VIP Lunabotics Senior Design Report

Nikita Ravi

***Abstract*— The goal of this senior design project is to support the Lunabotics club in their effort to develop an autonomous mining robot to compete in a NASA lunar-style mining competition. My contribution to making this goal one step closer to reality was to develop a pipeline that would transform a front-facing photo captured by the rover to a top-down view. This continuous transformation of photos would enable the development of a GPS mapping system. In addition to the pipeline, I implemented RGB-Depth reconstruction to provide the option of creating a custom dataset to train the pipeline on. Source code for this senior design is available at: `https://github.com/nikkiravi/VIP-Lunabotics-Software-Senior-Design`**

I. INTRODUCTION

This design project initially began with conducting extensive background research on the subject matter such as understanding what deep learning in computer vision entails, the different focus areas of computer vision: Object Detection, Object Tracking, Masking, Stereo Vision, and Noise Removal. In addition to the subject matter, I also studied different Graphing Algorithms like Dijkstras, A*, D* algorithms and as well as Sampling Based Algorithms like PRM, PRM*, RRT, RRT* algorithms to prepare for the possibility of switching from a computer vision focused project to a motion-planning project.

Once I completed the research phase, I found myself finding the computer vision (CV) aspect of robotics more interesting and therefore, decided to proceed with this focus area in my second semester of senior design with the objective of developing a pipeline that would transform a front-facing view to a top-down view of the lunar environment.

The implementation phase of the CV pipeline first began with finding two research papers on deep learning techniques for deblurring an image and its transformation from a front-facing view to a birds eye view (BeV) perspective. The deblurring research paper I decided to re-implement is called *DeblurGAN-v2: Deblurring (Orders-of-Magnitude) Faster and Better* [1]. The BeV transformation research paper I decided to re-implement is called *A Sim2Real Deep Learning Approach for the Transformation of Images from Multiple Vehicle-Mounted Cameras to a Semantically Segmented Image in Bird's Eye View* [2]. In order for the deblurred image to be passed as an input to the BeV deep learning network, the image first needs to be semantically segmented which is where training the detectron2 model comes in. Finally, I implemented RGB-Depth reconstruction to provide the option of creating a custom dataset to train the pipeline on.

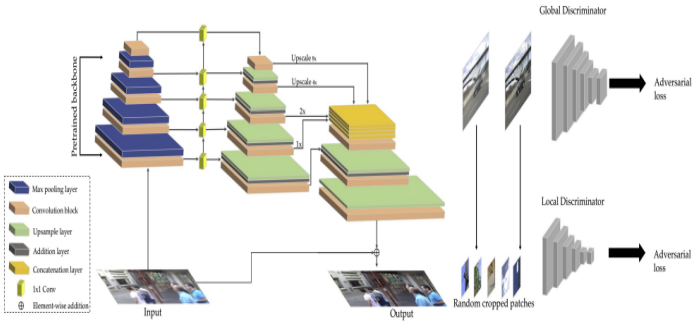

Figure 2: DeblurGAN-v2 pipeline architecture.

Fig. 1. Architecture of DeblurGAN-v2 [1]

II. IMPLEMENTATION OF DEBLURRING

A. Architecture of DeblurGAN-v2

1) Feature Pyramid Deblurring: Processing multiple scale images is time-consuming and memory demanding which is why the paper introduces Feature Pyramid Networks (FPN). The FPN module generates multiple feature map layers which encode different semantics and contain better quality of information [1]. FPN comprises a bottom-up and a top-down pathway where the bottom-up pathway is a CNN for feature extraction, with the spatial resolution being down-sampled which enables more semantic context information to be extracted and compressed [1]. Through the top-down pathway, the FPN reconstructs higher spatial resolution from the semantically rich layers [1]. The connections between the bottom-up and top-down pathways adds high-resolution details and help localize objects [1].

2) The Backbone: The FPN module is agnostic to the choice of feature extractor backbones. **Inception-ResNet-v2** was used to pursue a strong deblurring performance over **MobileNet V2** and **MobileNet-DSC** since we are not working with mobile-on-device image enhancement. The Inception-Resnet-v2 network has made great strides in image recognition performance due to it being a combination of the *inception* network with residual networks [3].

3) Double-Scale RaGAN-LS Discriminator: GAN stands for Generative Adversial Networks and it consists of two models: a discriminator *D* and a generator *G*, that form a two-player minimax game [1]. The generator learns to produce artificial samples and is trained to fool the discriminator, with the goal of capturing the real data distribution [1]. Since an ordinary GAN is difficult to optimize, the Least Square

GANs (LSGAN) discriminator introduced a loss function that provides a smoother and non-saturating gradient [1]. This log-type loss saturates quickly as it ignores the distance between x to the decision boundary [1]. On top of this, the paper adopts relativistic wrapping on the LSGAN cost function, thus creating **RaGAN-LS** loss. This allows the training to go much faster and the authors have empirically concluded that the generated results possess higher perceptual quality and overall sharper outputs [1].

4) The Training Algorithm: While training the networks on the blurred images, we need a metric that would assess how close the outputs of the network is to the original image. This metric or cost function is a combination of *pixel-space loss* L_p, *perceptual distance* L_x, and L_{Adv} containing both global and local discriminator losses, as shown below.

$$L_G = 0.5L_p + 0.006L_x + 0.01L_{Adv} \quad (1)$$

B. Training Datasets

For the purpose of training the DeblurGAN-v2 model, I used the **GoPro** dataset which uses the GoPro Hero 4 camera to capture 240 frames per second (fps) video sequences and generate blurred images through averaging consecutive short-exposure frames [1].

C. Implementation

The following table shows the parameters that were used to train the DeblurGAN-v2 network. The parameters are identical to the author's implementation to make an attempt at replicating their results.

Parameters	Value
Epochs	100
Learning Rate	0.01
Optimizer	Adam
Train Batch Size	1
Train Batch per Epoch	1000
Validation Batch per Epoch	100
Generator	FPN Inception
Discriminator	Double GAN

TABLE I
THE PARAMETERS USED TO TRAIN DEBLURGAN-V2

D. Results

Figure 2 shows the results of my training for the last few epochs out of which these are the best results:

- Best Validation G_{loss}: -1.3448
- Best Validation PSNR [1]: 24.6236
- Best Validation SSIM [2]: 0.8173

[1]Peak Signal to Noise Ratio is used for quality estimation for the loss of quality [4]

[2]Structural Similarity Index measures the similarity between two images [5]

Fig. 2. Deblur Training

III. IMPLEMENTATION OF DETECTRON2

Detectron2 was built and implemented in PyTorch by META for the purpose of Object Detection, Bounding-Box Detection, and Semantic Segmentation. For our purposes, in order to pass the deblurred input to the BeV network, Detectron2 was used for semantic segmentation. I used META's recommended Mask R-CNN model for semantic segmentation because it efficiently detects objects in an image while simultaneously generating a high-quality segmentation mask for each instance [6].

A. Training Datasets

For the purpose of training the Detectron2 model, I used the **Artificial Lunar Landscape** dataset which contains photorealistic images of the Moon's surface with ground truth rock segmentation [7]. For this project, the two features I decided to identify from the lunar surface and semantically segment are large rocks and small pebbles.

B. Implementation

The following table shows the parameters that were used to train the Detectron2 network.

Parameters	Value
Iterations	500
Learning Rate	0.00025
Classes	2
Threshold	0.3

TABLE II
THE PARAMETERS USED TO TRAIN DETECTRON2

C. Source Code

The following is a snippet of my source code for implementing detectron2 for the purposes of semantic segmentation.

```
classes = ["Pebbles", "Large Rock"]
cfg_path = r"cfg_model2.pickle"

def get_data_dicts(path, classes):
```

```
  dataset_dicts = []
  files = [file for file in os.listdir(path) if
    file.endswith(".json")]

  for idx, json_filename in enumerate(files):
    json_file = os.path.join(path, json_filename)
    with open(json_file) as fptr:
      img_annotations = json.load(fptr)

    record = {}
    image_filename = os.path.join(path,
    img_annotations["imagePath"])
    record["file_name"] = image_filename
    record["image_id"] = idx

    height, width = cv2.imread(image_filename).
    shape[:2]
    record["width"], record["height"] = width,
    height #720, 480

    annotations = img_annotations["shapes"]
    objs = []
    for ann in annotations:
      px = [a[0] for a in ann["points"]] # x-
    coordinate (top-left and bottom-right corners)
      py = [a[1] for a in ann["points"]] # y-
    coordinate (top-left and bottom-right corners)

      px = [px[0],px[1],px[0],px[1]]
      py = [py[0],py[0],py[1],py[1]]

      poly = [(x, y) for x, y in zip(px, py)]
      poly = [p for x in poly for p in x]

      obj = {"bbox": [np.min(px), np.min(py), np.
    max(px), np.max(py)],
            "bbox_mode": BoxMode.XYXY_ABS,
            "segmentation": [poly],
            "category_id": classes.index(ann['label'
    ]),
            "iscrowd": 0}

      objs.append(obj)

    record["annotations"] = objs
    dataset_dicts.append(record)
  return dataset_dicts

def set_up_model(max_iter=500, save=None):
  cfg = get_cfg()
  cfg.MODEL.DEVICE = 'cpu'
  cfg.merge_from_file(model_zoo.get_config_file("
    COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x
    .yaml"))
  cfg.DATASETS.TRAIN = ("category_train",)
  cfg.DATASETS.TEST = ()
  cfg.DATALOADER.NUM_WORKERS = 2
  cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url(
    "COCO-InstanceSegmentation/
    mask_rcnn_R_50_FPN_3x.yaml")
  cfg.SOLVER.IMS_PER_BATCH = 2
  cfg.SOLVER.BASE_LR = 0.00025
  cfg.SOLVER.MAX_ITER = max_iter
  cfg.MODEL.ROI_HEADS.NUM_CLASSES = 2

  if(save):
    with open(save, 'wb') as fptr:
      pickle.dump(cfg, fptr, protocol=pickle.
    HIGHEST_PROTOCOL)

  return cfg

def train_model(cfg):
  os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
  trainer = DefaultTrainer(cfg)
  trainer.resume_or_load(resume=False)
  trainer.train()

def semantic_segmentation(image, labels, masks):
  blank = np.zeros(image.shape)
  blank[:,:] = (139, 10, 80)
  color = None
  for label, mask in zip(labels, masks):
    if(not label):
      color = (255,0,0)
    else:
      color = (0,0,255)

    for row in range(len(mask)):
      for col in range(len(mask[row])):
        if(mask[row][col]):
          blank[row][col] = color
  return blank

def prediction(cfg_path, image_path, metadata,
    filename=None, bounding_box=False, thresh=0.3):
  with open(cfg_path, 'rb') as fptr:
    cfg = pickle.load(fptr)

  cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR,
    "model_final.pth")
  cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = thresh

  predictor = DefaultPredictor(cfg)
  image = cv2.imread(image_path)
  outputs = predictor(image)
  visualizer(image, outputs, metadata, filename=
    filename, bounding_box=bounding_box)
```

Listing 1. Detectron2 Source Code

D. Results

With a threshold of 0.3, the detectron2 network achieved an average precision of 95% with a training loss of 0.4077. Figures 3 - 6 are the outputs obtained from the detectron2 network.

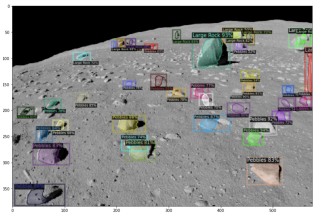

Fig. 3. Detectron2 Output Sample 1

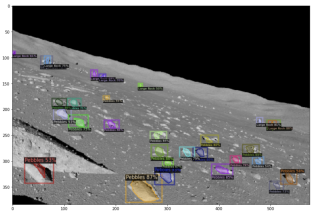

Fig. 4. Detectron2 Output Sample 2

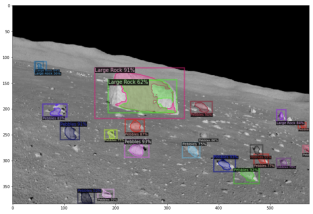

Fig. 5. Detectron2 Output Sample 3

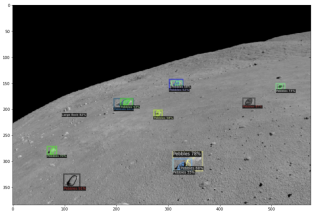

Fig. 6. Detectron2 Output Sample 4

IV. IMPLEMENTATION OF BEV NETWORK

A. Architecture of BeV Network

The paper proposes a neural network that processes all non-transformed images from a vehicle cameras as input.

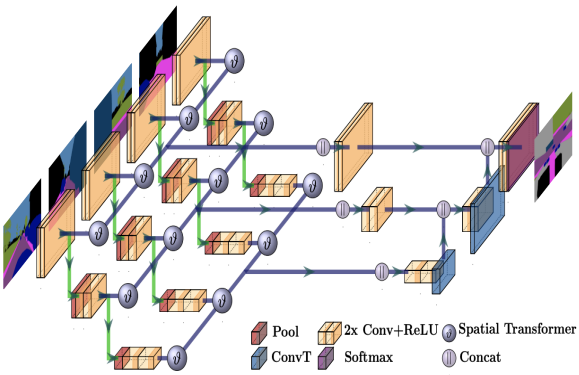

Fig. 7. Architecture of UnetXST [2]

It extracts features in the non-transformed camera views such that it is not subjected to the errors introduced by the IPM. To tackle the problem of spatial inconsistency, the paper integrates projective transformations into the network [2]. The architecture is designed such that it is capable of accepting multiple inputs for one output image. This is made possible by building on an existing network called *U-Net* [2].

The basic architecture as shown in figure 7 consists of a convolutional encoder and decoder path based on successive pooling and up-sampling, respectively [2]. In addition, high resolution features from the encoder side of the network are combined with up-sampled outputs on the decoder side via skip-connections on each scale [2]. Two extensions made to the *U-Net* network are:

1) The encoder path is separately replicated for each input image. For every scale, features from each input stream are concatenated and convoluted to build the skip connection to the single decoder path [2]
2) Before concatenating the input streams, *Spacial Transformer* units projectively transform the feature maps using the fixed homography as obtained by IPM [2]

These extensions are the reason for why the architecture is called *UnetXST* Network.

B. Training Datasets

The data used to train the *UnetXST* model to transform a front-facing image to a BeV perspective was created in the simulation environment Virtual Test Drive (VTD). In the simulation, the ego vehicle is equipped with a front-facing camera and the ground truth data is provided by a virtual drone camera centered above the ego vehicle. All virtual cameras produce both realistic and semantically segmented images [2].

C. Implementation

The table above shows the parameters that were used to train the UnetXST network. The parameters are identical to the author's implementation to make an attempt at replicating their results.

Parameters	Value
Epochs	100
Learning Rate	1e-4
Optimizer	Adam
Train Batch Size	2

TABLE III
THE PARAMETERS USED TO TRAIN UNETXST

D. Results

Figure 8 shows the results of my training for the last few epochs out of which these are the best results:

- Best Validation Loss: 0.1195
- Best Validation Categorical Accuracy: 0.9762
- Best Validation Mean IOU [3]: 0.8800

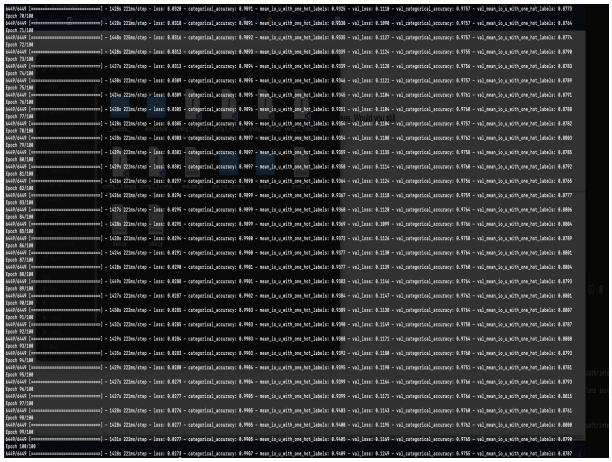

Fig. 8. UnetXST Training

V. CREATION OF THE PIPELINE

To combine the deblur, detectron2, and BeV networks and deploy them onto the lunar rover the club is currently building, I use an open source framework for robotic applications called ROS to get access to a continuous stream of images from the camera on the rover as my inputs to my networks.

A. Source Code

Below is the source code for the ROS deployment of the pipeline.

```
import rospy
import numpy as np
from sensor_msgs.msg import Image
import cv2
from deblurring.predict import Predictor
from detectron.detectron import prediction
from BeV.predict import predict
from cv_bridge import CvBridge

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

# Global Variables
bridge = CvBridge()
```

[3]Intersection or overlap between ground truth and result of network

```
publisher = rospy.Publisher("/cv_pipeline_result/", data_class=Image, queue_size=10)
result = None

def deblur(image):
    weights_path = r"/home/parallels/catkin_ws/src/cv/scripts/deblurring/best_fpn.h5"
    predictor = Predictor(weights_path=weights_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    deblurred = predictor(image, mask=None)
    deblurred = cv2.cvtColor(deblurred, cv2.COLOR_RGB2BGR)

    return deblurred

def segmentation(image):
    cfg_path = r"/home/parallels/catkin_ws/src/cv/scripts/detectron/cfg_model.pickle"
    meta_data = r"/home/parallels/catkin_ws/src/cv/scripts/detectron/output/category_test_coco_format.json"
    model_weights_path = r"/home/parallels/catkin_ws/src/cv/scripts/detectron/output/model_final.pth"

    segmented = prediction(cfg_path, model_weights_path, image, meta_data, thresh=0.3)

    return segmented

def birds_eye_view(image):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    config_file_path = r"/home/parallels/catkin_ws/src/cv/scripts/BeV/config.2_F.unetxst.yml"
    model_weight_path = r"/home/parallels/catkin_ws/src/cv/scripts/BeV/best_weights.hdf5"

    BeV_transformed = predict(image, config_file_path, model_weight_path)
    return BeV_transformed

def cv_pipeline(image, blur=False):
    rospy.loginfo("Image is loaded")
    cv2.imwrite("original.png", image)

    if(blur):
        # To purposefully blur an image for testing purposes
        image = cv2.GaussianBlur(image, ksize=(3,3), sigmaX=1.3)
        rospy.loginfo("Image is blurred")
        cv2.imwrite("Blurred.png", image)

    deblurred = deblur(image)
    rospy.loginfo("Image is deblurred")
    cv2.imwrite("deblurred.png", deblurred)

    segmented = segmentation(deblurred)
    rospy.loginfo("Image is segmented")
    cv2.imwrite("segmented.png", segmented)

    BeV_transformed = birds_eye_view(segmented)
    rospy.loginfo("Image transformed to BeV")
    cv2.imwrite("BeV.png", BeV_transformed)

    return BeV_transformed


def callback(message):
    image = bridge.imgmsg_to_cv2(message, desired_encoding='bgr8') # shape: 480 x 640 x 3
    image = cv2.imread("/home/parallels/catkin_ws/src/cv/scripts/render0909.png")
    result = cv_pipeline(image)
    result = bridge.cv2_to_imgmsg(result, encoding="passthrough")

    publisher.publish(result)
    rospy.loginfo("Image Published")


if __name__ == "__main__":
    rospy.init_node("ros_wrapper", anonymous=True)
    rospy.loginfo("Node has been initialized")

    rospy.Subscriber("/camera/color/image_raw", data_class=Image, callback=callback)
    rospy.loginfo("Node started") # simply keeps python from exiting until this node is stopped

    rospy.spin()
```

Listing 2. The Pipeline

B. Results

By passing a blurred image of the lunar surface through the pipeline, the results are as shown in figure 9

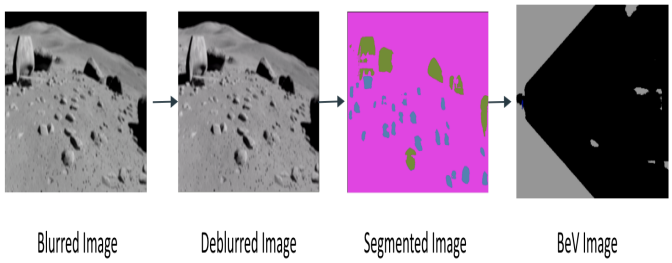

Fig. 9. Output of the Pipeline

VI. IMPLEMENTATION OF RGB-DEPTH RECONSTRUCTION

The last section of my senior design was to implement RGB-Depth reconstruction in order to provide the stakeholders of the option of creating a custom dataset to train the pipeline on.

A. Camera Calibration

Before I start implementing RGB-Depth reconstruction, it was vital to know what the **extrinsic** and **intrinsic** parameters of the camera is. The camera I used for this senior design is the one preferred by the club: Intel Realsense D455 camera.

The extrinsic and intrinsic parameters of a camera are transformation matrices that convert points from one coordinate system to the other [8]. The extrinsic matrix is a transformation matrix from the world coordinate system to the camera coordinate system, while the intrinsic matrix is a transformation matrix that converts points from the camera coordinate system to the pixel coordinate system [8].

The extrinsic and intrinsic parameters of the camera are obtained through *Zhangs'* Algorithm as shown in a snippet of my source code below

```python
### Global Variables
bridge = CvBridge()
CHECKERBOARD = (6, 9) # Dimensions of the Checkerboard
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001) # Termination Criteria

def get_corners(image, gray, objp, display=False):
    # Find the chessboard corners
    # If desired number of corners are found in the image then ret = true
    # cv2.CALIB_CB_ADAPTIVE_THRESH = Use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness)
    # cv2.CALIB_CB_FAST_CHECK = Run a fast check on the image that looks for chessboard corners, and shortcut the call if none is found. This can drastically speed up the call in the degenerate condition when no chessboard is observed.
    # cv2.CALIB_CB_NORMALIZE_IMAGE = Normalize the image gamma with equalizeHist before applying fixed or adaptive thresholding.
    ret, corners = cv2.findChessboardCorners(gray, patternSize=CHECKERBOARD, flags=cv2.CALIB_CB_ADAPTIVE_THRESH + cv2.CALIB_CB_FAST_CHECK + cv2.CALIB_CB_NORMALIZE_IMAGE)

    # If desired number of corners are detected, we refine the pixel coordinates and display them on the checkerboard
    if(ret):
        corners2 = cv2.cornerSubPix(gray, corners=corners, winSize=(11, 11), zeroZone=(-1, -1), criteria=criteria) # refining pixel coordinates for given 2d points
        if(display):
            display_corners(image, corners2, ret)

        return objp, corners2

    else:
        return None, None

def get_instrinsic_extrinsic_parameters(gray, obj_points, img_points):
    """
    Performing camera calibration by
    passing the value of known 3D points (objpoints)
    and corresponding pixel coordinates of the
    detected corners (imgpoints)
    """
    # K = Intrinsic Matrix
    # dist = Lens distortion coefficients. These coefficients will be explained in a future post.
    # R = Rotation Rodrigues Vector
    # t = Translation Vector

    ret, K, dist, rvecs, tvecs = cv2.calibrateCamera(obj_points, img_points, gray.shape[::-1], None, None)
    R, jacobian = cv2.Rodrigues(rvecs[0])
    t = tvecs[0]
    return K, R, t
```

Listing 3. Camera Parameters

B. Implementation

Once we get the camera parameters, I implement RGB-Depth reconstruction by creating a pointcloud by storing the depth value at each pixel representation of the 2d RGB image obtained from the camera attached on the lunar rover via ROS as shown in the following snippet of the code

```python
def get_camera_parameters(self):
    parameters = np.load(r"/home/parallels/catkin_ws/src/cv/scripts/RGBd/parameters.npz")
    self.K, self.R, self.t = np.array(parameters["K"]), np.array(parameters["R"]), np.array(parameters["t"])

    self.K = np.column_stack((self.K, np.array([0,0,0])))
    self.K = np.row_stack((self.K, np.array([0,0,0,1])))

    self.Rt = np.column_stack((self.R, self.t))
    self.Rt = np.row_stack((self.Rt, np.array([0,0,0,1])))

def display_point_cloud(self, display=False):
    self.get_camera_parameters()
    fx, cx, fy, cy = self.K[0,0], self.K[0,2], self.K[1,1], self.K[1,2]

    self.rgbd_image = o3d.geometry.RGBDImage.create_from_color_and_depth(self.geometry_rgb, self.geometry_depth, convert_rgb_to_intensity=False)
    height, width = self.rgb_image.shape[:2]
    point_cloud = o3d.geometry.PointCloud.create_from_rgbd_image(self.rgbd_image, o3d.camera.PinholeCameraIntrinsic(width=width, height=height, fx=fx, fy=fy, cx=cx, cy=cy))

    # Flip it, otherwise the pointcloud will be upside down
    point_cloud.transform([[1, 0, 0, 0], [0, -1, 0, 0], [0, 0, -1, 0], [0, 0, 0, 1]])

    if(display):
        o3d.visualization.draw_geometries([point_cloud])

    return point_cloud
```

Listing 4. RGB Depth Reconstruction

C. Results

Figure 10 illustrates the output of the RGB-Depth reconstruction. The result manages to capture the 3D essence from just an RGB image captured by the rover.

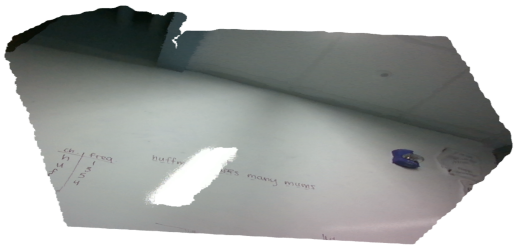

Fig. 10. Sample of RGB-Depth Result

VII. CONCLUSION

Through this senior design, I was able to successfully produce a prototype to transform a front-facing image to its BeV perspective which is essential when creating a localization map of the rover with respect to its surroundings. By successfully creating a prototype, I was able to assist the Lunabotics team in getting one step closer to making the rover autonomous which is a very significant metric in the NASA competition.

Some next steps for future teams would be to continue making improvements upon this prototype by gaining access to more datasets to train the networks on so that their validation metric scores could increase, by identifying craters on the surface of the moon during semantic segmentation, and finally using the outputs of the pipeline to create a localization system using SLAM.

VIII. ACKNOWLEDGEMENTS

I would like to thank Professor Nicole Ramirez and Graduate Teaching Assistant Prabhpreet Dhir for without whom I would not have been able to complete this senior design project.

REFERENCES

[1] Orest Kupyn, Tetiana Martyniuk, Junru Wu, Zhangyang Wang (2019) *DeblurGAN-v2: Deblurring (Orders-of-Magnitude) Faster and Better*

[2] Lennart Reiher, Bastian Lampe, Lutz Eckstein (2020) *A Sim2Real Deep Learning Approach for the Transformation of Images from Multiple Vehicle-Mounted Cameras to a Semantically Segmented Image in Bird's Eye View*

[3] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke (2016) *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*

[4] Christopher Thomas (2020) *Deep learning image enhancement insights on loss function engineering*

[5] Pranjal Datta (2020) *All about Structural Similarity Index (SSIM): Theory + Code in PyTorch*

[6] Kaiming He, Georgia Gkioxari, Piotr Dollar, Ross Girshick (2018) *Mask R-CNN*

[7] Romain Pessia (2018) *Artificial Lunar Landscape Dataset*

[8] Aqeel Anwar (02/28) *What are Intrinsic and Extrinsic Camera Parameters in Computer Vision?*