FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Immersive Redesign

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

BSc Medieninformatik und Visual Computing

eingereicht von

Manuel Keilman
Matrikelnummer 11824531

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Projektass.(FWF) Dr. Stefan Ohrhallinger

Wien, 01.07.2022

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Immersive Redesign

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

BSc Media Informatics and Visual Computing

by

Manuel Keilman
Registration Number 11824531

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass.(FWF) Dr. Stefan Ohrhallinger

Vienna, 01.07.2022

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Manuel Keilman

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Kurzfassung

In dieser Arbeit haben mein Kollege Ahmed El Agrod und ich eine Software implementiert, die es grundsätzlich ermöglicht Punktwolken zu bearbeiten. Durch Verschieben, Löschen, Speichern und Einfügen von ausgewählten Objekten soll die Punktwolke modifiziert werden können. Diese Bachelorarbeit beschreibt hauptsächlich, wie neu hinzugefügte Objekte automatisch auf dem Boden platziert werden indem dieser von einem Algorithmus erkannt wird. Weiters wird beschrieben wie ein Image Inpainting Algorithmus implementiert wurde, um unvollständige flache Regionen von Punktwolken mit neuen Punkten und zugehörigen passenden Farben zu füllen. Die Bodenerkennung wurde mithilfe des RANSAC-Algorithmus implementiert, welcher eine Ebene berechnet, die den Boden für die Szene repräsentiert. Für den Image Inpainting Algorithmus mussten wir die 3 dimensionalen Punkte der Punktwolke auf ein 2D-Bild abbilden, dann einen Image Inpainting Algorithmus verwenden, um die fehlenden Pixel zu füllen, und schließlich die 2D Pixel des inpainted Bildes wieder auf 3D-Punkte in unserer Szene abbilden. Außerdem wurde eine Evaluierung durchgeführt, bei der sowohl die automatische Bodenerkennung als auch der Image Inpainting Algorithmus hinsichtlich Laufzeit und visueller Qualität getestet wurden.

Abstract

In this thesis, my colleague Ahmed El Agrod and I implemented software that allows point clouds to be edited. By moving, deleting, saving, and inserting selected objects, the point cloud should be able to be modified. This bachelor thesis mainly describes how newly added objects are automatically placed on the ground being recognized by an algorithm. Furthermore, it is described how an image inpainting algorithm was implemented to fill incomplete flat regions of point clouds with new points and associated matching colors. The ground detection was performed using the RANSAC algorithm, which computes a plane representing the ground for the scene. For the image inpainting algorithm, three-dimensional point cloud points had to be mapped to a 2D image, then use an image inpainting algorithm to fill in the missing pixels, and finally, map the 2D pixels of the inpainted image back to 3D points in the scene. An evaluation was also conducted to test both the automatic ground detection and the image inpainting algorithm regarding runtime and visual quality.

Contents

CHAPTER 1

Introduction

The number of 3D scanners for creating point clouds is increasing daily, and making a point cloud with your smartphone is already possible. Scanning and digitizing your environment is a big step in computer graphics. For example, people can scan their living room and create 3D objects without modeling. Since self-driving vehicles are also evolving and most of these vehicles use LIDAR sensors or depth cameras that can be used to generate point clouds, we can talk about a trend of point clouds becoming more common in the future. The ability to reshape point clouds immersively, for example, by quickly selecting specific objects to move or delete, can open up new ideas and possibilities.

Our goal is to be able to redesign point cloud data immersively. Therefore, we created software where users can add multiple point clouds into the same scene. After the 3D model is displayed in the point cloud viewer scene, the user should be able to select specific subsets, which we will further refer to as „objects“. Deep learning methods should classify the point cloud into separate objects, e.g., trees, cars, passengers, etc. Then, the user should be able to select these objects simply by clicking on them. Alternatively, the user should be able to manually select objects using a box that can be moved, scaled, and rotated. Once an object is selected, it should be possible to delete, move, save and insert it again into the same scene or even a new one. When adding new objects, the program should automatically place them on the ground.

After deleting several objects from the scene, the ground may also be deleted. In addition, the point cloud is often initially full of holes and missing areas due to acquisition sensors poorly scanning the objects. Therefore, an image inpainting algorithm is implemented to smoothly fill these holes in flat areas such as sidewalks or street floors. A graphical user interface (GUI) is implemented to conveniently use all tools and features and make the software more user-friendly.

The project was carried out by my colleague Ahmed El Agrod and I. Ahmed was responsible for the machine learning that classifies objects of the point cloud and selects them to save, move or delete. I was responsible for implementing the image inpainting algorithm and automatically

detecting ground surfaces to place objects correctly on the ground. Nonetheless, we worked together, as some implementations were highly correlated, and basic implementations such as rendering the scene and implementing the graphical user interface were conducted together.

The ability to reshape point cloud scenes immersively can be a major step forward in being more comfortable with working and experimenting with point clouds. For example, cars, bikes, or other machines could be equipped with LIDAR sensors and record objects, specific areas, or an entire city. This would allow for the creation of a database of a wide variety of recorded data which can be used as assets for planning a room, house, or even a city.

The following chapter outlines related work. The methodology for this work is given in chapter 3, describing how the algorithm for ground detection and image inpainting algorithm, as well as how the selection of subsets of the point cloud was implemented. An evaluation of this study follows in chapter 4, where the efficiency of the implementation in terms of runtime and visible results is shown. The final chapter offers an outlook on what could be improved and what features could be implemented in the future.

CHAPTER 2

Related Work

2.1 Point Clouds

Point clouds are a set of unconnected points in 3D space. Those points consist of x, y, and z coordinates and r, g, and b color values. They can represent 3D objects and can be created with depth sensors or Light Detection and Ranging (LIDAR) sensors. [6]

2.2 Open3D

Open3D is an open-source library that focuses on processing 3D data. The initial goal was to create an easy-to-use, open-source software framework that supports rapid development for three-dimensional data, as there was no such framework. [18] The library contains many useful functions which we used in this thesis, for example:

The inbuilt point cloud data structure had several functions, such as voxel_down_sample(), which downsamples a point cloud and averages their normals and colors, segment_plane(), which uses a RANSAC implementation to segment a plane in the point cloud, select_by_index() to select a part of the point cloud using the indices of points as input parameters or even paint_uniform_color() to change the color of the point cloud. [2]

Open3D's geometry was also very beneficial. For example, we used BoundingBoxes, which contains a get_point_indices_within_bouding_box() method, which returns all point indices within the bounding box. [1]

There are also transformation functions such as rotate(), scale() and translate() available for all 3D data models.

The graphical user interface, setting up a scene, and rendering the data models was also implemented with Open3D's visualization functions. [3]

2.3 RANSAC

The Random Sample Consensus (RANSAC) algorithm can handle and interpret data models with a significant ratio of gross errors. In point cloud data, we have a vast number of outliers if we, for example, consider all points except ground points as outliers. Therefore we can use the RANSAC algorithm to detect planes in point cloud data, which we can further specify to recognize ground planes only to place all objects onto it. In general, the RANSAC paradigm works as follows:

If a set of data points P to a specific model M is to be set, which could be a plane, n random points (n is the minimum number of points required to instantiate the model. For a plane, three points would be needed) have to be selected from the set of points P. After that, a subset S containing all points within a certain error tolerance to the model M is determined. If S is greater than a specific threshold t, the subset S will be used to compute a new model $M1$, which will be the result. If S is less than t, a new subset, $S2$, will randomly be selected. This process is repeated for a certain number of iterations. If no consensus is found after all iterations are completed, the algorithm either terminates in failure or chooses the model with the largest consensus set S. [11]

2.4 Image Inpainting

Image inpainting is used to fill in damaged parts or missing data in an image or video using the surrounding pixels. The goal is to retouch photos with various types of defects, such as scratches or dust particles on the camera when the picture is taken. However, it can also remove particular objects from an image and then use the environment to reconstruct a complete image. Therefore, an attempt is made to fill in the damaged areas and put them together with the original image so that the average viewer cannot see that the image has been inpainted. Initially, this work was conducted by professional artists, but now there are several techniques to automate this work with an algorithm. [4]

The OpenCV library provides two different algorithms for inpainting images. One is the cv.INPAINT_TELEA algorithm, which is based on the paper „An Image Inpainting Technique Based on the Fast Marching Method“ [14], and the cv.INPAINT_NS algorithm based on the paper „Navier Strokes, Fluid Dynamics, and Image and Video Inpainting“ [4]. The decision on which algorithm to use was made after researching which would be better suited for our application. A paper was found comparing both algorithms in several tests, and as it turned out, they are very similar in terms of efficiency. To be precise, the differences are so small that it hardly makes a difference in which algorithm is chosen. The paper concludes that the TELEA algorithm is slightly more efficient. However, the comparison involved a test with random text as the error mask. This random text was generated and placed on the image as error regions.

The result was that the NS algorithm performed slightly better than the TELEA algorithm, and since the error masks in our work are also spread very widely and randomly over the image, just like the random text, the decision was made to use the NS algorithm. Although, as mentioned earlier, it would not make much difference if the TELEA algorithm would be used. [7]

2.5 Point Cloud Inpainting

We wanted to use the Image Inpainting algorithm to fill missing parts of flat regions in point clouds. A way to approach this is to directly fill the missing parts of the selected object, meaning that the missing points are created and added at the correct position. Point cloud completion algorithms can be used to complete specific point cloud objects as in [17] and [16]. For example, these algorithms use a point encoder GAN or a neural network such as PMP-Net++, which is used to complete all types of point cloud objects such as cars, airplanes, chairs, tables, or poles.

in the first paper [17], point cloud data is used directly as input without labeling the data. They can train their network very fast and it has a high generalization capability which can be used to create new point clouds and enlarge 3D point cloud data sets. In general, the results were very good although the local feature learning ability needs to be improved as some regions remained incomplete.

The second paper [16] completes their point cloud objects by moving points from the source to the target in multiple steps. The neural network PMP-Net++ learns to predict the complete shape of the incomplete 3D point cloud object. Compared to other point cloud completion algorithms, PMP-Net++ is superior in terms of quality and performance.

However, both of these studies [17] and [16] do not specifically address the efficiency of filling gaps on the ground or sidewalks, and the completion of specific objects is beyond the scope of this work.

There are also several other approaches as in [8] or the improved work [9] and [12] to fill holes in three-dimensional point cloud geometries. In [8], the first step is to determine where the hole is located. Then, points surrounding the detected hole and certain surface points are prioritized to determine when they will be processed. The filling is done iteratively by selecting the best template near the hole from which points are transferred to the hole. In particular, with the improvements from the following work [9], high quality results can be obtained. However, this work does not take into account the colors of the point clouds and again focuses more on individual objects rather than a street scene where multiple objects in the vicinity would need to be considered.

Another work [12] deals with dynamic point cloud inpainting via spatial-temporal graph learning. Here, it was attempted to fill holes in 3D point clouds in motion. The results are very promising, although for this work we need a static algorithm for inpainting point clouds instead of a dynamic algorithm.

2.6 Similar Projects

There are already several papers dealing with similar topics which will be discussed in the following. The first paper had the goal of processing street scenes by removing people, cars, or other objects and also adding them back, just as in our thesis. One difference was that they also calculated the sun direction to cast realistic shadows, but the most significant difference is that it only works on images. Our work will be able to remove, move and add those objects in a three-dimensional space, with extra features such as automatic classification of objects and image inpainting missing holes on the ground. [15]

Another work also deals with the selection and editing of large point clouds. Their goal was to develop an out-of-core editing system to interactively select and segment, delete or render specific parts of the point cloud and also insert new points. They also heuristically estimated point sizes to mimic closed surfaces when a point cloud had large discrepancies in density. However, the focus of this work was on using memory as efficiently as possible and achieving high performance and good rendering quality, so moving objects or inserting new objects into the scene, as well as automatically detecting objects with a classification algorithm and filling gaps in flat regions created when, for example, certain objects are deleted were not implemented. [13]

CHAPTER 3

Methodology

This thesis aims to write software that allows editing point clouds with the following specific operations.

The most crucial operation would be the selection of objects that are part of the point cloud. Furthermore, deleting, saving, or moving those selected objects and adding new objects to the scene should be possible. Also, filling gaps in ground regions such as the floor or a sidewalk will be implemented. Additionally, the software should be able to classify everyday objects found on streets, such as trees, cars, buildings, etc., to select these objects conveniently. This classification, however, is not explained in this thesis since my colleague Ahmed El Agrod describes this process in more detail in his work [10].

3.1 Ground Plane Detection with RANSAC

It is to be assumed that all our newly added objects are going to be placed on the ground. Therefore, an algorithm that recognizes a ground plane to implement the adding operation was required. So if a new point cloud, for example, a tree, is placed on a street, it would not be desirable for the tree to sink into the ground or float in the air. Thus, using the RANSAC algorithm [11], when a user wants to add the tree, it should be automatically placed at the correct height.

RANSAC

The RANSAC algorithm is beneficial for detecting the ground plane of a point cloud. Because capturing certain scenes with LIDAR sensors, a significant number of points are recorded on the ground.

The following section is a brief explanation of the RANSAC algorithm from Open3D's library that is used in this work:

The algorithm takes x points as input. From these points, it randomly selects a subset of 3 points and creates a plane with these points. Next, it is checked how many of those x points are exactly on that plane or within a specific threshold t. These points are called inliers. After that, it saves the plane and all inliers. The algorithm does this for i iterations, starting over again by selecting three more random points. If the number of inliers in this iteration is higher than the previously saved result, the result is updated and set to the current plane and inliers. After i iterations are completed, a suitable plane that contains as many points as possible is obtained.

Plain RANSAC implementation

The RANSAC algorithm was straightforward to utilize as Open3D's library [18] had already implemented it. The method is called with a point cloud and had the following three parameters: threshold t, number of randomly selected points n (which is 3 in our case since we want to create a plane), and iterations i, which have been briefly explained in 3.1.

Walls detected

A problem with using Open3D's RANSAC method was that it is not limited to only recognizing ground surfaces. As already described, the RANSAC algorithm returns a plane and all inliers, where the amount of inliers is maximized. Therefore, the RANSAC algorithm could detect walls instead of ground surfaces if a vertical plane contains more points than a horizontal plane, as seen in Figure 3.1.

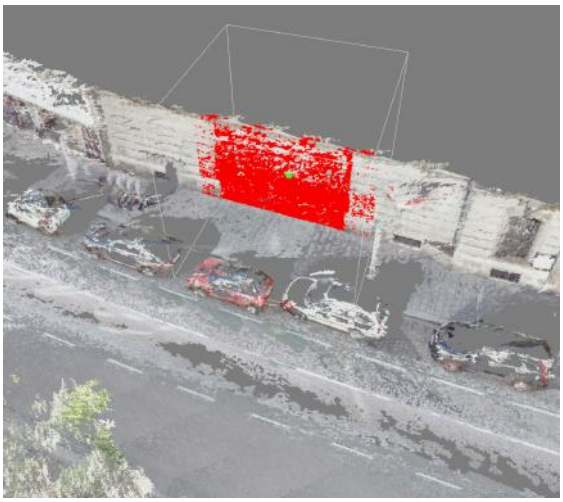

Figure 3.1: Point cloud where the wall was detected. Points marked red = inliers.

To solve the problem, the angle between the plane's normal vector $\vec{n}$ and the up vector

$$\vec{z} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

is calculated as follows:

$$\cos \alpha = \frac{\vec{n} \times \vec{z}}{|\vec{n}|}$$

If this calculated angle α is greater than 45 degrees (45 degrees has been found to work well), our RANSAC method is executed recursively, but this time with a point cloud that does not contain the inliers of the previous step. That means the new point cloud after each recursion step s is $P^{i+1} = P^i - I^i$, where I = inlier set.

With this implementation, only ground surfaces are detected, or to be more precise, surfaces with a maximum gradient of 45 degrees to the ground. The results are illustrated in Figure 3.2.

Figure 3.2: Point cloud where the floor was successfully detected. Points marked red = inliers.

Implementation

Several ideas and steps were attempted during this project to use the RANSAC algorithm efficiently and effectively. Various approaches that were implemented during the project and the final implementation are described in this section.

To improve usability, when adding new objects, it was decided to implement a preview that follows the mouse in real-time and shows where the object will be placed. Furthermore, since newly added objects are always placed on the ground, the preview must also move along the ground.

The first approach was to calculate the ground points locally in a sufficiently large area (for the RANSAC algorithm to work correctly) each time the mouse moves to a new position. Then the ground point closest to the mouse position was automatically selected to place the preview. However, the calculation of Open3D's RANSAC implementation took about two seconds. Thus it was not practical because the workflow would be severely restricted if users had to wait 2 seconds after each mouse movement.

So to solve this problem, it was decided that all ground points before adding new objects into the scene, should be calculated. To achieve this, the RANSAC algorithm is applied to the entire point cloud as soon as it is loaded into the scene. However, there is a downside to this as the RANSAC algorithm calculates a plane, but the ground of this point cloud may be somewhat curved. These gradients occur, for example, when a street is loaded into the scene since streets are not always flat, and their slope can vary. This curvature causes the RANSAC algorithm to not detect ground points in certain areas. Figures 3.3 and 3.4 show an example where the point cloud ground was not fully recognized. The points marked in red represent the inliers, and the grey area interior to the red area should also be marked.

Figure 3.3: Point cloud where the ground is not fully recognized. Points marked red = inliers, the grey area interior to the red area should also be marked.

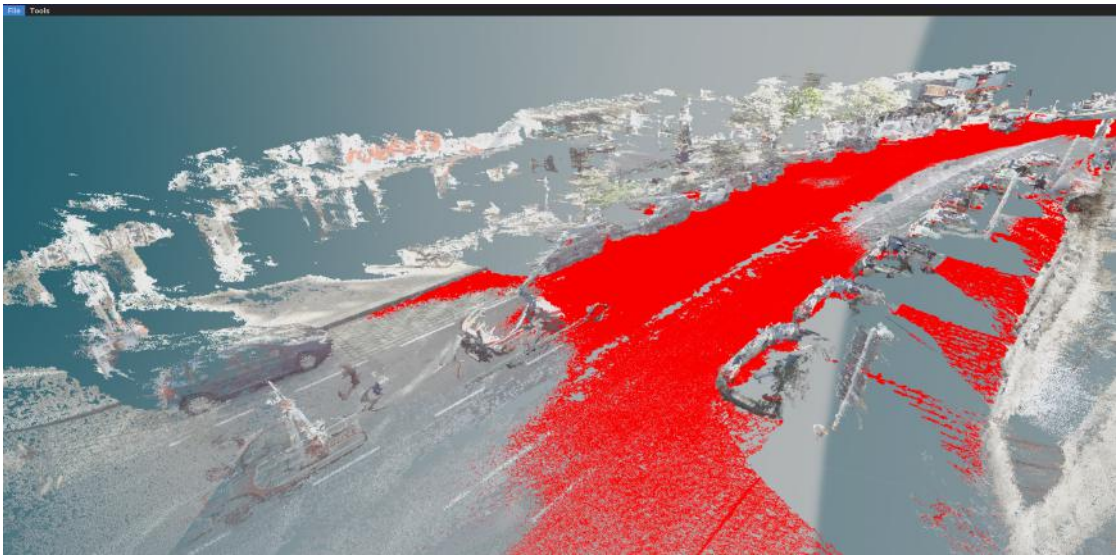

Figure 3.4: Point cloud where the ground is not fully recognized. Points marked red = inliers, the grey area interior to the red area should also be marked.

To counteract this, it was required to adjust the parameters for the RANSAC algorithm. More precisely, by increasing the threshold t, which specifies the distance to the plane at which points are marked as inliers. By doing that, we get all ground points as inliers, as seen in Figure 3.5. However, increasing the threshold results in another problem. In our case, the threshold

should be kept as small as possible because undesired outcomes are achieved if the value is set too high, as shown in Figures 3.6 and 3.7.

As can be seen, the tires and generally the underside of the car are also marked as inliers,

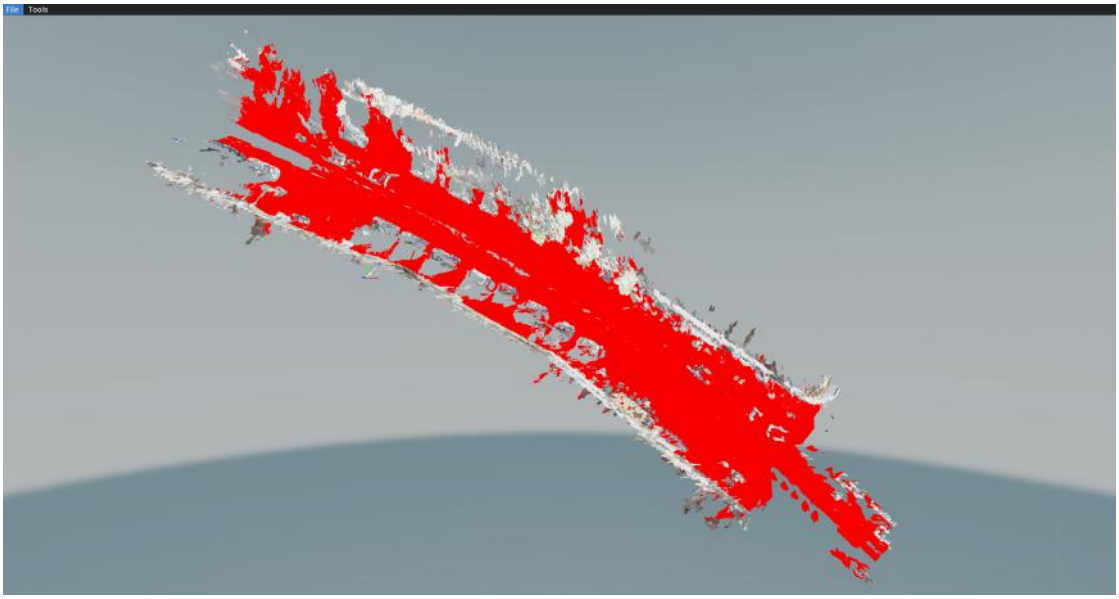

Figure 3.5: Point cloud where the ground is fully detected. Points marked red = inliers.

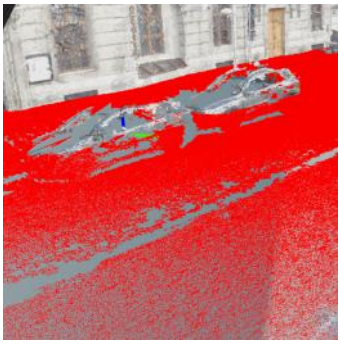

Figure 3.6: Cars wrongly contain inliers. Points marked red = inliers.

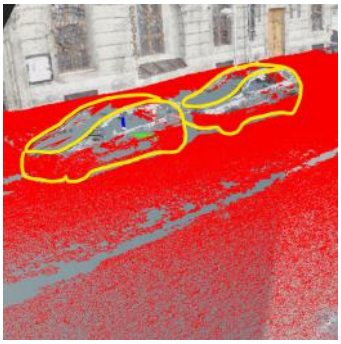

Figure 3.7: Cars (with clear outlines) wrongly contain inliers. Points marked red = inliers.

although they should not be part of the floor where the user could place objects. In addition to these problems, applying the RANSAC algorithm to the entire point cloud significantly increased the loading time. For an 8 million point cloud, there was approximately 17 seconds of delay before the point cloud loaded. Users would not tolerate such delays, so another solution had to be devised.

Eventually, we arrived at the final implementation, which was far more elegant and efficient than the other approaches. We still apply the RANSAC algorithm as soon as the point cloud is loaded into the scene, but this time we do not use all points of the point cloud but a subsampled set.

We get the subsampled set using the voxel_down_sample() method from Opend3D's library. This method has a parameter that specifies the size of a voxel. You can imagine that multiple neighboring voxels are superimposed on the entire point cloud. After that, we check which points are inside each voxel, and from these points, one average point is calculated. Therefore, each voxel will calculate precisely one point. This results in a point cloud containing significantly fewer points than the original, depending on the chosen voxel size.

With this method, we can significantly reduce the number of points from around 8 million to 50,000 points, improving performance from 17.45 seconds to approximately 0.45 seconds while still achieving good results. In our implementation, we use a fixed voxel size of 0.35 (this value worked very well) for every point cloud since all used models for this project are from the same dataset with approximately equal density.

Furthermore, we are no more calculating the closest ground point to the mouse position to obtain the location where the preview will be placed. Instead, we use the mouse's world x and y position and substitute them into the plane equation to get the z value. Therefore, we now have an x, y, and z value to place the preview. After each newly added object, the RANSAC algorithm is executed again to update this plane.

It has previously been mentioned that the ground of the point cloud can be curved. This curvature can still occur, meaning the preview is not perfect. It may float above the ground or sink into it. Therefore, we can no longer place the object in the same position as the preview. To solve this problem, when the user finally places the object, we apply the RANSAC algorithm locally, in a small area around the preview position, and substitute the mouse's x and y position in the new plane equation just computed, resulting in a more accurate result. This allows for a preview of the point cloud in real time and accurate placement on the ground.

3.2 Select Subsets of the Point Cloud

We implemented a modifiable box to enable the user to select a specific subsample of points. This box is of type oriented bounding box from Open3D's library. The user can transform the box by rotating, scaling, or translating it, which was implemented with Open3D's methods rotate(), translate(), and scale(). After the user places the box, we obtain all indices from the point cloud inside the box with the get_point_indices_within_bounding_box() function from Open3D's library. This function takes all point cloud points as its parameter and returns an array of all point indices inside the box. With these indices, it is possible to call the select_by_index() function on the original point cloud to obtain a new point cloud consisting only of points inside the box.

3.3 Image Inpainting

Since we can delete individual objects from the scene, it is not unlikely that ground points of the point cloud are also removed, or there may never have been ground points beneath the removed object because the LIDAR sensor could not detect them, resulting in missing parts on the ground. Also, scanned point clouds are often incomplete and generally contain many holes. However, we need a ground to place objects correctly. Therefore, the idea arose to apply an image inpainting algorithm to fill gaps in flat regions in the point clouds. Thus, the point cloud can be completed and embellished by implementing the image inpainting algorithm.

Image inpainting is mainly used to reconstruct incomplete two-dimensional images with holes or flaws. This raises the question of how we can use this algorithm on three-dimensional point clouds so that an image inpainting algorithm is still usable. The following section describes how we applied the image inpaining algorithm step by step.

Implementation

Here is a brief overview of how the implementation works. First, we need to select an area (with a gap and enough points to infer the pixels to fill) of the point cloud where we want to execute the algorithm. After that, the area will be converted to an image where point cloud points represent pixels. Then, each pixel receives the corresponding color of the point cloud point. If there is no point of the point cloud on a pixel, the pixel's color is black. The next step is to use a classic image inpainting algorithm to inpaint all black pixels. Last but not least, each pixel assigned a new color value through the image inpainting algorithm must be converted back into a point in the point cloud and placed correctly.

Step 1: Select an area

We use our modifiable box to select the area where the point cloud is to be inpainted, which is automatically created and has to be placed somewhere on the ground. After the box is placed, we obtain the point cloud with all the points inside the box, which was already mentioned and briefly explained in section 3.2. Subsequently, we downsample that point cloud to enhance performance, and with those points, we call our RANSAC algorithm to detect all ground points. This thesis previously mentioned that the algorithm returns inliers, and now we can create a point cloud with those inliers. Therefore, as seen in an example in Figure 3.8, the point cloud section is now inside the box, which contains only ground points.

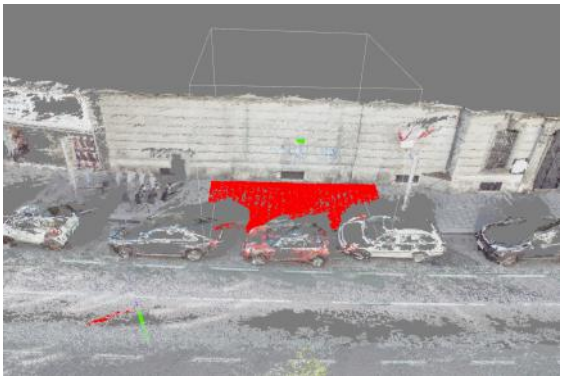

Figure 3.8: Only ground points are selected (= marked red), which are part of a separate new point cloud.

Step 2: Rotate the selected area to be axis parallel

Since the user can rotate the modifiable box, which is used to select the area, it is not always axis parallel. However, for the following steps to be able to map our points to pixels, it is necessary for the chosen area to be axis parallel. Therefore, we translate all points from the center of the modifiable box to the origin and afterward take the current rotation matrix of the box. Then, the inverse of the rotation matrix is calculated and applied to all of our points.

$$t := points.translate(-modifiable\_box.center)$$
$$r := points.rotate(inverse(modifiable\_box.R))$$
$$R = t * r$$

The entire transformation R is visualized from a top-down perspective in Figure 3.9.

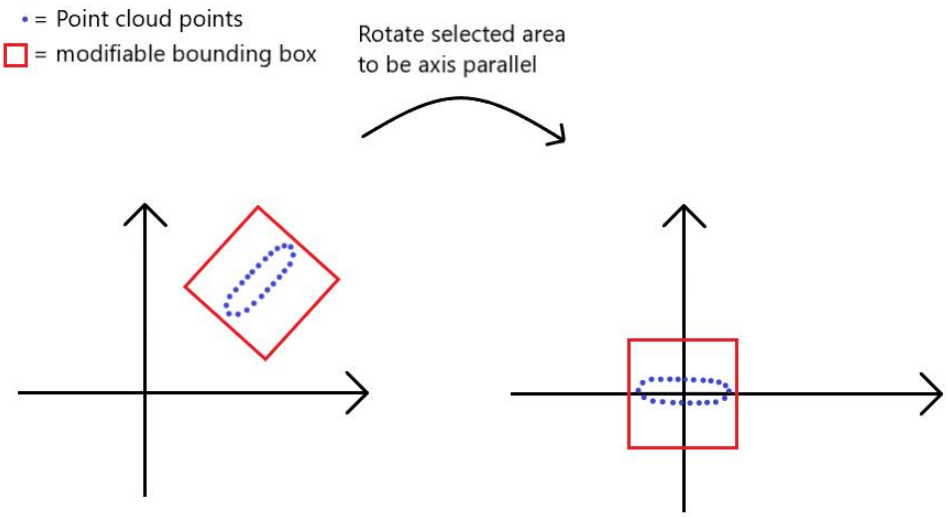

Figure 3.9: Top-down sketch before and after step 2.

Step 3: Convert points array for image

Now we can take all points and colors of the point cloud, remove the z value of the points and create an array called $points\_and\_colors$ which has the form x, y, r, g, and b, where x and y represent the position and r, g, and b are the colors.

Step 4: Calculate the average distance

To map the points to pixels, we need an average distance d to determine the image_height h and image_width w of the image we want to generate. The approach is first to calculate the area a of our modifiable box with

$$a = modifiable\_box\_height * modifiable\_box\_width$$

To obtain the average distance d, we calculate

$$d = \frac{a}{len(points\_and\_colors)}$$

After that, we compute the image_height h and image_width w of the image with the following equation:

$$h = ceil(\frac{modifiable\_box\_height}{d})$$

and

$$w = ceil(\frac{modifiable\_box\_width}{d})$$

If the average distance d is small due to the density of points in the area being very high, we will get a large image size. This is detrimental to the performance. Therefore, we cap the image size, not letting it surpass the maximum amount of pixels in the image max_pixel, which was chosen in this implementation to be 250,000. If

$$h * w > max\_pixel$$

, we increase the average distance d and calculate h and w again with the same equation. This is repeated until

$$h * w \leq max\_pixel$$

.

Step 5: Map points to pixels

In this step, we map our point cloud points, which have float values for x and y ranging from $[-inf, +inf]$, to pixels which are integer values ranging from $[0, h|w]$, where h or w are the previously calculated image_height/width. We first have to get a local position within the box. Therefore we subtract the upper left corner position of the modifiable box $corner\_pos = (modifiable\_box\_min\_x, modifiable\_box\_max\_y)$ from all point positions in our points_and_colors array. After that, we divide the x and y position of the points by the average distance d calculated in step 4 and truncate the result to obtain an integer pixel position.

$$t := points\_and\_colors.position.translate(-corner\_pos)$$
$$x := points\_and\_colors.position = ceil(\frac{points\_and\_colors.position}{d})$$
$$M = t * x$$

After this transformation M is completed, the x and y position of the points_and_colors array now represents the pixel position in which it should be placed. Some points may overlap and have the same pixel position. In this case, we will blend the colors of all points in that pixel equally to obtain the final pixel color.

Next, we create a new array containing all pixels for the desired image, with dimensions x = w and y = h, where w and h are the calculated image_width and image_height values from step 4 and dimension z which contains r, g, b as values initially set to 0 which represent the color and a for the number of points in that pixel. This results in a 3 dimensional array (image_width * image_height * r,g,b,a). Then we loop through our points_and_colors array. For each entry in points_and_colors, we do the following:

$$image[points\_and\_colors.position].rgb \mathrel{+}= points\_and\_colors.color$$

$$image[points\_and\_colors.position].a \mathrel{+}= 1$$

After the loop is finished, we have to correct the color value if more than 1 point was assigned to a single pixel:

$$pixel.rgb = pixel.rgb/pixel.a$$

Now we have an image array that contains all the ground points mapped to pixel values with associated color values, and in those pixel positions where no points were mapped, the r, g, and b values are 0, i.e., the color black. Figure 3.10 shows an example of the current image array data.

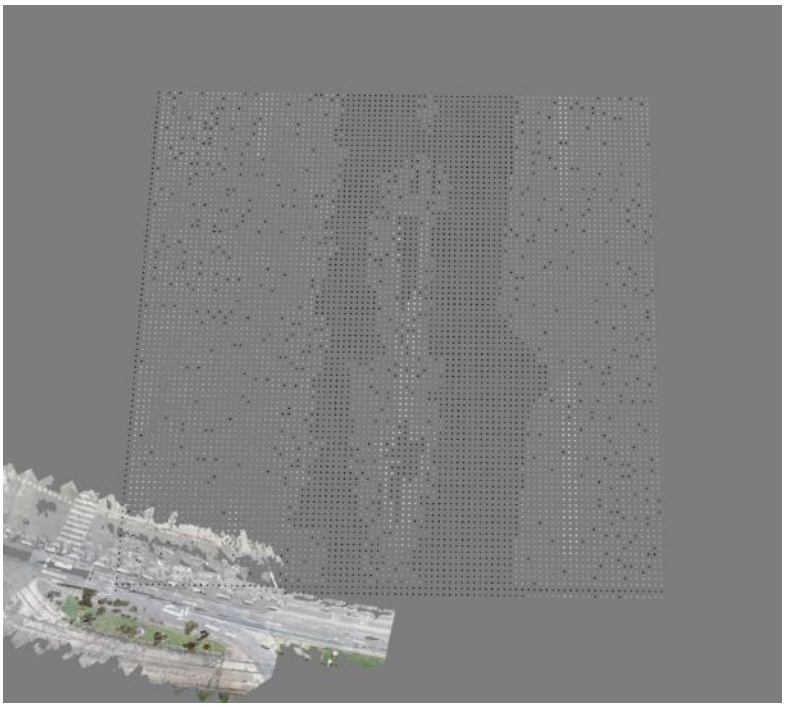

Figure 3.10: Point cloud, which visualizes the image array data.

Step 6: Image inpainting

For the image inpainting algorithm, we used a library called OpenCV [5]. This library provides the function inpaint() that takes two images as input. The first image is the distorted image that is going to be inpainted. An example of this is given in Figure 3.11. The second image is a mask where all pixels which will be inpainted from the distorted image are marked in white, as seen in Figure 3.12. The output of this function is the inpainted image, where all black pixels in the distorted image should be filled with a new color, observable in Figure 3.13.

To obtain the distorted image, it was necessary to multiply our color values from the image array by 255 and round the result to map the values which ranged from floating-point numbers between 0 and 1 to integers between 0 and 255. This is required because OpenCV uses a color range of 0-255.

To obtain the mask image, all black pixels of the distorted image were selected, converted into white pixels, and the rest set into black pixels.

After creating those image arrays, we used the imwrite() function from OpenCV to create .jpg images. At first glance, creating the .jpg files seems redundant and inefficient, but the imwrite() function is relatively fast, and it is no longer necessary to convert the image array to use the inpaint() function of OpenCV to work. Instead, we can directly use the .jpg images as input. Additionally, the user can use those images as a debug tool to see which areas are being recognized and how they were inpainted by the algorithm.

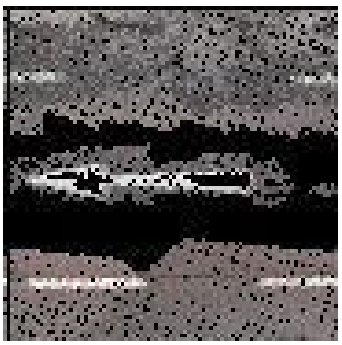

Figure 3.11: Incomplete image

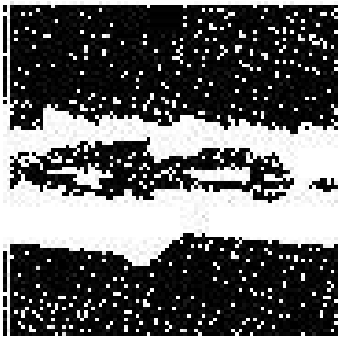

Figure 3.12: Mask image

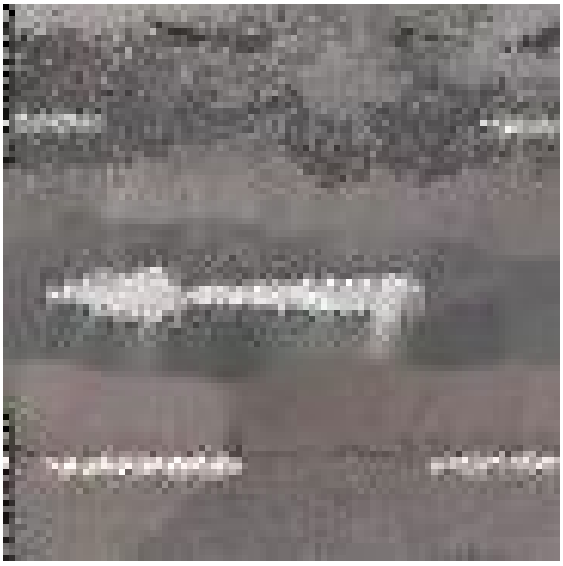

Figure 3.13: Inpainted image

Step 7: Map pixels to points

Only black pixels from the distorted image need to be selected as these are the inpainted pixels that need to be mapped back to points in the point cloud. Otherwise, we would not only create points where the gap is but everywhere. Therefore, we multiplied the mask image with the inpainted image to select exactly these inpainted pixels. This results in an array where only the inpainted pixels remain colored, and everything else is set to 0 and thus black.

Next, we traverse the entire image in the x and y directions and check if the r,g, and b values are not 0. If they are not, a new point is added to the point cloud. Therefore, we apply the inverse transformation M^{-1}, which was used in step 5, on the pixel position to return to the correct world position.

Additionally, we calculate a small random value with $random.uniform(-d, d)$, which gives an arbitrary value between $-d$ and $+d$, where d represents the average distance from step 4. This random value is added to the x and y position to obtain a more natural look for the point cloud. Otherwise, the added points are placed in a grid-shaped structure. Figures 3.14 and 3.15 show a comparison.

Depending on how often we increase the average distance in step 4, we add more than one point per pixel with identical color values to obtain a better-looking, denser cloud.

Figure 3.14: Image inpainting with random value added to x and y coordinates.

Figure 3.15: Image inpainting without random value added to x and y coordinates.

Step 8: Rotate back to the original rotation

Before adding the points to the original point cloud, it is needed to reverse transform and rotate them using the inverse transformation matrix R^{-1} mentioned in step 2.

When we apply this matrix, we have each point's x and y position and the corresponding pixel's color value that we can add to the original point cloud.

Step 9: Add new points to the original point cloud

The last element missing is a z value for the point position. To obtain an appropriate z value, we use an average z value of all initially selected points

$$avg\_z = \frac{sum(points\_and\_colors.z)}{len(points\_and\_colors)}$$

and assign this value to each point. The final step is to append the new points to the original point cloud points and our new colors to the original point cloud colors. Examples and tests of this implementation are shown in the evaluation section 4.3.

3.4 GUI

One of the goals was to write software that would be user-friendly; therefore, it was essential to implement a simple user interface. The software should also allow the user to operate the program as intuitively as possible. Therefore, we kept the GUI as minimalistic as possible and avoided confusion with too many buttons.

Implementation was straightforward because we could use Open3D's pre-built menu bar and callbacks that trigger when a button is pressed.

To avoid an excessive number of buttons, we implemented shortcuts. For example, the user must use shortcuts to transform the modifiable box. For example, a user wishes to move the modifiable box. In this case, they can press the key „T" and drag the mouse. Also, some buttons, such as „Remove Selected Object" and „Move Selected Object", have additional shortcuts „D" and „M" to achieve a more convenient workflow.

However, how does the user know which shortcuts exist or what a specific button does? We thought about adding a help tab, where the user can quickly check which keys must be pressed or which controls must be used to reach their goal. This approach works quite well since the software does not have too many features, and it is feasible to describe everything in a dialog box, as seen in Figure 3.16.

With „Open", a new point cloud can be added to the scene. It is assumed that the lowest point of each point cloud is always the point that should touch the ground since we presuppose that there are no outliers. Therefore, the first point cloud added is placed at the origin of the scene and shifted in the z-direction by the negative value of the smallest z-value in the point cloud. After this shift, the smallest z-value is therefore 0. Next, our RANSAC algorithm is executed to determine the ground surface as described in section 3.1, which is required for adding new objects.

If the „Open" button is clicked, and there is already a point cloud in the scene. The only difference is that users now see a subsampled preview of the point cloud they want to add and

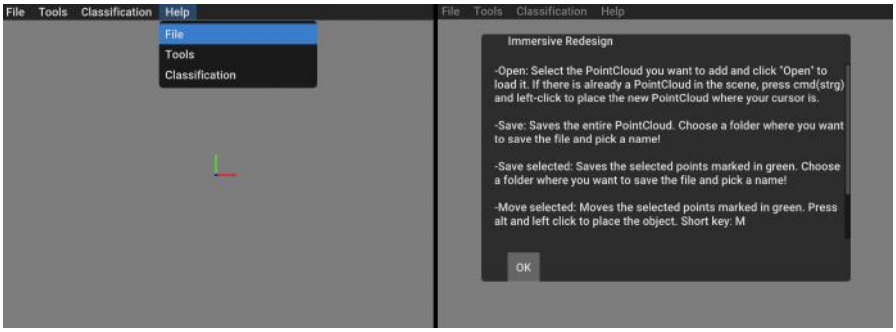

Figure 3.16: GUI of help tab on the left image and dialog box on the right image.

can choose where to place it. This preview is also placed on the ground with the smallest z value of the point cloud determined by the previous RANSAC calculation of the ground surface. After the user has decided to place the point cloud, the RANSAC algorithm is automatically executed again locally, as explained in 3.1, to position the point cloud ideally on the ground. Subsequently, the RANSAC algorithm is repeated to update the ground surface.

What is the difference between „Save“ and „Save selected“, seen in Figure 3.17? The „Save“ button is enabled as soon as the first point cloud is loaded into the scene and allows the user to save the entire point cloud. On the other hand, the „Save selected“ button is only enabled as soon as a specific part of the point cloud is selected. Selecting parts of the point cloud is possible with the tools „Select“ or „Select Object“. Then, for example, the selected area, which can be a tree or a car, can be saved separately.

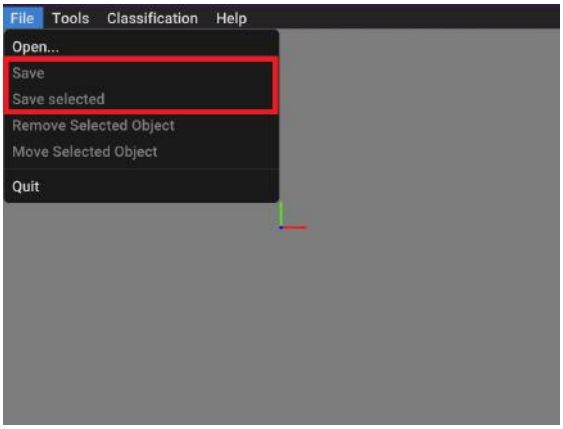

Figure 3.17: GUI of File tab where „Save“ and „Save selected“ are highlighted with a red box.

As can be seen in Figure 3.18, „Remove Selected Object“ and „Move Selected Object“ is also initially disabled. They can be enabled similarly to „Save Selected“ by selecting a particular part of the point cloud. If the user clicks on „Remove Selected Object“ or presses „D“, the selected part is deleted from the scene. If the user clicks on the „Move Selected Object“ button or presses „M“, the selected points can be moved, and we see a subsampled preview of the points following the user's mouse.

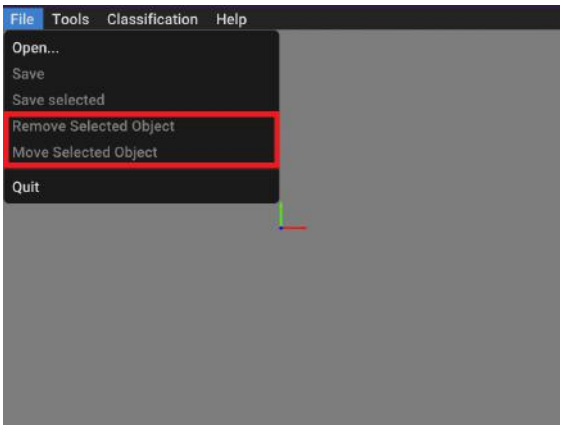

Figure 3.18: GUI of File tab where „Remove Selected Object“ and „Move Selected Object“ are highlighted with a red box.

When clicking „Select“, a modifiable box is created, and corresponding points within are selected as described in section 3.2. As seen in Figure 3.19, the selected points are also marked green for better visibility.

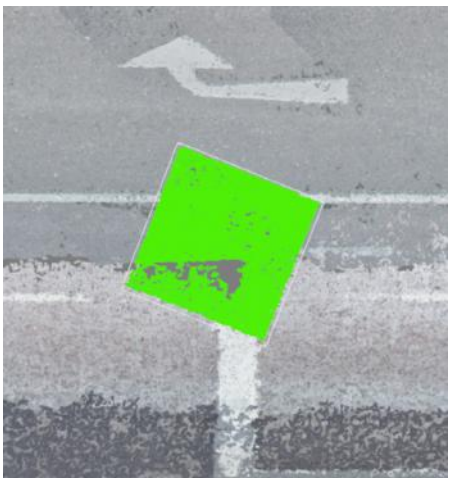

Figure 3.19: Selected points are marked in green.

If the user clicks on the „Image Inpainting“ button in the Tools tab, seen in Figure 3.20, the image inpainting algorithm will be executed as described in section 3.3. First, a modifiable box is automatically created, which the user can place as desired with a left click. Once the desired area is selected, the user must press „I“ to execute all further steps.

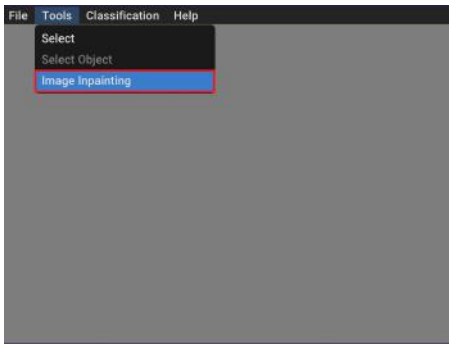

Figure 3.20: GUI of Tools tab where „Image Inpainting“ is highlighted with a red box.

Other tools, including filtering by classification, setting classification and selecting objects are described in Ahmed El Agrod’s work. [10]

CHAPTER 4

Evaluation

The evaluation was conducted using test data from the City of Vienna.

4.1 Software and Hardware Specification

My colleague Ahmed El Agrod and I used Python 3.9 for the project and developed it in the IDE PyCharm. We both used a Windows 10 operating system. However, the program was also tested on Linux Ubuntu 22.04.

The following components were used for the evaluation:

- GPU: NVIDIA GeForce GTX 980
- CPU: Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz, 4008 MHz, 4 Cores, 8 Threads
- RAM: 16.0 GB
- OS: Microsoft Windows 10 Home, 64-bit operating system x64-based processor

4.2 Adding Objects to the Ground

In this section, the efficiency of adding new objects is evaluated. We will measure execution time with different point cloud sizes to show how the implementation performs. We will also visually illustrate the distance between the ground and the added object.

Performance

To better understand how specific steps affect the performance, we have divided the adding process into three phases.

The first phase is the time required to translate the point cloud to the desired location and the locally executed RANSAC algorithm to more accurately place the point cloud on the ground. The second phase is, rendering the new point cloud and the last phase is computing the new plane by running the RANSAC algorithm on the entire downsampled point cloud. In the following graphs, these steps are referred to as „Local ransac“, „Rendering“, and „Update plane ransac“.

The first measurements were used to investigate the time needed to add a point cloud to the scene. Increasingly larger point clouds were added to determine if there was a correlation between the runtime and the size of the point cloud. The measurements are shown in Figure 4.1.

The results show that the runtime increases as the number of points in a point cloud increases.

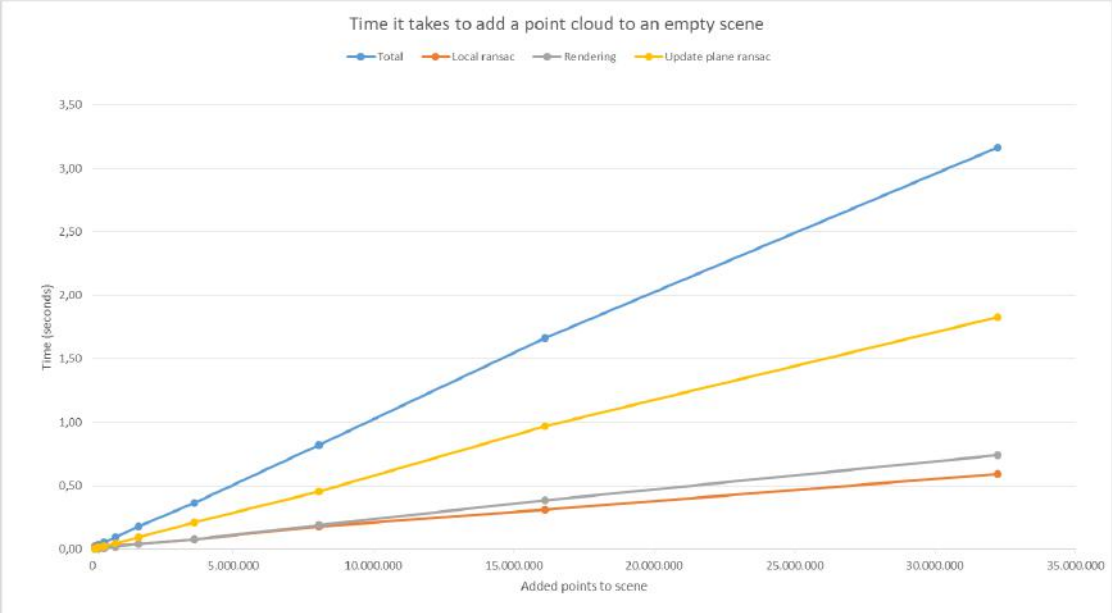

Figure 4.1: The time it takes to add a point cloud to an empty scene. Adding a larger point cloud increases the runtime—approximately linear growth.

It appears that a linear relationship exists between the runtime and the size of the added point cloud. It should be noted that all point clouds were added to an identical initial scene.

The second measurement was used to determine whether the same point cloud takes the same amount of time when added to the same scene multiple times.

In Figure 4.2, you can see the results of the time it takes if we add a point cloud of identical size multiple times to the same scene. In these measurements, we used a point cloud with around 400,000 points, and you can see the more often we add the point cloud, the more points there are in the scene, and the longer it takes to add the point cloud.

It could be assumed, based on the information in Figure 4.1, that the performance only de-

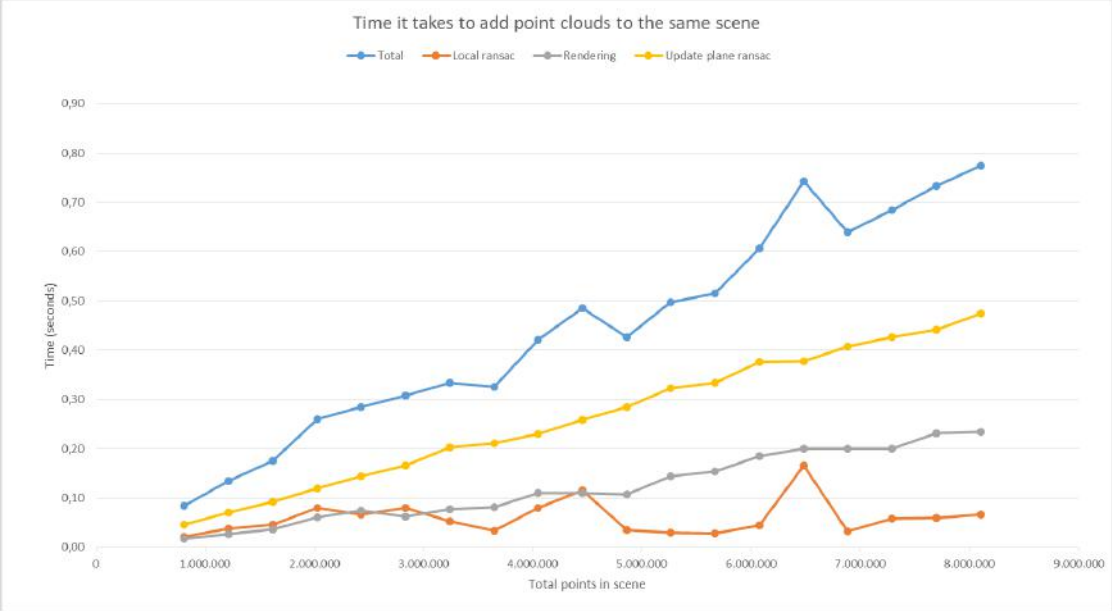

Figure 4.2: The time it takes to add a point cloud multiple times to the same scene. Linear growth is visible but dependent on Rendering and Update plane RANSAC.

pends on the size of the point cloud. However, the performance is heavily dependent on how many total points are in the scene since the execution time of the „Rendering“ and „Update plane ransac“ steps increase as the total number of points in the scene increase.

Here is a brief explanation of the reason for such runtimes:

- The duration of „Local ransac“ depends heavily on where the user clicks and adds the object because if there are many points in that area, the calculation takes longer than if there are fewer points. For this reason, there are peaks in the graph for the „Local ransac“ line.
- The „Update plane ransac“ shows a constantly increasing computation time, which is logical because if more points are in the scene, the RANSAC requires more time to calculate the new plane.
- The „Rendering“ step also takes more time as more points are in the scene as expected.

Best Case

In the following Figures 4.3, 4.4, 4.5, and 4.6, two highly successful examples of adding objects are illustrated. There is no visible gap between the ground and the objects. You can see that the object is placed on the ground. This functioned well because the RANSAC algorithm could

recognize the ground, and the object to be added had no outlier points. Every object shown in the following images was newly added at this location.

Figure 4.3 shows the addition of a vehicle to a scene, with the added vehicle visible in the right-hand image. Figure 4.4 shows the same example from another perspective to better visualize that the vehicle was perfectly placed on the ground. The red lines make it easier to recognize the car standing on the ground. Adding this point cloud took about 0.754 seconds; the vehicle has 19,644 points and the scene has 8,074,211 points.

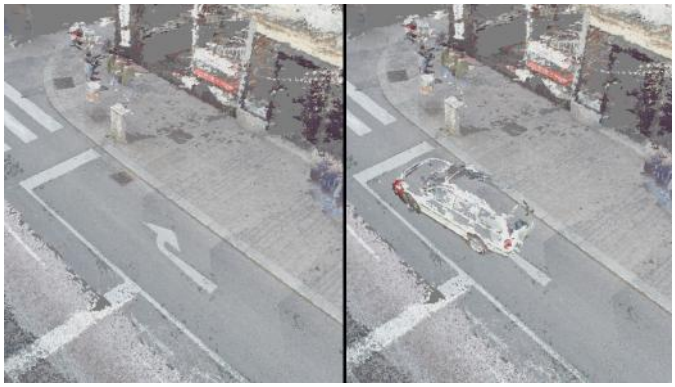

Figure 4.3: The addition of a vehicle to a scene.

Figure 4.4: Another perspective of the vehicle added to the scene. The red lines make it easier to recognize the vehicle standing on the ground.

Figure 4.5 shows the addition of a pole to a scene, with the added pole visible in the right-hand image. Figure 4.6 also shows the pole from another perspective to make it more recognizable that it was perfectly placed on the ground. The red circle makes it easier to recognize the pole standing on the ground. Adding that point cloud took about 0.736 seconds; the pole has 33,922 points and the scene has 8,088,489 points.

Figure 4.5: The addition of a pole to a scene.

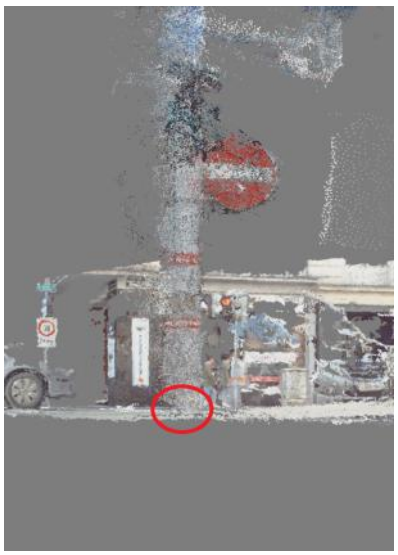

Figure 4.6: Another perspective of the pole added to the scene. The red circle is intended to make recognizing the pole standing on the ground easier.

Worst Case/Limitations

To achieve desired results, the point clouds must fulfill a requirement, as in the previous section 4.2. When adding new objects, there must be no or as few outlier points as possible. More specifically, these are outliers below the lowest point of the point cloud. This is important as the point clouds are placed at the lowest point on the ground. That means unwanted results will occur if the lowest point is an outlier, as shown in the following example. We wanted to add a wall that contained outliers below the actual lowermost point of the wall. This point cloud can be seen in Figure 4.7, where outliers are highlighted with a red circle.

Figure 4.8 shows the results of the adding process. The right-hand image shows a wall added

Figure 4.7: Wall with outliers below the lowest point of the wall. Outliers are highlighted with a red circle.

to the scene, which is above the ground. As always, the lowest point of the point cloud is placed on the ground, but in this case, the lowest point is an outlier, which causes the undesired result. In Figure 4.9, the outliers are highlighted with a red circle for better visibility.

Figure 4.8: The addition of a wall to a scene (right image). The wall is not on the ground as would be desired.

Figure 4.9: Outlier points of the wall are highlighted with a red circle.

Another known limitation of this project is that it is not possible to deliberately add objects above the ground. For example, as seen in Figure 4.10, the user wanted to add a traffic light, but they could only place it on the ground. Figure 4.11 shows how the user would like to position the traffic light.

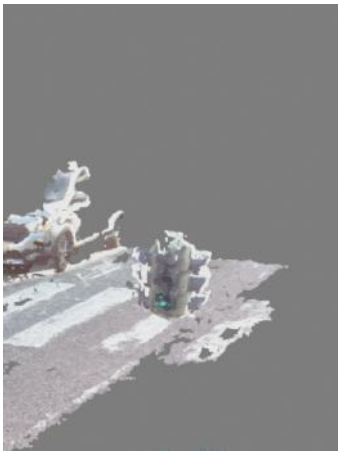

Figure 4.10: Shows that the traffic light is automatically placed on the ground.

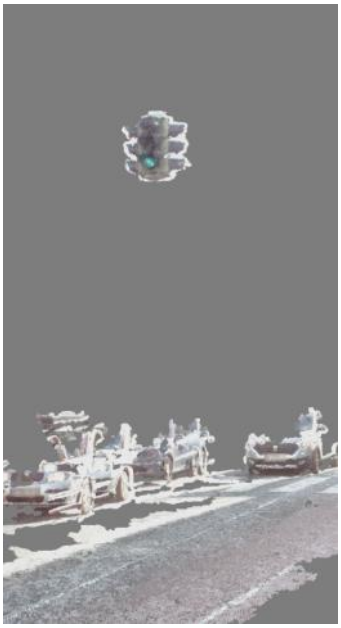

Figure 4.11: This image shows how the traffic light should be placed.

4.3 Image Inpainting

To evaluate the image inpainting implementation, we first examine how the performance changes as the selected area becomes larger. We also compare areas with different percentages of holes in the designated area for the same spot in terms of image quality and runtime. Finally, the overall quality is represented by erasing ground truth sections and then inpainting that section with the image inpainting implementation. Consequently, the degree to which the ground truth differs from the image inpainted region can be evaluated.

Performance and visual quality

The following measurements include the whole image inpainting process as described in 3.3. In Figure 4.12, we can see the measurements of the image inpainting as the area to inpaint becomes more extensive, and therefore, the points inside the selected area increase. It can be seen from the figure that the larger the area chosen, the longer the runtime.

In our tests, we found that, when selecting an area, the points do not exceed 100,000 points in

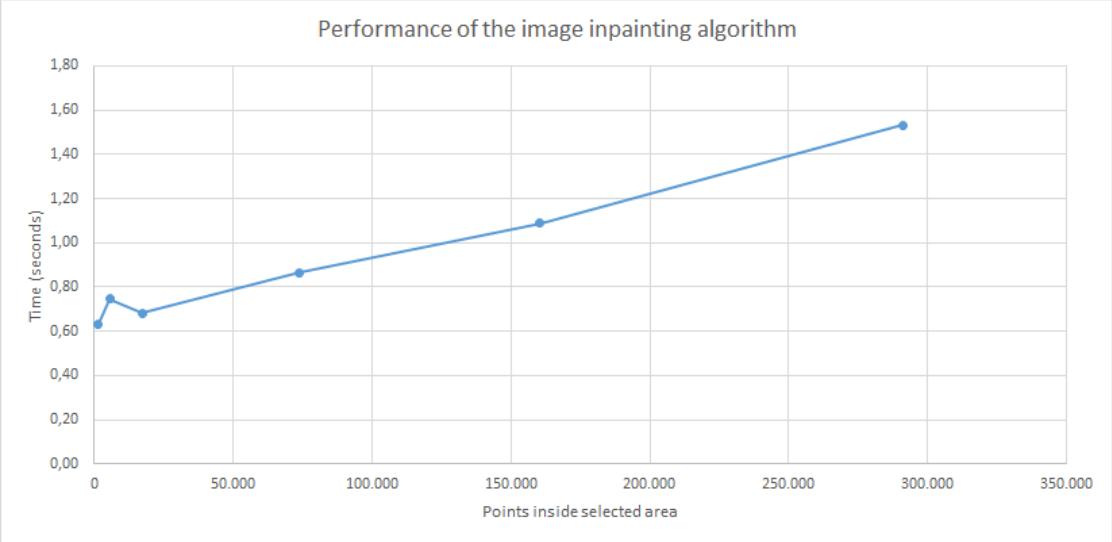

Figure 4.12: Performance of the image inpainting algorithm. The higher the point cloud size, the longer the runtime.

most cases, so the runtime is less than one second. Figure 4.13 shows an example of an area consisting of approximately 100,000 points, highlighting the area size which would have to be chosen. This box size is, in most cases, already much too large to fill individual separate holes.

Figure 4.13: Image inpainting on a selected area of approximately 100,000 points.

Next, the performance when choosing different percentages of the hole in the selected area will be considered. All measurements in Figure 4.14 were obtained from the same point cloud, which consisted of approximately 400,000 points, to ensure that the point cloud size was not affecting the measurements.

Performance-wise, as seen in Figure 4.14, the percentage of the hole has little to no effect on the algorithm's runtime. As the size of the hole (relative to the selected area) increases, the performance remains relatively the same or even has a slight downward trend. This downtrend can be explained as follows.

If the proportion of the hole increases, the calculated average distance becomes higher. Thus, the number of pixels in the created image is smaller, leading to a shorter runtime and fewer points that are inpainted. This leads to poor visual results because the average distance is almost certainly falsified, and as mentioned, fewer points than expected are added. So, ultimately, the hole is filled but not with as many points as it should be.

This effect can be seen in the following Figures. For the next two examples shown in Figure 4.15, we cut out a hole of the same size. In Figure 4.16, you can see the result of the image inpainted point cloud if the percentage of the hole is very high. It can be seen that the density of points in that area is lower than in the surrounding area. Also, the modifiable box (white square) is visible and shows how small the selected area was. In Figure 4.17, it can be seen that the density of points is close to the density of the surrounding area. We achieved this result by selecting a much larger area with the modifiable box and therefore reducing the hole percentage,

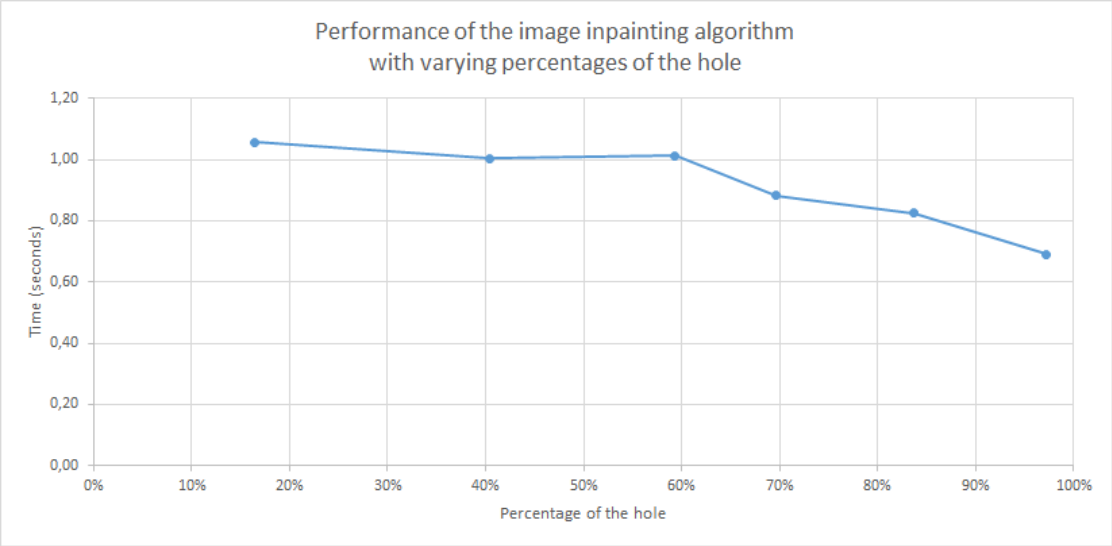

Figure 4.14: Performance of the image inpainting algorithm with varying percentages of the hole. With a point cloud of approximately 400,000 points.

as seen in Figure 4.18.

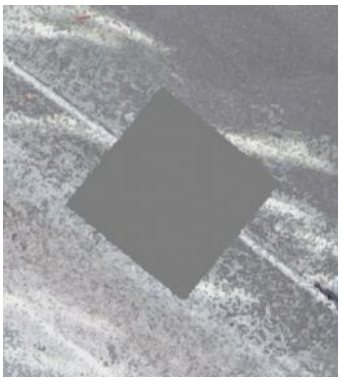

Figure 4.15: The cut-out hole.

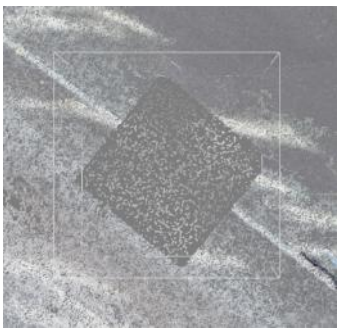

Figure 4.16: Image inpainted point cloud where the percentage of the hole was very high. Also, the modifiable box (white square) is visible and shows how small the selected area was.

Figure 4.17: Image inpainted point cloud with a lower percentage of the hole.

Figure 4.18: Image where the size of the modifiable box (white square) is much bigger, which leads to better image inpainting results as seen in Figure 4.17.

Best Case

Figures 4.19 and 4.21 show a point cloud with incomplete ground parts due to the LIDAR sensor not detecting them. With our image inpainting algorithm, we can fill those holes and achieve the following results, as seen in Figures 4.20 and 4.22. On close inspection, it can be seen that there was a hole, but overall the appearance has been improved, and the gap is patched. To achieve a good result, it is necessary that the selected area surrounds the entire hole and, as already described, the percentage of the gap must be small enough.

Figure 4.19: Point cloud with missing ground parts.

Figure 4.20: Image inpainted point cloud.

Furthermore, parts of the street were deleted and refilled with the image inpainting algorithm to compare the ground truth section with the image inpainted one. Two examples were made,

Figure 4.21: Point cloud with missing ground parts.

Figure 4.22: Image inpainted point cloud.

and the results are given in Figures 4.23 and 4.24. The picture with the cut-out hole is on the left side, the image inpainted point cloud can be seen in the middle, and the ground truth can be seen on the right side. As we can see, similar to Figures 4.20 and 4.22, it is still apparent that there was previously a hole that has been inpainted. The ground truth is visually superior to our implementation, even tough the gap has been filled with a convincing color and sufficient points.

Figure 4.23: Point cloud that has been image inpainted. On the right side, the ground truth is visible.

Figure 4.24: Point cloud that has been image inpainted. On the right side, the ground truth is visible.

Worst Case/Limitations

When trying to inpaint areas where the hole is extensive and at the edge of the selected area, the inpainting algorithm does not calculate the color for all pixels that should be inpainted. Thus some pixels remain still black. This effect is illustrated in the following two examples. You can see the distorted image on the left side of Figures 4.25 and 4.27, while on the right side of the respective Figures, the inpainted image is visible. Figures 4.26 and 4.28 show the result of the image inpainting on the point cloud. The red circle highlights the incorrectly colored black points.

Figure 4.25: Left: incomplete image. Right: inpainted image.

Figure 4.26: The inpainted point cloud. The red circle highlights black points.

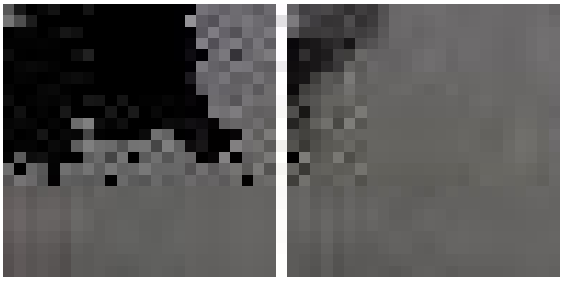

Figure 4.27: Left: incomplete image. Right: inpainted image.

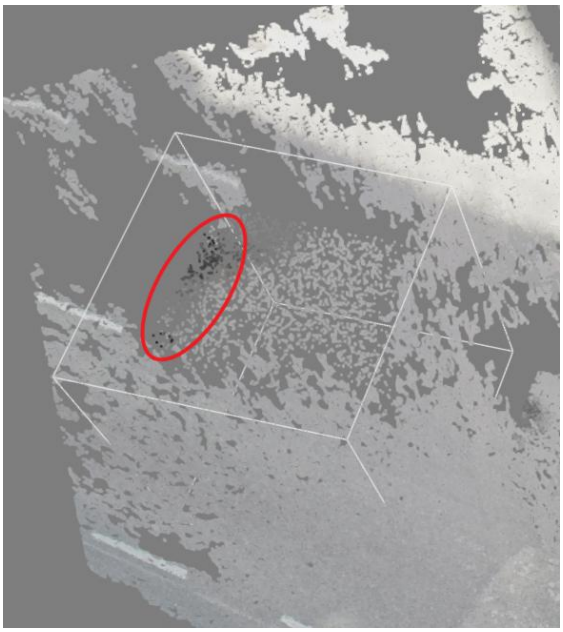

Figure 4.28: The inpainted point cloud. The red circle highlights black points.

CHAPTER 5

Future Work

5.1 Placing Objects

In our current implementation, it is impossible to deliberately place objects not on the detected ground. So, for example, if a user wishes to position a traffic light in the air, as shown in 4.10, the traffic light would snap to the ground.

To solve this issue, we could implement a feature where the user can manually move the object vertically after placing it on the ground.

5.2 Lighting and Shadows

Lighting and shadows have not yet been dealt with in this work. Currently, we have no light source or shadows projected onto the point clouds in our scene. Also, the colors of the point cloud points are not adjusted. We have deliberately omitted these features as too many aspects needed to be addressed to achieve good results and would therefore be beyond the scope of this work. Since the software can add new objects to different point clouds, we would have to adjust the colors of each point cloud accordingly. For example, if a tree were to be inserted from another scene where the lighting conditions are vastly different, we would have to change the color of every point in that tree to some degree so that it still looks realistic without the tree standing out conspicuously. Even if the tree is in the same scene but moved to another position, it is possible that it was initially captured in sunlight and now could theoretically be placed in a shady place. The tree would be much brighter as if still illuminated by sunlight, and everything around it would be darker because it is actually in shadow.

5.3 Multiple Ground Detection

Another limitation is the preview, which is shown before the user places an object on the ground. The preview of where the point cloud will be added is not always placed on the ground. This is because our preview uses only one plane representing the scene's entire ground. As soon as we have hilly terrain or, for example, already have a road where one section is straight and another section has a slope, the ground detection can no longer display the preview correctly because it would need at least two planes to detect the straight and sloped sections.

It should be noted, however, that this does not affect the point cloud that is finally added since the RANSAC algorithm is executed locally as soon as the user clicks to place the point cloud. Thus, the point cloud is placed on the ground as usual.

In future work, we could split our initial point cloud scene into several sections and calculate a plane for each. As soon as an object is added or moved, the correct plane must be selected for the respective x and y positions. After the correct plane has been chosen, the x and y position of the mouse can be substituted into the specific plane equation, as usual, to obtain the z coordinate for our preview.

5.4 User Feedback

Currently, our software does not offer much feedback for users if specific actions such as Image Inpainting are triggered. For example, the image inpainting algorithm is sometimes aborted because too few points are selected. Instead of just printing text to the console, we could inform the user with a label that shows an error message and automatically suggest an enlarged area with enough points to execute the algorithm correctly.

It would also be possible to add labels to the scene that pop up when other actions are performed that are not immediately apparent to the user to give better feedback. Another improvement would be a bar at the bottom where there are labels that always show which shortcuts or actions are possible with the current tool selection.

Bibliography

[1] http://www.open3d.org/docs/release/python_api/open3d.geometry.axisalignedboundingbox.html. Accessed: 2022-07-02.

[2] http://www.open3d.org/docs/release/python_api/open3d.geometry.pointcloud.html. Accessed: 2022-07-02.

[3] http://www.open3d.org/docs/release/tutorial/visualization/visualization.html. Accessed: 2022-07-02.

[4] M. Bertalmio, A.L. Bertozzi, and G. Sapiro. Navier-stokes, fluid dynamics, and image and video inpainting. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–I, 2001.

[5] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[6] Keming Cao and Pamela Cosman. Denoising and inpainting for point clouds compressed by v-pcc. *IEEE Access*, 9:107688–107700, 2021.

[7] Preeti Chatterjee, Subhadeep Jana, and Souradeep Ghosh. Comparative study of opencv inpainting algorithms. 2021.

[8] Chinthaka Dinesh, Ivan V. Bajić, and Gene Cheung. Exemplar-based framework for 3d point cloud hole filling. In *2017 IEEE Visual Communications and Image Processing (VCIP)*, pages 1–4, 2017.

[9] Chinthaka Dinesh, Ivan V. Bajić, and Gene Cheung. Adaptive nonrigid inpainting of three-dimensional point cloud geometry. *IEEE Signal Processing Letters*, 25(6):878–882, 2018.

[10] Ahmed El Agrod. Immersive Redesign, Bachelor thesis, Institute of Visual Computing Human-Centered Technology, Vienna University of Technology, 2022.

[11] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, jun 1981.

[12] Zeqing Fu and Wei Hu. Dynamic point cloud inpainting via spatial-temporal graph learning. *IEEE Transactions on Multimedia*, 23:3022–3034, 2021.

[13] Claus Scheiblauer and Michael Wimmer. Out-of-core selection and editing of huge point clouds. *Computers Graphics*, 35(2):342–351, 2011. Virtual Reality in Brazil Visual Computing in Biology and Medicine Semantic 3D media and content Cultural Heritage.

[14] Alexandru Telea. An image inpainting technique based on the fast marching method. *Journal of Graphics Tools*, 9(1):23–34, 2004.

[15] Yifan Wang, Andrew Liu, Richard Tucker, Jiajun Wu, Brian L. Curless, Steven M. Seitz, and Noah Snavely. Repopulating street scenes. *CoRR*, abs/2103.16183, 2021.

[16] Xin Wen, Peng Xiang, Zhizhong Han, Yan-Pei Cao, Pengfei Wan, Wen Zheng, and Yu-Shen Liu. Pmp-net++: Point cloud completion by transformer-enhanced multi-step point moving paths. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2022.

[17] Yikuan Yu, Zitian Huang, Fei Li, Haodong Zhang, and Xinyi Le. Point encoder gan: A deep learning model for 3d point cloud inpainting. *Neurocomputing*, 384:192–199, 2020.

[18] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.