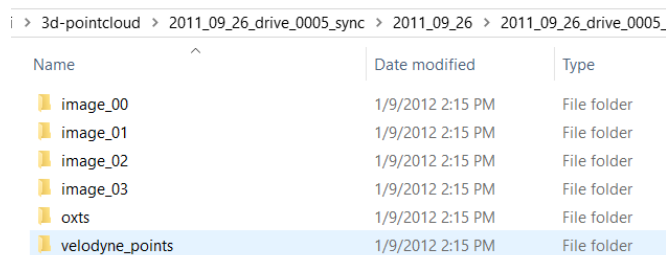


Overview for the pointcloud processing assignment

My assignment for the semester was to create a software that could display and analyze point clouds. In this short documentation I will explain the key aspects of the. My main goal for the project was to create an application that can automatically read the dataset I used and render it onto the screen in a comprehensive way. The calculations in the project are carried out by the Numpy library and for the display I used Open3D, although I have to mention that Open3D in this particular solution is mainly used to render a simple image to the screen.

Dataset

For the project I used the KITTI vision benchmarking suite. They support novel real-world computer vision benchmarks and large datasets are available for developing and testing self-driving applications. The vehicles used for the dataset recordings are equipped with LiDARs, cameras and internal measurement units. After using and testing their databases I also concluded that their datasets are great and well calibrated. The raw data section can be found on the website ([The KITTI Vision Benchmark Suite](http://www.kitti.vision.uni-luebeck.de/)) and I used synced and rectified data in a city environment.



Name	Date modified	Type
image_00	1/9/2012 2:15 PM	File folder
image_01	1/9/2012 2:15 PM	File folder
image_02	1/9/2012 2:15 PM	File folder
image_03	1/9/2012 2:15 PM	File folder
oxts	1/9/2012 2:15 PM	File folder
velodyne_points	1/9/2012 2:15 PM	File folder

The main data folder contains four subfolders. The folders named image_x are containing stereo images taken during the drive. The oxts folder contains location data and data from the IMU, velodyne_points contains the point clouds recorded by the LiDAR. The folders have timestamps associated with each individual data point creating a spatially and temporally coherent dataset.

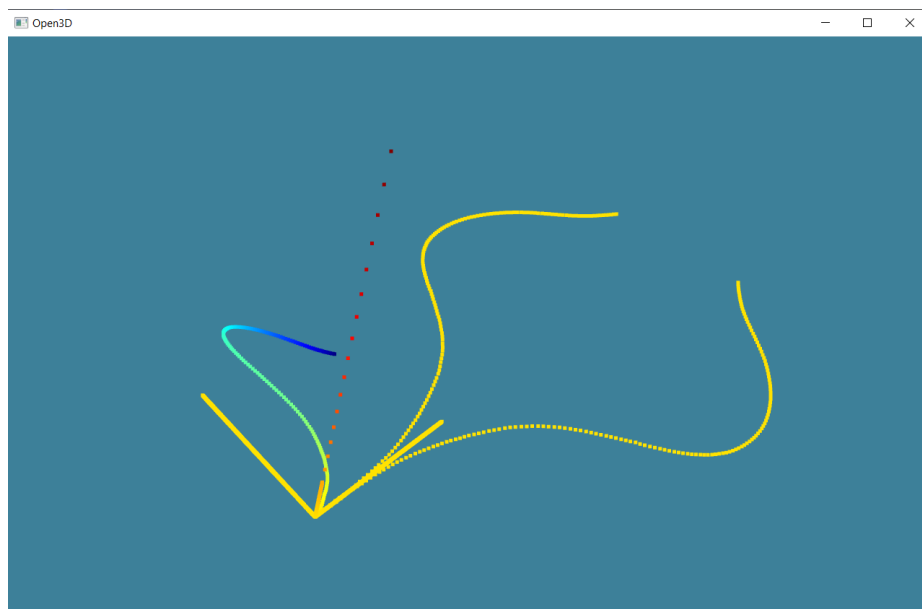
The application does not (yet) have an UI, therefor the path of the dataset has to be set from the main file in the solution. After that the reading and storing of the data is done automatically.

Application

After setting the path of the dataset that the user wants to display and pressing start the application will first load all the necessary data for it to display the recorded frames of the test drive. Four main classes will be instantiated here. In order they are the OxtsDataReader, the LidarDataReader, an OxtsAlignmentUtility and the PointCloudData class. The logic for the reader classes is very similar. They read the LiDAR and the oxts data into data maps, and the data is associated with a key. The key of the dictionaries are strings, the names of the LiDAR data files. (0000000001, ... , 0000000099, ...). The associated timestamps are also stored in corresponding dictionaries. The OxtsDataReader also stores the origin of the IMU, the rotation and the position. These will be the origin of the simulation. All of the reader

classes have a state based enumerating logic in place, the data from the dictionaries can be queried with the proper keys and getter methods, and the keys can be obtained by the `getCurrentName` and `getNextName` functions.

The next object created after the raw data is stored is the `OxtsAlignmentUtility`. It takes an `OxtsData` and a `LidarData` reader as an argument. It will read the oxts data from the corresponding reader, enumerating it with the keys of the `LidarDataReader`. The main aim of the utility is to convert location data to a local frame of reference for the application, so later it can be displayed coherently. The rotation data for each frame of the vehicle's path is calculated and stored. The location of the vehicle is calculated four different ways. First I used GPS data which appeared to be satisfactory, but the accuracy of the measurements showed big changes between datasets. The location data is also calculated from the vehicle's North-East velocity, parallel to Earth's surface, and furthermore calculated from the rotation and speed of the vehicle. Here I had to calculate the time between the timeframes and solve for distance from time and velocity. Setting which location mode the app should use for display can be changed with a key, and a solution for mixing this data is partially implemented. I was able to get the best results by using the rotation and speed of the vehicle to calculate the spatial coordinates. The utility also has many helper functions so the path can be rotated into place and mirrored if necessary.



The dotted axis is the Z axis, facing towards the camera. The other two axes are X and Y. The two yellow paths are the modified location data of the vehicle, the one with the green and blue colors is the original, about 45 degrees to each of the axes. The yellow paths are projected and parallel with the XY plane.

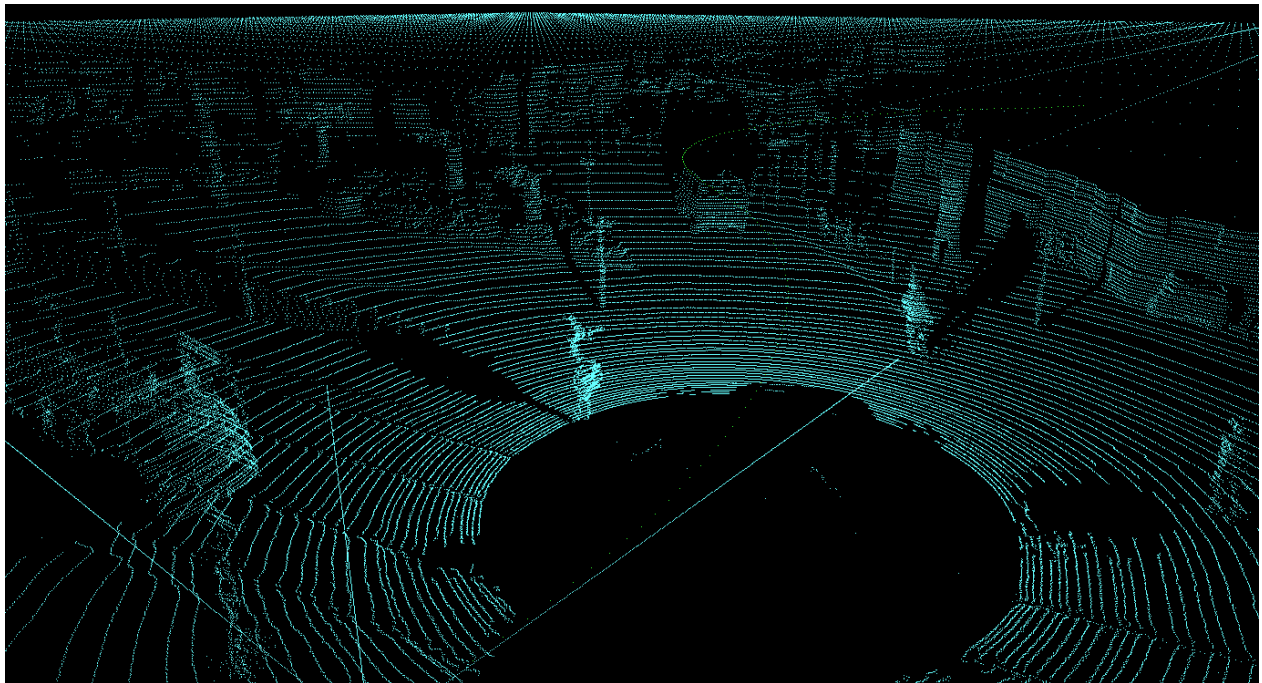
The last two classes instantiated here are the `PointcloudData` and the `PointcloudController` classes. The first class is for storing and segmenting the data coming from the utilities and reader classes, and the `PointcloudController` is for handling graphics and keypresses.

The `PointcloudData` class starts by segmenting each frame of the LiDAR scan into a `VoxelData` class. It then stores this data in a dictionary. The same keys can be used here as any other reader, and will return one `VoxelData` class per key. The `VoxelData` class takes a voxel size and a minimum number of points threshold for parameters. The first is for how large the voxels in a frame will be (meters) and the latter is for filtering

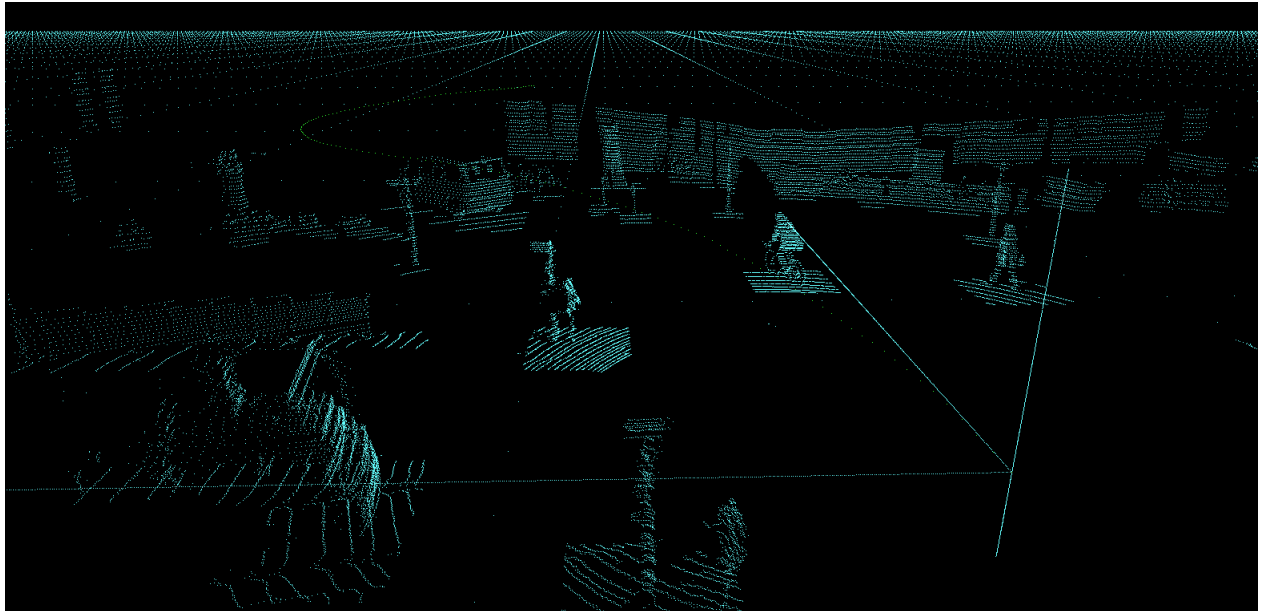
out voxels where there are not enough points. After a Voxel is created voxelize, clearNotDense, bake and bake_points can be called on it. Voxelize will take a numpy array and with vectorized operations will distribute these points into buckets. The clearNotDense method will delete buckets where the height is smaller than 25 centimeters or there are not enough points. Bake and bake_points are just utility functions so the whole voxel grid (and the points in the grid) can be returned as one numpy array and later rendered faster. All VoxelData class has an underlying dictionary of DynamicNP classes that can store points and reallocate their sizes if necessary. A voxel of a VoxelData class can be queried by the spatial coordinate of the voxels along the XY plane, that is if the voxel size was one, the keys to the voxels are numpy arrays of the form: [0, 0], [-42, 42], [11, 0], [0, 66] ...

The PointcloudController class has a Camera class associated with it that will do the rendering. The projectWithNp will take a numpy array of points and perform a perspective projection on them. The points then are projected into a 2D array that will be the final image for the frame. The point cloud controller will check for keypresses (W, A, S, D, Shift, Space, "P"ause) in a loop and project the outgoing images onto the screen. Colors of the projections can be set, and a composeImage function can overlay these images. Some static objects are rendered each frame, like the path of the vehicle, the coordinate axes, and a ground grid. These are only created only once in a fashion similar to functional programming, then are stored in memory. After the heavy lifting is done in the start when all the data is loaded, the app aims to keep a constant 60 fps, but this is heavily dependent on how many objects and points are rendered each frame.

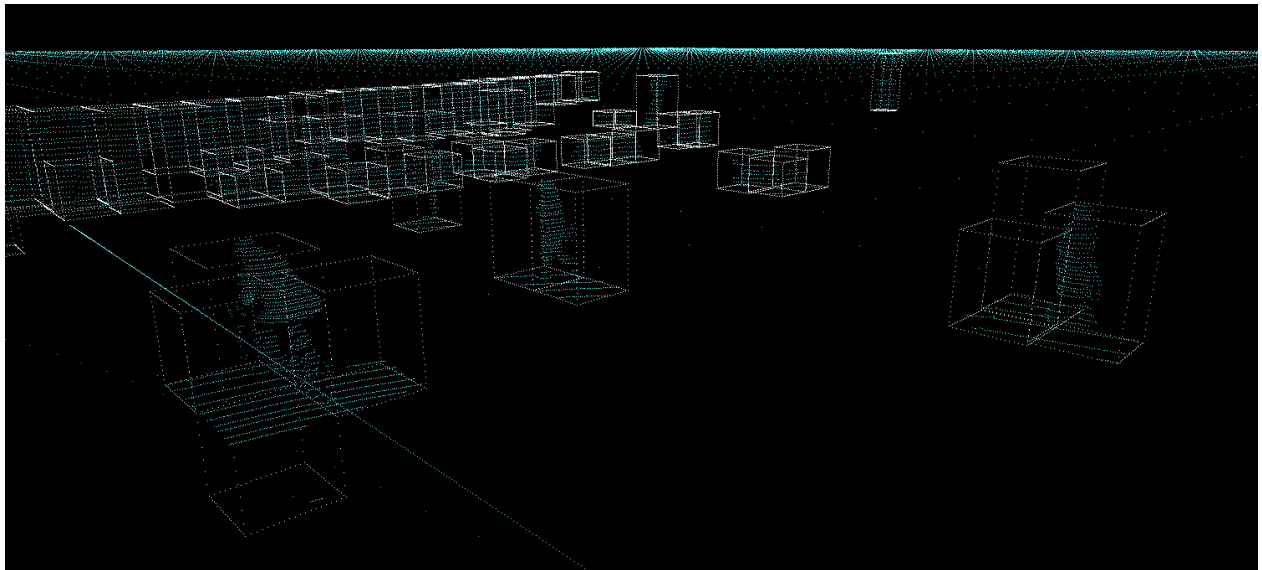
This the first frame of the dataset rendered over a grid.



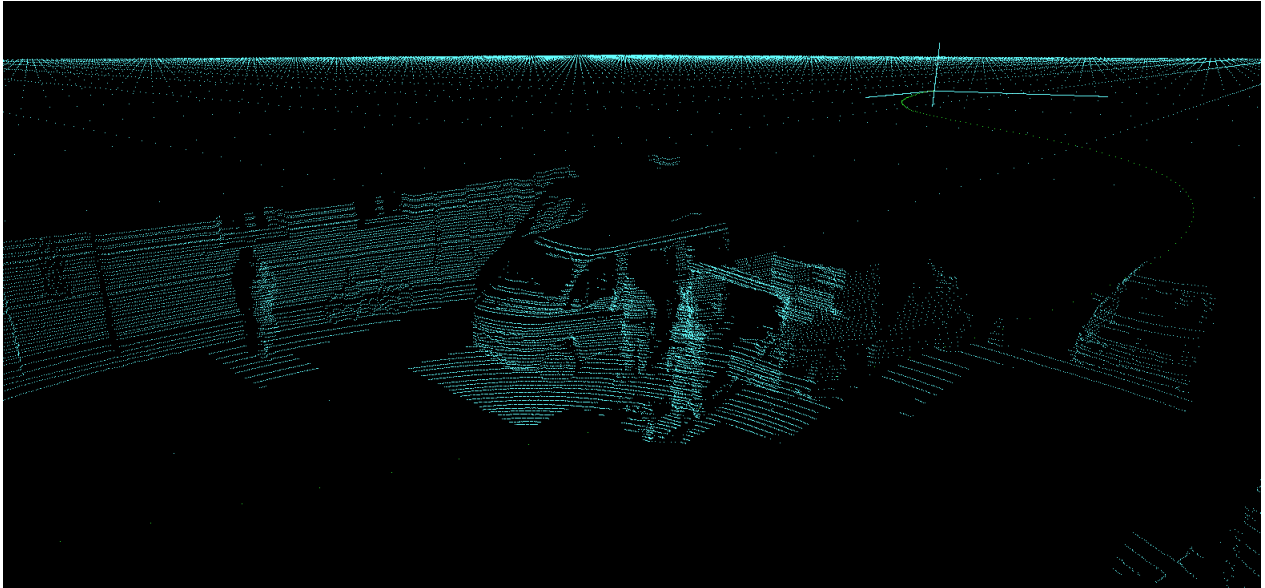
After voxelization and segmentation most of the ground gets filtered out and only tall objects remain.



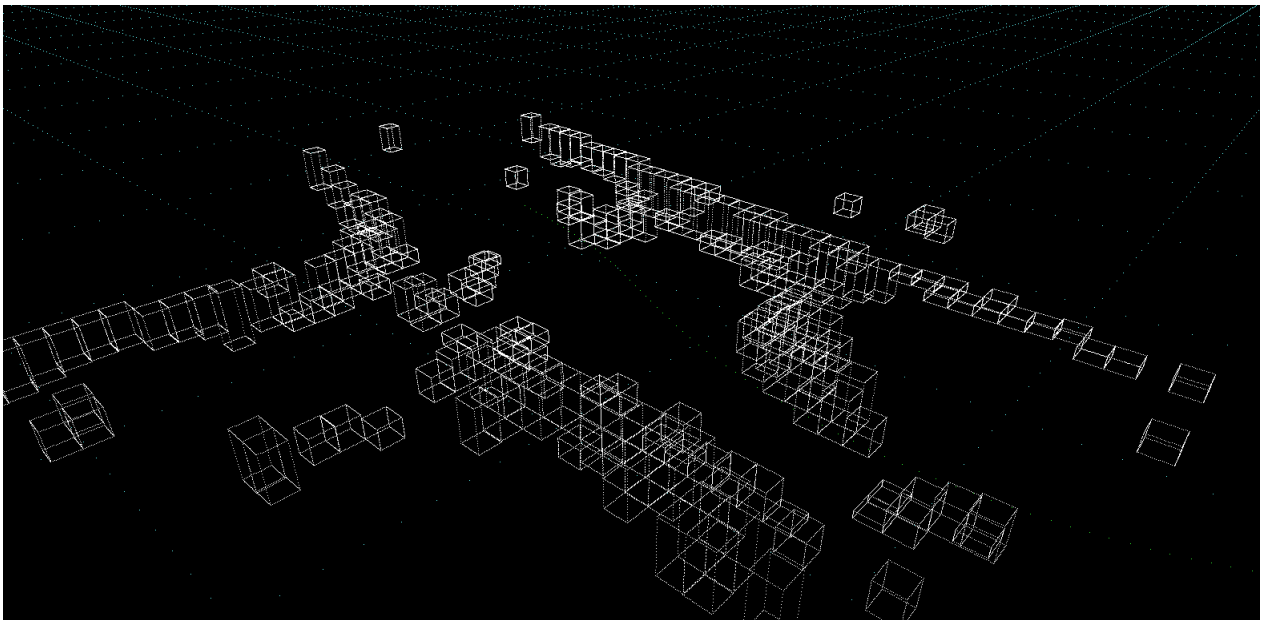
All of these segments have bounding boxes associated with them that correspond to the voxel size. On this snapshot a cyclist and two pedestrians can be seen, and in the background a wall.



On this snapshot a group of individuals can be seen around a vehicle, in front of a wall. Voxels are only around objects of interest.



The bounding boxes can be rendered separately from their points to give an overview of the scene. The PointcloudData class also contains an all-encompassing VoxelData that has all the points and bounding boxes from the whole dataset.



This concludes the overview of the assignment. In the next semester I'd like to continue the project as my master's thesis. There is still a lot of room for improvement, mostly on the rendering part and fine tuning some algorithms. Object detection can also be implemented in the future, with some new features.