature computation cost further, down-sampling was applied. A decrease in F-score was expected as more down-sampling was applied. However, the mean F-score fluctuated in a general downward trend. It is likely that the curvature computation, was affected by down-sampling, and that the optimal radius may need to be redetermined after

down-sampling. At a 0.06m resolution, all test cases could be run under 10 seconds with a mean F-score of 80%.

For user testing, a more conservative 0.05m resolution was used. We attempted to show that the use of our classifier reduced the overall segmentation time. While results showed a reduction in segmentation time of between 8.9% and 20.4%, this was not significant. The lack of significance can be attributed to an insufficient sample size and participant's lack of prior experience with the system that made segmentation times highly variable. It is expected that by using multi-core techniques to manage feature computation cost, better results could be achieved, by reducing down sampling and through the use of multi resolution features. The use of colour channels, when available, is also expected to improve accuracy and in turn reduce touch up time.

Chapter 6

Conclusion

The aim of this thesis was to accelerate the point cloud cleaning process. In line with this aim, we set three objectives:

- Reduce the impact of inefficient layering on system resources.
- Reduce the navigation overhead of the 3D workspace.
- Speed up segmentation by introducing a segmentation tool that can learn object classes and segment them on the fly.

We addressed the first two objectives as part of the implementation of our software framework and the last is realised as an extension to our framework.

To reduce the impact of inefficient layering on system resources, we introduced a novel layering technique. This technique supports a large number of layers while consuming a near constant amount of memory. It also supports extremely efficient set operations that can aid the cleaning work flow.

Navigation overhead is reduced via a roll-corrected first person camera mode. The camera mode assists users in recovering from potentially disorientating states while maximising rotational freedom. When a potentially disorientating state is detected, the camera is slowly nudged back out of this state over successive user movements. A user experiment showed this camera mode significantly speeds up navigation between 29.8% and 57.8%, when compared to non roll-corrected camera.

Lastly, we created a semi-automated segmentation tool that harnesses a Random Forest classifier to interactively learn new object classes from examples, and assists the user in segmenting the remainder of a scene. Preliminary findings from a user experiment show, without significance, that this method reduces the overall cleaning time between 8.9% and 20.4%.

6.1 Limitations

CloudClean system in itself has two noteworthy limitations. Firstly, it lacks support for rendering very large point clouds that exceed system memory. Secondly, the system currently does not support loading or rendering RGB colour channels in point clouds.

The main limitation of our layering system is that it does not support large numbers of overlapping layers. The number of layers is limited by the number of overlaps between layers. In the worse case, all layers will overlap. In this scenario only n layers are supported, where n is the number of bits allocated per point for layering purposes. If no overlaps occur 2^n layers can be supported. In real world work flows we do not expect a large number of overlap between layers.

Finally our classification tool is proof of concept and therefore has much room for improvement. Firstly a limited amount of training data was used to select features and tune the classifier. We therefore cannot predict with certainly how it will perform on a wide variety of scans. Establishing a set of performant features and random forest parameters that will produce robust results on a diverse range of data will require a substantial amount of training data and effort.

The limited of training data we used, did highlight a number of shortcomings in our work. These limitations and ways to address them is discussed in the next section.

6.2 Future work

Our classification tool leaves much of room for future work. A problem with the implementation is that the the classifier has a tendency to localise objects using XYZ coordinates in the absence of other strong features. The RGB channel of colour range images is likely to be a very strong feature and may mitigate this issue and result in higher segmentation accuracy. During feature evaluation it was seen that a feature's performance may vary depending on both the radius used and the target class. Including features such as curvature computed over multiple resolutions may therefore further improve accuracy. For this to work, however, the feature computation time would need to be reduced. Finding ways to cluster non-uniform point clouds into related patches could reduce computation overhead while sacrificing less accuracy compared to subsampling.

To ensure a robust set of features and classifier parameters, a larger set of more variable training data should be used in future work.

Another potential way to improve user efficiency would be to provide near real-time classification results after each stroke. Christian et al.'s [12] Random Forest implementation was specifically designed to learn from streaming data. It would thus not require relatively few adjustments for CloudClean to support this.

CloudClean's lack of support for data sets that exceed system memory could be addressed by adding a pre-processing step. Creating a multi-resolution data structure when loading files could enabled it to render regions with different levels of detail and thus not keep all data in memory. This would enable it to work with much larger workloads

Finally, the system currently only supports displaying grey scale point clouds. Adding support for colour range images would require minor changes to the renderer, but will make the software to exploit colour information.

6.3 Contributions

This principal contributions of this thesis are:

1. a new open source, cross-platform, point cloud cleaning framework designed for creating and evaluating new semi-automated segmentation methods. This framework is the first open source point cloud software that supports and cleaning iterative work flow with undo capabilities.
2. a novel point selection layering technique that can support a large number of layers with a near constant amount of memory.
3. a roll-corrected first person camera that maximises rotational freedom while at the same time avoiding confusing camera orientation states.
4. a semi-automated segmentation technique that can learn to new object classes and segment them on the fly.

Source code for CloudClean is available at `https://github.com/circlingthesun/CloudClean`.

Acknowledgements: spatial data provided by the Zamani heritage documentation project, University of Cape Town.

Bibliography

[1] Marc Alexa, Johannes Behr, Daniel Cohen-or, Shachar Fleishman, David Levin, and Claudio T Silva. Computing and Rendering Point Set Surfaces. *IEEE Transactions on visualization and computer graphics*, 9(1):3–15, 2003.

[2] D. Anguelov, B. Taskar, V. Chatalbashev, D. Koller, D. Gupta, G. Heitz, and A. Ng. Discriminative Learning of Markov Random Fields for Segmentation of 3D Scan Data. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 169–176. IEEE, 2005.

[3] Fausto Bernardini and H Rushmeier. The 3D model acquisition pipeline. *Computer Graphics Forum*, 21(2):149–172, 2002.

[4] PJ Besl and ND McKay. Method for registration of 3-D shapes. *Robotics-DL tentative*, 1992.

[5] Roshan Bhurtha and Christopher Held. Introduction to terrestrial laser scanning, 2007.

[6] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A Training Algorithm for Optimal Margin Classifiers. *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152, 1992.

[7] Yuri Y Boykov and Marie-Pierre Jolly. Interactive Graph Cuts for Optimal Boundary & Region Segmentation of Objects in N-D Images. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, number July, pages 105–112. IEEE, 2001.

[8] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[9] Daniel Brown. Short-range 3D scanning technologies: an overview, 2012.

[10] Cl'ement Chehata, Nesrine and Guo, Li and Mallet. Airborne lidar feature selection for urban classification using random forests. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 38(Part 3):207–212, 2009.

[11] Michael Chen, S. Joy Mountford, and Abigail Sellen. A study in interactive 3-D rotation using 2-D control devices. *15th annual conference on Computer graphics and interactive techniques - SIGGRAPH '88*, 22(4):121–129, 1988.

[12] Amir Saffari Christian, Christian Leistner, Jakob Santner, and Horst Bischof. On-line Random Forests. In *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, pages 1393–1400. IEEE, 2009.

[13] P Cignoni, M Callieri, M Corsini, M Dellepiane, F Ganovelli, and G Ranzuglia. Meshlab: an open-source mesh processing tool. In *Sixth Eurographics Italian Chapter Conference*, pages 129–136, 2008.

[14] Nicolas David, Bruno Vallet, Jerome Demantke, and Clément Mallet. Dimensionality based scale selection in 3D lidar point clouds. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2009.

[15] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems. *J. Mach. Learn. Res*, 15 (1):3133–3181, 2014.

[16] Delbert Ray Ford Jr, Lester Randolph and Fulkerson. Maximal flow through a networ. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[17] Gianfranco Forlani, Carla Nardinocchi, Marco Scaioni, and Primo Zingaretti. Complete classification of raw LIDAR data and 3D reconstruction of buildings. *Pattern Analysis and Applications*, 8(4):357–374, Jan 2006.

[18] Rapid Form. 3D Scanners A guide to 3D scanner technology, 2014.

[19] C Fröhlich and M Mettenleiter. Terrestrial laser scanning - new perspectives in 3D surveying. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2004.

[20] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-Wesley, 1995.

[21] Aleksey Golovinskiy, Vladimir G Kim, and Thomas Funkhouser. Shape-based recognition of 3D point clouds in urban environments. In *2009 IEEE 12th International Conference on Computer Vision*, pages 2154–2161. IEEE, Sep 2009.

[22] Hermann Gross and Ulrich Thoennessen. Extraction of lines from laser point clouds. *Symposium of ISPRS Commission III Photogrammetric Computer Vision*, 36:86–91, 2006.

[23] Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research (JMLR)*, 3(3):1157–1182, 2003.

[24] Chih-wei Hsu, Chih-chung Chang, and Chih-jen Lin. A Practical Guide to Support Vector Classification. 1(1):1–16, 2003.

[25] J. Huang and C. H. Menq. Automatic data segmentation for geometric feature extraction from unorganized 3-D coordinate points. *IEEE Transactions on Robotics and Automation*, 17(3):268–279, 2001.

[26] K Jeffery. Navigating in a 3D world. *Animal thinking: contemporary issues in comparative cognition*, pages 23–28, 2011.

[27] George H GH John and Pat Langley. Estimating Continuous Distributions in Bayesian Classifiers. *IN PROCEEDINGS OF THE ELEVENTH CONFERENCE ON UNCERTAINTY IN ARTIFICIAL INTELLIGENCE. Montreal, Quebec, Canada*, 1:338–345, 1995.

[28] Andrew E. Johnson and Martial Hebert. Using spin images for efficient object recognition in cluttered 3D scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(5):433–449, 1999.

[29] B Jutzi and H Gross. Nearest neighbour classification on laser point clouds to gain object structures from buildings. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 38(Part 1):4–7, 2009.

[30] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. *Eurographics Symposium on Geometry Processing*, 2006.

[31] Leica. Cyclone pointcloud export format - Description of ASCII .ptx format.

[32] Leica. Cyclone, 2012.

[33] M S Livingstone and D H Hubel. Psychophysical evidence for separate channels for the perception of form, color, movement, and depth. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 7(11):3416–3468, 1987.

[34] David K Mackinnon and Victor Aitken. A Comparison of Precision and Accuracy in Triangulation Laser Range Scanners. In *2006 Canadian Conference on Electrical and Computer Engineering*, number May, pages 832–837. IEEE, 2006.

[35] Bruce Merry, James Gain, and Patrick Marais. Moving least-squares reconstruction of large models with GPUs. *IEEE transactions on visualization and computer graphics*, 20 (2):249–61, Feb 2014.

[36] Lance A Miller and John C Thomas. Behavioral issues in the use of interactive systems, 1977.

[37] Eric N Mortensen and William a Barrett. Intelligent scissors for image composition. *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques SIGGRAPH 95*, 84602(801):191–198, 1995.

[38] Marius Muja and David Lowe. FLANN - Fast Library for Approximate Nearest Neighbors User Manual. *Visapp*, pages 331–340, 2011.

[39] D. Munoz, J.a. Bagnell, N. Vandapel, and M. Hebert. Contextual classification with functional Max-Margin Markov Networks. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 975–982, Jun 2009.

[40] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[41] Jakob Nielsen. 10 Usability Heuristics for User Interface Design. *Conference companion on Human factors in computing systems CHI 94*, pages 152–158, 2005.

[42] M Pauly, Richard Keiser, and M Gross. Multi-scale Feature Extraction on Point-Sampled Surfaces. *Computer graphics forum*, 22(3), 2003.

[43] Adobe Photoshop. 6.0 User Guide. *San Jose, CA: Adobe Systems*, 2000.

[44] Pointools. Pointools Edit, 2012.

[45] T Rabbani, F a van den Heuvel, and G Vosselman. Segmentation of point clouds using smoothness constraint. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 36(5):248–253, 2006.

[46] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. Grabcut: Interactive foreground extraction using iterated graph cuts. In *ACM transactions on graphics (TOG)*, pages 309–314. ACM, 2004.

[47] S. Rusinkiewicz and M. Levoy. Efficient variants of the ICP algorithm. In *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*, pages 145–152. IEEE, 2001.

[48] Radu Bogdan Rusu. *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Computer Science department, Technische Universitaet Muenchen, Germany, 2009.

[49] Radu Bogdan Rusu, Nico Blodow, and Michael Beetz. Fast point feature histograms (FPFH) for 3D registration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3212–3217. IEEE, 2009.

[50] RB Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). *2011 IEEE International Conference on Robotics and Automation (ICRA)*, 36(2):1–4, 2011.

[51] Heinz Rüther and Christoph Held. Challenges in Heritage Documentation with Terrestrial Laser Scanning. In *Proceedings of AfricaGeo*, 2011.

[52] Jakob Santner, Markus Unger, Thomas Pock, Christian Leistner, Amir Saffari, and Horst Bischof. Interactive Texture Segmentation using Random Forests and Total Variation. *Procedings of the British Machine Vision Conference 2009*, pages 66.1–66.12, 2009.

[53] Oliver Schall and Marie Samozino. Surface from scattered points. pages 138–147, 2005.

[54] Jürgen Schmidhuber. Deep Learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[55] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[56] Roman Shapovalov, Er Velizhev, and Olga Barinova. Nonassociative markov networks for 3d point cloud classification. the. In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences XXXVIII, Part 3A*. Citeseer, 2010.

[57] Andrei Sharf, Marc Alexa, and Daniel Cohen-Or. Context-based surface completion, 2004.

[58] Ken Shoemake. ARCBALL: A user interface for specifying three-dimensinal orientation using a mouse. *Proceedings of Graphics Interface '92*, pages 151–156, 1992.

[59] Akin Tatoglu and Kishore Pochiraju. Point cloud segmentation with LIDAR reflection intensity behavior. *2012 IEEE International Conference on Robotics and Automation*, pages 786–790, May 2012.

[60] Technodigit. 3DReshaper, 2012.

[61] Terrasolid. Terrascan, 2012.

[62] Georgios Toubekis, Irmengard Mayer, Marina Doring-Williams, Kosaku Maeda, Kazuya Yamauchi, Yoko Taniguchi, Susumu Morimoto, Michael Petzet, Matthias Jarke, and Michael Jansen. Preservation and management of the unesco world heritage site of bamiyan: Laser scan documentation and virtual reconstruction of the destroyed buddha figures and the archaeological remains. pages 185–2, 2009.

[63] Rudolph Triebel, Kristian Kersting, and Wolfram Burgard. Robust 3D scan point classification using associative Markov networks. In *International Conference on Robotics and Automation*, pages 2603–2608. IEEE, 2006.

[64] J Tuley and M Hebert. Analysis and Removal of artifacts in 3-D LADAR Data. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 2203–2210. IEEE, 2005.

[65] VirtualGrid. VR Mesh Studio, 2012.

[66] Visual Computing Laboratory. Meshlab, 2012.

[67] Yunsheng Wang, Holger Weinacker, and Barbara Koch. A Lidar Point Cloud Based Procedure for Vertical Canopy Structure Analysis And 3D Single Tree Modelling in Forest. *Sensors*, 8(6):3938–3951, 2008.

[68] Martin Weinmann, Boris Jutzi, and Clément Mallet. Feature relevance assessment for the semantic interpretation of 3D point cloud data. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 5(November):11–13, 2013.

[69] Karen F West, Brian N Webb, James R Lersch, Steven Pothier, Joseph M Triscari, and A Evan Iverson. Context-driven automated target detection in 3D data. In *Defense and Security*, pages 133–143. International Society for Optics and Photonics, 2004.

[70] Frank Wilcoxon and Roberta A Wilcox. *Some rapid approximate statistical procedures*. Lederle Laboratories, 1964.

[71] Z+F. Z+F LaserControl, 2014.

[72] Q Zhan, Yubin Liang, and Y Xiao. Color-based segmentation of point clouds. In *Proc. ISPRS Laser Scan. Workshop*, pages 248–252, 2009.

Appendices

Appendix A

Navigation experiment results

Participant	Palms (with roll-correction)	Palms	Stairs (with roll-correction)	Stairs
1	25	64.33	25.67	31.67
2	N/A	N/A	N/A	N/A
3	65.33	119.33	59	69.33
4	5.67	27	8	16.67
5	35.67	66	14	14.67
6	9	30	10.67	14.33
7	70	139.33	47.67	114.33
8	33	81.67	67	43.33
9	29.33	50.67	29.33	30
10	16	53.33	20.67	35.33
11	7.67	9.67	6.33	9.67
12	33	126	33	71
13	28	56	17.33	26
14	18	31.67	13.67	29.67
15	19	38	28.33	25.67
16	29.33	116	17	38
17	12.67	31.33	13	14.33
18	6	21.67	5.67	17
19	19	32	16	15.33

Table A.1: Navigation experiment results

Appendix B

Segmentation experiment results

Participant	Tree (manual)	Tree (with RF)	Wall (manual)	Wall (with RF)
1	314	159.7	243	447.64
2	88	142.3	151	149.99
3	N/A	N/A	N/A	N/A
4	132	208.56	220	294.39
5	365	162.79	198	198.54
6	234	71.2	157	60.11
7	459	316.35	253	711.57
8	250	397.41	663	271.84
9	299	176.78	441	329.27
10	87	129.78	176	445.63
11	110	110.94	194	142.02
12	184	87.91	525	132.73
13	197	130.04	326	134.24
14	150	152.08	305	166.92
15	138	129.75	240	186.27
16	165	122.51	177	226.97
17	186	142.08	194	154.76
18	138	176.56	193	221.59
19	221	143.48	174	125.92

Table B.1: Segmentation experiment results