Extracting and Rendering Primitives from Point Clouds

Thiago Barroso Perrotta, Daniel Pinto Coutinho, Ricardo Guerra Marroquim
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
email: perrotta.thiago@poli.ufrj.br, {dpcoutinho, marroquim}@cos.ufrj.br

***Abstract*—In this work we provide a method to extract and render geometric primitives from point clouds, without the need to convert to polygon meshes. The extraction is realized in two steps: first the basic shapes are detected, then their boundaries are analyzed and stored in a convenient spatial data-structure. Finally, a ray-tracer is employed to directly visualize the extracted primitives.**

***Keywords*-point cloud; primitive detection; ray tracing**

I. Introduction

Due to the popularization of acquisition devices and techniques, such as 3D scanners and photogrammetry, point clouds are becoming a common representation in many application domains. Their main advantage is the simplicity of the data, since no adjacency information is stored. However, this lack of connectivity also renders point clouds challenging in some important aspects, such as reconstruction and visualization.

An important challenge in this field is the extraction of structured information from the point sets. Even though scanners are able to produce datasets in the order of billions of points, often a more compact representation is possible and desired. For example, the set of points representing a plane can be highly redundant, and may be traded for a simple definition of the plane and its physical boundaries.

Another matter with this representations is visualization, since graphics hardware are tailored to deal with triangle meshes. Directly rendering point clouds has also been the focus of many research efforts, to avoid converting the point set to a mesh for example. Nevertheless, our interest lies on the direct visualization of the extracted primitives.

In this work, we propose a primitive extractor from point clouds, and a coupled ray-tracer to directly render our compact structure. An important step in our algorithm is the detection of geometric primitives, such as planes, spheres, cones, cylinders and tori. Having extracted these primitives we delimit their shapes and extract their boundaries to respect as closely as possible the point set. The shape representation is directly visualized using a ray-tracer, to avoid conversions.

II. Primitive Extraction from Point Clouds

Our algorithm is based on the Random Sample Consensus (RANSAC) paradigm [1], a way to fit a model into experimental data. Traditional methods for model fitting try to use most of the available points, where the RANSAC paradigm works in the opposite way, using only the minimum number of points to describe a model (ex. three points completely define a plane).

Briefly, the procedure works in the following way: given a number of points P and a minimal number of points n that describe a model, we randomly select a subset of n points from P and create a model M_1. Subsequently, we validate M_1 by checking how many n_1 points from P are within an error tolerance ϵ to M_1. If n_1 is larger than a given threshold t, we accept M_1 and use the selected n_1 points to calculate a new refined model M_1^*. Otherwise, we start the process again by choosing new random points and creating a new model. If, after some trials, no better consensus has been found, we either choose the one with largest consensus during previous iterations or we terminate the algorithm with failure (in case there was no large enough consensus).

A. Detecting shapes

We detect shapes in points clouds by using the algorithm developed by Schabel [2]. The algorithm works by using the RANSAC procedure where in each iteration a minimal number of points is randomly selected to create one of the possible primitives: sphere, plane, torus, cylinder and cones. When a model of a primitive is accepted, the candidates that fall on the primitive are extracted from the point cloud. The method continues until there are no points left, or no primitive can be extracted given the thresholds. Fig. 1 shows an example of an input point cloud to the algorithm.

Even though this basic method is sound, without further optimizations it suffers heavily of performance issues. However, under the assumption that shapes are a local phenomena, we do not need to create a minimal set by choosing randomly from all the points, we choose points that are spatially close. Exploiting this assumption, we use an octree to establish spatial proximity between points, so we only form a candidate set from points of the same cell.

Every time a primitive is extracted we perform a refitting step using non-linear least squares. Using Newton's algorithm, we refine the extracted primitive model by using all points that were considered to belong to that primitive, and not only the initial random candidates. Finally, we further refine our set of points that belong to the primitive as those that lie at most at a distance of 3ϵ, where ϵ is the distance threshold.

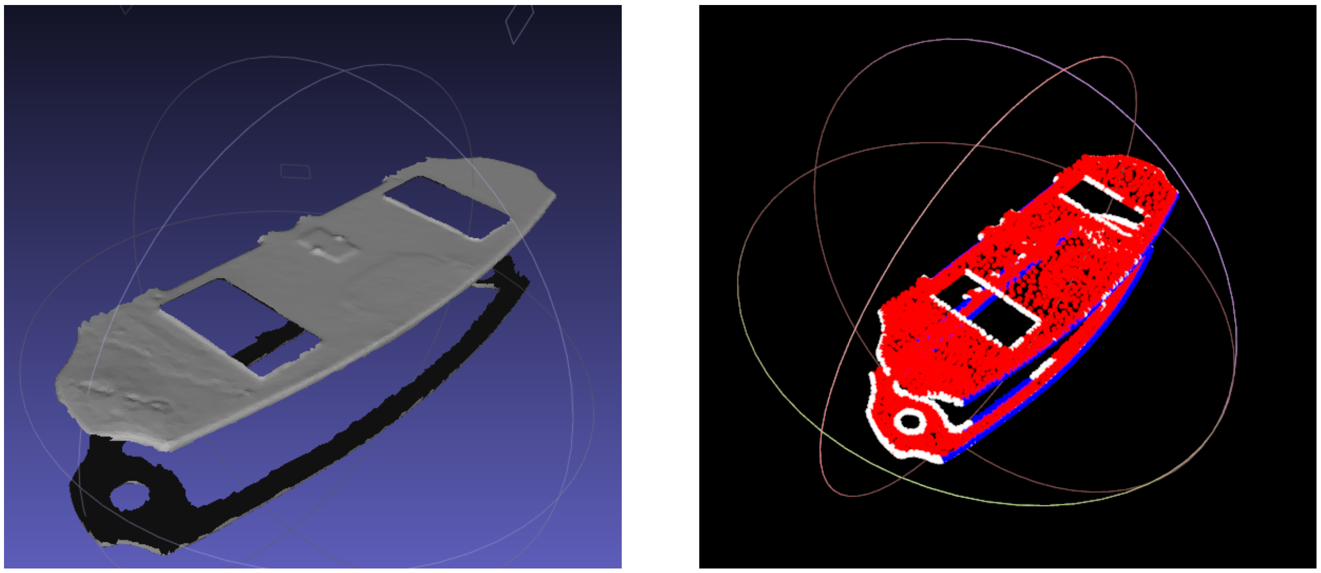

(a) Input mesh. (b) Detected primitives.

Fig. 1: A real case example, from the top and bottom sections of a scanned mechanical instrument. Points in red belong to planes; points in blue belong to cylinders; and white points were not assigned to any primitive. Even though the input is noisy, our extractor was able to retrieve most primitives.

B. Detecting boundaries

Mathematically, most of our primitives are infinite in space. However, to be able to effectively extract the model represented by the point set, we need to know their limits, e.g., the boundaries of plane in our points cloud. An assumption we made is that all of our inputs are closed models, meaning that every primitive is enclosed by others. From this assumption we can infer that the intersection points between the primitives are part of their boundary. Even so, a shape may still contain ”holes”, see Fig. 1a. Therefore, we not only need to detect outer boundaries, but possibly inner boundaries as well. We approach this issue by doing the following:

- Calculating the intersection points;
- Classifying the region in space of one primitive in respect to the other primitives.

Since we are dealing with any given type of intersection, many can be very challenging to solve analytically, such as the torus-torus intersection. Thus, we create a spatial data structure in order to detect the intersection points. More specifically, a KD-Tree is generated where a subdivision occurs every time a node contains more than one primitive. This process continues until the cell is small enough so that its center might be considered as a close approximation to the intersection point between the shapes.

Finally, we classify the region of one primitive. The idea behind our approach derives from the work of Tilove [3] and the main point is to create a series of membership classification (MC) functions for each primitive. Let P_i and P_j be primitives, then we can define a MC function $MC(P_i, P_j)$ which ”classifies” P_i with respect to P_j. For example, if P_i is a cylinder and P_j is a plane, there are three possible classification of the cylinder with respect to the plane:

- The cylinder lies in front of the plane (according to the plane’s normal);
- The cylinder lies behind the plane;
- The plane passes through the cylinder.

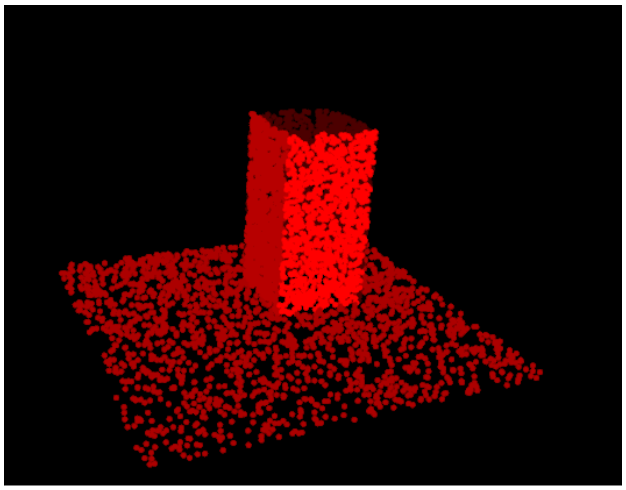
Note that in this last case, we can say that the cylinder is both in front and behind the plane. We follow this last approach in our work.

So, for each primitive we create a CSG tree where each node represents a classification function of the current primitive and another detected one. Each time we create a new node to add in the tree we classify our current primitive with respect to the new one generating two more nodes. Clearly, the height of our tree is $O(n)$, where n is the number of primitives.

In Fig. 2 depicts a simple case to illustrate the method. We can see the lines of intersection points resulting from each vertical plane. These lines illustrate the Membership Classification function as well. Each vertical plane divides the bottom plane in two regions, or patches. Taking into account all four vertical planes, a total of eight patches is generated, each of these representing a leaf node in the CSG tree. All plane’s points of interest are contained in these patches, so we use this information in the Ray Tracing algorithm.

In Fig. 3 an example of a cube is shown. In this case we can clearly see the intersection points delimiting the region of the plane.

Since each primitive requires a very reduced set of parameters to define it, a CSG tree of primitives is indeed very low memory consuming, even in the face of a large number of shapes. With our representation a primitive can be completely described by its boundary points and its CSG tree.


(a) Point cloud containing five planes.

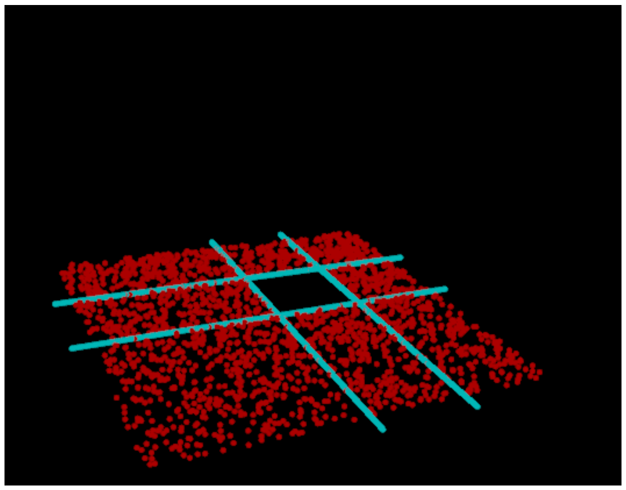
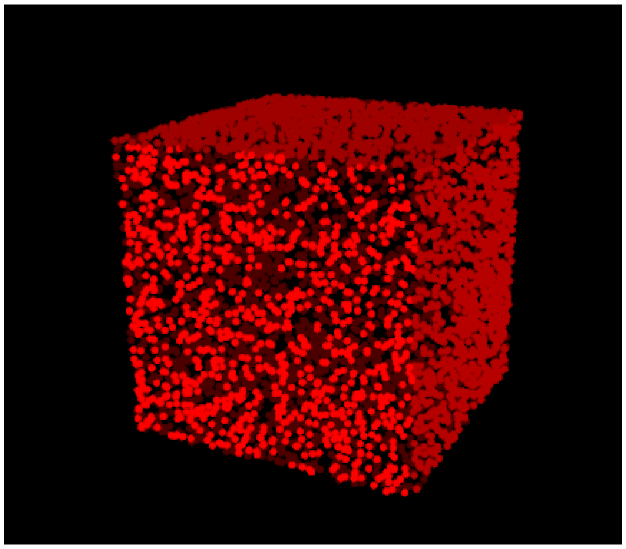

(b) The bottom plane with the intersection lines drawn in cyan.

Fig. 2: A simple point cloud generated from five planes. This division creates eight patches for the bottom plane and we use this information to determine the plane's boundaries for the ray tracing algorithm.


(a) Point cloud of the cube.

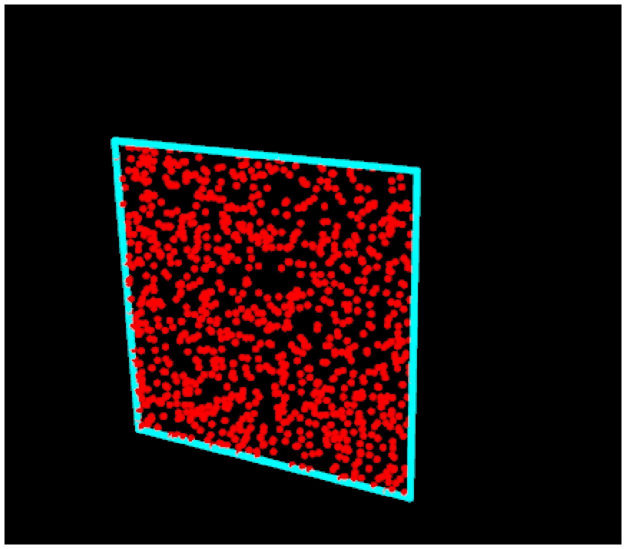

(b) Front side of the cube with the intersections.

Fig. 3: A point cloud representing a cube. Each edge represented by the intersection of the adjacent plane.

Even though the CSG tree alone can describe the spatial boundary of the primitive needed for the ray tracing algorithm, we also detect the intersection points because they may be necessary for other types of operations other than visualization.

III. Ray tracing

Ray tracing is a common technique used in computer graphics for rendering images by tracing the path of light sources through pixels in a viewplane, and simulating the effects of its intersections with virtual objects within a scene.

Its basic operation is to intersect rays with objects, such as the point cloud primitives extracted by our algorithm, then finding the closest hit point p from a given perspective. After finding p, it is only a matter of computing its correct color, primarily by analyzing the light sources around it. Although lights are not strictly needed – that is, we can just assign a simple shading to every point –, they are important for creating some realism in scenes. Another noticeable step is sampling, where several rays are shot through the same pixel of the viewplane to be interpolated later on. This process usually improves the overall anti-aliasing of scenes, and it is specially important when shooting rays into our primitives extracted from point clouds, instead of objects defined by implicit equations.

Fig. 4 illustrates an overview of our algorithm. It works like a conventional ray-tracer; the most remarkable difference is the intersection of rays using the structures from the extraction algorithm. First, a hit function computes the point of intersection between the primitives and a given ray. Then, the hit point is traversed through the CSG tree. If it reaches a leaf node that is not null, then the point has hit a patch and makes part of our primitive, on the contrary, we discard this point since it is outside the boundaries.

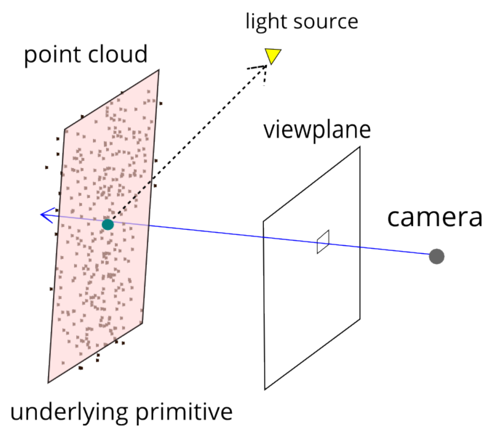


Fig. 4: The ray tracing algorithm builds an image by shooting and extending rays through the scene.

IV. Results

All code was developed in the C++ programming language, and the ray-tracer was primarily based on the work of Suffern[4], although we introduced some small differences in our design.

Fig. 5 shows the result of submitting the extracted primitives from a cube point cloud with 6000 points to our ray-tracer. The extraction process took about one second, while the ray tracing took over two seconds, using, however, ten samples per pixel. Even though the results are still preliminary, the extraction, boundary retrieval, and rendering were accurately achieved with this simple example.

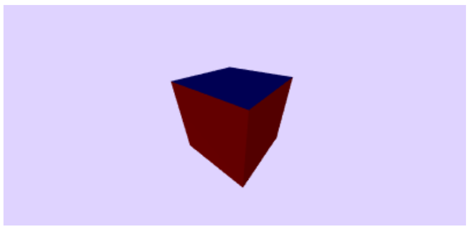


Fig. 5: The cube point cloud, rendered with the ray-tracer.

V. Conclusions

Once the primitives are extracted and their boundaries computed, we achieve a very compact representation that can replace the entire point cloud in many situations. This is specially interesting in some fields such as mechanical pieces, where the primitives can describe the objects in a very compact and precise way. Even more, this leads to a possibly noise free representation that is useful in many domains. The coupled ray-tracer brings even more power to our compact representation, since conversion to triangle meshes, for example, becomes unnecessary.

At this point, the strongest limitation is that our algorithm is not scalable to a large number of primitives, since the memory to calculate all intersections becomes impracticable.

VI. Future Work

We list some important aspects that we would like to improve in our approach:

- improve the overall performance of the primitive extraction. Although it is able to handle small datasets without significant noise, its accuracy drops considerably in real case scenarios;
- due to the lack of accuracy in more complex models, we are not yet able to produce good CSG trees for all primitives. Consequently, the regions are not well delimited , causing erroneous renderings with the ray tracing. This is the reason that only a simple primitive – the cube – was shown here as a result within the complete extraction and rendering pipeline. We are currently working on solving this issue to work with more complex point clouds.
- store and use only information from the boundary points and the CSG tree to improve memory efficiency;
- improve the overall performance and efficiency of the ray-tracer, for example, by parallelizing it using multiple processing cores, and by implementing some acceleration schemes, such as regular grids.

References

[1] M. A. Fischler and R. C. Bolles, “Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography,” *Commun. ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1981. [Online]. Available: http://doi.acm.org/10.1145/358669.358692

[2] R. Schnabel, R. Wahl, and R. Klein, “Efficient ransac for point-cloud shape detection,” *Computer Graphics Forum*, vol. 26, no. 2, pp. 214–226, Jun. 2007.

[3] R. Tilove, H. Voelcker, and A. Requicha, “A study of geometric set-membership classification,” Ph.D. dissertation, 1977.

[4] K. G. Suffern and K. Suffern, *Ray Tracing from the Ground up*. AK Peters, 2007.