Jon Rippe
PDAQS
Progress Report 21.9.0

Issues Addressed

- LiDAR data
- LiDAR clustering
- LiDAR point overlay on video

LiDAR Data Decoding/Extraction

The DataPacket module (and sub-modules) that was being used to decode and extract the PCAP files generated by the LiDAR device was not functional. This broke the previous team's workflow for all processing done with the LiDAR data. An attempt to obtain a bird's eye rendering of LiDAR data extracted by the module showed no discernable environment. To verify the integrity of the PCAP files, they were opened using 3rd party software VeloView (figure 1). To further verify the PCAP integrity, a public domain Python module created by Arash Javanmard (github.com/ArashJavan) was used to decode and extract raw point data. Each 360° frame is stored as a CSV in the project folder. The point data from the same frame as figure 1 is shown projected onto the x,y plane (figure 2). This new module has been incorporated into the project repository for use throughout the project. It may be modified to suit our needs if necessary.

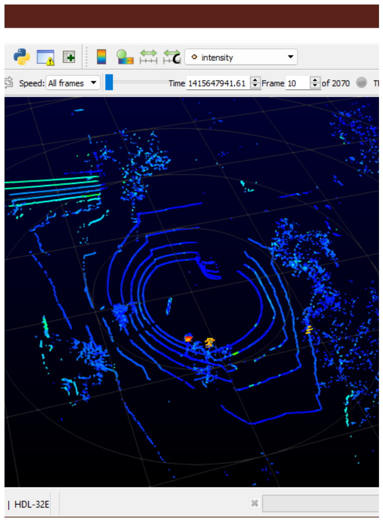

Figure 1: pcap file in VeloView.

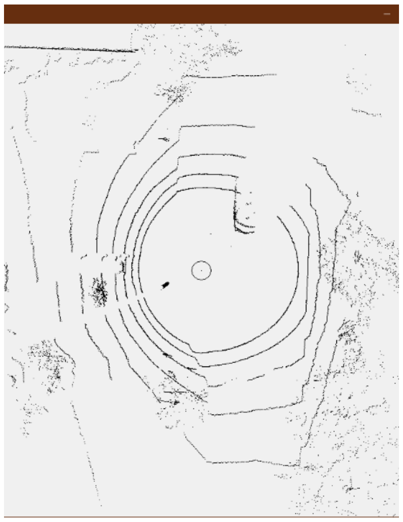

Figure 2: Bird's eye of raw points. 20px/m. 28000pts

Point Counts

The total number of points in a LiDAR frame causes some performance problems during visualization and will likely cause a time intensive process when overlaying points onto video data. This problem can be mitigated in two ways: using a clustering algorithm to create *N* new clusters or removing points at random. After testing several clustering algorithms, mini-batched K Means provided the best performance; approximately 5-6 seconds to generate 5000 clusters (new points). However, random point selection runs in a fraction of a second and will likely be sufficient for our needs. This can be revisited if necessary.

Notable Progress

With accurate point data broken out into frames, each frame can be used to map points onto a respective camera given the camera's relative location to the LiDAR device, its relative direction, its FOV, and its resolution.

Mapping LiDAR Point to Camera Pixels

Matrix transformations are used to map points in $\mathbb{R}^4$ vector space to pixel locations on a video frame:

$\begin{bmatrix} 1 & 0 & 0 & -x\text{ offset} \\ 0 & 1 & 0 & -y\text{ offset} \\ 0 & 0 & 1 & -z\text{ offset} \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Points are shifted to account for the camera's xyz offset to the LiDAR device. The camera now sits at $(0,0)$.
$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & -\cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Points are rotated to account for the direction the camera is facing with $\theta =$ angular offset clockwise from LiDAR. The camera is now facing in the direction of the y^+ axis

Points are checked to see if they're within the camera's FOV. Points with $y < 0$ (behind the camera) are discarded. Points with $\arctan\left(\left\lvert\frac{x}{y}\right\rvert\right) > FOV$ or $\arctan\left(\left\lvert\frac{x}{z}\right\rvert\right) > FOV$ (outside the FOV angle) are discarded.	
$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	The y and z axes are swapped to change from bird's eye to head on view. The camera is now facing in the direction of the z^+ axis
$\begin{bmatrix} {}^{1}/_{D} & 0 & 0 & 0 \\ 0 & -{}^{1}/_{D} & 0 & 0 \\ 0 & 0 & {}^{1}/_{D} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Points are normalized to $[-1,1]$ and the y values are reflected along the x axis to account for top-down video encoding. $D =$ maximum possible point distance, which is set when decoding PCAP data.
$x = \frac{x}{z}, \qquad y = \frac{y}{z}$	The x and y points are skewed based on distance (z) from the camera, effectively projecting them onto a normalized screen.
$\begin{bmatrix} S & 0 & 0 & 0 \\ 0 & S & 0 & 0 \\ 0 & 0 & D & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad S = \frac{R_x}{2\tan\left(\frac{FOV}{2}\right)}$ $\begin{bmatrix} 1 & 0 & 0 & {}^{R_x}/_{2} \\ 0 & 1 & 0 & {}^{R_y}/_{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	The points are normalized to the FOV of the camera ($\arctan(1,1) = \frac{FOV}{2}$), scaled to the camera's resolution, shifted to the correct pixel values, and saved in a Camera object.

Figure 3 shows test results of cameras at 0°,90°,180°, and 270° for the same frame shown in figures 1 and 2. The cameras are using a FOV of 60° and are positioned down 20cm and out 20cm from the LiDAR device. Given accurate camera data and positioning, this method can be used to generate points ready to be overlaid onto video. Additionally, these points retain all original LiDAR data.

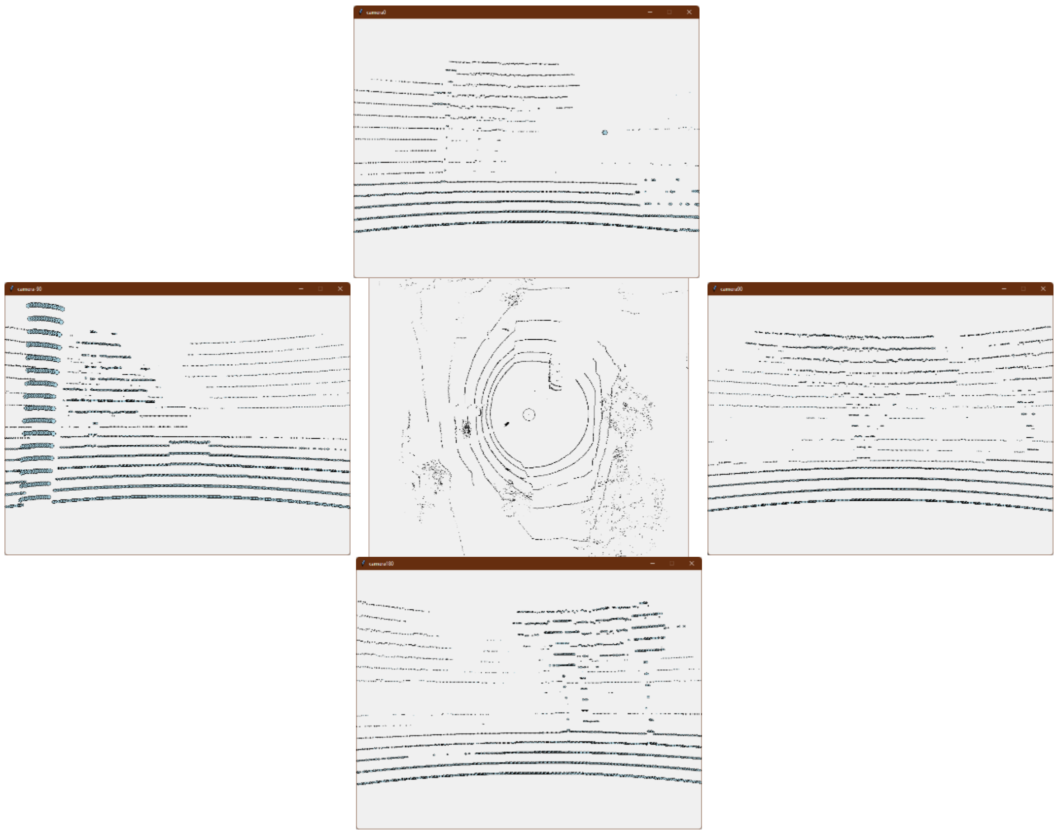

Figure 3: LiDAR points projected onto camera perspective.

New Modules

```
VelodyneManager(VLP type, pcap file, output folder, parameters)
       .run() -> Action: extracts pcap data and saves frames to output folder
       Parameters (yaml):
       gps-port: 8308
       data-port: 2368
       type: velodyneVLP16
       gps: False
       text: True
       ply: False
       from: 0  (first packet to read)
       to: 1000 (last packet to read)
       max points: 5000 (custom for this project)
       max distance: 15 (custom for this project)

Camera(x resolution, y resolution, rotation angle, FOV, xyz offset)
       Used by PointFilter object to transform points.  Can be extended to hold data from video as
       well, acting to stitch the two project aspects together.
       Member Variables:
       -camera_res_X: int (horizontal resolution)
       -camera_res_Y: int (vertical resolution)
       -camera_angle: int (rotation relative to LiDAR in degrees)
       -camera_fov: int (Field of view in degrees)
       -camera_offset: List(float) = [0, 0, 0] (offset relative to LiDAR in meters)
       -frames: List(array)

PointFilter(frames folder, camera list)
       ._import_frames() -> Returns frames imported from folder as a dict of numpy arrays with key =
       frame #
       ._get_camera_frames(camera) -> Translates points to match camera pixels.  Saves new frames to
       camera.frames member variable.
       Member Variables:
       -params: dict (from yaml) (Same params used by VelodyneManager)
       -point_data_path: str (path to files created by VelodyneManager)
       -cameras: List(Camera) (list of cameras)
       -frames: dict (key=frame #, value=numpy array of points)
```

Next Steps

Point data from the LiDAR device can now be overlaid onto video. Object tracking in the video processing side of the project can be used to create traffic objects with object classification and 2D positional data. Using this 2D positional data, the LiDAR distance data can be added using some basic collision detection. Clustering could also be used to eliminate accidental collision of outlier points. With a set of point collisions, object distance can be determined using point set mean or median. Additionally, if camera overlap exists, point collisions can be used across multiple cameras to track objects from one camera to another.

If the above doesn't work for inter-camera tracking, it may be possible to track objects from one camera to another by saving LiDAR points beyond each camera's FOV and using the retained metadata (i.e., point ID) for inter-camera tracking.

For the immediate future, focus should be placed on the video side of the project to prep video objects for LiDAR point injection. A frontend GUI should also be considered; It will likely be simple. If time permits, the VelodyneManager module could be better understood and modified to suit the needs of this project more closely.