Near real-time point cloud processing using the PCL

Marius Miknis[1], Ross Davies[1], Peter Plassmann[1], Andrew Ware[1]
[1]University of South Wales, Pontypridd, UK, CF37 1DL
{Marius.Miknis, Ross.Davies, Peter.Plassmann, Andrew.Ware}@southwales.ac.uk

Abstract – **Real-time 3D data processing is important in robotics, video games, environmental mapping, medical and many other fields. In this paper we propose a novel optimisation approach for the open source Point Cloud Library (PCL) that is frequently used for processing 3D data. Three aspects of the PCL are discussed: point cloud creation from disparity of colour image pairs, voxel grid downsample filtering to simplify point clouds and passthrough filtering to adjust the size of the point cloud. Additionally, rendering is examined. An optimisation technique based on CPU cycle measurement is proposed and applied in order to optimise those parts of the processing chain where measured performance is worst. The PCL modules thus optimised show on average an improvement in speed of 2.4x for point cloud creation, 91x for voxel grid filtering and 7.8x for the passthrough filter.**

Keywords **– Point clouds, PCL, Real-time**

I. Introduction

Point clouds are sparse spatial representations of 3D object shapes. Algorithms such as the ones in the frequently used RANSAC [1] method can then be applied to reconstruct the complete object shapes from the point clouds.

A popular toolkit for storing and manipulating point cloud data is the Point Cloud Library (PCL) [2]. The PCL is a large scale open source project that is focused on both 2D and 3D point clouds and includes some image processing functionality. Currently the Library has over 120 developers, from universities, commercial companies and research institutes. The PCL is released under the terms of the BSD licence, which means that it is free for commercial and research use. It can be cross-compiled for many different platforms including Windows, Linux, Mac OS, Android and iOS. The main algorithm groups in the PCL are for segmentation, registration, feature estimation, surface reconstruction, model fitting, visualisation and filtering.

In the work presented in this paper stereo-photogrammetry is used to acquire 3D data. This method is based on stereoscopy where two spatially separated images are obtained from different viewing positions [3]. The analysis of disparity (separation) between corresponding points in both images encodes the distance of object points which are then stored in a point cloud.

This paper is organised as follows: section II presents related work in the field of 3D data acquisition. This is followed in section III by a description of PCL modules and their optimisation. Conclusion and future work are discussed in sections IV and V.

II. Related Work

There are many uses for 3D data ranging from environmental perception for robots via autonomous car navigation, playing video games to medical uses such as wound measurement, facial reconstruction and more. A number of ways to capture 3D data have been proposed and implemented. Many existing technologies rely heavily on the use of structured or infrared lighting to extract the depth data [4]. The technique of structured lighting is widely used in computer vision for its many benefits [5] in terms of accuracy and ease of use. Over the last 15 years 3D laser scanners have been developed [6] as active remote sensing devices. Such scanners can quickly scan thousands or even millions of 3D cloud points in a scene. Time of flight cameras are also widely used in computer vision. The principle behind these cameras is similar to that of a sonar, but with light replacing sound. Such cameras were introduced into the wider public domain by the Microsoft Xbox One console [7] to replace its older structured lighting based Kinect sensor.

Once 3D data has been acquired by the above systems some kind of processing needs to be applied to extract useful information as well as to remove noise, outliers or any unnecessary information. There are software tools available for such processing [8] [9] [10] but very few provide a complete library framework to incorporate into software projects. The PCL is a very commonly used library for point cloud processing, thus the PCL was used as the main development library in this research.

The current application focus of the PCL library is in the field of robotics. For robots to sense, compute and interact with objects or whole scenes some way to perceive the world is needed, which is why the PCL is used as a part of the Robot Operating System (ROS) [11]. Using PCL as a part of ROS, robots can compute a 3D environment in order to understand it, detect objects and interact with them. Due to space and power restrictions such systems rarely use desktop-like computing devices and are therefore in most cases implemented on relatively small embedded systems. In these systems the universal nature of the PCL (many operating systems, many 3D data formats, etc) results in slow performance. The following section III proposes a range of optimisations in order to improve performance.

III. Point Cloud Processing Optimisations

Four key algorithm areas were selected for optimisation: point cloud creation (section A), rendering (section B), voxel grid down-sampling (section C) and pass through filtering (section D). For the stereo test data the New Tsukuba Stereo Dataset [12] was used. This is a collection of synthetic stereo image pairs created using computer graphics. Additionally the OpenCV (Open Source Computer Vision Library) [13] was used for image loading. The project code was run on a desktop machine: Intel i7-950 CPU @ 3.06 GHz, 8GB DDR3 @ 1600 MHz RAM, NVidia 460 GPU. The first set of tests were performed using the Microsoft Visual Studio 2013 code analyser was used to inspect code and its performance statistics. The purpose of the tests were to identify which parts of the code are using up the most of the CPU calls and then to optimise those.

A. *Point Cloud Creation Speed Improvements*

When using a stereo camera setup depth values are represented as a disparity map which in most cases is a greyscale image where the brightness of pixels represents depth values. A second output is a colour image that stores information of the actual colour value of the point. From the disparity and colour images a point cloud can be produced. The PCL provides the *OrganisedConversion<>::convert()* method which uses the disparity map, colour image and the focal length of the camera to produce a point cloud.

Point cloud generation is in 3 stages: first the input images are loaded into memory using OpenCV which converts them to vectors that can be passed as parameters to the second stage, PCL point cloud creation. The point cloud is then rendered on screen in the final stage. Using the Microsoft Visual Studio 2013 code profiler CPU cycles were measured per line of code. In order to average-out operating system specific random overheads all following test were performed three times. Results are shown in Fig. 1.

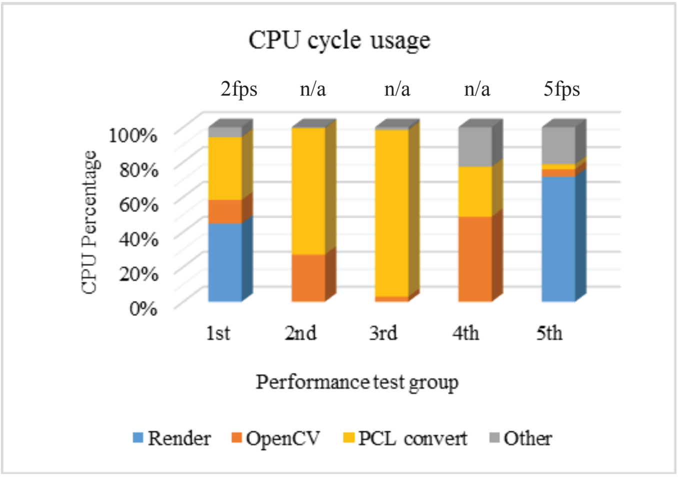

Figure 1. Test figures for CPU usage of different stages. Test 1 – 3 show pre-optimised PCL code, while tests 4 and 5 show optimised conversion.

- For the first test OpenCV was used to read the Tsukuba dataset as a sequence of images, loaded one at a time. OpenCV, PCL point cloud generation and rendering algorithms were used 'as is' without changes and as provided from public repositories. The results are shown in the first bar in Fig. 1. PCL point cloud generation required 36% of CPU cycles, rendering 45%. This resulted in a processing speed of 2 frames per second (fps).
- In the second test rendering was disabled to identify CPU load more accurately when OpenCV loaded images one at a time.
- This is contrasted by the third test where OpenCV loaded images not as individual stills but as a video sequence. Encoding the still images into a video sequence was achieved by using the OpenCV Intel IYUV. This had a dramatic effect as OpenCV CPU cycles reduced from 27% to only 3%, leaving the remaining almost 97% to the PCL conversion.
- In order to improve PCL performance numerous optimisations were made. In particular these were a) bit-shifting pointer incrementation of colour values to allow faster access and modification of values, b) vector clear and resize checks to avoid clearing and resizing a new vector when it is the same size as the previous one and c) a host of several minor optimisations. The source code and documentation of these changes are available in the PCL code repository [14]. The 4th bar in Fig. 1 shows that as a result the CPU cycles needed for PCL conversion reduced by 66% to less than the cycles needed for image loading by OpenCV.
- The two improvements documented in tests 3 and 4 were finally tested in the same way as in the first test of this series, i.e. with rendering switched on again. With image loading replaced by video loading and conversion optimised the total cycle usage of these two components now consumes less than 10% of processor cycles while rendering now takes 72%. Importantly, the overall frame rate increased to 5 frames per second.

B. *Rendering Speed Improvements*

Since rendering is now the new bottleneck, steps were taken to improve its performance.
By default, rendering for the PCL is done by The Visualisation Toolkit (VTK) [15] which is an open source library for 3D computer graphics and image processing. This was replaced with a shader (i.e. graphics processor) based OpenGL rendering implementation for desktop PCs.
The basic data structure inside the PCL is the point cloud. This is an assembly of sub-fields. The main ones are 'width', 'height' and 'points'. 'Points' is a vector that stores points of `PointT` type which in turn can be `PointXYZ`, `PointRGB`, `PointRGBA` (and some other basic types). Under the existing PCL data structure non-coloured point clouds of type `PointXYZ` could be rendered with our new OpenGL implementation but not coloured ones. To enable this several changes were made to the PCL:

- A fourth float value was added to the point cloud type union. This was easy to do since the union already had memory allocated for four float values but only x, y and z floats were declared. The fourth parameter added now stores the colour value to be passed to the OpenGL shaders.
- To store the colour values the 3 constituent independent integer values were bit-shifted into a

single float which was then stored as the 4th value of the above union. This was done to avoid integer calculations having to be done in the shaders while at the same time having minimal impact on the PCL.

- However, OpenGL shaders do not support bit shifting. The colour values were therefore extracted in the shader by a complex manipulation documented in [14].

The result of the above manipulations are shown in Fig. 2. The two bars labelled 'VTK' are unchanged re-runs of the first and fifth group tests from the previous section (see Fig. 1). When in the first test VTK is replaced by OpenGL the frame rate increases by a modest 50% to 3 fps. When, however, this is done in the optimised system produced in the previous section the speed improvement is considerable: 38 fps. In this final system where all three components are optimised, OpenGL rendering uses only 8.5% of the processor cycles while before VTK used up 72%.

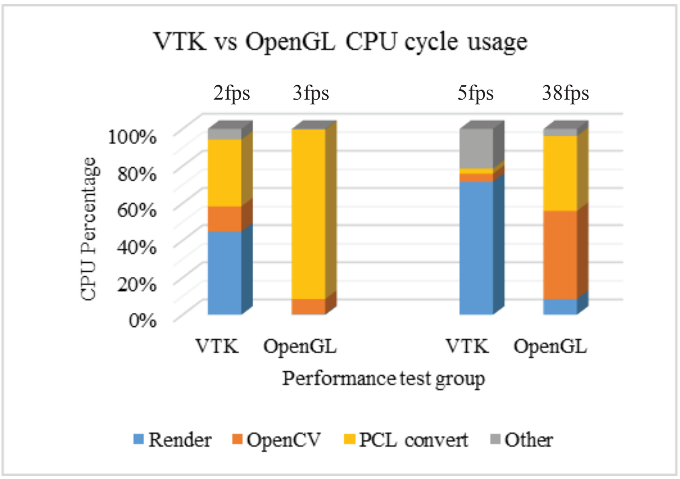

Figure 2. 1st and 5th re-tests (using the standard VTK renderer) compared to new OpenGL renderer. The first 2 bars represent performance of the non-optimised PCL code and the 3rd and 4th bar the optimised PCL/OpenCV code.

C. *Voxel grid downsample filter improvements*

After the point cloud has been produced further processing is usually required, e.g. for data reduction and filtering operations. A relatively low resolution point cloud of 640 x 480 (e.g. produced by the Kinect) results in 307,200 points. While for some operations (e.g. thresholding) point processing follows an O(n) notation a more complex algorithm (e.g. k nearest neighbour filtering) becomes O(nk). This can place a heavy workload on the processor.
One of the methods frequently used to lower the amount of points in a point cloud and unnecessary complexity while retaining detail and information is voxel grid downsampling. The downsampling is performed by using an octree to sub-divide the point cloud into multiple cube shaped regions (voxels). After processing, all points in the voxel are reduced to a single one. This results in a point cloud that is smaller in size and complexity but is still precise enough to work with and has a smaller cost in terms of CPU performance. The PCL has a dedicated method, for this called *voxelGrid.filter()*. For testing the leaf size values of the filter were 0.03f, 0.03f, 0.03f (3x3x3cm). Three groups of tests were performed as shown in Fig. 3.

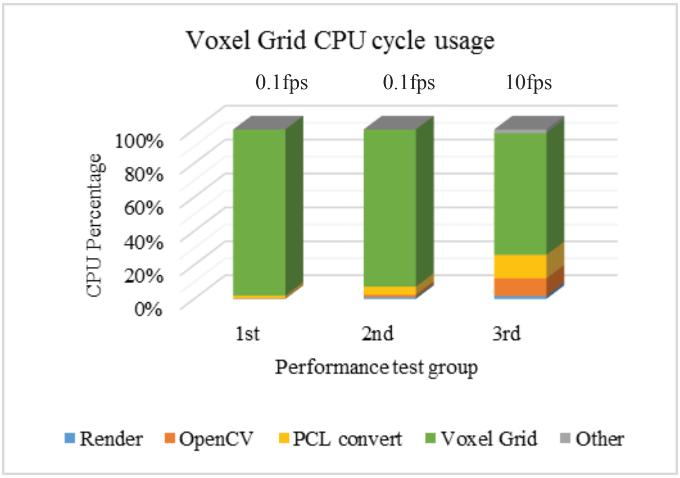

Figure 3. Test figures for CPU usage of voxel grid. Test 1 shows the stock code, test 2 shows results with Quicksort algorithm implemented and test 3 shows overall optimised voxel grid performance

- In the first group of tests voxel filtering was added to the optimised processing chain developed in the previous two sections A and B. Voxel grid computation proved to be very CPU intensive with overall CPU cycle usage of 98%. This also resulted in a poor frame rate of under 0.1 fps (8.6 seconds per frame). Analysis of the filter code revealed that 30% of the processing was spent on sorting the points using a standard C++ library vector sort method.
- The second group of tests was therefore performed with sort method replaced by a Quicksort algorithm [16]. This algorithm takes on average O(n log n) steps to sort n points, but in the worst case scenario when a chosen pivot value is the smallest or largest of the points to sort the algorithm has to make O(n^2) comparisons. To avoid this possible issue a mean value is computed before the sorting to avoid using very small or very large values as the pivot. Compared to the standard C++ sort with 30% of processor cycles used Quicksort was significantly more efficient, using only 0.9%. This unfortunately improved the overall filter method by only 5.2% as the computation shifted to different parts of the algorithm, mostly to vector access overheads.
- For the third test group vector access was therefore optimised as detailed in [14]. This reduced the voxel filter computation time by 26% to an overall contribution of only 72% of CPU cycles.

The combined changes to the sorting and vector processes increased the frame rate 91-fold to an average frame rate of about 10 fps.

D. *Pass through filter improvements*

Another PCL provided post-processing method is *passthrough.filter()* which is as a means to allow the removal of points from the cloud which are not within a specified range. This allows to adjust the point cloud in any coordinate direction similar to a frustum cut-off.

The *passthrough.filter()* method accepts parameters for upper and lower limits and a direction along the x, y or z axis. For the Tsukuba dataset the depth range values of 3 and 12 were used for testing in the z coordinate direction. Two groups of tests were performed with results shown in Fig. 4.

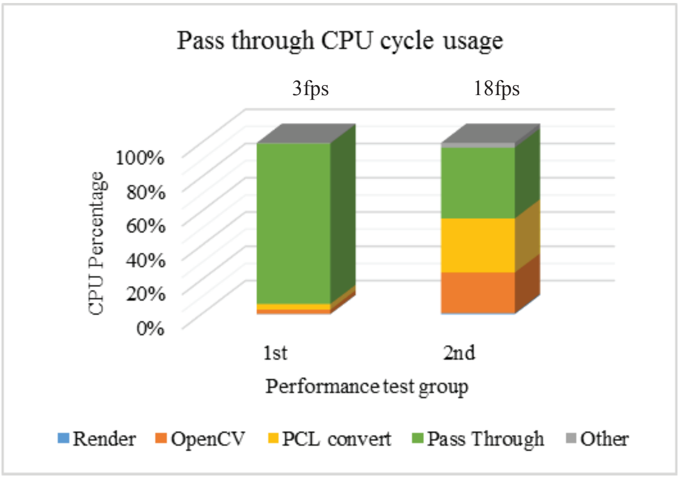

Figure 4. Test figures for CPU usage of Passthrough filter. 1st test showing the stock code performance and 2nd test showing the improved code module

- In the first test the pass through filter was appended to the optimised processing chain outlined previously in sections A and B. The filter was very CPU intensive using 93.6% of cycles bringing down the frame rate to 3 fps. Analysis of the code showed that (as before with voxel filtering) vector accesses were inefficient.
- After vector access optimisation along the lines outlined before with voxel filtering the passthrough filter now only consumes 41% of CPU cycles with the frame rate rising to 18 fps.

IV. Discussion & Conclusion

To develop a code base to work with point clouds would be a very time consuming task as most parts would have to be written from scratch. The benefit, however, would be dedicated code with optimised overall performance. A second option is to alter an already developed library such as the PCL. Being general purpose and multi-platform means that several aspects are generalised, not all parts are optimised and performance can suffer on time sensitive processing. A disadvantage of this second approach is that code becomes specialised and is no longer as versatile as the original. However, as shown in section III the optimised PCL modules provide significant performance gains over the stock modules. Neglecting the minimal cost of performance testing measurement overheads speed improvements were 2.4 times for the organised PCL conversion, 91 times for voxel grid filtering and 7.8 times for pass through filtering. This allows for the use of multiple PCL modules together while still maintaining near real-time frame rate.

V. Future Work

Future work is focused on submitting the optimised code to the official PCL repository through PCL Developers community to contribute to the project. Another part of research has already been started to allow the PCL to be used with embedded devices to perform real time point cloud processing.

References

[1] R. Schnabel, R. Wahl. and R. Klein, "Efficient RANSAC for Point-Cloud Shape Detection" *Computer Graphics Forum,* vol. 26, no. 2, p. 214–226, 2007.

[2] S. C. Rusu Radu Bogdan, "3d is here: Point cloud library (pcl)," in *Robotics and Automation (ICRA), 2011 IEEE International Conference*, Shanghai , 2011.

[3] C. Sun, "A Fast Stereo Matching Method," in *Digital Image Computing: Techniques and Applications*, Auckland, 1997.

[4] S. Izadi, D. Kim and O. Hiliges, "Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera," in *24th annual ACM Symposium on User Interface Software and Technology*, New York, NY, 2011.

[5] D. Lanman, D. Crispell and G. Taubin, "Surround Structured Lighting for Full Object Scanning," in *Sixth International Conference on 3-D Digital Imaging and Modeling*, Montreal, Aug. 2007.

[6] A. Zhang, S. Hu, Y. Chen, H. Liu, F. Yang and J. Liu, "Fast Continuous 360 Degree Color 3D Laser Scanner," in *The Internal Archives of the Photogrammetry, Remote Sensing and Spatial Information sciences, Volume XXXVII*, Beijing, 2008.

[7] Microsoft, "Kinect for Windows," Microsoft, [Online]. Available: https://www.microsoft.com/en-us/kinectforwindows/develop/. [Accessed 2 June 2015].

[8] Bentley Systems, "Bentley Pointools V8i," Bentley Systems, [Online]. Available: http://www.bentley.com/en-US/Promo/Pointools/pointools.htm. [Accessed 16 June 2015].

[9] Mirage-Technologies, "Home: PointCloudViz," Mirage-Technologies, [Online]. Available: http://www.pointcloudviz.com/. [Accessed 16 June 2015].

[10] Faro, "Home: PointSense," Faro, [Online]. Available: http://faro-3d-software.com/CAD/Products/PointSense/index.php. [Accessed 16 June 2015].

[11] Willow Garage, "Software: ROS," Willow Garage, 3 June 2015. [Online]. Available: https://www.willowgarage.com/pages/software/ros-platform.

[12] S. Martull, M. Peris and K. Fukui, "Realistic CG stereo image dataset with ground truth disparity maps," *Trak-Mark,* 2012.

[13] Itseez, "Home page: OpenCV," Itseez, [Online]. Available: http://opencv.org/. [Accessed 15 January 2015].

[14] GiHub, "Point Cloud Library Repository," [Online]. Available: https://github.com/PointCloudLibrary/pcl. [Accessed 23 June 2015].

[15] Kitware, "Home: VTK," Kitware, [Online]. Available: http://www.vtk.org/. [Accessed 15 June 2015].

[16] C. A. R. Hoare, "Quicksort," *The Computer Journal,* pp. 10-16 , 1962.