Finding Planes in Point Clouds

COSC450 Assignment 1

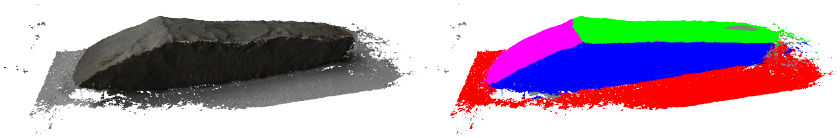
Due: 9th April 2018, 12 midnight

This assignment is worth 20% of your final grade.

1 Overview

Many methods for making 3D models of the world create *point clouds*. These are collections of unconnected points, although they may have information such as colour, normal estimates, etc. associated with them. Such models can come from LiDAR, stereo, or consumer depth sensors such as the Kinect.

While methods exist to estimate a surface that passes through, or approximates, a point cloud, many types of scene or object have large flat surfaces. These are well approximates by planes, and the goal of this assignment is to identify the dominant or significant planes in a point cloud model. For example, here is a model of *toki* (a Māori stone adze) created using multi-view stereo methods, and the main planes identified in it:



2 Plane Finding

Suppose we knew three points on a plane in the scene. We could fit a plane to those points, and then find all the other points that lie near that plane. The problem is that we do not have the initial set of points. One solution for this is Random Sample and Consensus (RANSAC) [3]. Three points are chosen at random, and we assume that these lie on a plane. If that is really a plane in the scene then there will be a lot of other points that lie close to that plane. The three points that are chosen at random define a model (the plane), and those points which lie close to the plane are known as the *consensus set* for that model.

In RANSAC we generate many models at random, and then accept the one with the largest consensus set. In the case of fitting planes to point clouds, we select many sets of three points at random, and accept the one with the largest

consensus set. The points in that consensus set are assigned to the first plane in the scene.

Once we have one plane, we can remove the points that are explained by that model, and then look for the next largest plane. Applying this iteratively allows us to find all of the planes in a scene. This gives us the following algorithm:

```
Input: X = {x_1, x_2, ..., x_n}, a set of 3D points
       P the number of planes to find
       T the point-plane distance threshold
       R the number of RANSAC trials
for p = 1 to P:
    bestPlane = {0,0}
    bestPoints = {}
    for r = 1 to R:
        S = {s_1, s_2, s_3} = 3 points at random from X
        thisPlane = fitPlane(|$S$|)
        thisPoints = {}
        for x_i in X:
            if (distance(thisPlane, x_i) < T):
                thisPoints = thisPoints + x_i
        if |thisPoints| > |bestPoints|:
            bestPlane = thisPlane
            bestPoints = thisPoints
    output bestPlane
    X = X - bestPoints
```

2.1 Improving the Algorithm

The basic RANSAC algorithm has several parameters that need to be provided. Choosing these can be difficult, and ideally the algorithm would be able to determine these automatically.

The number of RANSAC trials (R in the algorithm) can be determined based on a probabilistic approach. It is possible to choose R so that the probability of not finding the largest plane in a scene is less than some threshold probability, say 1% or 0.1%. Details of how to do this are given on the Wikipedia page for RANSAC.

Determining the number of planes, P, and the plane-point threshold, T is more difficult. Here are some ideas, but you might have some of your own:

- For P you could keep finding planes until some percentage of the scene is explained by planes.
- Alternatively, you could keep generating planes until the next plane is too small (has a small consensus set).
- For T you could look at some small local groups of points. Suppose you examine all points in a small region. These probably lie on a surface, and

locally most surfaces can be approximated by a plane. The maximum distance of these points from the best fit plane might be used as T.

Note that in many cases this just moves the definition of the parameter. If planes are generated until the next suggested plane is too small, we need a definition of 'too small'. However, this might be easier to estimate than the number of planes in a scene – particularly if we're not in a position to examine every scene before processing.

Also, some parameters are defined in terms of a number of points. For example, suppose you generate planes until the next one has less than 50 points. This threshold might be fine for a scene with 500 points, but what about a scene with 1,000,000 points? It may be better to define the threshold as a proportion of the scene, such as 10% of the total number of points.

Another way to improve the algorithm could be to do further analysis of the planes that are generated. Again, there are a number of different options here:

- You could make a better fit plane by applying a least-squares method to the consensus set.
- You could move the points so that they lie on the fitted planes, removing noise in the data.
- You could use the fact that the points are well approximated by a 2D co-ordinate frame to apply a triangulation technique (such as Delaunay triangulation) to the fitted points.
- You could explicitly represent the planes to make a simplified model of the scene. This would start by finding the extent of the planes (such as a bounding rectangle or the convex hull), but could then be further extended to find the intersections of the planes as edges of a polygonal model of the scene or object.

You could also look for other types of surface in a scene, such as spheres or conic sections. The challenge here would be to decide what range of surfaces to look for and to decide how to decide what to look for. How would you find surfaces in a scene that might contain a mixture of spheres, cylinders, and planes?

Yet another possible extension is to explore alternatives to RANSAC. Many alternatives and modifications have been suggested, aiming at making the algorithm faster, more robust, or more accurate. Such variants include MLESAC[5], NAPSAC [4], PROSAC [2], and BaySAC [1].

3 Requirements

A sample program is provided which reads information from a file in the ASCII version of the PLY format, recolours them according to some algorithm, and then writes them to a new file. The sample code just colours the points according

to their order in the file, and *the main task for this assignment is to modify this program to colour points according to the planes that you find in the data.*

The core requirements for your program are:

- To implement a RANSAC algorithm to find planes in point cloud data sets and recolour the points appropriately.
- To implement the probabilistic method for determining the number of RANSAC iterations.

The output of your program should be a point cloud with the planes identified by colour. Points in the data set that do not belong to any plane in the data set should retain their original colour.

Once you have implemented the program you should conduct a series of experiments to demonstrate its usefulness and limitations. These experiments should investigate the effects of different parameters on your algorithm. Key questions you should attempt to address include:

- What is the useful range of values for a given parameter?
- How sensitive are the results to changes in a parameter's value?
- Do the same parameter values work well for different data sets, or do they depend on the input point cloud?
- If they depend on the input point cloud, how can you decide on a good value?
- Are there any interactions between the parameter values, or can they be set independently?

You should present the results of these experiments in a short report to be handed in along with your code.

3.1 Extensions

For very high marks you should implement some extension beyond the basic plane detection algorithm described above. Some suggestions for these are given above, but you should be aware that some are more complicated than others. Very simple extensions (such as generating planes until 90% of the points are explained) may not earn high marks. Others, such as generating full polygonal models from the scene, are beyond the scope of this assignment. Extensions which are about the right size could be:

- Implementing methods for automatically estimating both P and T (as well as R, which is a core requirement).
- Finding a best fit plane through the consensus set and using this to triangulate the points (a third party library could be used for Delaunay triangulation).

- Extending the algorithm to look for different types of surfaces in addition to planes.
- Implementing an alternative to RANSAC and comparing its results to the basic RANSAC approach.

Note that (as usual) you get diminishing returns here, and so focussing on doing a good job of the core implementation, experimentation, and report writing is recommended.

4 Resources

Source code for a skeleton of the assignment is available on the departmental servers in `~steven/Public/COSC450/PlaneFinder`. This skeleton program reads a PLY file and re-colours the points but does not detect planes in the data, but should be used as the basis for your assignment. You may wish to change the parameters which the program takes, and this should be clearly documented both in the code and your report.

A number of PLY files will be made available on the website as well. You should not restrict yourself to this set. Other models may be found online, or you may wish to write a program which generates simple models for testing. Such generated models can be useful for testing specific cases such as a scene with a particular number of planes; scenes with varying numbers of points; or with different levels of noise in the 3D co-ordinates. If you write a generator, then discuss what it does in your report.

Additional PLY files can also be made with 3D sensors such as the Kinect, or from images via multi-view stereo. If you have a specific scene or object you'd like captured then email me (steven@cs.otago.ac.nz) to sort that out.

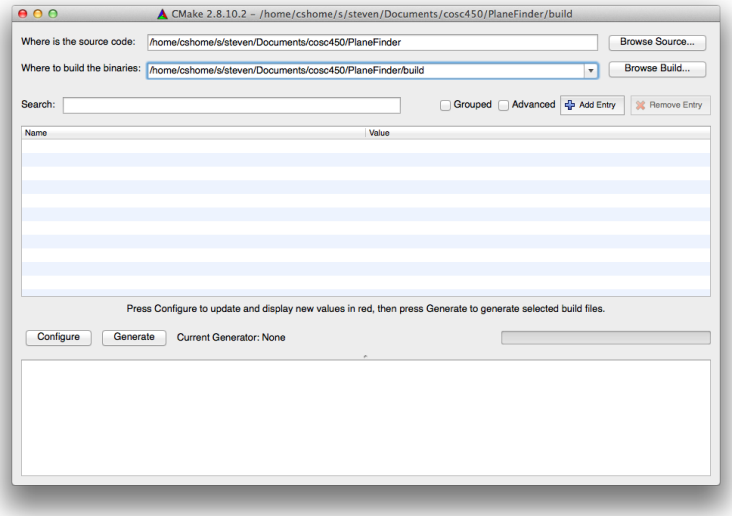
The free program Meshlab is useful for viewing PLY files, or for converting other 3D formats to PLY. Meshlab is *not* installed on the lab machines, but there is an online version available at `http://www.meshlabjs.net/`.

4.1 Building the Skeleton Code

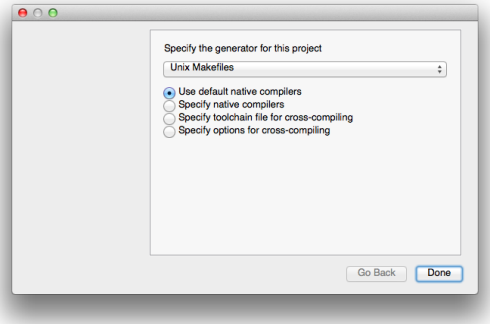
The skeleton solution can be built using CMake, which is installed on the lab machines or available from `https://cmake.org/`. CMake lets you create build environments with many different toolchains. The example here is with Makefiles, but XCode is supported as well on the lab machines. The skeleton also uses the Eigen matrix library, which is in my Public folder, or downloadable from `http://eigen.tuxfamily.org`. Documentation for Eigen is also available on that site.

To build the skeleton code on the lab machines make a copy of the PlaneFinder directory in your own space, then open CMake. If you can't find CMake, press `command-space`, and type CMake. CMake will ask for the directory where the source code is, and where you want to build it. Give it the path to your copy of the `PlaneFinder` directory in the first box. It is common to use a

`.../PlaneFinder/build` directory for building the project, but this is up to you.



CMake will prompt you and create the `build` directory if it does not exist. Next, click *Configure* and you will be prompted for what development tools you want to use. For this example I'll be using plain Makefiles, but you can use XCode or whatever if you prefer.



Finally, press *Generate* and the project will be made in the `build` directories.

4.2 Building on Your Own Machine

The code uses the Eigen matrix library, which is in my Public directory on the departmental servers. If you want to build on your own machine, you will need to download a copy from http://eigen.tuxfamily.org. You will then need

to edit the `CMakeLists.txt` file so that the `include_directories` line has the correct path to the Eigen library. Apart from that, CMake should work with whatever build environments are installed on your machine.

4.3 Running the Program

The skeleton program expects five command line arguments:

- The input PLY file
- The output PLY file
- The number of planes to find
- The threshold distance for a point to be 'on' a plane
- The number of RANSAC iterations to apply

As written the number of RANSAC iterations is an integer, but if the number is to be automatically determined, then that could be changed to a probability of success (say 0.99 or 0.999). You could also have an option where a number in the range $(0, 1)$ is interpreted as a probability of success and a positive integer as a fixed number of iterations to perform.

If you choose to automatically determine the number of planes and/or the distance threshold, you could use a value of -1 to indicate automatic detection, or some other type of input as required. This should be clearly documented in your code, the message printed when the program receives the wrong number of inputs, *and* your report.

5 Deliverables

You should send your report (PDFs preferred), source code (including build and execution instructions), and any supporting files required to steven@cs.otago.ac.nz. You may find it convenient to archive your files as a .tar or .zip, but be aware that archives sent from outside the department often get caught in the University's spam filters. This can be easily circumvented by renaming your files to remove the archive extension. If your submission is zipped up as astudent450ass1.zip then just rename it to astudent450ass1.piz or something and tell me what you've done in the body of your email. I will reply to acknowledge receipt of your assignment in any case.

6 Marking Scheme

The marks for this assignment will be allocated as follows:

- 30% for your code. Clarity and correctness are the main concerns here. Comments, naming conventions, and appropriate division of code into

functions and classes all help with this. While efficient methods are preferred, you should worry primarily about making sure that your program does what it should, and that this is clear to people reading your code.

- 30% for your experimental design. You should clearly explain what experiments you did and why. I will be looking at how you evaluated your algorithms, what data sets you used, and how you analysed the results.
- 20% for your report. In addition to your experiments your report should discuss the approach you took to implementing your algorithms, any resources you used, and how to use your program. As with your code, clarity of expression is the key factor in a good report.
- 20% for your extension, including a combination of code, evaluation, and report.

Late assignment submissions will be penalised at the rate of 10% per working day. Extensions to the deadline may be granted where appropriate, but should be sought well in advance. The usual university regulations relating to academic integrity apply (http://www.otago.ac.nz/study/academicintegrity/), and any work you submit must be your own, or be clearly attributed to the original author.

References

[1] T. Botterill, S. Mills, and R. Green. New conditional sampling strategies for speeded-up RANSAC. In *Proc. British Machine Vision Conference*, 2009.

[2] O. Chum and J. Matas. Matching with PROSAC – progressive sample consensus. In *Proc. Computer Vision and Pattern Recognition (CVPR)*, 2005.

[3] M. A. Fischler and R. C. Bolles. Random sample and consensus: A paradigm for model fitting with applications to image analysis and automated cartogrphy. *Comm. of the ACM*, 24(6):381–395, 1981.

[4] D. R. Myatt, P. H. S. Torr, S. J. Nasuto, J. M. Bishop, and R. Craddock. NAPSAC: High noise, high dimensional robust estimation - it's in the bag. In *Proc. British Machine Vision Conference (BMVC)*, 2002.

[5] P. H. S. Torr and A. Zisserman. MLESAC: A new robust estimator with application to estimating image geometry. *Computer Vision and Image Understanding*, 78:138–156, 2000.