RANSAC : Geometry Processing

Maud Buffier *(Co-worker : Hugo Germain)*

May 26, 2017

1 Introduction

For this project, Hugo and I decided to look into the RANSAC algorithm in 3D to detect primitives. The idea was to implement an algorithm based on this paper : «Efficient RANSAC for Point-Cloud Shape Detection» [1]. At the beginning we decided to implement 2 primitive detections : planes and sphere. We implemented the core of the algorithm together and set the best parameters for a scene containing spheres and planar objects (cubes and pyramids). After the core RANSAC algorithm, we implemented a function to fuse the primitives with quasi-similar attributes to achieve better results. To go further, on one hand, I implemented a connected component algorithm to separate distinct objects on the same plane. On the other hand, Hugo implemented a function to be able to recover from missing data points in the mesh.
Here is a photo of a complex scene showing the result of our work :

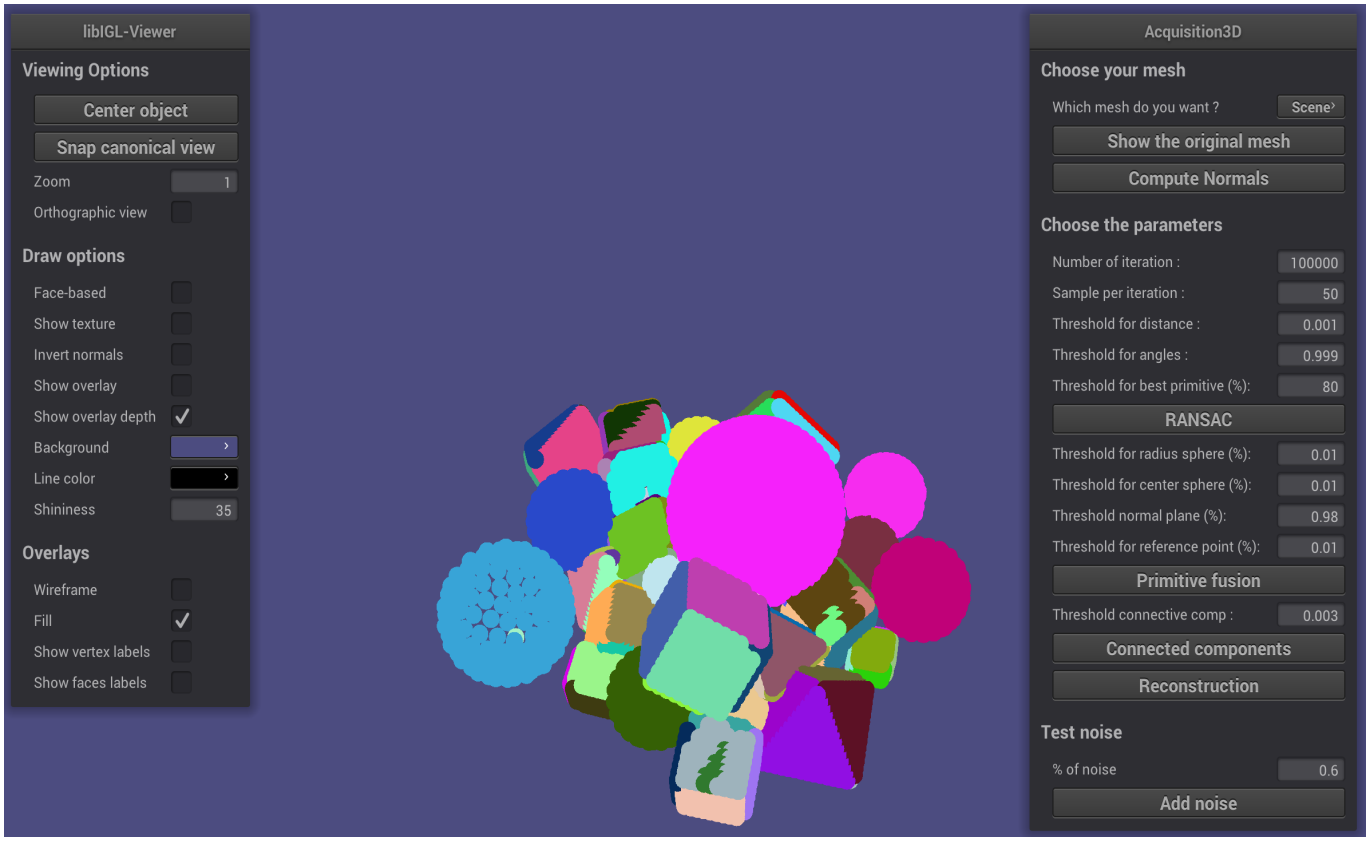


Figure 1: Output of our algorithm for a complex scene with 13000 vertex

During this report, I'll outline the algorithms we used as well as the difficulties we faced and how we overcame them.

Contents

2 RANSAC algorithm

2.1 Description of the algorithm

The main algorithm is in "ransac.cpp". The principal idea is to sample points from the point cloud to test if they could define a primitive. If they do, the inliers that fit this primitive are computed. According to the number of inliers, a score for each primitive is set. The primitive with the "best enough" score (both the best score but which also satisfies a threshold) is accepted and its attributes computed again using all the inliers found. The points are then removed from the cloud in order to continue finding other primitives in the cloud.
Here is the algorithm we implemented :

Algorithm 1 RANSAC ALGORITHM

Input: A 3D point cloud
Output: A cloud manager with one point cloud per primitive found

```
for i < numberIteration do
  for j < numberSample do
    Sample 3 point in the point cloud
    if points describe a plane then
      Compute the normal and a reference point and the score before storing it
    end if
    if points describe a sphere then
      Compute the center and a radius and the score before storing it
    end if
  end for
  Find the primitive with the best score
  if thisScore > 80% then
    Compute and store the inliers
    Clean the cloud from the inliers
  end if
end for
```

Some interesting points can be underlined :

- 3 points are sampled "numberSample" times before extracting the primitive with the best score. This way there is a higher probability to have found something that could be a valid primitive with a high score. Moreover, it's avoiding computing a test each time 3 points are sampled.
- The score needs to be applied to spheres as well as plane
- At each sample, 3 points are sampled with their normals. This way, those points can describe uniquely both a sphere or a plane. On one hand, it offers a "double-check" to avoid misclassification but on the other hand, accurate normals are needed for the algorithm to be efficient.

2.2 Implementation consideration

We implemented our algorithm on the libGl framework in C++. We defined a class "Primitive" and used inheritance to define 2 other classes "Sphere" and "Planes". Some attributes are shared (score, the matrix of inliers index) and other attributes are specific to each (center and radius for the sphere and normal and reference point for the plane).
To store the primitives, we implemented a class "PrimitiveManager" which has a vector of pointers on primitive as attribute. In the RANSAC implementation, a primitiveManager object is used to store all the primitives found. It's freed at the end of the function. An other contains only the best primitives which are stored for further use. The output of our RANSAC algorithm is a "cloudManager" (containing the point cloud for each primitive) and a "PrimitiveManager" (containing the best primitives). The first one is used in the function "gatherCloud" is gestion.cpp to construct one bigger mesh with 1 color per primitive to visualize the result. The primitiveManager is used to improved our classification as we'll see.

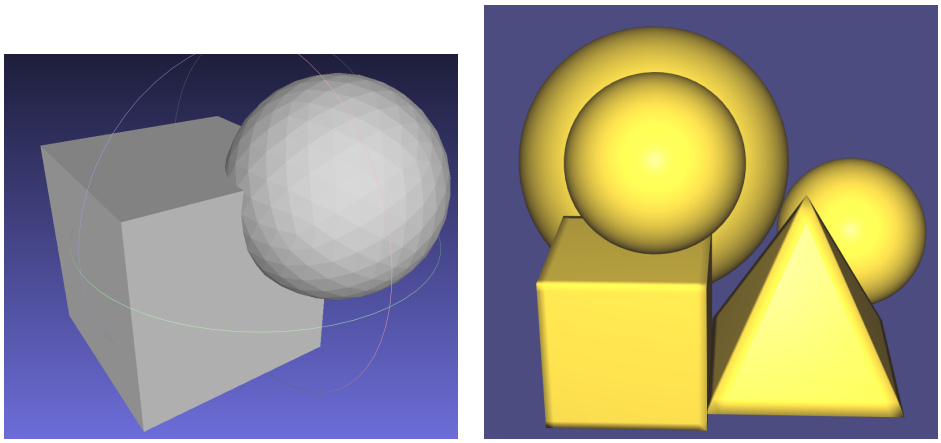
Hence in the code files :

- ransac.cpp is containing the core algorithm
- gestion.cpp contains all the function to deal with the output and compute primitives from the samples
- sphere.cpp and planes.cpp are classes to handle primitives
- evaluation.cpp contains the function to separate objects in the same plane and to add the noise
- reconstruction.cpp contains the function to recover from missing datas in a plane
- the main deals with the interface

We learned a lot about C++ using polymorphism, overwriting functions, building copy constructor or casting pointers during this project.

2.3 Problems when merging meshes

However, we faced an issue during our first tests. As I explained at the beginning, we needed accurate normals to compute our primitives. At first, we tried the algorithm on meshes created and fused using "Cinema4D". We obtained meshes like in fig2 a).



(a) Merged mesh leading to problem (b) Separated objects

Figure 2: Illustration of the different meshes we could have used

However, when fusing the mesh, the software created an unmanifold mesh (a edge was shared by 3 triangles) and we didn't succeed to obtain a manifold mesh from meshlab. Hence, when computing normals from faces using this unmanifold mesh, they weren't accurate. We couldn't cope from that. Moreover, if we computed the normals from vertices they weren't accurate neither. Hence, the results weren't convincing during testing.
Hence, we tried our algorithm on a mesh with separate objects like in the fig 2 b). However, on any mesh with accurate normals the result should be similar.

3 Primitive detections

One of the first challenge in the project was to find a way to define primitives and their attributes. Hugo focused on planes and I, on sphere. Hence, I'll oversee planes implementation and spend more time explaining the implementation for spheres.

3.1 General detection

First, a sample composed of 3 distinct indexes is generated (using the function "sample" in gestion.cpp) . Then, the functions "computeSphere" and "computePlane" are used to test if those 3 points could define a valid primitive using "isSphere" or "isPlane". If it does define one, a primitive is created, and its attributes and score are computed. Then, it's stored in a cloudPrimitive.

3.2 The planes

Planes are described using a normal and a reference point. Given 3 points associated with their normals, a test determines if they could lie in a plane. The normal to the plane defined by those points (3 points define 1 unique plane) is computed and compared to the normal at each point. If the dot product for each point is below a threshold, the plane is accepted.
If the 3 points pass this test, a primitive can be coputed. It will have the mean normal as normal and the mean of the 3 points as reference point.

3.3 The spheres

To detect a sphere from 3 points associated with their normal, a center and a radius are found using 2 points. Then, the third point is used to reject the sphere if it can't lie on it. Indeed, an unique sphere can be created using 3 points with their normals.
Hence, a center C is computed by minimizing the distance between 2 rays defined by 2 points and their 2 normals. A mean radius R is then computed by averaging the distance of the 2 points to the center C. We now want to estimate if the third point could lie on this estimated sphere. To do so, we both look if the distance between this point and the center is approximately equal to the radius. Then, if the vector between the center and the third point is in the same direction as the normal at this point. Hence, only point at the right distance and with a good normal could pass the tests and assure the 3 points sampled are forming a sphere.
The functions to perform this are in "computerRadius" and "computerCenter" in gestion.cpp. "computerRadius" just computes the average distance between a set of points and the center defines in "computerCenter". To compute the center, a least square approach is used to find the point which minimizes all the distances to a set of rays. Using this technique, the best center can be estimated for 2 points with their normals (in "isSphere") as well as with a higher number of points in the next functions. To do so, this linear system of equations is build (from [2]) :

$$\mathbf{R}\mathbf{p} = \mathbf{q} \ ,$$
$$\mathbf{R} = \sum_{j=1}^{K} c_j \left(\mathbf{I} - \mathbf{n}_j \mathbf{n}_j^T\right) \quad , \quad \mathbf{q} = \sum_{j=1}^{K} c_j \left(\mathbf{I} - \mathbf{n}_j \mathbf{n}_j^T\right) \mathbf{a}_j$$

The more inliers on a sphere will be used, the more accurate the center will be.
To solve this equation, I used the Jacobi solver which is numerically very accurate and fast for small matrices (as it's the case here).

Hence, in both cases, if the sample defines a primitive, we need to set a score in function of the number of inliers we obtained for those attributes.

4 Scores computation

To compute scores, we tried several approaches to be able to cope with primitives with different densities and sizes in the same cloud. Moreover, we also needed to have scores we could compare for both our primitives to treat them similarly when looking for the best score.

4.1 First idea

At the beginning we wanted to find a function that takes the variance of the point cloud (seen as the mean distance between points in the principal directions) and the attributes of the primitives to find the "optimum number of points in this primitive". Then the difference between this optimum and the number of inliers would have given us a score. It would have been consistent for both the planes and the sphere.
For the sphere more specifically, I implemented a function "findBestNumberPoints" in "Sphere.cpp" which computed this optimum number. By dividing the area of the sphere (obtained by its radius) by the area of small circle defined using the variance of the point cloud (basically the area around a point which is not occupied by an other points according to the norm of the variance) a number of "optimum points" was obtained for a sphere of this radius given this variance.
However, we computed the variance globally and not locally. Hence, the function weren't really accurate. Moreover, it was hard to obtain a consistence for planes and spheres scores. Hence, we changed strategy.

4.2 Inliers computation

The score implementation is in the "computeScore" function in Sphere.cpp and Plane.cpp. After finding a sample of 3 points which defines a primitive, a good estimate of its attributes is computed. Then, a primitive is created and its score needs to be set. First, the inliers for this primitive are set by iterating on the cloud to find which points could satisfy the primitive's attributes.

4.2.1 Inliers for the planes

To find inliers in planes, the first test verifies if the normal to the point is aligned with the plane's normal. The second test verifies that the normal to the plane and the vector formed by the point and the reference point of the plane are orthogonal. If both tests satisfy the threshold, the point is accepted as an inlier.

4.2.2 Inliers for the spheres

To find inliers in planes, the first test verifies if the distance between the center of the sphere and the point is close to the radius of the primitive. Then, we need to check for normals consistency. As we did for the third points earlier, we now look if the vector formed by the center and the point is aligned with the normal at this point. If tests are satisfied, the point is an inliers.

4.2.3 Implementation issue

Actually, we add to face a problem of normals orientation. Hence, we computed 2 tests for normals consistency in case where the normals at the points were inverted.

4.3 Score computation

Now that we have the number of inliers that fits the primitive we can compute the score. We processed the same way for planes and sphere. We defined a low and a high threshold for the number of inliers. The score is computed only if the number of inliers is above the low threshold for the primitive and as a linear function defined by the high threshold and the actual number of inliers. The code is in the "computeScore" function (sphere.cpp).
To have a consistency between both primitives the score is directly set to 80% if the number of inliers is above the low threshold. The higher number of inliers the primitive has, the higher the score.

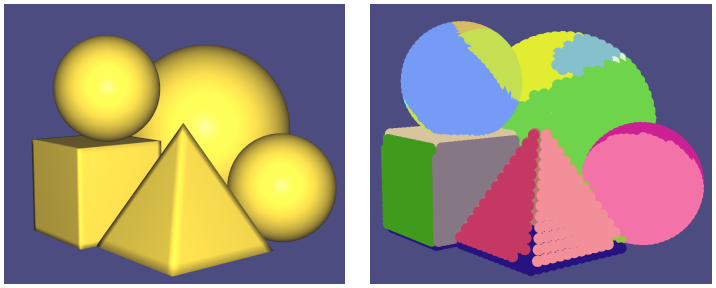
4.4 Amelioration

However, we are aware that the method isn't flexible as we could have a sparse primitive where the number of point is bellow the threshold. Hence, it will never be accepted as a valid primitive and its score always sets to zero. But, beside the number of inliers we weren't able to find a solution to compute a score independent to the mesh density.
However, with our meshes, it's not a problem as we knew the number of point per primitive but it could be in a more general case.

5 Visualization and some ameliorations

5.1 End of the algorithm

Now that we had a score for each primitive, we can select the primitive with the best score. If the score is above the threshold, the primitive is stored in a primitiveCloud, the cloud cleaned from the inliers and the inliers stored in a new cloud in a cloudManager (see function "cleanCloud" in gestion.cpp).
Hence, at the end of the RANSAC function a cloudManager is obtained. The meshes inside can be fused using "gatherClouds". A single big cloud is returned with 1 color per primitive. Here is an example of the output :



(a) Original mesh (b) Output of RANSAC

Figure 3: Output of the algorithm with very high threshold

However, we can see that the output isn't accurate as some sphere are sampled as different primitive as they should be only one.

5.2 Fusion of primitives

Hence we implemented a function which uses the output of RANSAC to fuse the primitives with attributes almost similar. For the spheres, the distance between the 2 centers and the difference of radius are compared. For the planes, the normals and the dot product between one normal and a vector defined by the 2 reference points are compared. This function "fuse" was a challenge to deal with because of memory issue.
We first iterated across all the primitives to label them similarly if they have close attributes. Then, the clouds with similar label are fused and the old clouds deleted. The output of this function is a cloudManager with fused clouds and a cloudPrimitive indicating the type of the cloud at each index (used later). Here is the result of this function :

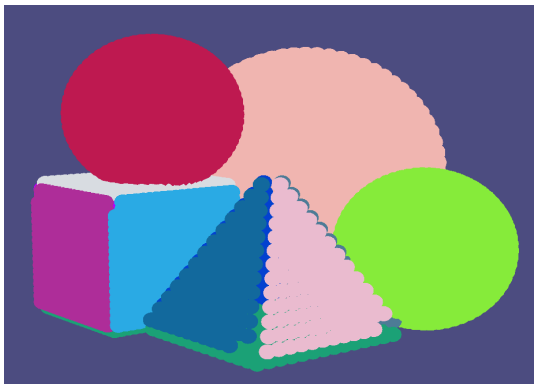


Figure 4: Result after fusion

The advantage of this technique is that we can set high thresholds to avoid misclassifying vertices between planes and sphere and still obtained a good result at the end. It also very useful in the case of noise as we'll see.

6 Parameters

The RANSAC algorithm major parameter is the number of samples used to find the primitives. Indeed, this number is specific to the mesh we have. It depends both on the number of vertices but also the number of primitive to estimate in the mesh. Hence, we'll start by looking at this parameter and then look at the thresholds we need to put to avoid misclassification of planes and spheres.
To compute efficient testing we implemented an interface to be able to move between meshes, and applied the algorithm with different parameters :

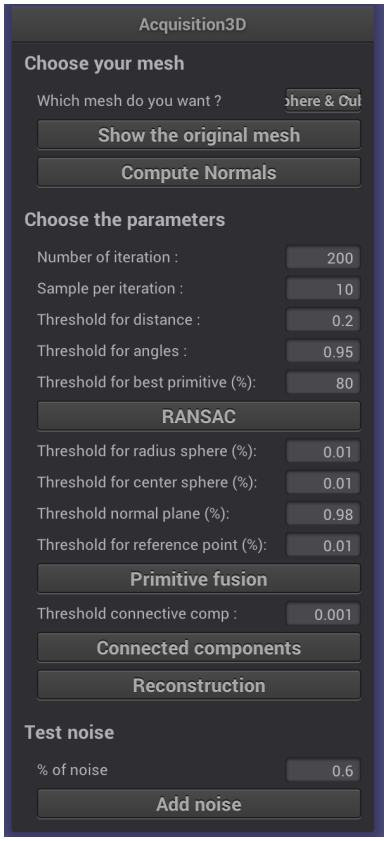
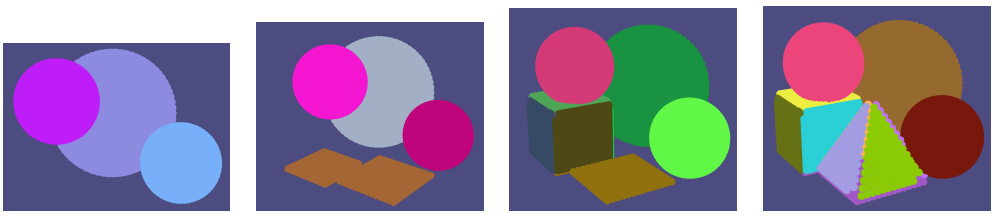


Figure 5: Interface computed

6.1 Number of samples

This test was computed on a small mesh. The number of sample includes both the number of sample and the number of iteration from the previous pseudo-code. Then, the number of primitive detected over the total number of primitives (14) is computed. The percentage of points found as inliers in a primitive with respect to the number of point in the original mesh is also computed.
Here are the results of my experiment in term of visualization :



(a) 140 samples (b) 220 samples (c) 500 samples (d) 1000 samples

Figure 6: Result by varying the number of samples in the mesh

and in term of testing :

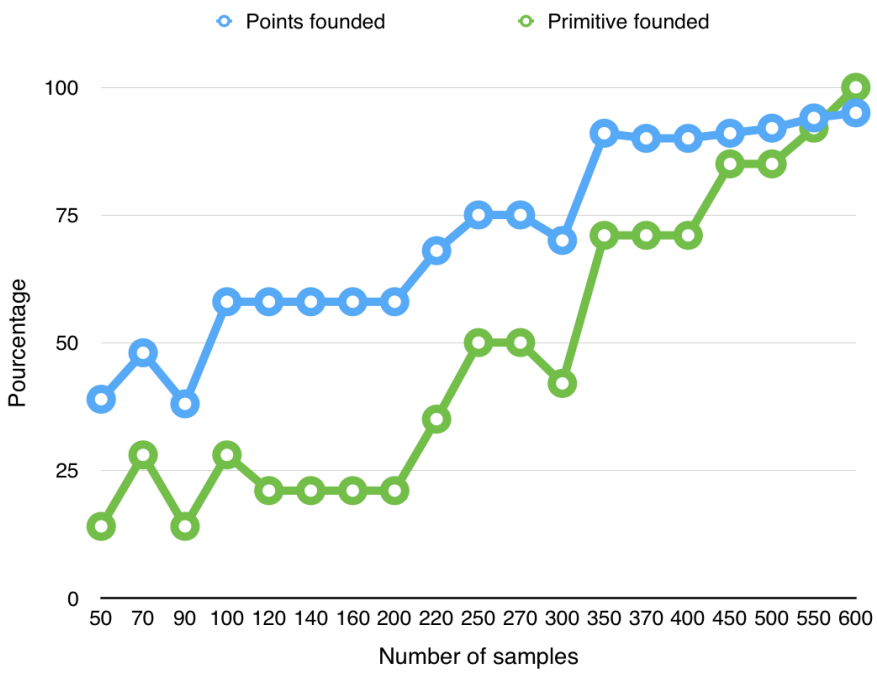


Figure 7: Graphical result showing the evolution (on %) of the number of points in the RANSAC result and the number of primitive detected in function of the number of samples

According to my testing, for this mesh (composed of 7000 vertices and 14 primitives) I need to put at least 1000 samples to be sure to fine, in most of the case, every primitives.
We can also underline that spheres are founded more easily than planes as they have a higher number of points.
Of course, this number is specific to this mesh and the number of sample needs to be increased in the case of a bigger mesh with more primitives.

6.2 Thresholds for primitives

The other important factor in our RANSAC algorithm is the way to define primitives. In both cases, we use 2 thresholds :

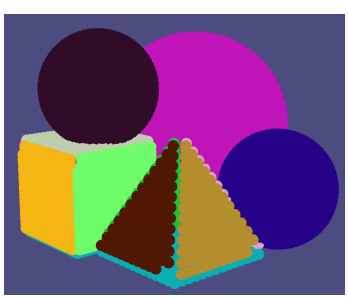
- One for the distance between points and the primitive
- One for the angle between normals of point and the normal estimate by the primitives

We'll try to understand how we need to adjust those thresholds to obtain the best result possible and observe their influence on the primitive computed.

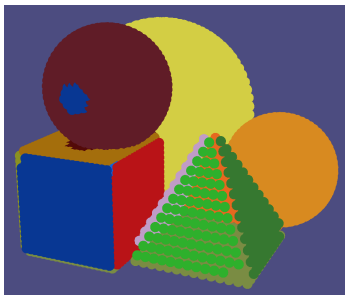
6.2.1 Distance threshold

When computing the inliers, this threshold rejects points to distant from the center of the sphere or too far from the plane. During the testings, the threshold for angles was set to a middle value (0.97) and 1000 samples were used.

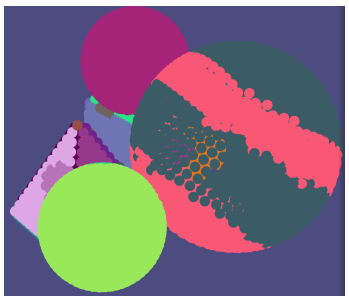
Here are some results :



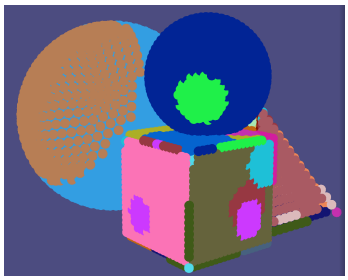
(a) Threshold at : 0.001



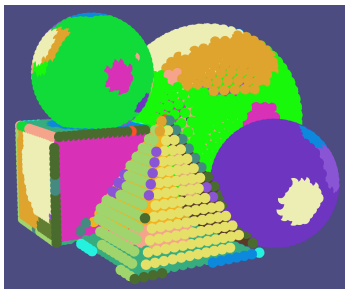
(b) Threshold at : 0.09



(c) Threshold at : 0.15



(d) Threshold at : 0.2



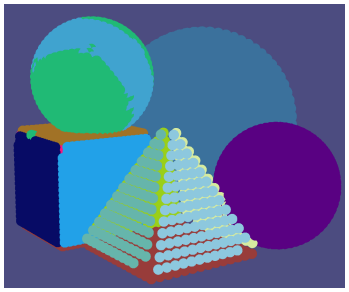
(e) Threshold at : 0.3

Figure 8: Result by varying the threshold for the distance

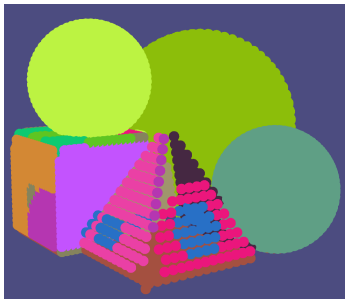
We can observe the effect of increasing the threshold (for this threshold of normal). First, in b) some points of a sphere are selected in the plane because their distance is small. Then, some entire slices of spheres (c) and d)) are selected as planes. When the threshold is way too high (e)) we can't even interpret the result and points are added to primitives too easily.
We can notice that having a high distance threshold tends to mis-classify spheres into planes.

6.2.2 Normal threshold

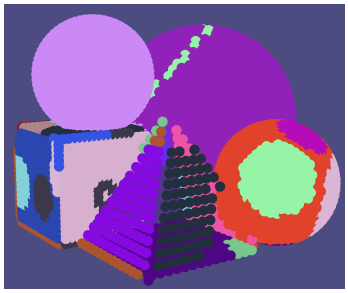
The threshold for the normals compares the normal of the point to the normal estimate by the primitive (directly the normal for the planes and the vector from the center of the point for the sphere). It acts as a double check to avoid misclassification. The following test were computed with a fixed distance threshold of 0.05.
Here are my results :



(a) Threshold at : 0.99



(b) Threshold at : 0.95



(c) Threshold at : 0.9

Figure 9: Result by varying the threshold for the angle

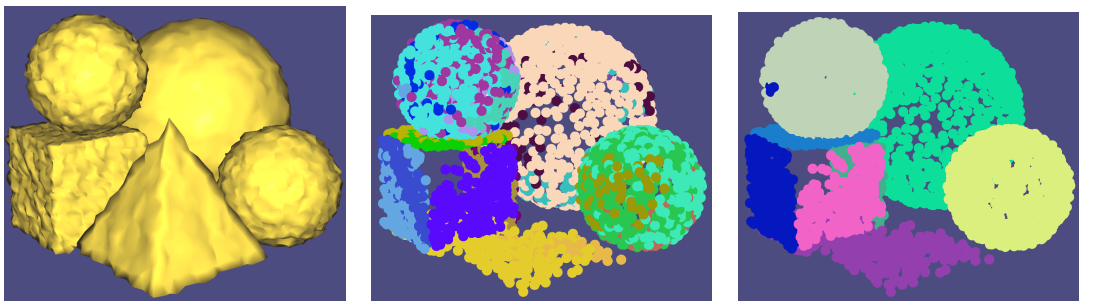
Planes are still fitted on spheres, but now, part of the cube are also fitted as a spheres, like the blue part on b) for example.
Hence, I would say this threshold tends to prevent over-fitting spheres to the mesh.

6.3 Score threshold

An other parameters we could have played with is the threshold to accept a primitive. However, in our case it's not very relevant as our score can only go from 80 and 100 as soon as a certain number of inliers is achieved. But, a better computation of the score could have allow us to test this parameter as well.

7 Behavior with noise

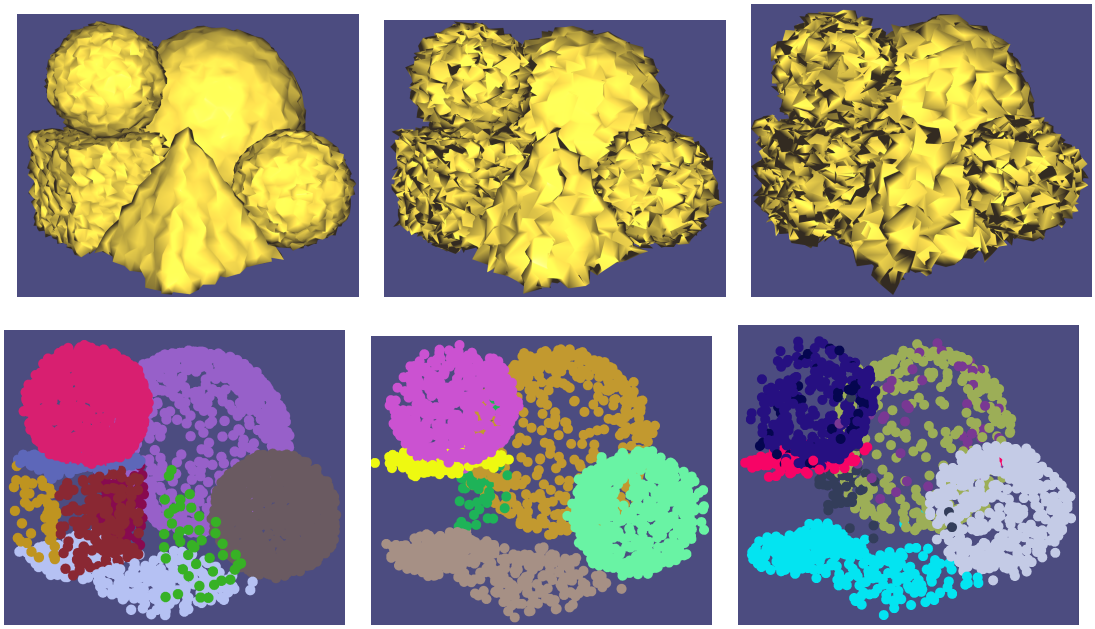
An other interesting point for the algorithm is its resilience to noise. A function to add noise to the mesh (both to the vertices and the normals) was implemented. The amount of noise added is described as a percentage of the bounding box for the mesh (the code is in "evaluation.cpp"). For the algorithm to continue working, a larger number of samples is needed (over 10000). The algorithm can then recover from noise by having more chances to find points less affected by noise. Moreover, some choices had to be made in this case : do we prefer find less primitives (but accurate ones) or, over-segment the mesh by setting less selective thresholds ? In our case, we preferred to look for "the most complete" primitives possible by keeping the same selective thresholds as before (0.01 for the distance and 0.99 for the angle). Moreover, a function to fuse primitives with close attributes was implemented. Hence, we can lower thresholds on this function to merge primitives more easily and then, obtain a good result even in case of noisy mesh.
Here is an example of merging primitives in the case of a noisy mesh :



(a) Noisy mesh (b) Output of RANSAC (c) Output of the fusion algorithm

Figure 10: Utility of the function "fuse" to recover from noise

Here is some tests with more noise :



(a) Noise of 0.03 (b) Noise of 0.06 (c) Noise of 0.08

Figure 11: Effect of noise on the algorithm after applying the fusion of primitives

Hence, even if the mesh is very noisy, primitives can still be found. Spheres are better recovered because they contains a greater number of points. Similarly, the plane at the bottom is still present as it has a high number of vertices. Once again, the fusion algorithm is very useful in this case to recover from noise.
Here is a chart showing the effect of the noise on the inliers in the algorithm :

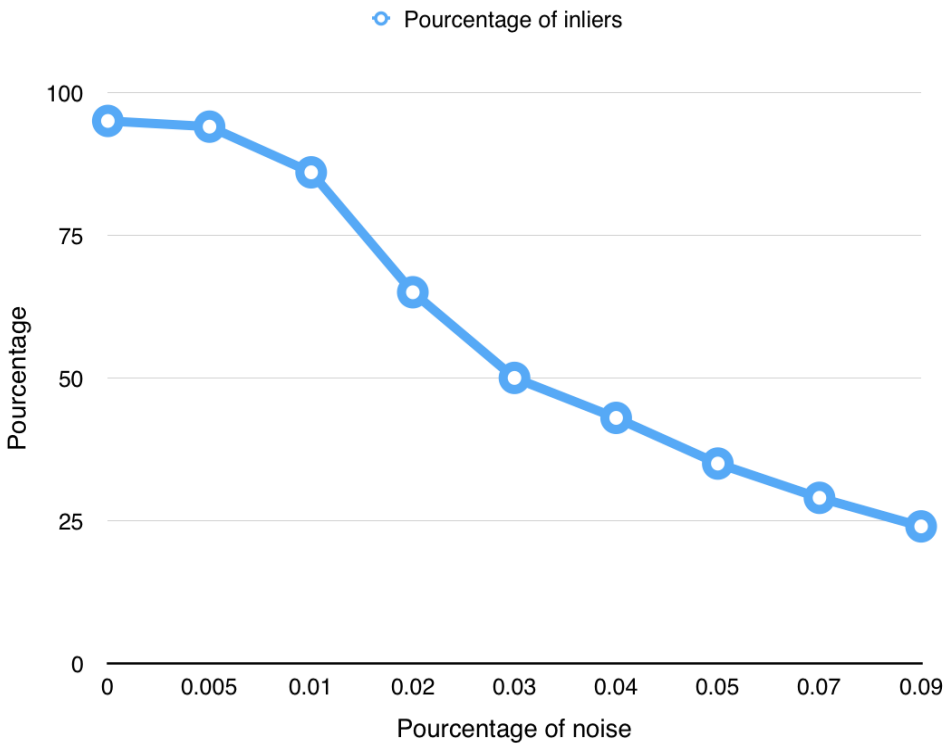


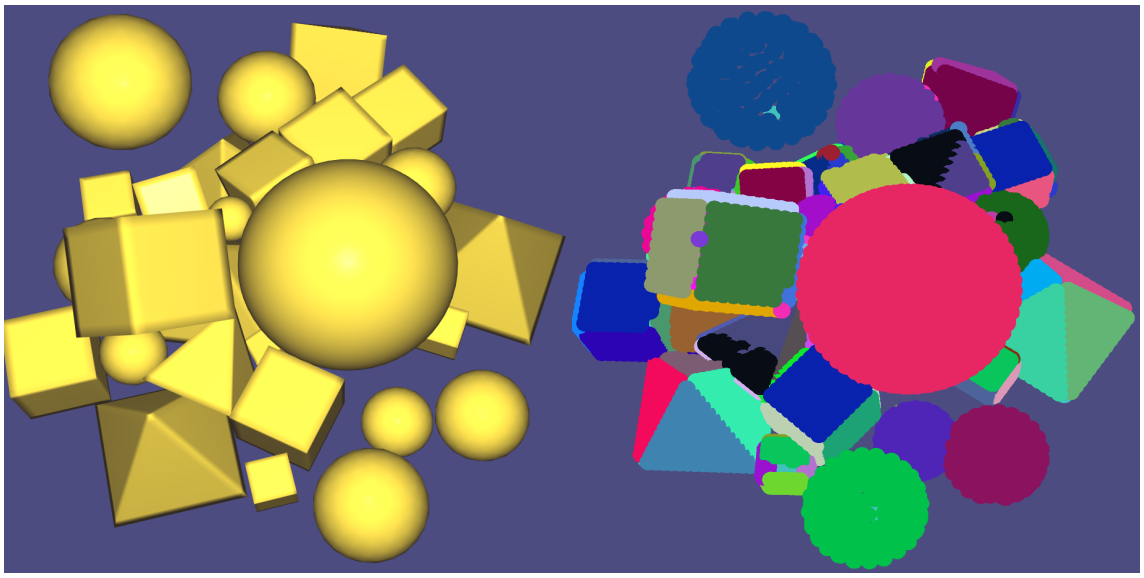
Figure 12: Curve of the percentage of inliers (with respect to the number of vertex in the original mesh) in function of the percentage of noise added

Hence, even if the primitives are still visible in the output, the number of inliers decreases drastically with the noise.

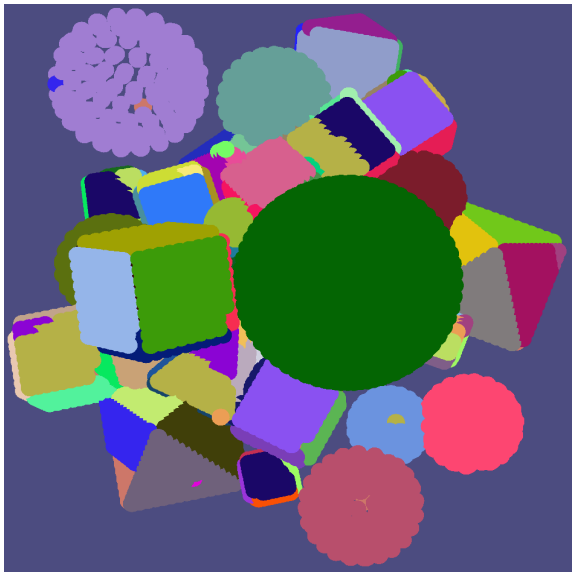
8 Complex scene

8.1 Results

To test our algorithm in more challenging situations, we created a more complex scene with more primitives of various sizes and densities. To detect all primitives (more than 20 with 13702 vertex in total), a huge number of samples were needed (100 000).
Here is the result when applying the RANSAC algorithm as well as the primitive fusions :



(a) Complex mesh (b) RANSAC output



(c) Primitive fusion output

Figure 13: A more complex scene

It's interesting to notice that the percentage of point detected to be a part of a primitive with respect to the number of point in the original mesh is almost the same that in the previous example : 94.84%. Hence, the number of point "missed" is sensitively the same between meshes.

8.2 Possible ameliorations

Here is some ameliorations we could have worked on :

- We can notice that there are some isolated points we could get rid off by looking the distance between the points in the mesh of a primitive. Points which are not surrounded by points of the same label could be eliminated.
- Some planes are "cut in half" and our "fuse" function doesn't bring them back together. It may also be due to problems in the normals computation where normals get inverted in the middle of a face. We weren't able to cope with this normals issue.

9 Team work

To be able to work as a team we use a git repository which can be found at this address : https://github.com/germain-hug/RANSAC.
At the beginning of the project, we discussed the core algorithm together and designed the code by setting all the classes and function prototypes. Then, he implemented functions related to planes and I did the ones related to spheres. For the rest of the functions, we worked together, debugging or improving the code already made using GIT.
For the following sections, we both implemented an amelioration of the algorithm individually. However, we discussed the results and the issues we faced together as a team.

10 Amelioration

We computed 2 ameliorations for our project, the first one is a connected component algorithm to separate object in the same plane and the second one is a filling function to recover from missing data. Hugo implemented the last one and I did the connected component algorithm.

10.1 Connected components

Our algorithm works but we wanted to be able to separate faces of the object on the same plane. For example, in this example :

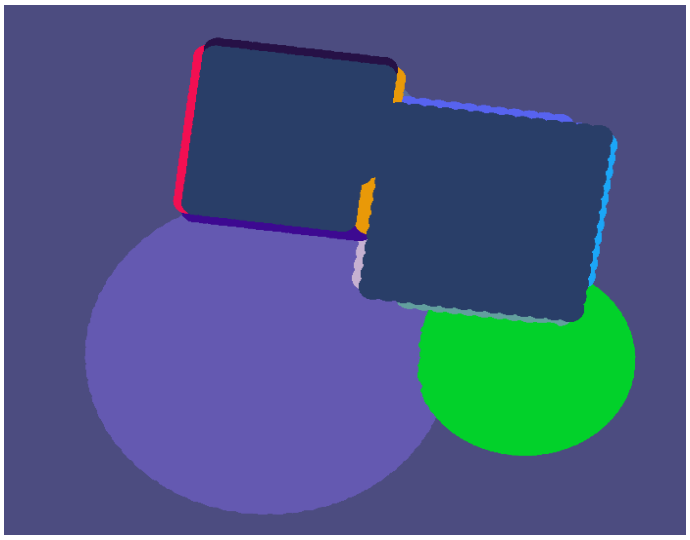


Figure 14: Before connected components algorithm

We would like to be able to differentiate the bases of the cube and the pyramid as 2 distinct objects even if they are in the same plane.

10.1.1 Algorithms

This algorithm is in "evaluation.cpp" and can be describe using 2 pseudo-code algorithms. It's a recursive algorithm which is taking the distance as threshold

Algorithm 2 Connected components

Input: A 3D point cloud, a threshold
Output: A color matrix the same size as the vertices
  $visited \leftarrow vector(0)$
  $label \leftarrow rand(color)$
  $colors \leftarrow vector(0)$
  Compute a kd-tree for the vertices of this mesh
  for i < numberVertices **do**
    if visited(i)=0 **then**
      $labelVertices(i, label, colors, vertices, visited, kdTree, threshold)$
      $label = rand(color)$
    end if
  end for
  Return the color matrix

which call recursively :

Algorithm 3 Label function

Input: Index, label, colors, vertices, visited, kdTree, threshold
Output: A label and a color set for this Index position
  $visited(Index) \leftarrow 1$
  $colors(Index) \leftarrow label$
  Find the 8 closest neighbors of the vertex index in vertices using the kdTree
  for i $<$ 8 **do**
    if visited(thisNeighborIndex) $=$ 0 **then**
      if distance(thisNeighborIndex) $<$ threshold **then**
        $labelVertices(thisNeighborIndex, label, colors, vertices, visited, kdTree, threshold)$
      end if
    end if
  end for

Hence, we can see that if the distance is above the threshold, the recursive function will go back at the beginning, change the label and look for unvisited vertex to label.

10.1.2 Implementation

To compute the kd-Tree and find the nearest neighbors I used the ANN library. Indeed, it's more accurate that FLANN and the speed was good enough for our purpose.
I also apply the connected component algorithm only on the plane clouds because the vertices on the sphere are more spaced. Indeed, to separate all the planes in the complex scene, I needed to have a low threshold and I arrived to artifacts like this one :

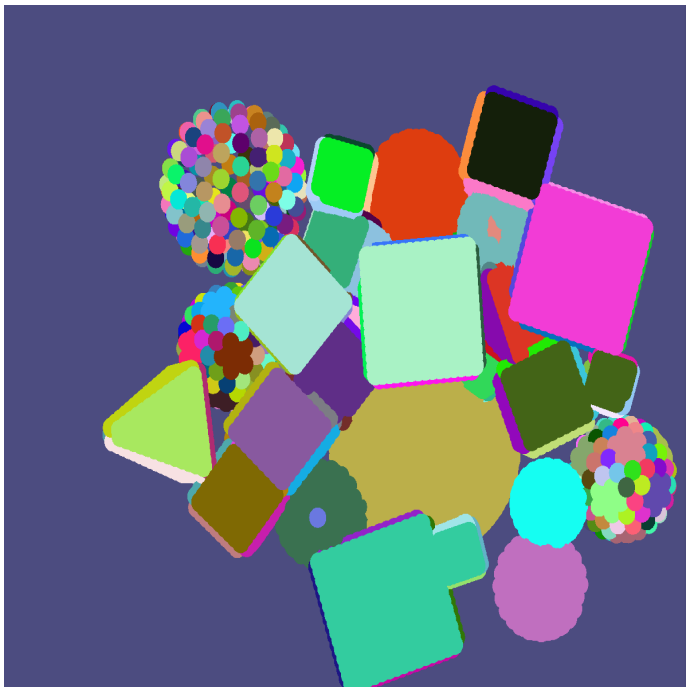
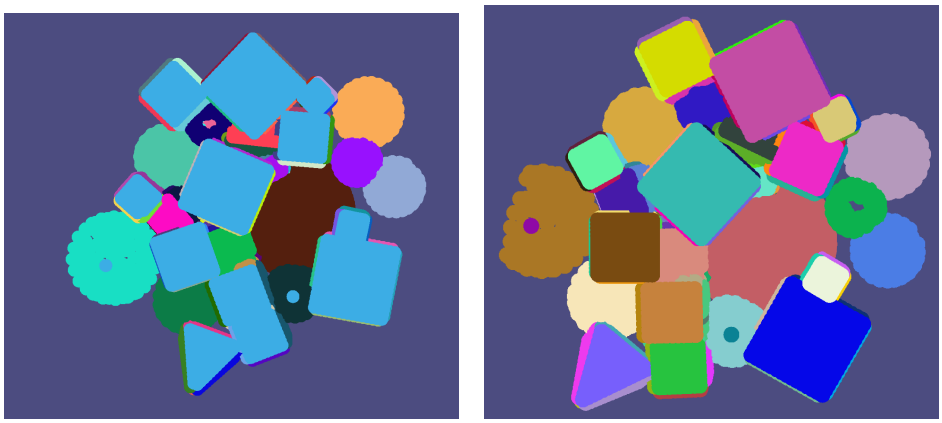


Figure 15: Artifact using the connected component algorithm on spheres. All points are thought to be from different meshes

where point on the sphere are considered to be from different meshes.
Hence, I used the primitiveManager output from the fuse function to determine if the cloud was a sphere or a plane and if I add to apply the connected components or not.

10.1.3 Result

Here is the result for the complex scene :



(a) Before (b) After

Figure 16: Result of the connected components algorithm

We can see that all the objects on the same plane are now labeled with a different colors even if they are very close to each other.

10.1.4 Possible ameliorations

We can see that some points (purple in the brown sphere at the left for example) are isolated. They probably were associated with the plane which was found before the sphere. Hence, now they look like outliers in the mesh. We could have tried to erase points which are too far from the others in the cloud (obviously outliers).

10.2 Filling holes in a plane

Hugo implemented this section. The idea was to be able to reconstruct plane which have been damaged like this one :

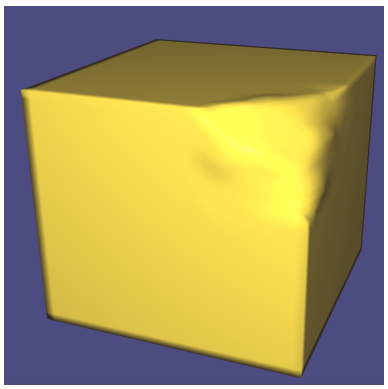
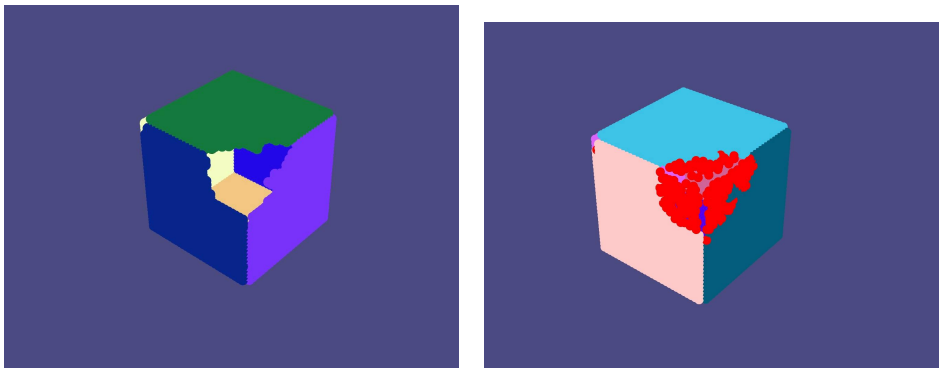


Figure 17: Result of the connected components algorithm

To do so, for each planar primitive, he computed a bounding box around the primitive cloud. Point on the bounding box are projected on a plane defined by 2 vectors formed by 3 points in the mesh. A "bounding square" in 2D is computed using those projections. Then, he sampled a high number of time in this 2D plane adding the point if the smallest distance to its neighbors is above a given threshold. The 2D points are then projected back in 3D using the frame computed above and the normals of the plane.
Here is an example of the result he obtained :



(a) Planes fitted by the RANSAC function

(b) Red point have been added to the mesh to fill the holes

Figure 18: Result of the filling holes function

11 Conclusion

We achieved an implementation of the algorithm to detect 2 types of primitive. Moreover, we tried to implement it in order to be able to add new primitives easily (inheritance, no hard coding for the size of the primitives and cloud manager ect..). We also implemented 2 extensions which convincing results.
However, it would have been nice to test the algorithm on more complex and "professional" meshes like the ones in the article.

References

[1] Reinhard Klein Ruwen Schnabel, Roland Wahl. Efficient ransac for point-cloud shape detection. 2007.

[2] Johannes Traa. Least-squares intersection of lines.