Title:
Using Data Indexing for Remote Visualization of Point Cloud Data

Authors:
Paul Rosen, prosen@usf.edu, University of South Florida
Les Piegl, lespiegl@mail.usf.edu, University of South Florida

Keywords:
point cloud, visualization, data indexing, data structures, remote visualization

DOI: 10.14733/cadconfP.2016.296-300

Introduction:

For decades, point cloud-based datasets have been critical to a number of research communities, including graphics, robotics, and CAD. In many ways, points better represent many data sources than do triangulations. Take a laser scan for example. The laser collects a series of point that represent a surface. Then, other algorithms take over to produce surfaces, such as triangulations, NURBS, etc. These surfaces must make assumptions about continuity or connectedness that may or may not be valid. Point clouds also have the advantage of being more naturally representable in multiple resolutions. For example, clusters of points can be culled to achieve a desired resolution.

Visualizing points on remote devices remains somewhat problematic to this day. Very large datasets have the problem of needing to be transmitted, have any data structures built locally, and rendered on the local machine. Even with advances in low-power computing, such as tablets, the trend in computing is to compute as little as possible locally and rely on cloud-based systems to store and process large data. For point-based data this approach is highly relevant. In fact, data should only be transmitted at a resolution visible on the output display. What's the point in rendering billions of points, when the display only has 2 million pixels?

There are further challenges to visualizing points on remote devices for in situ applications where, for example, mechanical part is being laser scanned. New points that are scanned should be added to the database and become visible in near-realtime. While not technically impossible, it can be technically challenging to implement in an efficient manner.

To address these challenges, we have built a system that used data indexing to provide remote access to point cloud data. The advantages of the system are 3-fold. First, the system seamlessly delivers point-based data at a variety of resolutions. Secondly, the system is able to import new data, build data structures, and make it available to clients in near real time. Finally, the system provides flexibility to heterogeneous data types. For example, each point may have color, texture value, normal direction, etc. This can be achieved without any modification to the system beyond the rendering itself.

Background and Prior Work:
For context to our approach, we discuss standard approaches of using spatial data structures for visualizing point cloud data. We also discuss how indexing engines are being used in data analysis.

Spatial Data Structures:

Traditionally, storing and accessing point-based data has been the job of spatial data structures, most commonly including Octrees [2], Binary Space Partitions (BSP) [6], Kd-Trees [4], Bounded Volume Hierarchies (BVH) [3], etc. Though each of these techniques has a different take on the problem, they

generally approach the problem the same way. First, a tree or hierarchy is created that recursively subdivides the data into sets that are spatially similar (i.e. close in Euclidean space). Then, in applications such as ours, queries can efficiently access objects within subregions of space. Additional operations can be performed, such as nearest neighbor finding, but these tasks often require additional data be stored or complicated algorithms be developed. The difference between the individual approaches (Octree, BSP, etc.) generally lies in how the algorithms subdivide data.

Indexing Engines in Data Analysis:

The idea of using indexing engines in spatial data analysis is a relatively new one. The premise, first introduced by the ColumbuScout [1] and its successor, STORM [5] is simple. By indexing data, instead of storing it in traditional rigid data structures, we can build a system that enables fast queries on data of mixed-schema. The ultimate goal of the approach is to create "Google for data". The architecture for STORM is one in which a powerful indexing engine is built that supports a wide variety of queries related to data analysis. Then, a lightweight frontend is applied. The crux of STORM is that the power lies solely in the indexer itself.

On the other hand, Klareco (Fig. 1) is a decentralized architecture. Klareco uses an off the shelf indexer, Apache Lucene, to index data. Apache SOLR provides a web-based interface to Lucene. Next, microservices are hosted on Apache Tomcat webservers. Those microservices are lightweight user-specified algorithms that provide the intelligence for the architecture. 2-way communication between the microservices and indexer are trivial to accomplish. This means that results calculated in the microservice can be stored in the index for later retrieval. Finally, lightweight clients are connected to these components using HTTP and JSON for communication. Each Klareco component can run on a different machine operating over standard communication channels. Apache SOLR and Tomcat are Java-based, enabling execution in many environments. The front-end may be written in any language with support for HTTP- and JSON-based communication.

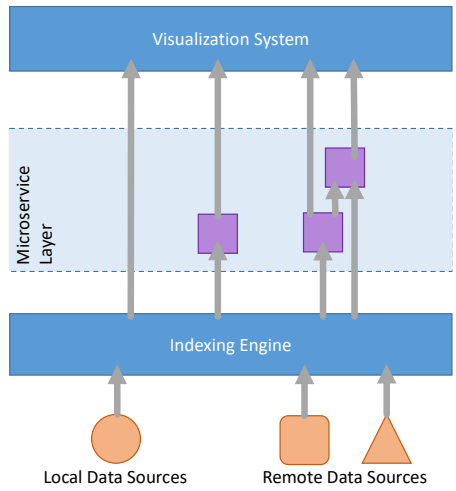

Fig. 1: The Klareco System Architecture

Remote Visualization of Point Clouds:

When attempting to render point cloud data at remote locations, a number of problems present themselves. One assumption should be that communication and computational resources are limited (bandwidth, power, etc.). Thus, not all of the data can be transferred, nor should all of the data be rendered. For example, the resolution needed by the client varies based upon their display resolution, as well as viewpoint of the data. Additionally, the data may come in a variety of resolutions. Rarely are laser scans or other point cloud data sets sampled in a spatially uniform manner. A final important assumption is that there is an extensive variety in the metadata associated with points. The may include a scalar value, a color values, a texture location, etc. To address these challenges, we built a system for visualizing point cloud data using Klareco.

Supporting Range Queries and Metadata:

The system we initially built only includes the indexer and visualization frontend. First, a parser was required to insert point cloud data into the indexer. This parser inserted the positional data, along with any metadata. Each data point was stored as it's own "document", which is the Lucene equivalent of a record. Second, we built a Javascript/WebGL-based frontend for visualizing the data.

Our frontend directly queried the indexer. Since the indexer automatically provides range query capabilities, the frontend merely needs to specify the range of the view frustum, and the indexer

returns the relevant points with all of their metadata. Finally, it is up to the frontend to decide what to do with that metadata. For this paper, metadata was simply unused.

Supporting Multiresolution Queries:

The indexer does not explicitly support multiresolution queries. However, with the addition of the right microservice and additional metadata, we can help it support multiresolution queries.

We first discuss the multiresolution technique we apply. We use a hierarchical gridding approach where each level of the hierarchy has twice the resolution in each direction as the previous.

The algorithm is as follows:

All points are marked with their minimum possible resolution, 0.
Starting at the highest desired resolution (L)
Place the points in the grid
For each grid cell
If there are 0 or 1 points in a grid cell, do nothing
If there is a conflict (i.e. more than one point in the cell)
The lowest left point (by Euclidean distance) is retained
Set resolution of other points to L+1 and disregarded from future computation
Repeat for L-1, until reaching level 0

The algorithm is demonstrated in Fig. 2. Each point is marked with its minimum possible level in the hierarchy, starting with 0 (orange). We begin at L=2, a grid resolution of 2^2 (i.e. 4x4 cells). At this resolution each point occupies its own cell. Therefore, no further action is taken. Next we proceed to L=1, grid resolution 2^1 (i.e. 2x2 or 2x2x2 in the case of 3D). We now have 3 conflict cells. The points that are closest to the lower left corner retain their current mark. Others are marked with L+1, 2 (green) in this case, and disregarded from future computation. Finally, we complete the process with L=0, grid resolution 2^0. We only have 1 cell that is conflicted. The lower left point is untouched, and all other are marked with L+1 or 1.

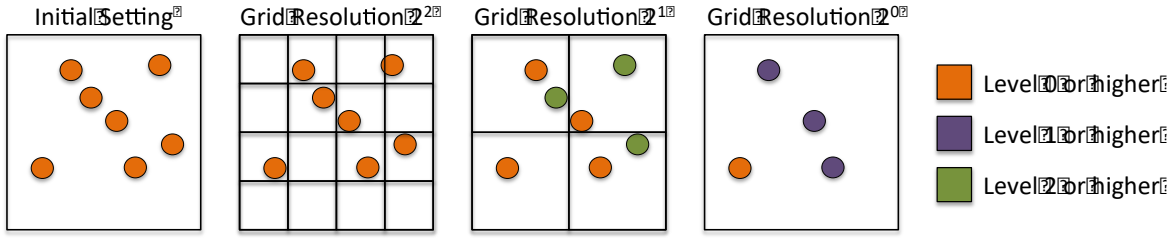

Fig. 2: Multiresolution Gridding Algorithm

Once all the points have been marked, the grid itself no longer needs to be stored. The grid resolution for each point is stored in the index as additional metadata. Then, when queried, not only can a spatial range be specified, but also only points at or below a desired resolution can be requested (i.e. resolution<=3).

Assuming the initial resolution is selected correctly, the first iteration of the algorithm requires processing *n* nodes. Each additional iteration should only require *log n* points.

Supporting Multiple Files and Streaming Data:

The algorithm as described requires having all of the data points available in order to compute the multiresolution index. However, this does not need to be the case. We compute the grid on subsets of data points. These can be from separate files, partial files, or subsets of points streaming from a source, such as a laser scanner.

The problem with computing subsets individually is that the results from each subset needs to be merged with previous results. Instead of processing at load, this operation is performed at runtime. The process is simple, a client will perform query on range $\boldsymbol{X}$ and resolution L. The microservice will query the index for data fitting those requirements. However, since the data have not been gridded globally, the data returned by the index is conservative, containing more data points than needed. To get the data properly gridded, the gridding algorithm described above is performed on that data, but the algorithm is only run at one resolution, L, the resolution of the query. Any data points modified by the gridding operation can be updated in the index. The other data points are returned to the client.

We have no method for determining convergence. Thus, the operation continues to be performed on the dataset. This sounds worse than it actually is, as the operation is not that costly, given that is linear with the number of data points returned by the range and resolution query.

Visualization Front-End:

To demonstrate the effectiveness of the approach we have built a visualization frontend using JavaScript and WebGL. The visualization frontend is responsible for 3 tasks: rendering the data, handing user interaction, and querying and storing data at appropriate resolutions. The former 2 tasks are performed using standard techniques. The latter task requires some special attention.

For querying and storing the data, the spatial domain is divided into a grid. The resolution of the grid has to balance two aspects. First, higher resolution grids give more fine grained control of cell resolution. Second, higher resolution grids demand a greater number queries—too many simultaneous queries will cause bottlenecks at both the client and server.

Each grid cell retrieves data at a resolution that is most appropriate to its needs. This is determined by looking at the viewpoint and view direction of the camera. The closer to each a grid cell lives, the higher the resolution of data it needs. As the camera moves, any grid cells whose resolution needs change update their data accordingly. In addition, the data in each grid cell is additionally updated at regular intervals to accommodate any streaming data.

Results:

To test our system, we have applied the approach to point cloud datasets obtained from the internet, in particular the Andreas Haus and Andreas Garten [7]. The indexer and microservice were run on a Linux PC with an Intel Core i7-3770 CPU @ 3.40GHz and 24 GB of RAM. The web-based frontend was tested using both a MacBook Pro (early 2015 model) and Apple iPad (Gen 3). The server was located on the University of South Florida campus in Tampa, Florida. The clients where tested in home locations in Tampa, Florida and Salt Lake City, Utah, both with broadband internet.

Fig. 3 shows the results of loading the Andreas Garten datasets, which is an outdoor scene containing trees (which is what is mostly visible in the figure) and a few small structures. The dataset contains approximately 45M points, far more than needed for high quality rendering. In total, it took approximately 6 hours to initially index the data. The majority of this time is taken by the indexer, not the gridding, which takes a matter of seconds. When viewing the scene, the loading takes a few

Fig. 3: Example of system on Andreas Garten dataset (45M data points). Left shows the data at points from selecting grid levels 0-7 only (48K data points). The center shows data from grid levels 0-10 (1.1M data points). The right image shows the data points gathered for a view dependent (red arrow) resolution selection (635K data points).

minutes, depending upon the resolution and internet speed. Updates to the data resolutions occur as the camera moves, with each query taking 5-20 seconds, depending upon the data size.

Acknowledgements:
We thank our funding sponsor NSF DIBBs ACI-1443046.

References:

[1] Hansen, C.; Li, F.: ColumbuScout: towards building local search engines over large databases, In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012, 617-620. http://dx.doi.org/10.1145/2213836.2213914

[2] Meagher, D.: Geometric modeling using octree encoding, Computer Graphics and Image Processing, 19(2), 1982, 129-147. http://dx.doi.org/10.1016/0146-664X(82)90104-6

[3] Klosowski, J. T.; Held, M.; Mitchell, J. S. B.; Sowizral, H.; Zikan, K.: Efficient collision detection using bounding volume hierarchies of k-DOPs, IEEE Transactions on Visualization and Computer Graphics, 4(1), 1998, 21-36. http://dx.doi.org/10.1109/2945.675649

[4] Bentley, J. L.: Multidimensional binary search trees used for associative searching, Communications of the ACM, 18(9), 1975, 509-517. http://dx.doi.org/10.1145/361002.361007

[5] Christensen, R.; Wang, L.; Li, F.; Yi, K.; Tang, J.; Villa, N.: STORM: Spatio-Temporal Online Reasoning and Management of Large Spatio-Temporal Data, In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, 1111-1116. http://dx.doi.org/10.1145/2723372.2735373

[6] Fuchs, H.; Zvi, M. K.; Naylor, B. F.: On visible surface generation by a priori tree structures, In ACM Siggraph Computer Graphics, 14(3), 1980, 124-133. http://dx.doi.org/10.1145/800250.807481

[7] Borrmann, D.; Nüchter, A.: Robotic 3D Scan Repository, http://kos.informatik.uni-osnabrueck.de/3Dscans/