IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 26, 2021, accepted May 19, 2021, date of publication May 31, 2021, date of current version June 8, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3084954

RayCloudTools: A Concise Interface for Analysis and Manipulation of Ray Clouds

THOMAS D. LOWE AND KAZYS STEPANAS
CSIRO, QCAT, Pullenvale, QLD 4069, Australia

Corresponding author: Thomas D. Lowe (thomas.lowe@csiro.au)

This work was supported by Commonwealth Scientific and Industrial Research Organisation's (CSIRO's) Data61, Land and Water, Wine Australia, and the Department of Agriculture's Rural Research and Development for Profit Program.

ABSTRACT We describe a new toolset for the manipulation and analysis of *ray clouds* (3D maps defined by a set of rays from a moving lidar to the scanned surfaces). Unlike point clouds, ray clouds contain information on free-space (air) as well as surface geometry. This allows the toolset to perform volumetric functions and analysis that cannot be done on point clouds alone. The presented toolset consists of seventeen command-line functions, with a C++ library available for those who require more control or tight integration. Our aim is that RayCloudTools is as useful and simple as possible, and we use this paper to demonstrate its utility, and to assess its ease of use, with comparison to established cloud processing libraries.

INDEX TERMS Ray cloud, point cloud, lidar, 3D mapping, decimation, alignment.

I. INTRODUCTION

Robotic perception has come a long way with the advent of spinning lidar, depth cameras and computer vision systems for generating 3D maps directly from the robot. These maps are typically point clouds, which contain a 3D location per point and possible additional data such as colour. Many libraries have been developed for manipulating and analysing these point clouds [1]–[8] to aid in perception and our understanding of what the robot has observed. Unfortunately, the point cloud data alone is lacking an important piece of information: the knowledge of the free space between the sensor and contact point. Algorithms based only on point clouds therefore lack volumetric information about the scene and are impaired in their ability to correctly analyse it.

Occupancy gridmaps [9] provide a way to approximate this volumetric information. They voxelise a scanned area and discretise the ray information into per-voxel statistics representing attributes such as occupancy [9], partial occupancy [10], and surface covariance [11], [12]. They are valuable in real-time scene perception and are common in robotic navigation. However, as a raster data format, gridmaps are imprecise for analysis, and lose quality under repeated manipulations such as aligning maps. A vector format representing the exact sensor observations is better suited to the analysis and manipulation workflow that is the subject of this paper. That vector format is the ray cloud.

We present an open-source suite of tools where the fundamental data structure is the ray cloud.[1] This is the set of directed line segments from sensor to contact point, together with a time stamp, and optional colour and intensity information. The library of tools treats this data as a volumetric description of the scene, rather than as a set of surfaces. We describe the design philosophy, structure and contents of this library, and demonstrate how this volumetric component of ray clouds enables long-term mapping techniques and cloud analysis that is not possible on point clouds alone.

The aim of this library is to provide a set of building-blocks that are useful to as many users as possible. We define this usefulness by five criteria, which we demonstrate within the main sections of the paper:

1) it performs functions that cannot be performed on just point clouds - Section VI
2) it is viable on large data sets - Section VII
3) inclusion of all the common functions - Section VIII
4) functions are reusable building-blocks - Section IX
5) simplicity of interface - Section X

II. RELATED WORK

Tools for processing point clouds have been available since the early 2000s, for instance the CloudCompare [1] software originated in 2003. While primarily a graphical

The associate editor coordinating the review of this manuscript and approving it for publication was Hui Liu.

[1]https://github.com/csiro-robotics/raycloudtools

tool for manipulating point clouds, it also comes with a command-line interface to allow simple manipulations to be automated. A similar command-line interface was created by the developers of the.las/.laz point cloud data format, called LasTools [2]. This included more complex techniques such as ground extraction, but with a far larger parameter space from which to find the appropriate function call.

The growing popularity of 3D mapping led to the creation and growth of Point Cloud Library [3], a C++ library for processing point clouds. It contains hundreds of classes, with multiple member functions, which can be connected and used in many different ways. The benefit of this approach is its large functionality, but its drawback is the steep learning curve involved in understanding how to interact with this wide interface effectively.

Largely due to the above drawback, Open3D [4] was developed. The resulting interface is simpler than Point Cloud Library. Even so, its C++ interface contains over 200 classes, and tools such as aligning clouds have a large search space of methods, options and parameters.

Other point cloud tools also exist [5]–[8], [13], [14] but all of these cited tools have the drawback that point clouds lack the volumetric information required to perform many of the more useful functions. Examples of these are given in Section VI.

Additionally, we believe that the growing popularity of 3D mapping supports a tool suite for non-expert users, where ease of use is even more important than for existing packages. For this reason, we have developed RayCloudTools as a primarily command-line interface with just seventeen core functions, and with ray clouds as the underlying data structure.

III. DESIGN PHILOSOPHY

Our goal is to make ray clouds a viable data format by providing tools for their manipulation that anyone can pick up and use. For this general audience it is important that tools are as simple as possible. This leads to our main design goals:

A. TURN-KEY

- command-line tools, no programming knowledge required
- `cmake`, `make` and `make install` are enough to start running the command-line tools from anywhere.
- automatic help text whenever the command-line is incorrect
- minimising the parameter search space where possible

B. INFORMATION HIDING

Basic functionality is turn-key, but advanced functionality is still available through less visible options. This is a form of information hiding, which we support in four ways:

- build options allow more features through additional third party dependencies. Examples include.laz file and convex hull support.
- optional command-line parameters allow additional control
- more complex functionality through sequencing multiple commands
- using the C++ interface for greater control

C. NOT REINVENTING THE WHEEL

The ray cloud is stored in a point cloud file format (Stanford PoLYgon .ply files) with an additional attribute to represent the ray. This means that existing rendering software can display the clouds, and specialist libraries for point cloud processing can continue to be used. Additionally, ray clouds are imported from an existing point cloud and associated trajectory file. This means that RayCloudTools is not required to perform 3D mapping algorithms such as SLAM. So the generation of the 3D map from the raw sensor data is not part of the scope of the library. Instead, this library is focused on the easy manipulation of ray clouds, and on tools that use the full ray information.

IV. FILE FORMAT

The Stanford polygon (.ply) format is sufficiently versatile to represent polygon meshes, point clouds and also ray clouds. It requires a list of vertex positions (x,y,z) and the remainder are user-defined attributes.

We choose to re-purpose the common user attribute for a vertex normal (nx,ny,nz) to represent the ray from the end point to the sensor location. This allows existing rendering software [1], [13] to display the full set of rays by toggling vertex normal rendering. Representing ray clouds within established rendering software is sufficiently valuable that it more than compensates for re-purposed attribute. See Table 1.

TABLE 1. Ray cloud .ply file attributes.

attribute	data type	meaning
x,y,z	float	end position
time	double	system time
nx,ny,nz	float	end-to-sensor vector
red,green,blue	uchar	point colour
alpha	uchar	intensity

We have also chosen to utilise the `alpha` value to store per-point intensity, such as lidar return intensity. Currently this comes from an `intensity` field in the imported point cloud, if one exists. In addition to its use in visualisation as per-point opacity, the `alpha` value has one functional role: any out-of-range ray is represented with an end point at the sensor's maximum range and `alpha` set to zero. This flags it as an 'unbounded ray' with a non-physical end point.

V. FILE NAMES

We have chosen to use suffixes in naming output files. This leads to default behaviour where a string of functions is visible in the file name. For instance, if we see the file `farm_decimated_smoothed_aligned.ply` then we have a good idea of how the farm map has been adjusted. Moreover, usually `farm_decimated.ply` and `farm_decimated_smoothed.ply` will be available if the user wants to adjust from an earlier point.

When file names get too long they can of course be renamed, and `farm_*.ply` can be removed to clean up the intermediates.

The output file name option `-o` is used only when there are a choice of file types that can be determined from the extension (**rayrender**) or when the output file prefix is ambiguous (**raycombine**).

VI. THE CASE FOR RAY CLOUDS

Point clouds typically represent surfaces, whereas ray clouds represent free space, with boundary surfaces. As such, tools that operate on volumes can only feasibly be performed on ray clouds. For example, ray information is essential in:

- distinguishing between a thin wall observed from both sides and a noisy wall observed from one side
- distinguishing between a solid wall and a porous fence
- estimating the density of vegetation

The clearest case of volumetric operations are the set operators, such as a set union or intersection. These are useful operators. For instance, if you map a street at two different times, the union of the two volumes of free space will exclude any pedestrians that have moved between the two times. In principle this union can be used for removing transient objects in maps. The intersection operation does the opposite, it will include all pedestrian locations, providing a 3D map of the "maximum" of all transient objects. See Figure 1. We apply these form of operators in the **raytransients** tool.

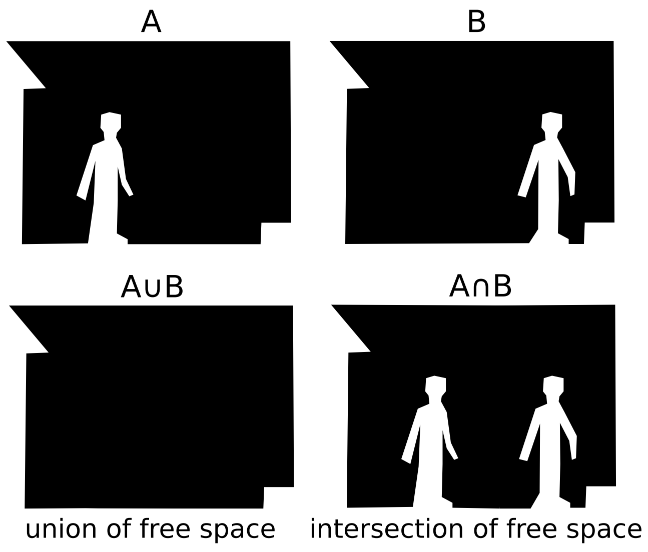


FIGURE 1. Set operations on free space allow transient object removal (bottom left), or the maximum set of objects seen (bottom right).

A. TRANSIENT OBJECT REMOVAL

The tool **raytransients** splits a ray cloud into its static and transient components, suffixed with `_fixed.ply` and `_transient.ply` respectively. A ray cloud however,

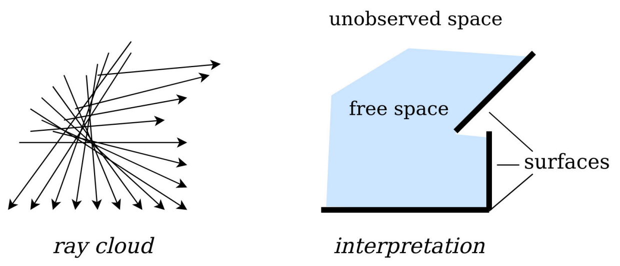


FIGURE 2. The ray cloud as a set of rays (left) typically approximates a volume of free space, unobserved space and a set of surfaces.

is not a boolean volume, instead it represents three states: free space (approximated by rays), contact surfaces (approximated by end points) and unobserved space. Consequently, the operations are modified forms of the boolean set operations, as they need to respect the unknown status of the unobserved space. See Figure 2.

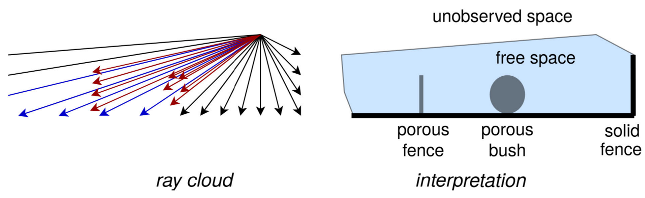


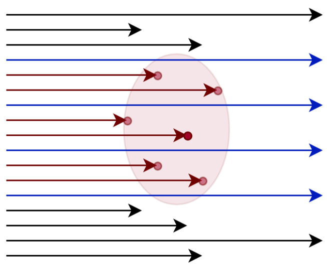
FIGURE 3. A general ray cloud interpretation identifies *porous* volumes and surfaces in addition to solid surfaces. This is required for raytransients and raycombine. The rays shown in blue pass through the vicinity of the red ray end-points, identifying them as porous regions.

Moreover, many scenes contain intermediate regions: porous volumes such as a bush, and porous surfaces such as a wire fence, as shown in Figure 3. The **raytransients** tool generalises these geometries using small ellipsoidal regions around the neighbourhood of each point in the scene (Figure 4(a)). Each ellipsoid contains an *opacity* value representing the proportion of intersecting rays that end within it. Bushes are built from multiple spherical low-opacity ellipsoids, surfaces from flat ellipsoids, and branches and wires are built from thin ellipsoids.

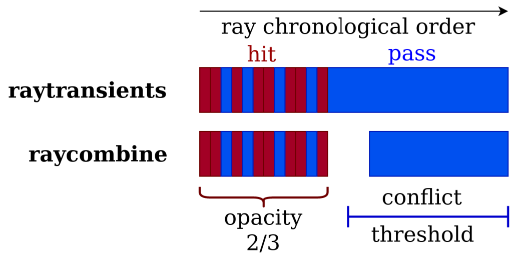
With this representation, **raytransients** looks for ellipsoids that start or end with a string of r rays passing through them. This indicates the geometry has changed, and so the point is transient (Figure 4(b)). How readily the tool classifies transients is controlled by its **n rays** argument, where transience $= r > n/\text{opacity}$. Figure 5 shows a real-world application of the tool where there is a person walking in the scan, the method removes the moving person without removing the porous row of bushes.

B. COMBINING CLOUDS

In addition to its use in transient object removal, volumetric operators provide a consistent approach to combining separate maps together. The tool for this is **raycombine**, it is the same as **raytransients** but acts on multiple clouds, rather than within one cloud. This provides the ability to support

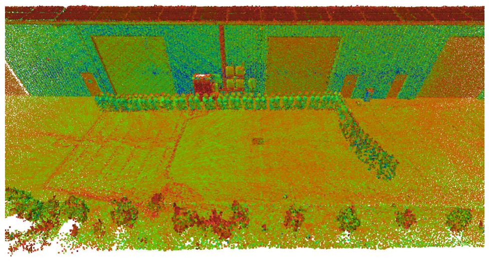


(a) The volume around the dark red point is the ellipsoid of the covariance matrix of the neighbouring points (light red). Blue rays pass through the ellipsoid, red hit inside the ellipsoid, this defines an opacity: hits/(hits+passes).

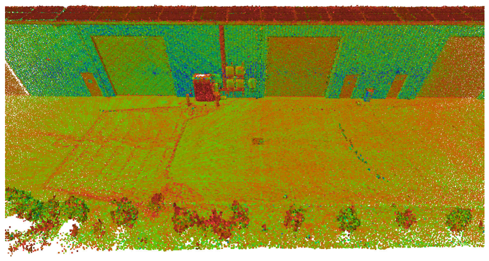


(b) The rays intersecting the ellipsoid are ordered chronologically and a conflict occurs when there are more contiguous passes (before or after the hits) than the threshold: n/opacity, from the command argument **n rays**.

FIGURE 4. Porous volume patches are estimated per-point. In the single cloud example (raytransients), sufficient rays pass through the ellipsoid at the later time that the object is deemed to have disappeared (a conflict). Depending on the merge type, the point in question is removed, or the pass through rays are removed. In the double cloud example (raycombine) there are not enough pass through rays in the second cloud to be certain that the porous object has disappeared.



(a) Person captured walking in the scan is typically undesired.



(b) **raytransients min scan.ply 10 rays** removes the changed geometry while preserving porous geometry such as the row of bushes.

FIGURE 5. Transient object removal from a single ray cloud, coloured by lidar return intensity.

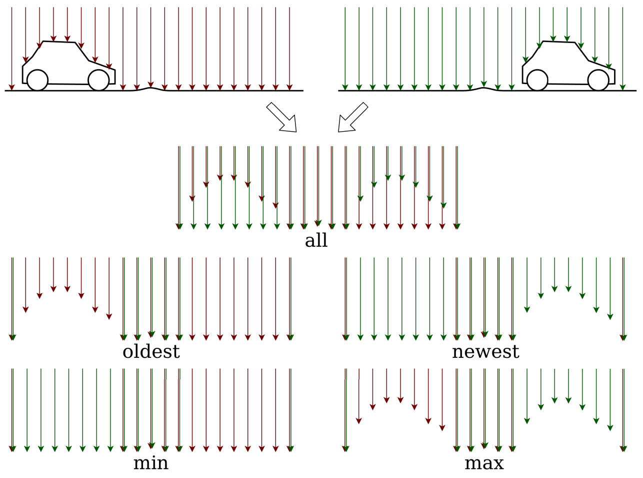


FIGURE 6. The five automatic merge types, for an aerial ray cloud of a road scanned once (red) and again later (green). If these two scans are within a single ray cloud then the merge is performed with raytransients, otherwise it is performed with raycombine.

long-term mapping, as it allows newly acquired 3D maps to be merged over previous ones in a meaningful manner. The only user decision is which merge type to employ.

The merge type only makes a difference in regions where a ray in one map passes through an ellipsoid in the other map. If we refer to these cases as merge conflicts, then we can interpret the merge type as the choice of merge conflict resolution. This is a 3D map equivalent of the merge conflict auto-resolution options that are available on text files under source control. Text options include using the old text, using the new text, or including both for manual resolution. We support using the oldest geometry, using the newest, and including both for editing in a 3D editor [1], [13]. We also support use of the minimal, and maximal geometry on merge conflicts. These options are displayed in Figure 6.

Version-controlled 3D maps allow the step-change from using isolated maps to continued mapping operations. However, there is an important operation that is still required, the three-way merge. This operation allows multiple users to edit the same map. It uses the closest common parent map to discern which map makes which change. An illustration of the method is given in Figure 7, and a real-world example of the method is shown in Figure 8, with the ray cloud end points coloured by lidar return intensity.

raycombine base.ply min cloud1.ply cloud2.ply 10 rays performs the three-way merge of cloud1 and cloud2 both onto the base cloud. The operation is a simple variant of **raycombine**:

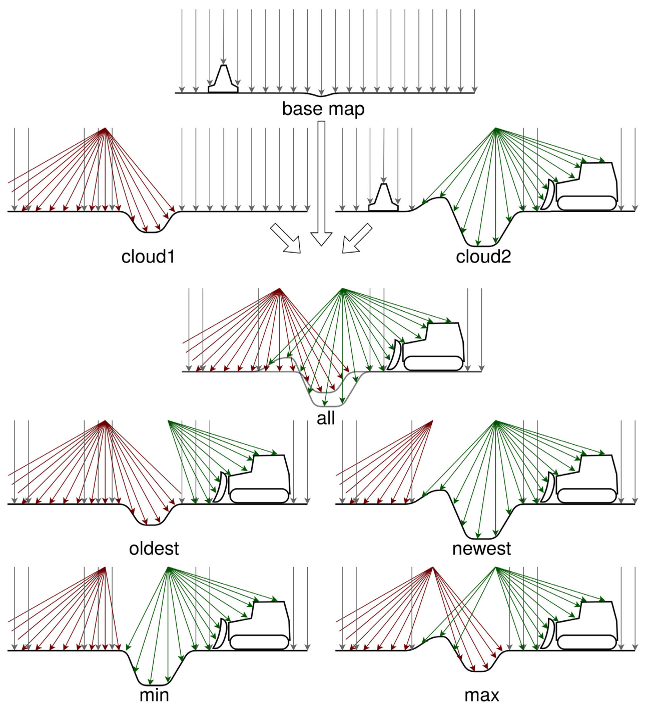


FIGURE 7. Three-way merge types. Here red and green are two small static scans that have been merged onto the base map using raycombine newest, green was captured later. We then merge the two branches using the base map. The merge types only affect what is done at merge conflicts, which are where red and green overlap. The disappearance of the traffic cone and appearance of the bulldozer are therefore in all merge types.

1) all rays that are present in all three clouds move to the output cloud
2) all rays in cloud1 and cloud2 that are present in base are removed
3) cloud1 and cloud2 are then merged with the standard **raycombine** and the result added to the output cloud

These operations can be assigned to merges involving the .ply file type. With the git[2] version control system on Linux this is done by adding to the files[3]:

```
~/.config/git/attributes:
*.ply merge=plymerge
~/.config/git/config:
[merge ``plymerge'']
name = Custom merge for ply files
driver = raycombine %O all %A %B -o %A
```

In all the above examples, the free-space volume is essential in performing the function, and ray clouds are the most direct representation of this observed space. An alternative is to store the point cloud and sensor trajectory in separate files, but every transformation and combination of maps would require the same operation to be performed on the trajectory files. Furthermore, the above merge operations allow sites to be re-mapped indefinitely, with new maps being merged in countless times per day. There would be a growing and unbounded list of trajectory files, or one giant and complex merged trajectory file. It is simpler and more robust to store each ray independently when there is the potential for this level of continued combining (and splitting) of maps.

C. SPLITTING CLOUDS

Splitting of clouds is our next example of the benefits of ray clouds. There are many ways that a user may wish to split a 3D map. Using ray clouds enables several forms of splitting that are not possible with point clouds alone, one can crop out rays:

- longer than a distance: which may contribute more to surface noise
- shorter than a distance: to remove the user from the scans
- in a certain direction: as a quick way to remove ceilings

See Figure 9. Our ray cloud tool **raysplit** allows all of these split types, in addition to the more common types such as cropping according to a bounding box or imported triangle mesh. These functions are different to their point cloud equivalent because they must split each individual ray, to maintain the validity of the output ray clouds. Split rays that no longer end within the volume are flagged as unbounded in the usual fashion, by setting their alpha component to zero.

D. ACCURATE SURFACE ESTIMATES

Lastly, ray information is important in extracting more representative local surface geometry. In particular, the ray direction allows us to differentiate between a thin wall observed from two sides, and a one-sided wall with noise, see Table 2 top row. This allows flat walls to be interpreted as flat, which improves the quality of many functions, including: smoothing, denoising, colouring and aligning clouds together. Table 2 demonstrates the difference that this more accurate surface estimate makes.

VII. MANAGING SIZE

The amount of acquired cloud data is growing rapidly every year, due primarily to hardware advances such as multi-beam lidar and depth cameras. While 5 million points was a large point cloud ten years ago, 500 million is not uncommon today.[4] This poses several problems for ray cloud processing, namely: clouds do not fit in working memory, clouds take a long time to process, and clouds cannot be visualised at interactive rates. We include three mechanisms to alleviate this issue of size:

A. DECIMATE - RESTORE

We implement a decimate-restore workflow whereby the majority of functions (X,Y,...) by the user are performed on a decimated ray cloud, in the manner of:

[2]Git is used for this example, but we note that there are more suitable version control systems for operating on data.

[3]On earlier systems the files ~/.gitattributes and ~/.gitconfig may be used instead

[4]illustrated by lidar data rates [15], such as 2010's Hokuyo: 40Hz to 2019's Velodyne Alpha Prime: 2,400Hz.

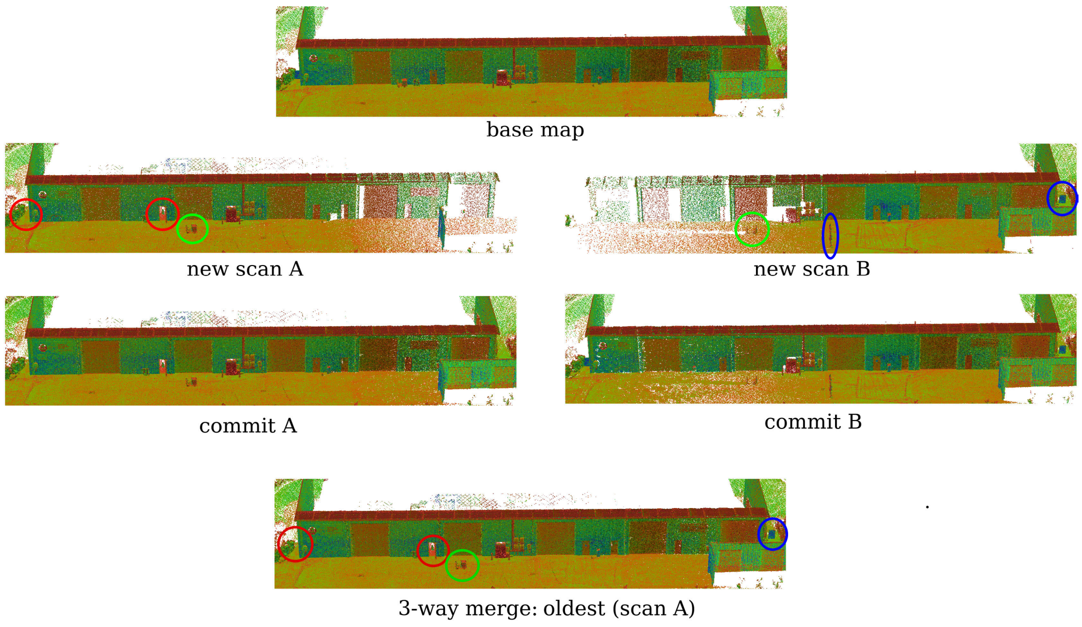


FIGURE 8. **Three-way merge on a base map of an industrial building. New scan A and B are taken from the left and right side of the building respectively. Each are merged onto the base map in commit A and B, using rayalign then raycombine newest. They contain three highlighted changes each, with the change circled in green (an added cart) common to both A and B. The three-way merged map contains both sets of changes, apart from where the changes conflict (the opened door in A and the added barrier in B), we resolve with the** `oldest` **merge type here, which keeps the open door and removes the barrier.**

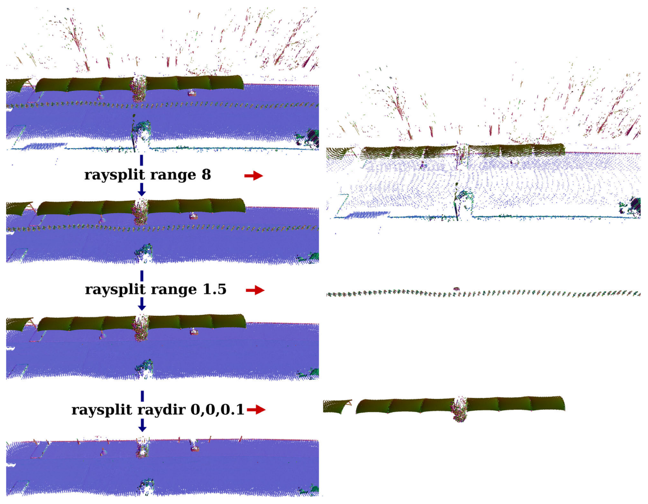


FIGURE 9. **Example raysplit operations in a car park. Top: splitting at long range removes the partially observed trees. Middle: splitting at short range removes the user from the scan. Bottom: splitting by ray direction removes the shade cloths. None of these operations are possible on only a point cloud.**

raydecimate $\rightarrow$ **rayX** $\rightarrow$ **rayY** $\rightarrow$ $\ldots \rightarrow$ **rayrestore**

raydecimate generates a subsampled ray cloud. It can either pick every n'th ray, or it picks one ray that ends in each voxel of a specified width.

rayrestore takes the processed decimated ray cloud, and applies the changes on a per-voxel (or per n'th ray) basis, to the undecimated ray cloud. For example, if **raydenoise** has removed a single ray from the decimated cloud, **rayrestore** will remove the whole voxel worth of rays from the undecimated cloud. This decimate-restore workflow is therefore coarser than operating on the undecimated cloud, but it allows the vast majority of processing to be fast, low memory and possible to visualise at interactive rates.

B. CHUNKED I/O

In order to avoid running out of working memory, a number of key tools are processed in a serial fashion, processing one chunk at a time. This is applied to all tools that are amenable to serial processing, and it includes all those that operate on undecimated clouds: **rayimport**, **raydecimate**,[5] **rayrestore** and **rayexport**. The consequence of serial processing is that the memory usage has a fixed upper bound, which is approximately 1Gb.

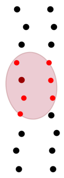
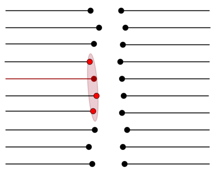
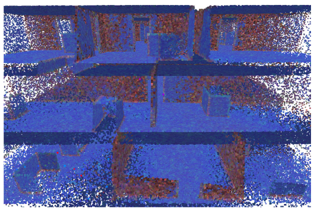
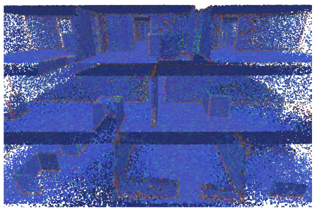
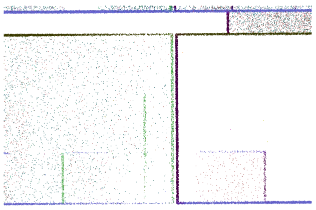
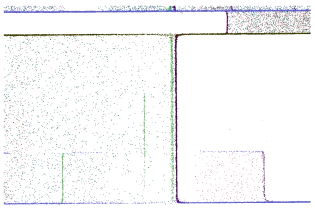
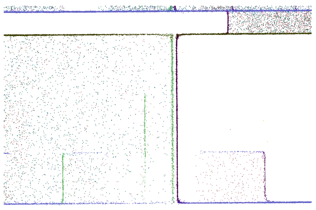
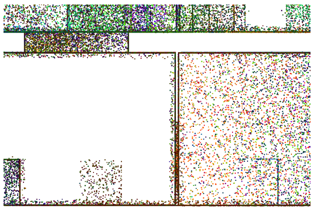
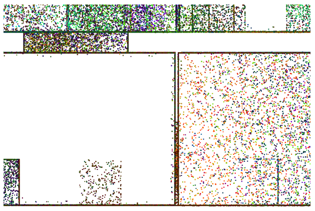
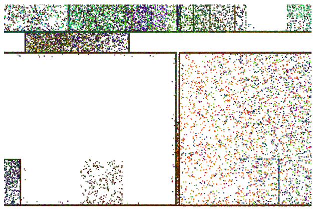
While processing of undecimated clouds can still be slow, we note that the above listed operations on undecimated clouds are fairly simple operations, which are typically run a single time. Additionally, we implement a text counter to track the progress of the function on large clouds.

C. COMPLEXITY

Under the wide range of possible cloud sizes (n rays) the performance of the tools ought to be a consistent and predictable function of n. We aim for each tool to have computational complexity O(n). This is because loading the cloud is already O(n), so it can't be any lower, and superlinear complexity

[5]The memory of spatially decimated clouds is proportional to the size of the decimated cloud

TABLE 2. Ray information allows a better representation of each point's neighbourhood. In particular, it allows two-sided walls to be represented as two flat surfaces, rather than as one thick volume. Here we show the improvement of several functions, on a building generated with several cm of range noise (using raycreate building 1).

Tool	before	point cloud only	with ray clouds
			
illustration:	on a two-sided wall the neighbourhood of the dark red point above is approximated by a thick ellipsoid, but with ray information (right) we can exclude backward facing rays, to obtain a flat surface element (surfel).		
			
raycolour:	when colouring by surfel shape our method (right) displays all blue walls (flat surfels) rather than red (thick surfels).		
			
raysmooth	our method (right) keeps two-sided walls distinct. Without ray information (middle) the two sides pull towards a single average plane.		
			
raydenoise	**2 sigmas** removes each point that is statistically more than two standard deviations away from the covariance of its neighbour points. When the covariance is excessively thick (middle, vertical wall) not as many outliers are removed.		

will require too much computation on large datasets. We also aim for memory complexity O(n), or O(1) where possible.

Since processing performance is proportional to number of rays (rather than map size, or surface area covered) the decimation function directly controls the performance of all the tools, and the degree of decimation allows users to trade off resolution with computational cost.

In addition to the three cited mechanisms, we also include a **raysplit** option to split the cloud into a grid of separate files. This can be a useful final step in the processing pipeline, for working with very large files that may need to be streamed in from a server [16]. Finally, we note that the Stanford polygon file format (.ply) is chosen for its ubiquity and ease of use. But once the ray clouds are fully processed, they can be exported to a point cloud in more compressed formats,[6] and as a separate trajectory file. These can make preferable long-term-storage formats, or data transfer formats.

VIII. THE FULL SUITE OF TOOLS

We will now describe the full suite of seventeen tools, which are summarised in Table 3. Most of these fall into natural pairs or triplets (see Figure 13) so we will describe these first, followed by the remaining ones.

A. IMPORT/EXPORT

The **rayimport** tool takes a specified point cloud and trajectory file and generates a ray cloud file. The point cloud may

[6]directly as a.laz file, or further compressed, for instance as a DRACO file [17]

TABLE 3. Overview of the full list of tools with typical command-line arguments.

Tool	Example arguments	Description
rayalign	cloud.ply	axis-aligns cloud to its principle planes (or aligns to a second cloud)
raycolour	cloud.ply time	colours the cloud based on colour type (e.g. by time) with optional lighting (--lit)
raycombine	all cloud1.ply .. cloudN.ply	combines together multiple clouds (or 3-way merge if base cloud supplied)
raycreate	building 1	generation of example clouds: rooms, trees, forests and buildings, using random seed 1
raydecimate	cloud.ply 10 cm	spatially sub-sample the cloud, e.g. one point per 10 cm wide voxel
raydenoise	cloud.ply 4 cm	remove outlier end points, e.g. points more than 4 cm from any other
rayexport	cloud.ply points.laz traj.txt	convert to a point cloud and trajectory of start points
rayextract	terrain cloud.ply	estimate geometric information about the cloud
rayimport	points.laz traj.txt	import from a point cloud and trajectory file
rayrender	cloud.ply top ends	render cloud to an image file
rayrestore	decimated.ply 10 cm full.ply	apply changes to decimated.ply onto full.ply
rayrotate	cloud.ply 0,0,90	rotate around the origin by the supplied rotation vector in degrees
raysmooth	cloud.ply	pull points towards their local surface
raysplit	cloud.ply range 10	split a cloud on a specified condition, e.g. ray length of 10 metres
raytransients	min cloud.ply 10 rays	remove moving objects from a scene
raytranslate	cloud.ply 10,0,0	translate the cloud by the supplied vector, in metres
raywrap	cloud.ply upwards 0.01	generate a highest lower-bound mesh of cloud, using penetration curvature 0.01

be a .laz file or .ply file, and the trajectory file is a text file which lists the position (and orientation) of the sensor for each time stamp in order. These set of times are independent of the time stamps in the point cloud, and are interpolated to generate the start point for each ray.

The **rayexport** command converts the ray cloud back into a separate point cloud (.ply or .laz) and trajectory file (.ply or .txt). This allows the user to view the sensor trajectory, and the point cloud file is also a smaller file.

B. DECIMATE/RESTORE

Described in section VII-A

C. DENOISE/SMOOTH/TRANSIENTS

There are three methods for addressing noise and clutter in ray clouds, one can remove the rays with outlier end points (**raydenoise**) or you can shift these end points onto the nearest surface (**raysmooth**). Lastly, you can remove geometry that is not consistent over time (**raytransients**).

raydenoise cloud.ply 4 **cm** removes rays with end points more than the specified distance from other points. The drawback of this is that distant and sparsely scanned regions will also have their rays removed. For these cases, the command **raydenoise cloud.ply** 2 **sigmas** will remove each ray with end point more than the specified number of standard deviations from its nearest neighbour points, represented as a covariance. Table 2 provides an example of this.

raysmooth shifts each ray's end point onto the nearest estimated surface, as shown in Table 2. This is the only tool that **rayrestore** does not support. So if you decimate a cloud and smooth it, then rayrestore will not apply that smoothing to the original cloud, and will not function correctly.

raytransients was described in Section VI. It splits a ray cloud into the rays corresponding to geometry that has changed, and the rays corresponding to static geometry. This is based on the choice of merge type. Typically, the `min` merge type is used, and the static (.._static.ply) output cloud retained, this has the effect of removing transient objects from the scene.

D. SPLIT/COMBINE

raysplit splits a ray cloud around a given criterion into two ray clouds. The criterion may be the equation of a plane in space (given by a single vector), it may be a point in time, or a range value for the ray. There are multiple criteria, and they are typically used to discard parts of the map that are not required. See Figure 9 for examples.

The most versatile option splits the map based on the specified signed distance from an input polygonal mesh, which is a .ply file. This allows parts of the map to be cut out, based on any shape defined in a 3D modelling package.

raycombine does the opposite, it takes multiple ray clouds and combines them into a single cloud. The merge type parameter defines which of the rays from each cloud are included in the final ray cloud when there is a conflict (see Section VI-B for details). The simplest merge type `all` includes every ray, and so is equivalent to a concatenation of the two sets of rays. The other merge types are illustrated in Figure 6.

E. TRANSLATE/ROTATE/ALIGN

The two manual transformation functions **raytranslate** and **rayrotate** act in-place; they are the only tools that modify the supplied ray cloud directly. This is because they both have a simple and lossless inverse transformation, in order to undo any mistakes. This also supports a trial-and-improvement method of aligning a ray cloud, without generating new files each time. The interfaces are simple:

raytranslate cloud.ply 1,2,3
rayrotate cloud.ply 0,0,90

The rotate command argument is a rotation vector in degrees: its axis is the axis of rotation and its magnitude is the rotation angle in degrees. We consider this rotation representation to be the most user friendly, without over- or under-parameterising the rotation.

As well as manual alignment, we can also automatically align a cloud using the **rayalign** tool. When the tool is used on a single cloud, it axis-aligns the major orthogonal planes in the cloud. For example, translating the bottom corner of the largest external walls of a building to the origin, and axis-aligning them. Typically the ground acts as the principle horizontal plane, and so takes a height of zero. The major orthogonal planes are extracted using a radon transform [18] technique, by discretising the cloud to a horizontal grid of end-point densities. We therefore are making the assumption that the clouds are already vertically aligned. We have found this to be a fair assumption for mapping sensors that contain an Inertial Measurement Unit (IMU).

When **rayalign** specifies a second cloud file, it aligns the first cloud onto the second. The alignment is based on a cross-correlation of the density of end points, with respect to 3D translation and yaw angle.[7] As such it is recommended that clouds are spatially decimated so that density represents the geometry rather than time spent observing an object. The peak correlation is used as the coarse alignment, from where it performs a gradient-descent minimisation of the difference between nearby surfaces. See Figure 10 for example. This second stage generates a rigid transformation of best fit by default, but can also generate a non-rigid (quadratic) transformation under the *--nonrigid* option. This latter option is primarily useful for long maps that may have a small amount of bend along their length.

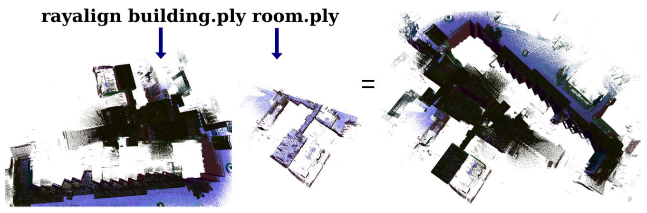


FIGURE 10. Example of aligning the large cloud onto the smaller cloud, for a scene including rooms and outdoor areas.

F. EXTRACT/CREATE

rayextract is the most experimental of the ray cloud tools.[8] It is for extracting meaningful real-world geometries from a ray cloud. Currently this includes extraction of rough terrain and extraction of forests. The terrain is extracted as a mesh, and forests are extracted as text files containing the estimated location and size of trees.

The **raycreate** tool acts in the opposite fashion, it takes in a reduced description (a single random seed, a mesh, or forest text file) and generates a ray cloud to fit this description. The current example types are: tree, forest, terrain and building.

G. COLOUR

raycolour colours the end points of the ray cloud based on the specified criterion. This may be based on the time stamp, the local shape, the normal direction, lidar intensity or several other options, including choosing a uniform colour directly. The **--lit** option provides lighting based on a diffuse illumination model, in order to aid in visual interpretation of the 3D scene. A good default colouring is **normal --lit**, but **time --lit** is useful in analysing how a scan unfolded, it cycles from red to green to blue every minute of the scan.

H. RENDER

rayrender renders the ray cloud to an image from the chosen view. The ends of each ray can be rendered to display the map, or the start points can be rendered to show the trajectory. The full rays can also be rendered, to show the free-space volumes. A particularly useful render option for vegetation is to colour the map by surface area density, see [20] for method. Real, unscaled values can be obtained by rendering to a High Dynamic Range (.hdr) file. See Figure 12 for a visual overview.

I. WRAP

The process of converting point clouds to meshes is an ambiguous one in general, but one process that is well-defined is ‘wrapping’. We perform this on the surfaces (the end points) of a ray cloud, as shown in Figure 11.

raywrap takes a ray cloud, a direction type and a curvature value, and wraps the cloud in the given direction up to the given curvature of penetration. The direction types are *inwards*, *outwards*, *upwards* and *downwards*. The *inwards* type is closest to our notion of wrapping, when the curvature is zero it generates the convex hull of the end points of the ray cloud. For larger curvatures it acts like an increasingly tight vacuum packing of a surface to the cloud. It generates a concave hull up to the specified curvature.

Direction type *outwards* inflates a closed surface onto an enclosed space such as a room or cave, up to the specified penetration curvature. *upwards* wraps an open surface up to the cloud from below, which is particularly effective at extracting the ground surface underneath surface geometry such as grass, trees or stones. *downwards* wraps an open surface onto the cloud from above, which is useful in extracting forest canopy shape, or modeling the obstacles that a drone might fly into. These all use the *--full* wrap option, which can be processor intensive.

The default (non-full) option uses a quadratic approximation to the above functions, based around the visibility methods of Katz and Tal [21]. This is much faster, but less complete as it ignores ‘overhangs’ in the cloud geometry. Take for instance the *upwards* direction, the curvature value no longer represents a spherical curvature but the vertical curvature of a paraboloid (the second differential of height with respect to lateral distance). The mesh is the set of points that such an upwards probing paraboloid can reach, and as such the mesh will not contain any overhangs. It represents a form of highest lower-bound surface for the ray cloud [20].

In all cases the resulting mesh is ‘well behaved’: it is a triangular mesh that is topologically a sphere (*inwards*, *outwards*) or a plane (*upwards*, *downwards*). It contains no holes,

[7]This is a form of Fourier-Mellin technique, similar to the ideas of Rufus *et al.* [19].

[8]as such, it currently resides in the experimental branch of the RayCloudTools software repository.

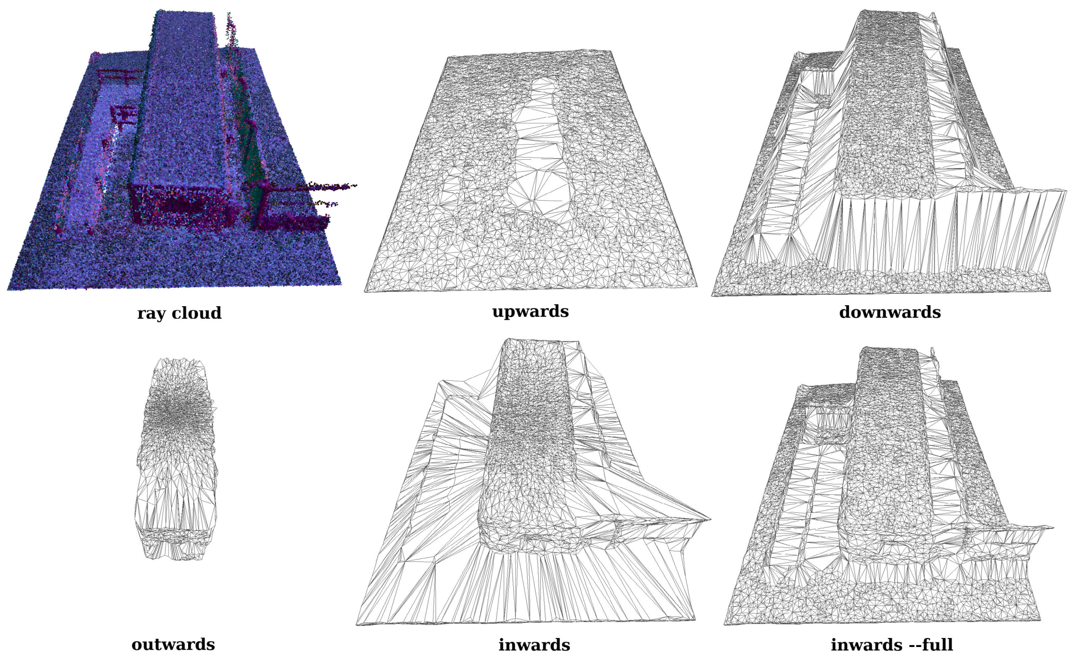


FIGURE 11. **Raywrap: applied to a cabin beside a fence, showing the four wrap directions. These project the mesh in vertical and radial directions. The '--full' option projects iteratively in each triangle's normal direction, which allows more thorough wrapping, at a greater computational cost.**

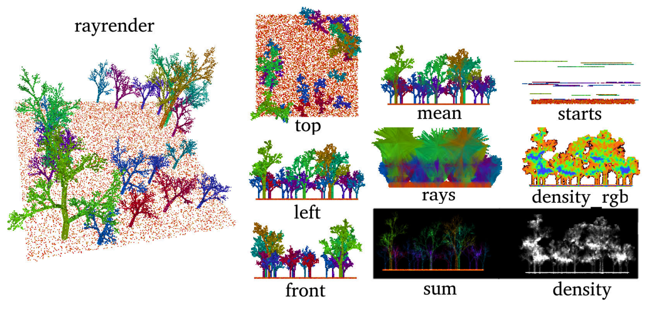


FIGURE 12. **Rayrender: multiple views and render types displayed. Cloud example was generated with raycreate forest.**

split edges or branching surfaces, and thin triangles are very rare as the process is related to the Delauney triangulation.

IX. SEQUENCES

These tools are useful individually, but their main strength is when combined together in sequences. We present a visual example of how functions could be sequenced in Figure 13. In this figure, we visualise the ray cloud tools as the interface between raw 3D maps (top) and the user information (bottom). Analysis tasks tend to flow downwards, interpreting the ray clouds into higher level information for the user, this may involve segmenting and splitting the maps. Conversely, manipulation tasks flow upwards, taking information from the user to generate, modify and construct 3D maps.

We now present some more specific use cases of sequenced operations, which are useful in a variety of scenarios. In each case we assume that **rayimport** has already been used to convert into the ray cloud format. We also rename the output cloud name to cloud.ply after each command is called, in order to simplify the presentation of these sequences.

A. CLEAR VISUALISATION

This cleans the data and colours it for clear viewing.

1) **raysplit cloud.ply alpha 0** - remove zero-intensity (unbounded) rays
2) **raysplit cloud.ply plane 0,0,2** - remove ceilings so you can see in
3) **raydecimate cloud.ply** 4 **cm** - even density
4) **raydenoise cloud.ply** 3 **sigmas** - remove outliers
5) **raytransients min cloud.ply 10 rays** - remove moving objects
6) **raysmooth cloud.ply**
7) **raycolour cloud.ply normal --lit**

B. LONG-TERM MAPPING

We have a large map.ply ray cloud being maintained, and wish to merge a newly acquired cloud on top.

1) **raydecimate cloud.ply** 4 **cm**
2) **raytransients min cloud.ply 10 rays** - remove moving objects
3) **rayalign cloud.ply map.ply** - fit onto the existing map
4) **raycombine newest cloud.ply full_map.ply 10 rays**
5) **raydecimate map.ply** 4 **cm** - prevent density increasing
6) commit changes

TABLE 4. Existence of similar tools within existing open source point cloud packages, coloured by the number of tool parameters. The table illustrates the low cognitive load of our method (green colouring), the small number of functions to learn ('Functions' row) and the clarity of function naming.

	RayCloudTools	CloudCompare	LasTools	Open3D	PCL
Interface	command-line C++	command-line	command-line C++	Python C++	C++
Paper	2021	2003	2004	2018	2010
Tool	rayalign	-ICP (local only)		registration_*[a]	Correspondence*[b]
	raycolour	-PCV		paint_uniform_color	
	raycombine	-MERGE_CLOUDS			
	raycreate				
	raydecimate	-SS	lasvoxel	voxel/uniform_down_sample	VoxelGrid[c]
	raydenoise	-SOR		statistical/radius_outlier_removal	*OutlierRemoval[d]
	rayextract		lasclassify		
	rayrender	-RASTERIZE	las2den/lasgrid		
	rayrestore				
	rayrotate	-APPLY_TRANS		rotate	
	raysmooth				MovingLeastSquares
	raysplit	-FILTER_SF/CROP	lasclip/lassplit	crop_point_cloud	Crop*/*Clipper3D[e]
	raytransients				
	raytranslate	-APPLY_TRANS		translate	
	raywrap		lasground	compute_point_cloud_convex_hull	ConcaveHull
Functions	17	69	26	+200	+200

Independent parameters: 🟩🟩🟩🟩🟩🟨🟨🟨🟨🟨🟧🟧🟧🟧🟧🟧🟧🟧🟧🟧🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

[a] registration_icp/registration_fast_based_on_feature_matching/registration_ransac_based_on_correspondence/registration_ransac_based_on_feature_matching
[b] Local: GeneralisedIterativeClosestPoint/Registration base class. Global: Keypoint, Feature, CorrespondenceEstimationBase, CorrespondenceRejector, TransformationEstimation
[c] ApproximateVoxelGrid/UniformSampling/VoxelGrid
[d] RadiusOutlierRemoval/StatisticalOutlierRemoval
[e] CropBox/CropHull/BoxClipper3D/PlaneClipper3D

C. TRAVERSIBILITY

Obtain a mesh that defines the traversibility of the terrain.

1) **raydecimate cloud.ply** 4 **cm** - even density
2) **raywrap cloud.ply upwards 0.1** - get ground mesh
3) **raysplit cloud.ply cloud_mesh.ply distance 0.5** - get first 50 cm above ground
4) **raywrap cloud.ply downwards 2.0** - mesh of ground obstacles

The slope of the output mesh triangles gives a gradient cost, and the circumradius of each triangle represents the size of the (paraboloidal) concavity that sits between its three corners. Graph-based planning can be performed on the mesh, based on these two costs.

X. EASE OF USE

In addition to its functionality, a library should be easy to use if it is to aid as large an audience as possible. This requires its interface to be a low cognitive burden on the user. Poorly named interfaces with a large set of functions demand a heavier learning effort of the user. Similarly, dozens of parameters included per function leads to a wide search space, taking time to search, understand, and placing the burden on the user to find the solution that works.

With this in mind, we compare our interface in Table 4 against the four established cloud libraries: CloudCompare, LasTools, Open3D and PCL.

Cognitive burden is challenging to quantify, we acknowledge this by providing a coarse colouring per tool based on the number of independent parameters that are presented to the user, from under five (green) to over 20 (red). Under this metric, an enumeration such as top/front/side is one parameter, and a 3D vector is three independent parameters. For the LasTools library, all optional parameters listed under ‘‘overview of all tool-specific switches’’ on the web site are included. For Open3D and PCL, the C++ interfaces make parameter counting more complicated as there may be several functions for the same tool, or multiple functions may be required in sequence. In both cases, we sum the parameter degrees of freedom for each function, or for each modifier function when the tool is wrapped in a class.

Additionally, we measure the interface size in the bottom row of Table 4, this enumerates the total number of functions that are presented to the user, either on the library's web site or in documentation. For the libraries that have no command-line interface we use the number of available function calls, in both cases there are well over two hundred.

Lastly, we note that functions that rely heavily on the full ray cloud (**raycombine**, **raytransients**) are not present in Table 4 for the point cloud based libraries (only the simple concatenation of clouds is supported in CloudCompare), the useful **raycreate** and **rayrestore** functions are also missing in the point cloud libraries. These demonstrate

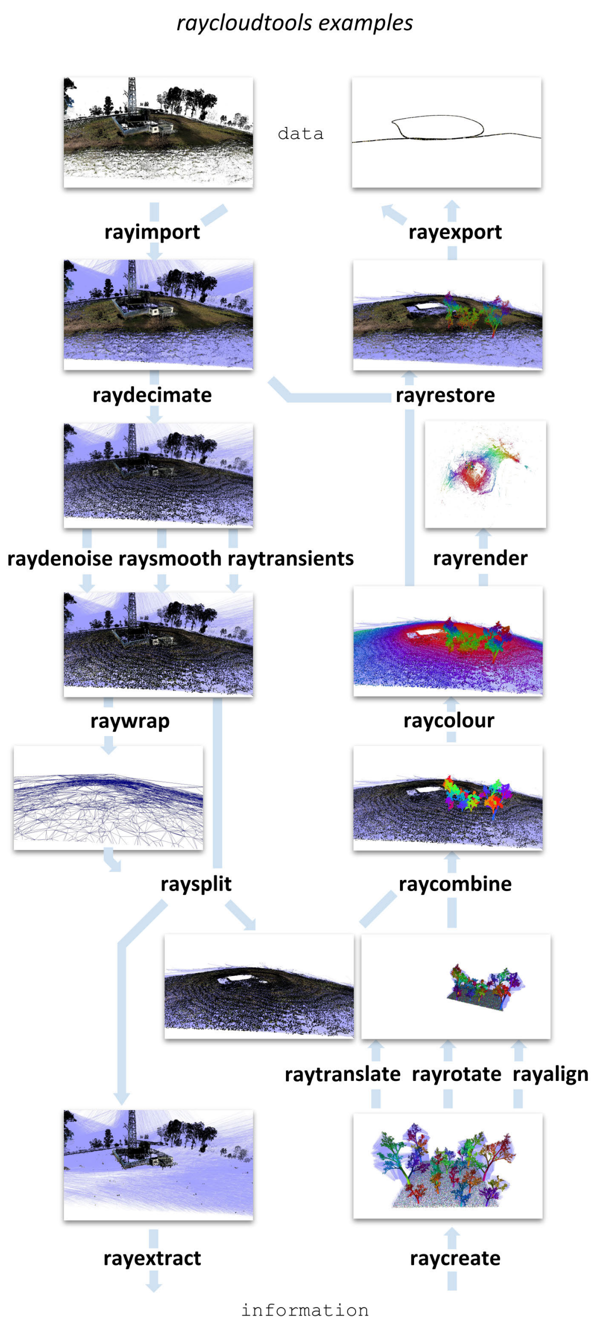


FIGURE 13. Typical sequences, combined in a single image. In this schematic, analysis flows downwards from raw data to information, and manipulation flows upwards. The example cloud is of a hill with trees and huts, and the manipulation involves replacing the structures on the hill with generated trees.

the functionality gap that ray cloud processing is able to fill.

XI. CONCLUSION

We have described a toolset for easy-to-use fundamental manipulation and analysis of ray clouds. We have shown that the choice of ray cloud as a format supports crucial functions for long term mapping, and that ray information is valuable for many of the tools. In comparison to established libraries (Table 4) we have shown both the similarities and differences against the toolset that we provide, and have provided measures that indicate a relatively low cognitive burden for our toolset.

It is our hope that this toolset will encourage people to adopt the ray cloud as the primary data structure for 3D mapping, and will aid in making ray cloud processing available to a broad community. We also hope that these tools will encourage the native support for ray clouds within existing cloud visualisation software.

APPENDIX
PARAMETER COUNTS

As a record, the individual parameter counts from Table 4 are given here in row order, with multiple functions remaining as sums for convenience:

RayCloudTools: 3, 2, 4, 2, 2, 2, 5, 4, 3, 3, 0, 5, 3, 4, 2
CloudCompare: 10, 4, 0, 2, 2, 11, 9, 2 + 6, 3
LasTools: 20, 28, 37 + 71, 16 + 24, 62
Open3D: 14 + 3 + 6 + 8, 3, 1 + 1, 2 + 2, 5, 6, 3, 0
PCL: 8 + 13 + 4 + 4 + 7 + 3 + 0, 3 + 1 + 11, 5 + 2, 15, 9 + 14 + 4 + 4, 4

ACKNOWLEDGMENT

The authors would like to thank Riley Bowyer for his efforts in scanning fieldwork, and in supporting the low level requirements of this software.

REFERENCES

[1] D. Girardeau-Montaut, "Détection de changement sur des données géométriques tridimensionnelles," M.S. thesis, Laboratoire Traitement et Communication de l'Information, Ecole Télécom Paris, Paris, France, 2006.
[2] C. Hug, P. Krzystek, and W. Fuchs, "Advanced LiDAR data processing with lastools," in *Proc. 10th ISPRS Congr.*, 2004, pp. 12–23.
[3] R. B. Rusu and S. Cousins, "3D is here: Point cloud library (PCL)," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2011, pp. 1–4.
[4] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A modern library for 3D data processing," 2018, *arXiv:1801.09847*. [Online]. Available: http://arxiv.org/abs/1801.09847
[5] K. Zampogiannis, C. Fermuller, and Y. Aloimonos, "Cilantro: A lean, versatile, and efficient library for point cloud data processing," in *Proc. 26th ACM Int. Conf. Multimedia*. New York, NY, USA: ACM, Oct. 2018, pp. 1364–1367.
[6] A. Bell, B. Chambers, and H. Butler. *Point Cloud Abstraction Library*. [Online]. Available: https://github.com/PDAL/PDAL
[7] D. Borrmann, A. N. Josch. *3Dtk—The 3D Toolkit*. [Online]. Available: http://slam6d.sourceforge.net/
[8] D. de la Iglesia Castro. *Pyntcloud*. [Online]. Available: https://github.com/daavoo/pyntcloud
[9] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *Proc. IEEE Int. Conf. Robot. Autom.*, vol. 2, Mar. 1985, pp. 116–121.
[10] C. Schulz and A. Zell, "Sub-pixel resolution techniques for ray casting in low-resolution occupancy grid maps," in *Proc. Eur. Conf. Mobile Robots (ECMR)*, Sep. 2019, pp. 1–6.
[11] J. P. Saarinen, H. Andreasson, T. Stoyanov, and A. J. Lilienthal, "3D normal distributions transform occupancy maps: An efficient representation for mapping in dynamic environments," *Int. J. Robot. Res.*, vol. 32, no. 14, pp. 1627–1644, Dec. 2013.
[12] J. Ahtiainen, T. Stoyanov, and J. Saarinen, "Normal distributions transform traversability maps: LiDAR-only approach for traversability mapping in outdoor environments," *J. Field Robot.*, vol. 34, no. 3, pp. 600–621, May 2017.

[13] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia, "MeshLAB: An open-source mesh processing tool," in *Proc. Eurograph. Italian Chapter Conf.*, Salerno, Italy, vol. 2008, 2008, pp. 129–136.
[14] Connor Manning. *Entwine*. [Online]. Available: https://github.com/connormanning/entwine
[15] Hexagon. *LiDAR Comparison Chart*. [Online]. Available: https://autonomoustuff.com/lidar-chart
[16] M. Schütz, "Potree: Rendering large point clouds in Web browsers," Ph.D. dissertation, Institut für Computergraphik und Algorithmen, Technische Universität Wien, Vienna, Austria, 2015.
[17] Chrome Media. *Draco 3D Data Compression*. [Online]. Available: https://github.com/google/draco
[18] S. Helgason and S. Helgason, *The Radon Transform*, vol. 2. Boston, MA, USA: Springer, 1980.
[19] N. Rufus, U. Krishnan R. Nair, A. V. S. S. B. Kumar, V. Madiraju, and K. M. Krishna, "SROM: Simple real-time odometry and mapping using LiDAR data for autonomous vehicles," 2020, *arXiv:2005.02042*. [Online]. Available: http://arxiv.org/abs/2005.02042
[20] T. Lowe, P. Moghadam, E. Edwards, and J. Williams, "Canopy density estimation in perennial horticulture crops using 3D spinning lidar SLAM," *J. Field Robot.*, vol. 38, no. 4, pp. 598–618, Jun. 2021.
[21] S. Katz and A. Tal, "On the visibility of point clouds," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1350–1358.

THOMAS D. LOWE was born in Essex, U.K., in 1978. He received the degree (Hons.) in computer science from the University of Warwick, Coventry, U.K., in 1999.

From 2000 to 2005, he developed 3-D video game mechanics and game engines for Krome Studios, Brisbane, Australia. From 2005 to 2011, he developed physically driven humanoid behaviours for NaturalMotion Ltd., Oxford, U.K. Since 2012, he has been a Senior Experimental Scientist with CSIRO, Brisbane, where he has helped develop mobile mapping (SLAM) systems that have been commercialized as the Zeb and Revo devices through GeoSLAM Ltd., and Emesent Pty Ltd. He is the author of the book *Exploring Scale Symmetry* (World Scientific), and of multiple articles through journals, including *JFR* and *RAL*. His ongoing research interests are in ray cloud analysis and robot dynamics.

Mr. Lowe was a recipient of the Eureka Award for Innovative Use of Technology, in 2013, and has received one national and two state I Awards, in 2020. In 2014, he received the CSIRO Medal for Scientific Excellence.

KAZYS STEPANAS received the B.Eng. degree in computer engineering from the University of Canberra, in 1998. He has over a decade of industry computer engineering experience, primarily in the video games industry, specializing in software architecture. He has spent another decade applying his industry experience to robotics research and development. He is a champion of software best practice, improving software, and developing software support tools.