COMPM080: Next View Planning Project

Due on Wednesday, June 1st, 2016

Maria Ruxandra Robu- 14042500

Introduction

This project introduces the concept of Next View Planning for structure from motion. The papers I consulted for developing this project were [1, 2, 3, 4]. Therefore, the concept of Next View Planning can be summarised as:

Given an unknown 3D point cloud and scans from k viewpoints, what is the next best viewpoint that we can select for the next scan? The aim would be to have the shortest path of next best views that would allow for a complete reconstruction of the original model.

Algorithm 1: 3D object reconstruction with next view planning

Input: k Camera positions: (cam_k)
Result: Object Model: $\tilde{M}$
for *each* cam_i *in the k positions* **do**
    $scan_i \leftarrow$ Projected point cloud from cam_i;
    $\tilde{M} \leftarrow$ Update reconstructed model by merging $scan_i$;
while *true* **do**
    $nbv \leftarrow$ Compute next best view (Section 2);
    if *Optional: Is nbv really the best view?* **then**
        $score_nbv \leftarrow$ Evaluate using ground truth model M;
    if *nbv does not provide new information* **then**
        Return $\tilde{M}$;
        Finish reconstruction;
    $scan_{k+1} \leftarrow$ Projected point cloud from $nbv = cam_{k+1}$;
    $\tilde{M} \leftarrow$ Update reconstructed model by merging $scan_{k+1}$;

The aim of the project was:

- to build the core simulation framework (which was developed in a group with Karina Mady)
- to build upon the framework and explore different solutions for choosing the next best view (individual task)

What we have achieved:

- to build a working simulation framework that can process point clouds (i.e. read/write, add cameras, project and backproject the points, evaluation), which will be detailed in Section 1
- to implement a fast search based strategy to find the next best view, by using 2 different scores (to maximize the new information brought by each scan and the quality of the points scanned), which will be detailed in Section 2

Libraries used:

- Eigen - whole project
- OpenMesh - I used it for reading and writing point clouds
- ANN - I used it to compute the normals for the point cloud

The project was developed on both Mac OS(Maria) and Windows(Karina) and we used the same IDE: Clion. However, we could not make the debugger work (most probably due to the use of libraries). As a result, I used the terminal and the CMakeLists for compiling and running. The GitHub repository for the core simulation

Introduction continued on next page...

framework is: `https://github.com/xmariuca/NextViewPlanning`. The list of commits along with this report should help clarify our individual contributions. Karina had trouble integrating OpenMesh, so the Master branch uses reading and writing functions developed by Karina, whereas the testWOpenMesh branch uses OpenMesh for I/O. We used MeshLab to visualize the results.

Please note that the paths for the meshes used are hard coded in defs.h, which would have to be modified according to your project folder structure for running.

1. Core simulation framework

The core framework was structured around the main classes: **PointCloud** and **Camera** using helper functions from **utilities** and **NextBestView**. We have discussed all of the implementation details together. Figure 1 shows the division of tasks. However, in most of the cases in this common part of the project, we collaborated through feedback and testing to improve each other's functions. Throughout this report, I will specify whenever I discuss functions that Karina wrote. If nothing is stated, it is my implementation.

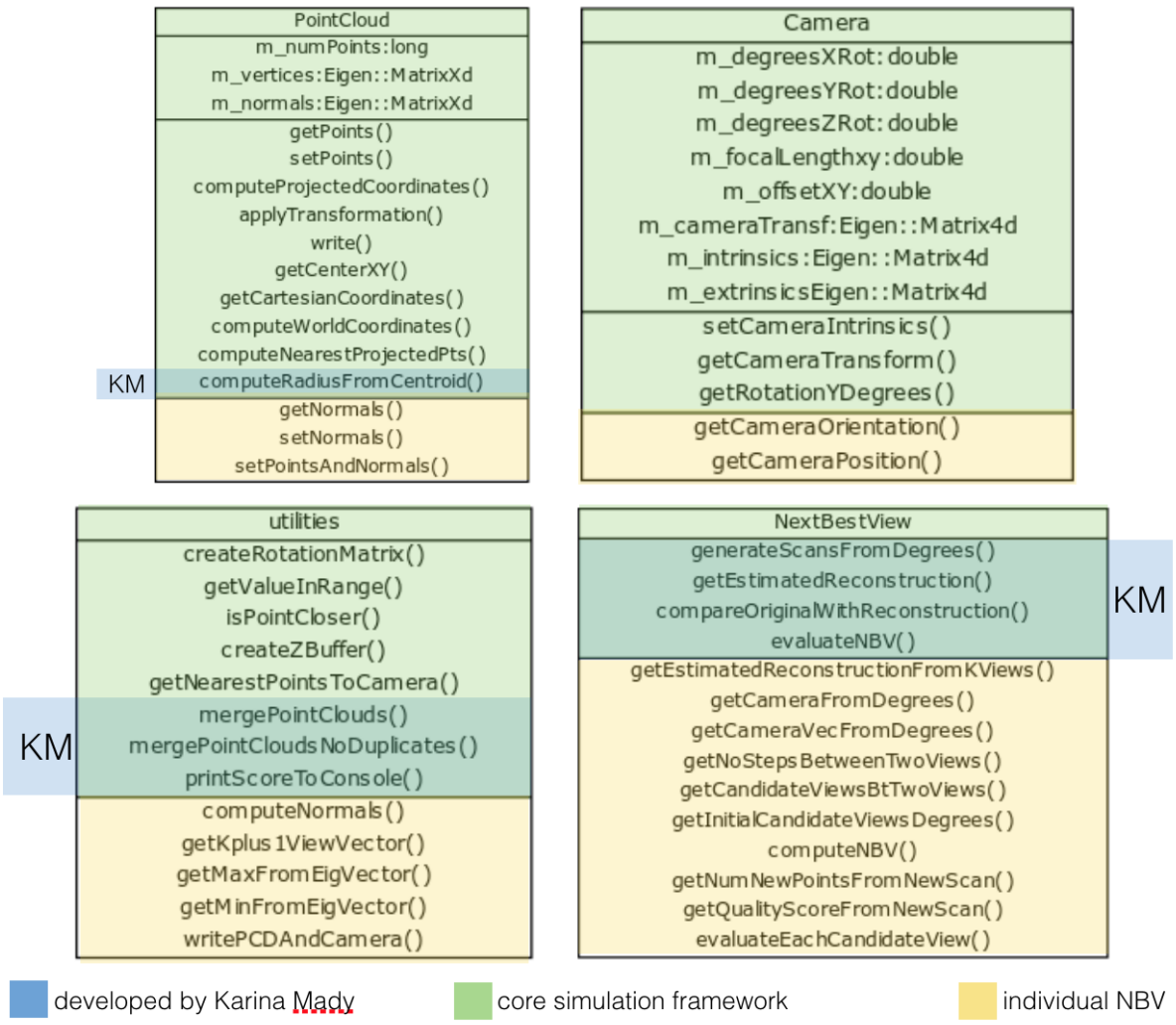

Figure 1: UML structure for the main functions of the Next View Planning project. The main structure of the framework was created by me while Karina set up her development environment. The functions wrote by her are highlighted with blue. The green and yellow tints differentiate between the core framework and the individual task.

It was difficult to get the framework starting in the beginning because of we were working on different operating systems. While I/O operations worked fine with OpenMesh on my laptop, Karina could not make it work. We chose to use the reader and writer in OpenMesh because of its ease to read any input format for the mesh. Also, it made the I/O operations faster since it loads the mesh in a half edge data structure,

which was more efficient to traverse. While Karina set up her read and write custom functions, I created the project folder and file structure. I used a class structure similar to my previous courseworks in this module. Note that some of the functions (i.e. in the PointCloud class) are copied from my previous implementations and adapted for this particular scenario.

I will go over a few design choices made and implemented in the core framework. I will mostly explain in more details the functions I wrote and mention brief explanations about Karina's work and how they all fit together.

Camera model

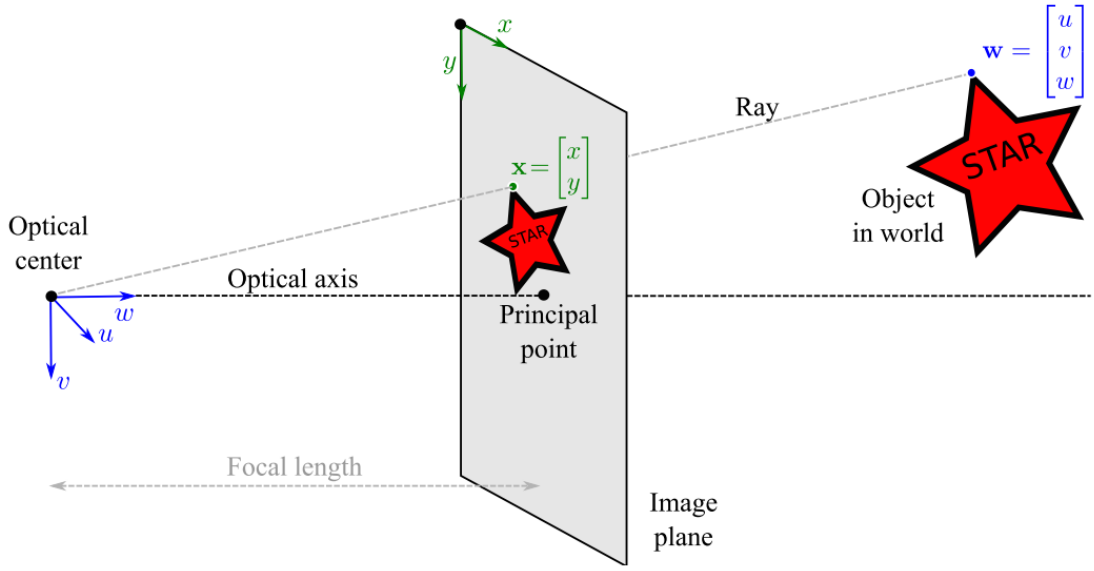

Figure 2: The pinhole camera model. The optical center corresponds to the world origin in camera coordinates. The virtual image is formed on the image plane, which is displaced from the optical center (focal length). The principal point is the location where the optical axis intersects with the image plane. Figure taken from [5].

The pinhole camera model was used for simulating a camera from a specific viewpoint. A schematic of the main intrinsic parameters is shown in Figure 2. Note that we have used homogeneous coordinates for the following calculations to make the system linear [5]. A pinhole camera model has an intrinsic matrix:

$$M_{intrinsic} = \begin{bmatrix} \theta_x & \gamma & \delta_x & 0 \\ 0 & \theta_y & \delta_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where θ_x and θ_y are the focal lengths along the x and y directions, γ is the skew and δ_x and δ_y are the offsets on x and y to the principal point. In this implementation, we set the skew to 0 and we keep the focal length and the offsets fixed throughout all the computations. The parameters are equal for both x and y and are set empirically. It was difficult to figure out how their values influence the scans given since we did not find any intuitive visualisation in MeshLab. This is the reason why they are not used at all in the simulation of

the scans, where a z buffer is used instead.

The extrinsic matrix of the camera is the one that we modify to get different camera positions.

$$M_{extrinsic} = \begin{bmatrix} R_{3x3} & t_{3x1} \\ 0_{1x3} & 1 \end{bmatrix}$$

The 3 by 3 rotation matrix changes with each new camera position, since we are looking for different rotations around the y axis. The 3 by 1 translation vector is chosen correspondingly for each new point clouds. The x and y translations are set to be the equal to the centroid of the point cloud (the mean on x and y). The z translation is chosen to be twice the longest distance between two points inside the input point cloud (function written by Karina).

Another theoretical aspect I wanted to understand better was the projection and back-projection of the point cloud. This website: `http://ksimek.github.io/2012/08/22/extrinsic/` helped me understand understand what the relationship is between the translation column in the extrinsic matrix and the actual pose of the camera. Hence, I managed to extract the camera position and orientation from each camera's extrinsic matrix and I visualized them with MeshLab (see Figure 3). The camera position and orientation will be used in the individual part in the evaluation of new scans.

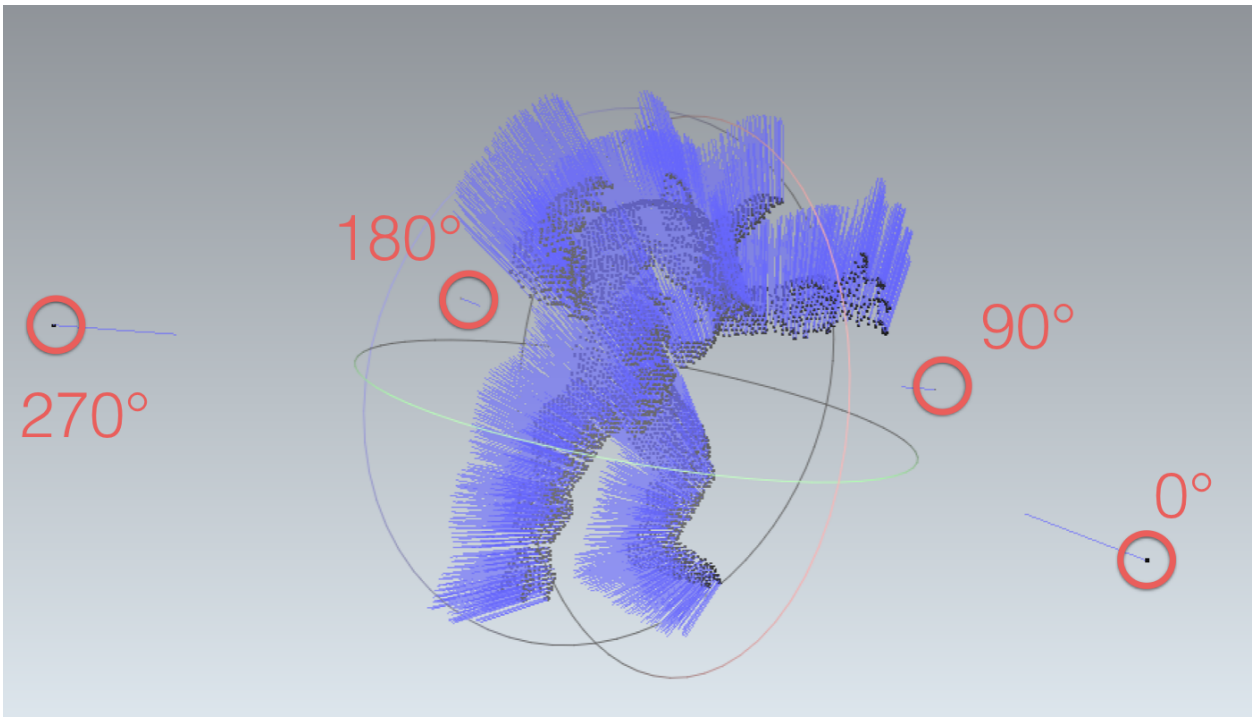

Figure 3: 4 Camera positions and orientations with respect to the armadillo point cloud

Finally, using this camera model, a transformation from world coordinates to camera coordinates $T_{world2cam}$ can be computed as:

$$T_{world2cam} = M_{intrinsic} * M_{extrinsic}$$

```
void PointCloud::computeProjectedCoordinates(Camera &camera);
```

Similarly, the back projection from camera coordinates to world coordinates $T_{cam2world}$ is computed as the inverse:

$$T_{cam2world} = T_{world2cam}^{-1}$$

```
void PointCloud::computeWorldCoordinates(Camera &camera);
```

These functions project the whole point cloud into a scan. However, we wanted to simulate a scan with occlusions which contains only the closest points, as would happen in a realistic scenario.

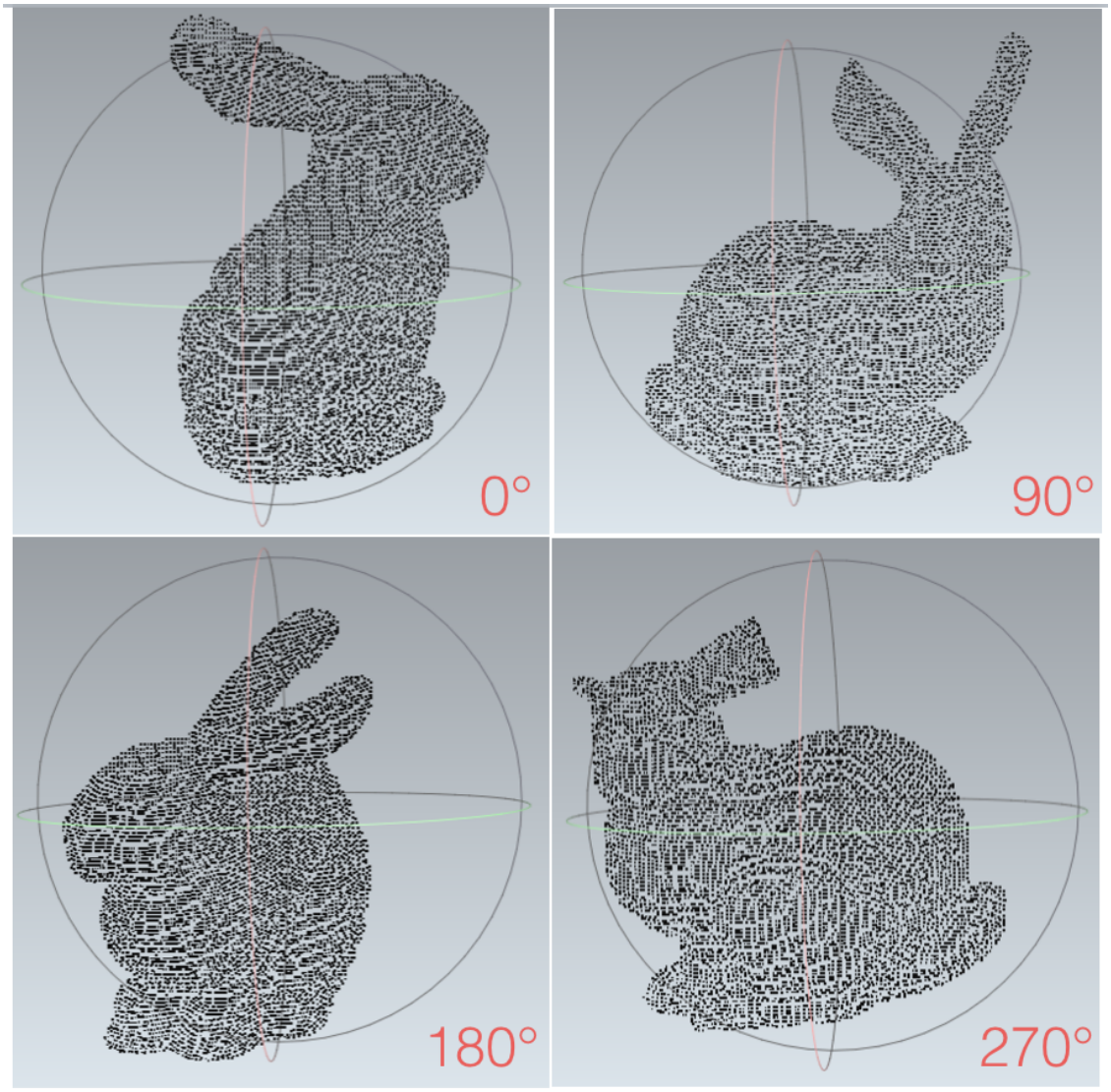

Figure 4: Simulated scans for 4 camera positions on the bunny point cloud (using the z buffer detailed in this section)

1. Core simulation framework continued on next page...

Simulation of a scan - ZBuffer

Once we obtain all of the 3D points in camera coordinates, how do we decide which ones to keep? I tried obtaining the closest points by only keeping the ones that had their depth(z) smaller than half the maximum depth. However, it was not a good approach, as there might be points further behind which are not occluded and will not appear in the final scan. So a version of z buffer was implemented.

The z buffer is a matrix which encodes the nearest points seen by the camera along with their depth. Once again, it was difficult to design this function to keep only the first points the camera sees. This is due to the fact that we are working with point clouds and we have no information about how they are related to each other (i.e. no faces). So, a very important parameter in this function is the size of the matrix (zbufferSideSize). This variable sets the resolution of the final scan. The bigger zbufferSideSize is, the more points will be seen by the camera.

Algorithm 2: Scan simulation with a z buffer

Input: PointCloud pcd, Camera cam_i, Size zBuffer $zbufferSideSize$
Result: Scan $scan_i$ with the nearest projected points to the camera
$projectedPCD \leftarrow$ Project all the points in pcd using cam_i;
$zBuffer \leftarrow$ Initialize the matrix with biggest z depth;
$indexBuffer \leftarrow$ Initialize the matrix with -1;
for *each point* p_i *in* $projectedPCD$ **do**
  $xIndex, yIndex \leftarrow$ Map the corresponding p_i to the desired range of the zBuffer to get the row and column indices;
  if *point* p_i *is closer than* $zBuffer[xIndex, yIndex]$ **then**
    $zBuffer[xIndex, yIndex] \leftarrow p_i$ Save the current closest point;
    $indexBuffer[xIndex, yIndex] \leftarrow i$ Save the index of the current closest point ;
$scan_i \leftarrow$ Save all the closest points to the camera cam_i by taking their indices from $indexBuffer$;

In most of the calls, this function uses a zbufferSideSize of 100. However, depending on the part of the algorithm, this can be changed. For example, for bigger and more complex meshes, the resolution could be bigger to capture more points. Finally, here are the function signatures that output the nearest projected points by using the technique described in this part.

```
void computeNearestProjectedPts(Camera &camera,
                                int zbufferSideSize = 100);

\\which calls the following function from utilities, where the z Buffer is built

void getNearestPointsToCamera(Eigen::MatrixXd &projectedPts,
                          Eigen::MatrixXd &out_nearestProjectedPts,
                          int zbufferSideSize = 100);
```

Evaluation with ground truth of a new scan

We decided what method we wanted to use and Karina implemented it. Once we have a new scan, we back project the points back to world coordinates and merge them with the current $\tilde{M}$ obtained from the k scans.

$$score_{gt} = \frac{numPoints(\tilde{M})}{numPoints(M)} \tag{1}$$

1. Core simulation framework continued on next page...

where $\tilde{M}$ is the estimated point cloud from the k+1 views, and M is the original model. The ideal $score_{gt}$ is 1 (100%), but it is highly improbable it will ever be obtained with our setup. Even if we use only synthetic data, with no noise, we cannot get all of the points if we are restricted to rotate the camera only around one axis (y).

This approach for the evaluation works because we update $\tilde{M}$ by merging point clouds with no duplicates.

Merging point clouds with no duplicates

This function was implemented by Karina. The final merged point clouds is saved in an accumulator matrix which is updated with each new scan.

Algorithm 3: Merge point clouds without duplicates

Input: PointClouds $pc1$, $pc2$
Result: Merged point clouds $mergedPCD$
for *each point* p_i *in* $pc1$ **do**
  for *each point* p_j *in* $pc2$ **do**
    if p_i *is not* p_j **then**
      $mergedPCD \leftarrow$ Add new unique point
Return $mergedPCD$; Finish merging;

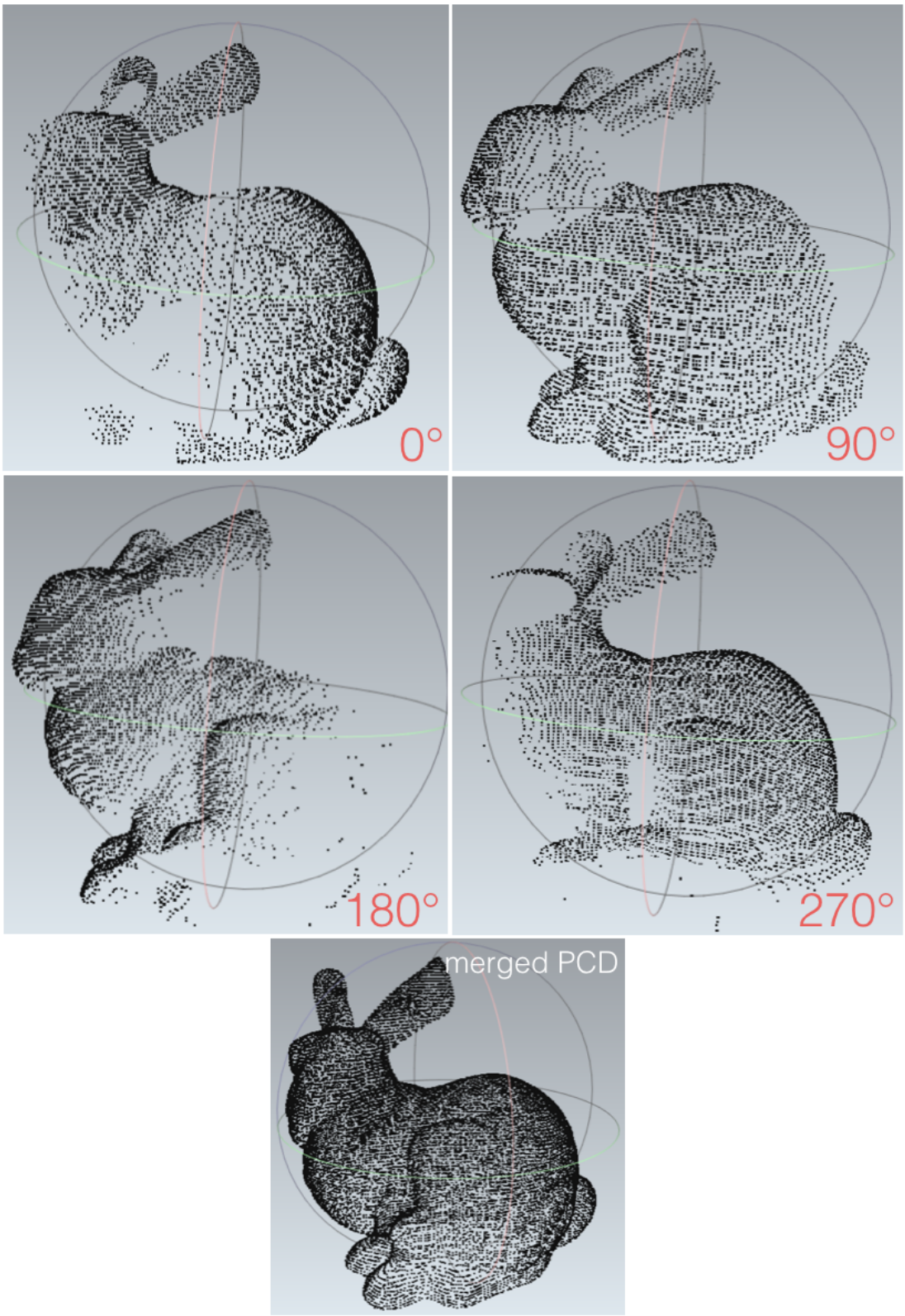

Figure 5: Back-projected points the 4 scans in Figure 4

2. Individual part

The aim of this part of the Next View Planning project was to implement a strategy to choose the next best view, given k views and the simulation framework.

A helpful guide in planning this part of the project was found in [3], where they present a list of view constraints that need to be met for choosing the next best view:

- new information - the new scan must see unknown surfaces
- positioning constraint - the new scan must be reachable by the robot
- sensing constraint - the surfaces must be in the camera's field of view and they should have a good quality
- registration constraint - overlap with previous scans

In the context of our project, the registration constraint is not a problem since we have the exact position of the cameras. So, we can use it to back project the points, obtain their world coordinates and simply merge the scans. The fact that we can rotate the camera around only one axis with no translation gives us the positioning constraint. Therefore, I have decided to choose an evaluation strategy that will incorporate the other two view constraints. So, each candidate view is given a score based on the amount of new information it brings and the quality of the points. The candidate views are generated through a fast two step hierarchical strategy.

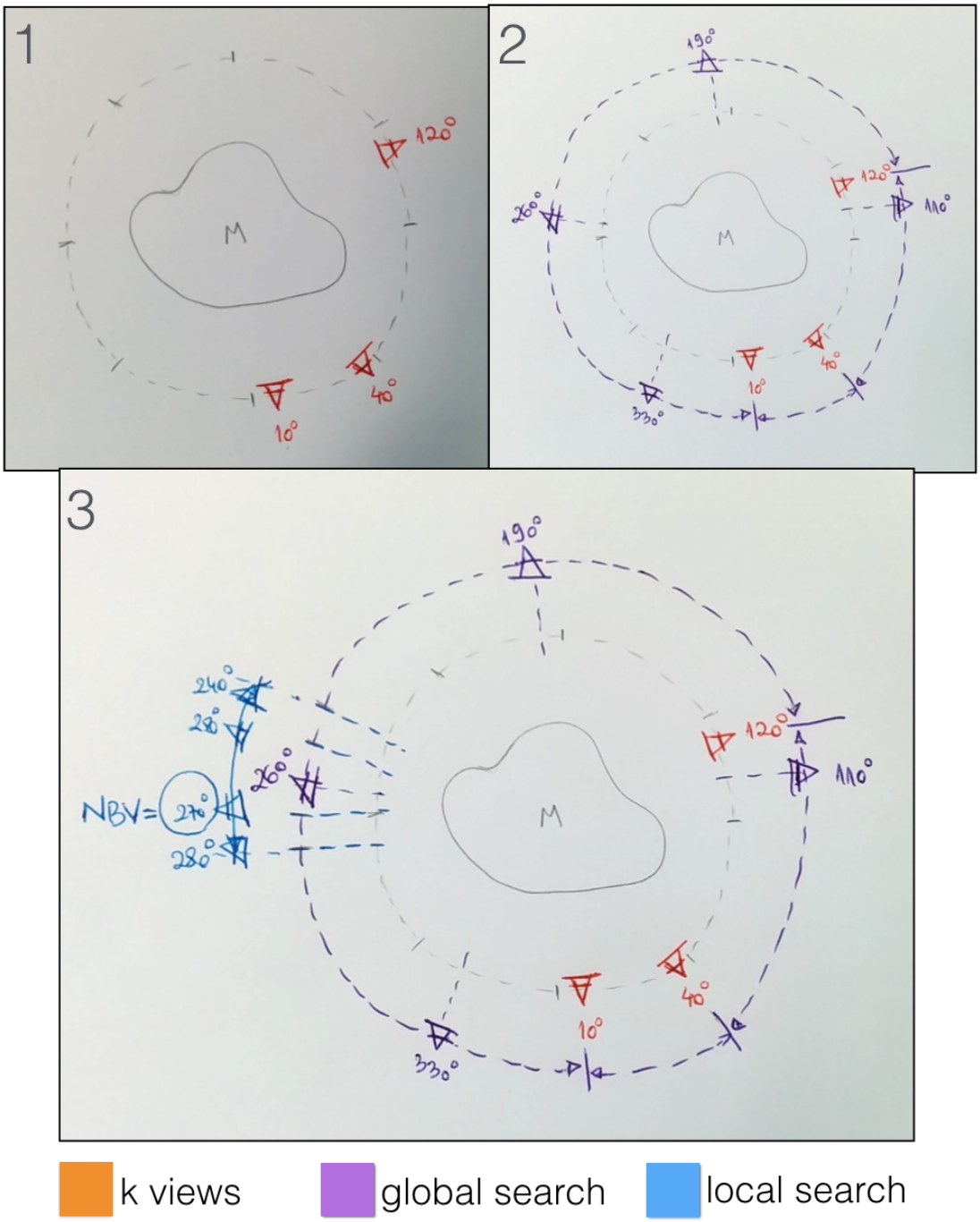

Figure 6: Schematic with an example of how the implemented search strategy generates candidate views. Image 1 shows the k views given as input at 10, 40 and 120 degrees. Image 2 shows a global search for candidate views. New views are generated at intervals of 70 degrees for each pair of cameras (110, 190, 260 and 330). Image 3 shows a local search around the candidate view with the best score from the global search. The final next best view is found at 270. This values were worked out on paper in the planning on this algorithm and they are supported by the algorithm results (see Figure 7 with the console output).

```
[Mariucas-MacBook-Pro:build ucaHome$ ./nextViewPlanning
 Enter <1> for the armadillo, <2> for the bunny or <3> for the dragon mesh
 2
 Reading 3D model from file:../models/_bunny.ply...
 35947 points read

 Candidate search:

 Global sparse search - Found 4 candidate views
 110 190 260 330
 Computing score for candidate view #1...
 Computing score for candidate view #2...
 Computing score for candidate view #3...
 Computing score for candidate view #4...
 Candidate views at:
 110 190 260 330
 Scores - new information:
 2038 3203 4727 2792
 Scores - scan quality:
  1952.7 1956.85 2990.75  1841.4
 Final Scores:
 2038 3203 4727 2792
 Global sparse search - Found maximum score of 4727 for 260 degrees

 Local search - Found new candidate views
 240 250 260 270 280
 Computing score for candidate view #1...
 Computing score for candidate view #2...
 Computing score for candidate view #3...
 Computing score for candidate view #4...
 Computing score for candidate view #5...
 Candidate views at:
 240 250 260 270 280
 Scores - new information:
 4238 4593 4727 4783 4545
 Scores - scan quality:
 2502.46 2775.97 2990.75 3151.48 3060.34
 Final Scores:
 4238 4593 4727 4783 4545
 Local search - Found maximum score of 4783 for 270 degrees
 Final NEXT BEST VIEW = 270 with a score of 4783

 Eval with GT - Compute score for chosen NBV with 270 degrees
 ***
 Not bad, human ->0.807383
 ***
 Estimate PCD from k+1 views...
 Writing file: ../output/estimatedPCD.ply...
 29023 points written
 Done!
```

Figure 7: This figure shows the console output for the algorithm presented in Section 2 for the individual part of the NVP project. Note that in this example, the final score for the evaluation of a candidate view is chosen purely based the amount of new information each scan brings. Also, note that an evaluation by comparison with the ground truth is made for the final next best view chosen (270 degrees) and it has a score of 0.80.

Methods

The easiest way to illustrate the search strategy is Figure 6. Given the k views, a global search is performed on the 360 degrees ring around the point cloud. For each pair of cameras, new candidate views are selected every 70 degrees. This is a very sparse sampling that ensures a small number of candidates that are well distributed over the empty regions of the ring. Each candidate view is evaluated (a low resolution z buffer is used to make the computation faster) and the one with the highest score passes to the next stage. The local search happens in a range of $[-20, 20]$ degrees around the best view from the global step. Thus, 4 new candidate views are generated every 10 degrees around the position which we already know is a good match. This local search allows us to refine the selection of the next based view and choose the candidate view with the highest score. This search strategy is a faster version of a sampling that proposed candidate views every 10 degrees and evaluated them all.

```
\\ main function that outputs the next best view
Camera computeNBV(PointCloud &pc,
                  std::vector<Camera> &kViewVector,
                  bool FLAG_EVALwGT= false);
```

In order to choose a good scoring system for the evaluation of a candidate view, I looked into two methods:

- $score_N$ = compute the number of new points a scan brings into the $\tilde{M}$

$$score_N = |\tilde{M_{k+1}}| - |\tilde{M_k}| \tag{2}$$

```
int getNumNewPointsFromNewScan(PointCloud &pc,
                               std::vector<Camera> &kViews,
                               Camera &newView,
                               int numPoints_kViews,
                               int zbufferSideSize = 100);
```

- $score_Q$ = look at the quality of the points scanned (adapted from [3])

$$score_Q = \sum |cos(\alpha_i)| \tag{3}$$

where α_i is the angle formed by each normal and the camera orientation.

```
double getQualityScoreFromNewScan(PointCloud &pc,
                                  Camera &newView,
                                  int zbufferSideSize = 100);
```

In [3], they measure the quality of the scanned points with a similar formulation. The reason behind it is that a scan will have a higher quality if the camera is orthogonal to the surface. Consequently, by adding up the cosine of the angles between each point normal and the camera orientation, we are aiming for a score as close to the number of points in the scan. This evaluation method is needed to filter out views that would introduce new information, but they would be isolated points, or points only on the boundaries, since they have poor normal estimation. In this way, the two chosen scoring systems are complementary and can be adapted based on the requirements for the estimated model $\tilde{M}$.

The normals of the point cloud are calculated as in Coursework 1, using the library ANN. For each point, a plane is fit to its neighbourhood and the corresponding normal is estimated. This formulation only gives the direction of the normals, without correcting for the orientation. This implies that the score for a view

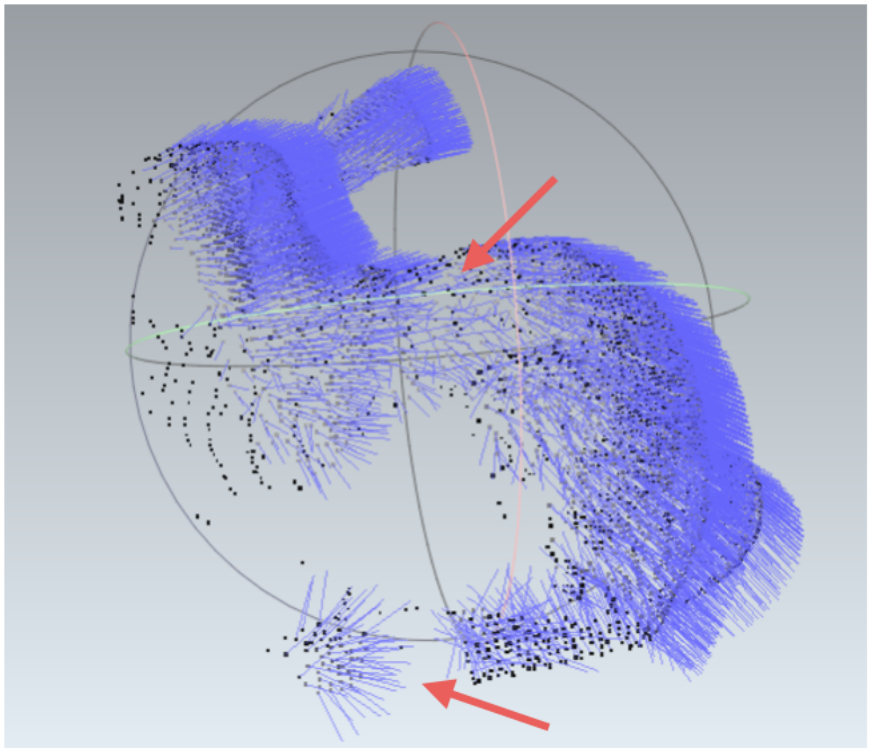

Figure 8: Problems with normal estimation in regions with sparse points or boundaries

at position x will be equal to the score at position $x + \pi$.

Algorithm 4: Evaluation of a candidate view

Input: PointCloud *pcd*, k Camera positions cam_k, k+1 candidate view cam_{k+1}
Result: $score_{fin}$
$\tilde{M}_k \leftarrow$ Reconstruct the estimated model from the k scans;
$\tilde{M}_{k+1} \leftarrow$ Reconstruct the estimated model from the k+1 scans;
$score_N \leftarrow$ Difference between the number of points in $\tilde{M}_{k+1}$ and $\tilde{M}_k$;
Compute normals for the point cloud *pcd*;
cam_position $\leftarrow$ Compute the camera position out of the extrinsic matrix;
cam_orientation $\leftarrow$ Compute the camera orientation out of the extrinsic matrix;
for *each point* p_i *in pcd* **do**
  $score_Q \leftarrow$ Add the cosine of the angle formed by the current normal and the *cam_orientation*;
$score_{fin} \leftarrow$ Combine $score_N$ and $score_Q$ into the final score;

Experiments
Starting with the 3 scans shown as an example on paper in Figure 6, we aim to compute the next best view for the bunny point cloud. In these experiments, we will use the method detailed in Section 1 for the evaluation with the ground truth.

For each candidate view in both the global and local search, we compute the score with the ground truth to validate if my strategy picks the right camera position. The following list shows the results (the values are taken from the console output):

- Global search
  - Candidate view: $110° \rightarrow score_{gt} = 0.72$
  - Candidate view: $190° \rightarrow score_{gt} = 0.75$
  - Candidate view: $260° \rightarrow score_{gt} = 0.80$
  - Candidate view: $330° \rightarrow score_{gt} = 0.75$
- Local search
  - Candidate view: $240° \rightarrow score_{gt} = 0.8051$
  - Candidate view: $250° \rightarrow score_{gt} = 0.8071$
  - Candidate view: $270° \rightarrow score_{gt} = 0.8073$
  - Candidate view: $280° \rightarrow score_{gt} = 0.8049$

This list of values will be used as proof that the camera position at 270° is actually the next best view, for this particular setup with three initial views. So, I use this as validation against three scenarios:

- 1. Evaluate each candidate view based on only the new information it brings, $score_N$
- 2. Evaluate each candidate view based on only the quality of the points, $score_Q$
- 3. Evaluate each candidate view based on a linear combination of both:

$$score_{fin} = \alpha score_N + (1 - \alpha) score_Q \tag{4}$$

Results

- 1. I ran the algorithm with $score_{fin} = score_N$ and the computation of the next best view arrived at the same camera position: 270°
- 2. I ran the algorithm with $score_{fin} = score_Q$ and the computation of the next best view arrived at the same camera position: 270°
- 3. I ran the algorithm with several values for α and I set it empirically to 0.6. So, for $score_{fin} = 0.6 score_N + 0.4 score_Q$, the computation of the next best view arrived at the same camera position: 270°

I chose to set α to 0.6 to give more weight to the amount of new information each new scan brings. Furthermore, as I mentioned above, the orientation of the normals is not computed. So, while $score_Q$ is a helpful indication of the quality of a scan, it does not discerne between opposite camera poses. However, this trade-off between between new information and point quality can be chosen by the user for each input

mesh.

The results presented so far represent the ideal case where the estimation of the best next view matches the validation with ground truth. Furthermore, both scores pointed in the same direction, making the decision easy. However, this was also due to the fact that the bunny point cloud is a relatively simple model with few points and without a lot of distinctive features. So, the same algorithm with the same k views (10,40,120) was ran on the armadillo mesh which is significantly more complex (see Figure 9). In this scenario, the two scores chosen for the evaluation of a candidate view contradict each other by suggesting different positions, showing the benefit of the linear combination used for the final score. This difference between the input meshes illustrates perfectly the need for a decision from the user on the requirements of the estimated point cloud. Is it desirable to have more points at the expense of a lower quality or is it the quality of the estimated model that would bring the most benefits?

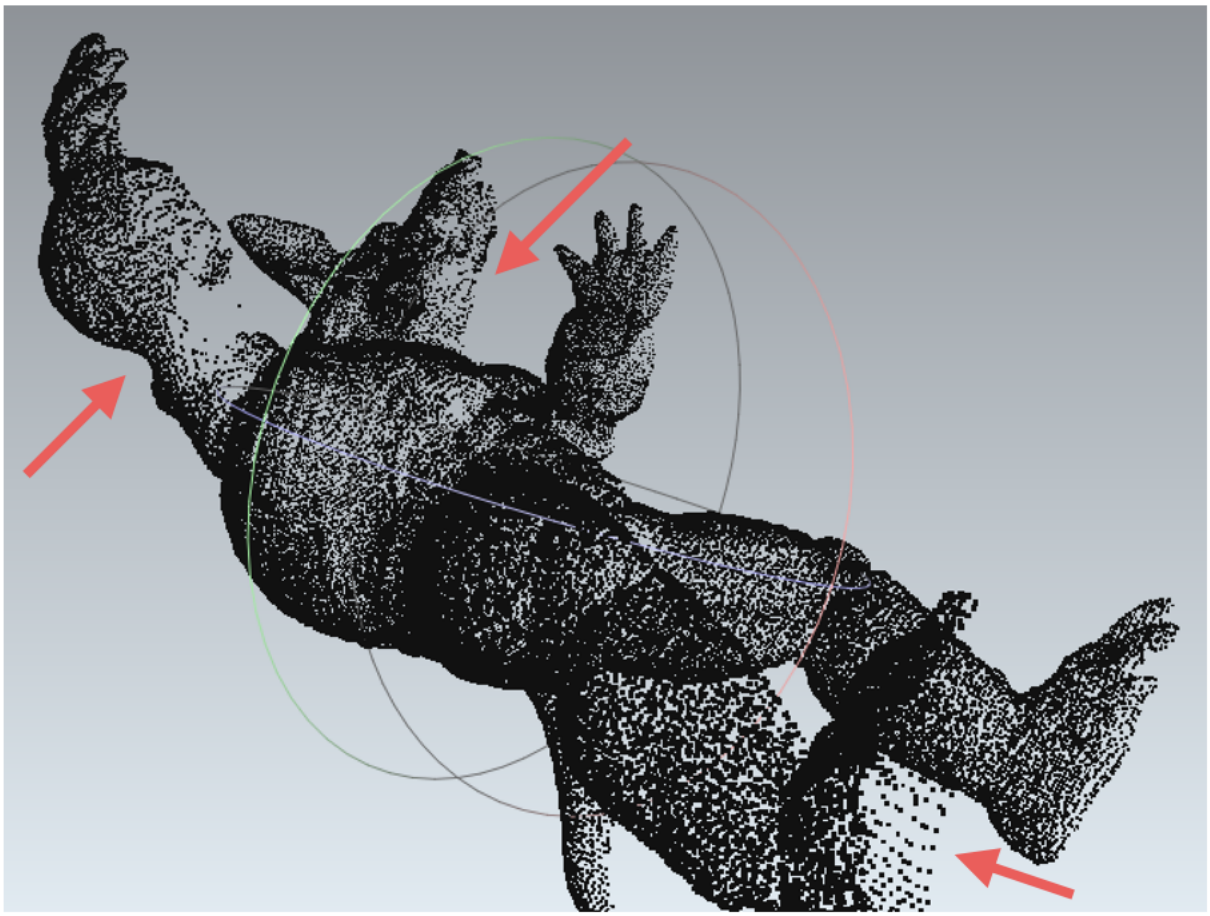

Figure 9: Reconstruction of the armadillo point cloud from k+1 views with $score_{gt} = 0.85$ and a resolution of 400 for the zBuffer. The initial camera poses were at 10, 40 and 120 degrees. The next best view was estimated to be at 200 degrees. The arrows point in areas where the model is still missing points, due to the complexity of the surface in the original model

Limitations

Clearly, one of the biggest limitations of the current implementation is that it was tested only on synthetic data. However, the results are promising for an extension to a real model. Furthermore, there are inherent limitations in the assumptions that he have made in the beginning in order to make this project achievable in a short period of time:

- the positions of the camera are known, without any errors in the measurement
- the point clouds have no noise and no artefacts
- we are only aloud to rotate the camera around one axis and position it on a ring
- it was difficult to visualise the entire setup, so developing a specific viewer would be helpful

Conclusions

In this Next View Planning project, we (me and Karina) developed a core framework in C++ to simulate the 3D reconstruction of a synthetic 3D object (detailed in Section 1). Given k camera poses, the aim was to compute and evaluate the next camera pose which would bring the most information to the final reconstruction. The camera was restricted to rotate around one axis in a ring surrounding the object. In the individual part of the project, a two-step fast hierarchical search strategy was developed to propose candidate views. Each candidate view was evaluated (without using the original model), based on how many new points it brings and their quality. Section 2 shows that next best view computation depends on the complexity of the mesh and on the resolution of the z buffer. However, even for more complicated meshes, the results are close to what the ground truth suggests. Also, the user has to decide on the desired trade-off between the two scores for the evaluation of each candidate mesh. In conclusion, the results presented in this report are promising on synthetic data, given the assumptions made.

References

[1] J. Maver and R. Bajcsy, "Occlusions as a guide for planning the next view," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 15, pp. 417–433, may 1993.

[2] W. R. Scott, G. Roth, and J.-F. Rivest, "View planning for automated three-dimensional object reconstruction and inspection," *ACM Comput. Surv.*, vol. 35, pp. 64–96, Mar. 2003.

[3] J. I. Vasquez-Gomez, L. E. Sucar, R. Murrieta-Cid, and E. Lopez-Damian, "Volumetric Next-best-view Planning for 3D Object Reconstruction with Positioning Error," *Int. J. Adv. Robot. Syst.*, vol. 11, p. 1, 2014.

[4] S. Isler, R. Sabzevari, J. Delmerico, and D. Scaramuzza, "An information gain formulation for active volumetric 3d reconstruction,"

[5] D. S. J. D. Prince, "Computer Vision: Models, Learning, and Inference," p. 598, 2012.