Visual Computing - Point Cloud Rendering

Kranzl, Manuel
ai22m038@technikum-wien.at

June 14, 2023

1 Specification of the task

This paper represents the analysis of the solution of the third topic - Point Cloud Rendering. The main task was to visualize a point cloud, given in the file *model.pts*. The color of every point was also given in this file. When rendering the point cloud the size of the points had to be changeable.

As my specialisation is AI technologies I wanted to focus on improving my python skills, therefore I solved this exercise in python. I used the *Open3D* package for the visualisation itself.

2 Solution

2.1 Used Packages

Below are the used packages and their versions. Note that the used python version (3.6) is older then the current standards, as it was not possible to install the *Open3D* package with a newer version.

```
import os
import open3d as o3d
import numpy as np
from preprocessing import preprocessing
```

2.2 Loading Data

In this first part of the code the data points were loaded into numpy-arrays. Loading the file directly into a point cloud variable caused problems, as only the first two points were detected.

```
def load_points(file_name):
    # Get full file path
    main_dir = os.path.dirname(os.path.dirname(os.path.
    abspath(__file__)))
    pts_file_path = os.path.join(main_dir, 'data', file_name)
    # Load the points and RGB values from the file
    points = []
    colors = []
    with open(pts_file_path, 'r') as file:
        for line in file:
            values = line.strip().split(' ')
```

```
            x, y, z = map(float, values[:3])
            r, g, b = map(int, values[3:])
            points.append([x, y, z])
            colors.append([r, g, b])
    # Convert the lists to numpy arrays and return them
    points = np.asarray(points)
    colors = np.asarray(colors)
    print("Points loaded successfully!")
    return points, colors
```

2.3 Converting into a point cloud

In this subroutine a new point cloud gets initialized and the points and their colors are saved into it.

```
def convert_to_pointcloud(points, colors):
    point_cloud = o3d.geometry.PointCloud()
    point_cloud.points = o3d.utility.Vector3dVector(points)
    point_cloud.colors = o3d.utility.Vector3dVector(colors / 255.0)  # Normalize RGB values to [0, 1]
    print("PointCloud created!")
    return point_cloud
```

2.4 Visualisation

As the direct visualisation does not allow for a change in the point size, the visualizer had to built manually.

```
def visualisation(point_cloud, point_size=2, zoom=False):
    # Create a visualizer object
    visualizer = o3d.visualization.Visualizer()
    visualizer.create_window()
    # Add the PointCloud to the visualizer
    visualizer.add_geometry(point_cloud)
    # Get the rendering option and modify the point size
    render_option = visualizer.get_render_option()
    render_option.point_size = point_size  # Set point size
    # Set camera (used for comparison pictures in document)
    if zoom:
        ctr = visualizer.get_view_control()
        parameters = o3d.io.read_pinhole_camera_parameters("data\\ScreenCamera_01.json")
        ctr.convert_from_pinhole_camera_parameters(parameters)
    # Run the visualizer
    visualizer.run()
    visualizer.destroy_window()
```

2.5 Main program

```
if __name__ == '__main__':
    point_size = 2
    # Loading data from model.pts file
    points, colors = load_points('model.pts')
    # Preprocessing
    points, colors = preprocessing(points, colors, eps=0.05, min_samples=40, subset_size=10000)
    # Converting the numpy arrays to a point_cloud
    point_cloud = convert_to_pointcloud(points, colors)
    # Visualisation using the Open3D library, zoom=True will put camera in front of dinosaur
    visualisation(point_cloud, point_size=point_size, zoom=False)
```

3 Results

The next two figures show the outcome using 2 as point size. The left image (Figure 1) represents the output with standard rotation while on the right image (Figure 2) the

dinosaur has focused using zooming and turning, which is possible with the *Open3D* package.

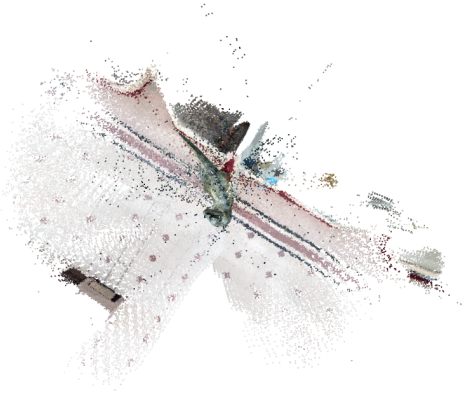

Figure 1: results/pc1.png

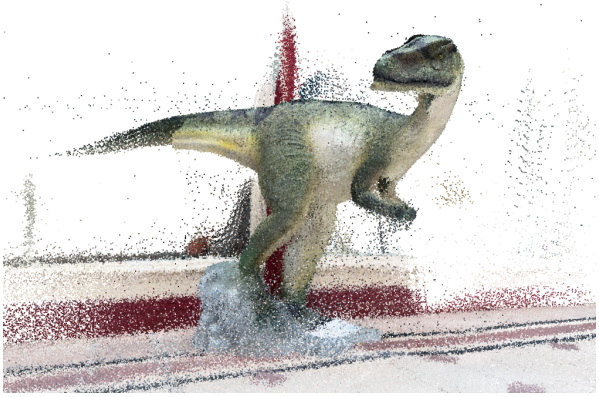

Figure 2: results/pc2.png

4 Discussion

Besides the implementation itself there were three topics to discuss. This discussion will be the content on this section.

4.1 Exploring the dataset

There are 1.797.943 points in this data set, each line of the file representing one point with it's six values ((x, y, z) and (r, g, b) for its position and color). When looking at the created image itself we can see lots of noise points (especially in front of the dinosaur). Besides that, the floor and wall are also in the picture (or rather in the data set). This might be a mistake in capturing the data.

4.2 Challenges

One particular challenge with the approach using python and Open3D was the memory usage, especially when trying to use preprocessing algorithms, as the numpy arrays get loaded into the RAM. Having a larger data set might cause problems with Open3D and this approach might not work anymore. However, there are ways to only partially load the data set, one method can be seen in the preprocessing method in the submission.

4.3 Improving the outcome

There are different approaches for improving the output picture.

- Cleaning up the data set: Removing outliers and noise from the point set could improve the rendered picture's quality. As the points (and RGB values) were first stored as a numpy array it would be easy to find outliers and/or noise. Inside the submission there is a subscript *preprocessing.py* which uses DBSCAN for filtering out points without neighbours. There is a problem with it, as the tail of the dinosaur

has the same density of points than some noise clouds. Therefore not all noise points can be removed.

- Point size: Playing around with different point sizes may in- or decrease the quality of the rendered image. As seen above the dinosaur may appear more clear when using larger points. However when using larger point sizes outliers and noise points become more and more visible, this is why increasing the point size should not be done without using data preprocessing before.
- Adding lightning and shading: As the picture from the output has no shadows it appears unreal. In *Open3D* there are built-in options to include Phong shading in the rendering.

The first two points were implemented in the project (the third one not, as writing a shader was the topic of the second chosen exercise). The results can be seen in the following pictures. At the left (Figure 3) the original picture ($point\_size = 2$) is shown while preprocessing was used for the right picture (Figure 4).

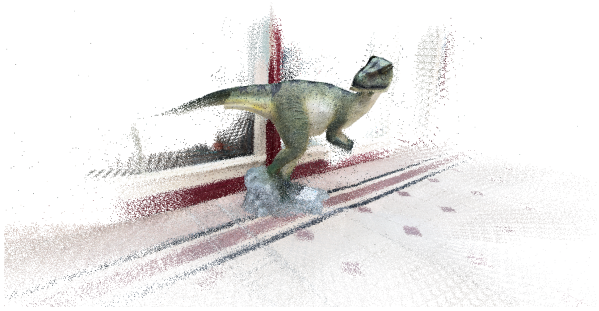

Figure 3: results/pc3.png

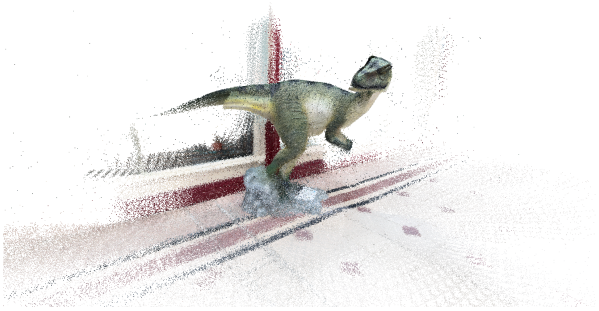

Figure 4: results/pc4.png

In the next two pictures (Figure 5, Figure 6) we see the same results, only for an increased point size ($point\_size = 4$).

Figure 5: results/pc5.png

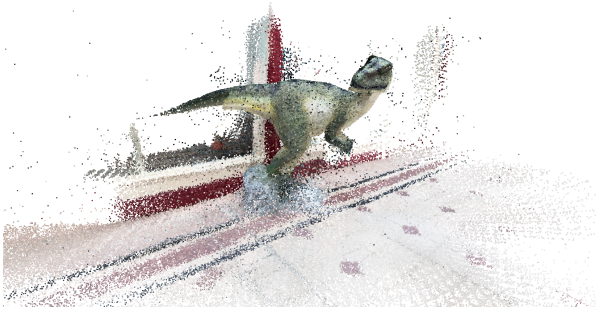

Figure 6: results/pc6.png

As the results are not very visible in these small pictures, all the result image files are provided in the project.