COMPUTER-AIDED DESIGN & APPLICATIONS, 2017
VOL. 14, NO. 6, 789–795
https://doi.org/10.1080/16864360.2017.1287754

Using data indexing for remote visualization of point cloud data

Paul Rosen and Les A. Piegl

University of South Florida, USA

ABSTRACT
We present a new approach for accessing and visualizing point-based data in CAD applications. Instead of developing a traditional database around spatial data structures, our approach augments a data indexing engine to enable quick access to data. The primary advantage of an indexing engine is flexibility. The approach enables both range queries for accessing data spatially and resolution queries to access data at appropriate spatial resolutions. Our approach is robust to very large datasets, naturally supporting remote visualization and near-real-time input data streams. We demonstrate our approach on 2 large datasets, one 45M points, the other 53M points.

KEYWORDS
Point cloud; visualization; data indexing; remote visualization

1. Introduction

For decades, point cloud-based datasets have been critical to a number of research communities, including computer 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 points that represent a surface. Then, algorithms, such as triangulations, NURBS, etc., take over to produce surfaces. These surfaces are, at their core, assumptions about shape, continuity, and connectedness that may or may not be valid. When the original points are instead visualized, the human perceptual machinery can reconstruct those surfaces instead. Point clouds also have the advantage of being more naturally representable at multiple resolutions. For example, points can be culled from clusters to achieve a desired density. With polygonal meshes, many options exist for simplification [17], but they all optimize on different conditions, none of which is clearly best. Parameterized surfaces can extract data at any resolution, but choosing and tuning the sampling rate can be challenging.

Visualizing points on remote devices remains somewhat problematic to this day. Very large datasets have the problem that they needing to be transmitted, have any data structures built locally, and be rendered on the local machine. With advances in low-power computing, such as tablets, the trend 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. Point of fact, the amount of data to draw should be directly linked to the resolution of the output display. There is no value 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, a mechanical part is being laser scanned. As new points are scanned, they should be added to the database and become visible in near-real-time. While not a technically impossible task, it can be practically challenging to implement in an efficient manner without the assistance of a database designed for just such a purpose.

To address these challenges, we have built a cloud-based system that uses a data indexing engine 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 over an open API. 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.

2. 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 currently being used in data analysis.

CONTACT Paul Rosen prosen@usf.edu; Les A. Piegl lpiegl@gmail.com

2.1. Spatial data structures

Traditionally, storing and accessing point-based data has been the job of spatial data structures, most commonly including Octrees [13], Binary Space Partitions (BSP) [9], Kd-Trees [4], Bounded Volume Hierarchies (BVH) [14], 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. Other operations can be performed, such as nearest neighbor finding, but these tasks often require additional data be stored or complicated algorithms be developed. For example, the data structures have no sense of spatial resolution making a query of that nature challenging to develop efficiently. The difference between the individual approaches (Octree, BSP, etc.) generally lies in how the algorithms subdivide data.

2.2. Indexing engines

The traditional use of indexing engines is for fast searches among large document collections. For example, when you use your hard drive file search functionality, you search for a paper on your favorite publisher's website, or you perform a Google search, you are using an indexing engine. Given a set of search criteria, the indexer returns a set of documents. This is an ideal platform for unstructured data, such as a document collection.

Increasingly however, indexing engines are supporting mixing of both structured and unstructured data in document queries. This seemingly minor change opens up the possibility of using these engines in more data intensive applications, including spatial data analysis.

2.2.1. Indexing engines in spatial data analysis

The traditional approach of accessing point data relies on flat file storage. In the naïve case this makes queries slow. LAStools [11] rely on a data reorganization scheme to make spatial queries of a flat point-based data files fast. Such systems can scale to billions of points but require direct access to the data.

On the other hand, the idea of using indexing engines in spatial data analysis is a relatively new one. The premise, first introduced by the ColumbuScout [10] and its successor, STORM [7] is simple. By indexing data, instead of storing it in rigid data structures, we can build a system that enables fast queries on data of mixed-schema (mixed structure). The ultimate goal of the approach is to create "Google for data". The architecture for STORM is a tight 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 making it rigid and its functionality difficult to extend.

On the other hand, Klareco [16] (Fig. 1) is a decentralized data indexing system. Klareco has a 4-layer architecture that we will reference throughout the remainder of this paper.

- At the bottom, the *parser* layer provides standard or, as in our case, specialized functions for parsing data sources and inserting them into the indexer.
- Next, Klareco uses an off the shelf *indexer*, Apache Lucene [1], to store and query documents (i.e. records). Apache SOLR [2] provides a web-based interface to Lucene for both input and output of data.
- Third, *microservices* are hosted on Apache Tomcat [3] 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 not only can microservices return results to the front-end, but the calculations performed by the microservice can be stored in the index for later retrieval.
- Finally, *lightweight visualization clients* are connected to these components using HTTP and JSON for communication.

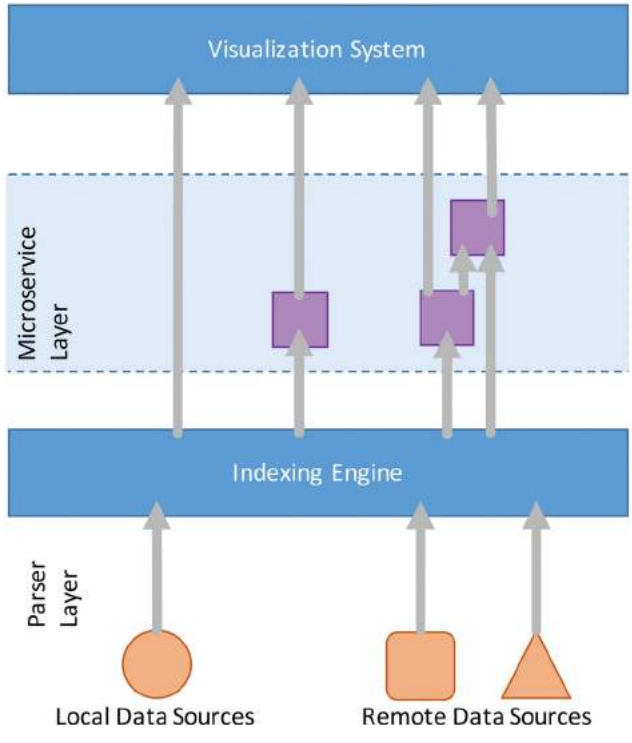

Figure 1. The Klareco System Architecture.

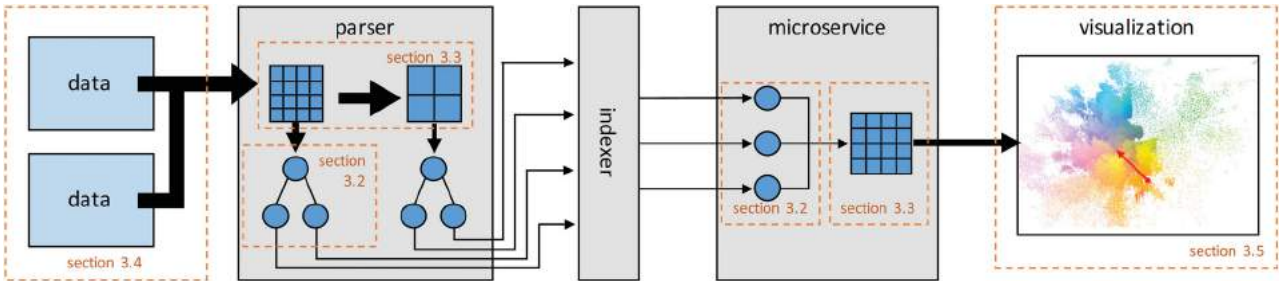

Figure 2. Schematic of the system outlined in this paper. The regions with dotted orange lines indicate the section that describes an element.

Each Klareco component can run on a different machine operating over standard communication technologies. Apache SOLR and Tomcat are Java-based, enabling execution in the vast majority of hardware and software environments. The front-end may be written in any language with support for HTTP- and JSON-based communication.

3. 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. This may include scalar values, color values, a texture location, etc. To address these challenges, we built a system for visualizing point cloud data using Klareco. The final system construction is outlined in Fig. 2. The following sections will build the system, piece by piece.

3.1. Supporting range queries and metadata

The system we initially built included a custom parser, the indexer, and a visualization frontend. First, the parser's job was to read a file and insert the point cloud data, along with any metadata, into the indexer. Each data point was stored as its own "document", which is the Lucene equivalent of a record. The parser would insert the positional data into specially marked fields, enabling the range queries already built into Lucene.

Second, we built a JavaScript/WebGL-based frontend for visualizing the data. Our frontend directly queried the indexer. Since the indexer automatically provided the range query capabilities, the frontend merely needs to specify the range of the view frustum, and the indexer returned the relevant points with all of their metadata.

This implementation had 2 very important shortcomings. First, the time to index the entire dataset was very high. Our datasets with 45M and 53M point would take between 24 and 48 hours to load into the indexer. Nevertheless, once indexed, queries were swift, taking at most a few seconds to execute and transmit (depending upon data size). The second shortcoming was the lack of support for multiresolution queries. All the points within the specified range would be sent with a query. We first address the former problem, indexing time.

3.2. Using kd-trees to build multi-point documents

We considered 24+ hour indexing as being a significant enough problem that we made an effort to improve the performance. As it turns out, the time to build the index is more significantly impacted by the number of documents in the index, instead of the size of the documents. So, the simple solution to the problem was to insert fewer documents by merging multiple points into a single document. However, that brings up the important question of how best to merge points.

Given that our primary query of interest are spatial, we chose to use a balanced kd-tree [6] to segment points into equally sized, spatially similar documents. The parser would load an input file and build the kd-tree (with a stopping condition of 256 points). Then, each leaf of the kd-tree would be converted into a single document in the index.

With this change, the data returned by the indexer is conservative—a document (kd-tree leaf) is returned whenever a single point in that leaf fit the range search criteria. Unfortunately, this also meant that points outside of the range would also be returned, breaking one of our original design goals—keeping client bandwidth and computation low. However, adding a simple

microservice, the system can be realigned to the design criteria. The new microservice takes the documents returned by the indexer and filters out points that do not fit the range criteria. This is a linear time operation per query, but only on the subset of points returned by the indexer.

With this change, indexing time is significantly reduced, down to 30 minutes and 45 minutes, respectively, for the 2 datasets. The query performance takes only a negligible hit (practically impossible to measure relative to the variance in bandwidth and latency).

3.3. Supporting multiresolution queries

Originally, support for multiresolution queries was an important design criterion. Unfortunately, the indexer does not explicitly support multiresolution queries. However, with modifications to the parser and microservice and some additional metadata, the system can support multiresolution queries.

3.3.1. Selecting points at a specific resolution

We first discuss an algorithm for taking a set of points and selecting those which are appropriate to keep for a given resolution.

The algorithm is as follows:

Select a resolution and create a grid based upon it
Place points, one at a time, in the grid
If their grid cell is empty, do nothing else
If there is a conflict (i.e. a point already in the cell)
The lowest left point (by Euclidean distance) is retained
The other point is discarded

The result is a set of points that best fit the given grid resolution. According the the Nyquist—Shannon theorem [15,18], these points provide enough sampling for a resolution half that of the grid.

From an implementation perspective, if the space is expected to be dense, the grid can be implemented with a multidimensional array. If the space is expected to be sparse, a map data structure can be used. We expected mostly sparse space, leading us to chose the latter approach.

3.3.2. Efficiently selecting points at multiple resolutions

We next discuss the multiresolution technique used. We use a hierarchical gridding approach where each level of the hierarchy has twice the resolution in each dimension as the previous.

The algorithm is as follows:

All points are marked with their minimum possible resolution, $L = 0$.
Starting at the highest desired resolution (2^L)
Run gridding algorithm from Section 3.3.1
Retained points are kept for additional processing
Discarded points have resolution set to L+1 and are ignored in future processing
Repeat for L-1, until reaching $L = 0$

The algorithm is demonstrated visually in Fig. 3. 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. 4×4 cells). At this resolution each point occupies its own cell. Therefore, no further action is taken. Next, proceeding to $L = 1$, the grid resolution is 2^1 (i.e. 2×2 or 2×2×2 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 further computation. Finally, the process is completed with $L = 0$, grid resolution 2^0. Only 1 conflicted cell remains. The lower left point is untouched, and all other are marked with L+1 (i.e. 1). Once all the points have been marked, the grid no longer needs to be stored. The marked 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 (maximum) resolution is selected correctly, the first iteration of the algorithm requires processing n nodes. Each additional iteration should only require $\log n$ of the previous.

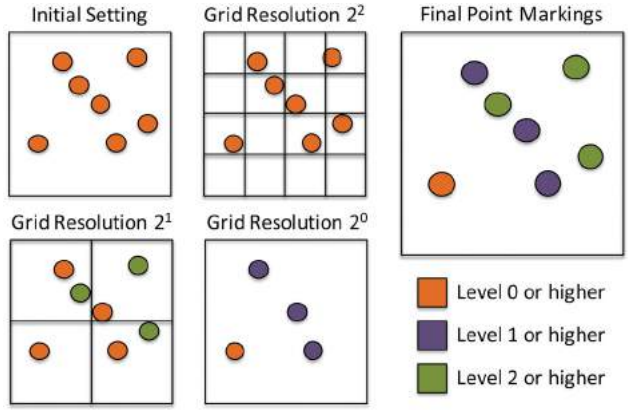

Figure 3. Multiresolution Gridding Algorithm.

This particular algorithm gets inserted into the parser. The specific modification to the parser is that once the data is read, it has the gridding applied. Then, the kd-tree is applied once to each level of the grid hierarchy.

3.4. Supporting multiple files and large or streaming data

The algorithms as described require having all of the data points available in order to compute the proper kd-trees and multiresolution index. However, this does not need to be the case. We actually assume that the grid and kd-trees are calculated on subsets of data points. These can be from separate files, partial files for large data, or subsets of points streaming from multiple sources, such as a laser scanner. In fact, our datasets of 45M and 53M point are divided into 99 and 153 files, respectively.

The main problem with computing subsets individually is that the gridding results from each subset need to be merged with previous results. Two possible choices present themselves.

The first option would be re-gridding results in the parser at load time. When considering this option, a number of problem present, including querying the right ranges of data, rebuilding (and possible splintering) of kd-trees, and not wanting to overload the indexer with calls to add and delete documents (as this causes rebuilds of the index that can be expensive in time).

The second option, which is the option we chose, is to perform the operation at runtime or query time in a microservice. To do this, a client will perform a query on a range and resolution. The microservice will query the index for data fitting those requirements. First, the points are filtered by range, as described in Section 3.2. The gridding algorithm described in Section 3.3.1 is then performed on that data, but the algorithm is only run at one resolution, the maximum resolution requested in the query. At the desired resolution, the grid level is calculated. Points in the grid are retained and output. Conflict cases are discarded.

If desired, the results of the new gridding can be stored back into the index. However, this carries the same limitation of rebuilding and splintering the kd-trees and potentially overloading the index with change requests. Furthermore, we have no method for determining global convergence. Thus, the operation would still need to be performed on the dataset.

All told, this operation sounds worse than it actually is, as the operation is not that costly, given that is only linear with the number of data points returned by the initial range and resolution query.

3.5. Visualization front-end

We have built two visualization frontends. The first is web-based using JavaScript and WebGL, while the other is desktop-based, using Processing [8], a Java-based visualization sketching language. The visualization frontends are responsible for 3 tasks: rendering the data, handling 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. On the plus, higher resolution grids give more fine grained control of resolution. On the negative, 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 a combination of the distance to the viewpoint and position relative to the 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 changing, updates their data accordingly. In addition, the data in each grid cell can be updated at regular intervals to accommodate streaming data.

Finally, it is up to the frontend to decide what to do with metadata. For both systems, we experimented with coloring the data based upon the metadata values. However, the images we present do not use that feature.

4. Results

To demonstrate our system, we have applied the approach to point cloud datasets obtained from the internet, in particular the Andreas Haus and Andreas Garten datasets [6]. The 1.3GB and 1.6GB datasets contain 45M and 53M point that are divided into 99 and 153 files, respectively. After gridding and kd-tree construction, the parser created 253k and 318k documents, respectively, for indexing. The indexer and microservice were run on the same machine, a Linux PC with an Intel Core i7-3770 CPU @ 3.40 GHz 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 desktop-based system was tested using only the MacBook Pro. The server was located on the University of South Florida campus in Tampa, Florida. The clients where tested in residential locations in Tampa, Florida and Salt Lake City, Utah, both with broadband internet.

Figure 4. Example of web-based viewing system on Andreas Garten dataset (45M data points). Left shows the data at points from selecting grid levels 0-7 only (48 K 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 (635 K data points).

The parser is a Java-based application that can be run in the background. For the two datasets tested, the parsing portion of the process took 34 minutes and 45 minutes, respectively. The majority of this time is taken by the indexer, not the gridding or kd-tree construction, which each take a matter of seconds.

The indexer, Apache Lucene/SOLR, is also a Java-based application. The indexer opens a port (8983 to be specific) as a query interface. The indexer can then be queried using a standard web browser, specifying query parameters using the GET protocol. Results are returned as XML or JSON type documents.

The microservice operates under the servlet model with Apache Tomcat hosting the servlet. Under this model Apache Tomcat hosts a webserver on port 8985. Then, specific URLs can be used to launch Java servlet applications, in our case, the microservice. The microservice communicates with the client in much the same way the indexer does (using the GET protocol but returning JSON only). In addition, microservice communicates with the index directly to retrieve data for processing.

Fig. 4 shows the results of loading the Andreas Garten dataset into the web-based viewer. Andreas Garten is an outdoor scene containing many trees (which is what is mostly visible in the figure) and a few small structures. This viewer renders the data as point primitives that are colored by their location relative to some origin. The dataset contains approximately 45M points, far more than needed for high quality rendering. When viewing the scene, the loading takes a few 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.

Fig. 5 shows the result of loading the Andreas Haus dataset into the desktop-based viewer. The scene is a laser scan of the inside of a house containing more than 53M data points. The points in this case were rendered as spheres. Three colored lights were used to help differentiate surfaces. This camera position used approximate 6.3M points totaling 185MB. The loading time for those data points was 124 seconds on residential broadband. We suspect that this performance is bandwidth limited, as those numbers point to 1.5 MB/s (or 12 Mbps) downstream rate.

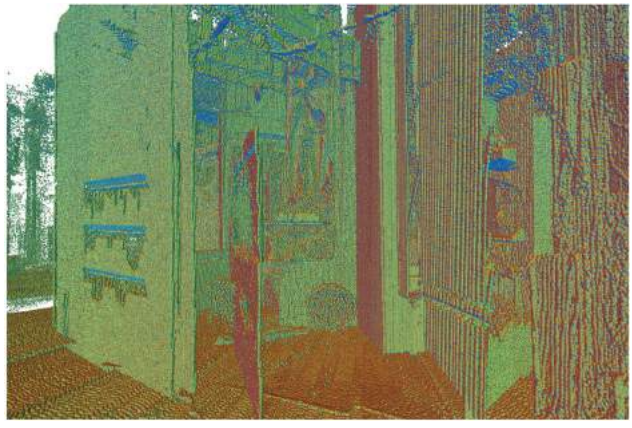

Figure 5. Image from inside of the Andreas Haus model that contains over 53M data points in total. Points were rendered as spheres, with 3 lights enabled to help differentiate surfaces. About 6.3M points were queried and drawn, which totaled about 185MB of data.

Overall, the biggest problem we ran into in both of these cases was downstream bandwidth limits. Two remedies for this problem present themselves. First, data compression could be very useful. A simple solution would be to gzip the JSON data. JSON is a text-based format (containing mostly numbers, brackets, periods, and commas in our case), so undoubtedly, this would yield significant improvements. Lossless floating point compression [12] could also be used. Given the structure of our points (bounded in a region at regular intervals), high-levels of compression are likely. The second remedy would be improvements in the visualization systems.

The methods currently in use for querying data are quite naïve. Approaches such as progressive detail querying and reuse of data on successive queries could prove quite beneficial.

5. Conclusions

We have presented a new approach for remotely visualizing point clouds using an architecture based upon an indexing engine instead of a database specifically designed for spatial data queries. This approach has a number of important features. First, our system is able to support both range and resolution queries on data. This helps keep the bandwidth and computational load low on the client. Second, the indexing engine supports data type flexibility. We can trivially support storage and association of any metadata. Furthermore, we can support fast queries on that metadata without building any additional software. Finally, our system is loosely coupled. This enables dividing resources among many different computer systems. It would be easy to have many machines hosting microservices or even splitting data among multiple machines hosting indexers. The only major disadvantage we have been able to identify with our system is in real performance. Flexibility does not come without a cost. The cost is that our queries sometimes take a few seconds to complete. A custom database built using the same algorithms presented in this paper could return results orders of magnitude more quickly. Speeding up the application could easily be achieved by adding additional servers index and/or microservice servers into the mix. Nevertheless, given the nature of the application, the bottleneck in bandwidth remains a larger factor in performance than the indexer delay.

Acknowledgements

This work was supported by NSF DIBBs [ACI-1443046]. All opinions, findings, conclusions and recommendations expressed in this paper are those of the authors and do not necessarily reflect the National Science Foundation or the University of South Florida.

ORCID

Paul Rosen http://orcid.org/0000-0002-0873-9518
Les A. Piegl http://orcid.org/0000-0003-0629-8496

References

[1] Apache Lucene, http://lucene.apache.org
[2] Apache SOLR, http://lucene.apache.org/solr
[3] Apache Tomcat, http://tomcat.apache.org
[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] Borrmann, D.; Nüchter, A.: Robotic 3D Scan Repository, http://kos.informatik.uni-osnabrueck.de/3Dscans/
[6] Brown, R. A.: Building k-d Tree in O (knlog n) Time, Journal of Computer Graphics Techniques, 4(1), 2015.
[7] Christensen, R.; Wang, L.; Li, F.; Yi, K.; Tang, J.; Natalee 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
[8] Fry, B.; Reas, C.: Processing, http://www.processing.org
[9] Fuchs, H.; Kedem, Z. M.; 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
[10] 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, 617–620. ACM, 2012. http://dx.doi.org/10.1145/2213836.2213914
[11] Isenburg, M.: LAStools: converting, filtering, viewing, processing, and compressing LIDAR data, http://rapidlasso.com/lastools/
[12] Isenburg, M.; Lindstrom, P.; Snoeyink, J.: Lossless compression of predicted floating-point geometry, Computer-Aided Design, 37(8), 2005, 869–877. http://dx.doi.org/10.1016/j.cad.2004.09.015
[13] 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
[14] Klosowski, J. T.; Held, M.; Mitchell, J.; Sowizral, H.; Zikan, K.: Efficient collision detection using bounding volume hierarchies of k-DOPs, Visualization and Computer Graphics, IEEE Transactions on, 4(1), 1998, 21–36. http://dx.doi.org/10.1109/2945.675649
[15] Nyquist, H.: Certain topics in telegraph transmission theory, Journal of the A.I.E.E. 47(3), 1928, 617–644. http://dx.doi.org/10.1109/jaiee.1928.6538024
[16] Rosen, P.; Morris, A.; Payne G.; Keach, B.; Watson, I., Richards-McClung B.; McLennan, J.; Polson, R.; Levey, R.; Ring, T.; Jurrus, E.; Jones, G.M.: Klareco: An Indexing-based Architecture for Interactive Visualization of Heterogeneous Data Sources, 1st Workshop on Data Systems for Interactive Analysis (DSIA), 2015.
[17] Shamir, A.: A survey on mesh segmentation techniques, Computer Graphics Forum, 27(6), 2008, Blackwell Publishing Ltd. http://dx.doi.org/10.1111/j.1467-8659.2007.01103.x
[18] Shannon, C.: Communication in the presence of noise, Proceedings of the IRE, 37(1), 1949, 10–21. http://dx.doi.org/10.1109/jrproc.1949.232969