


Blackboard - Advancing Autonomous Target Detection and Mapping: Development of Integrated Software Systems for Monocular Depth Sensing and Dynamic Target Labeling

Brandon C. Colelough, 

School of Engineering and Information Technology, University of New South Wales, Australia

Abstract—This study aimed to implement a system capable of autonomously detecting sheep targets and representing them on a 2D occupancy map, with the ultimate goal of facilitating autonomous shepherding on a UxV platform. This paper details the development of the Blackboard system, a software solution designed for autonomous target detection and mapping. Employing Python and C programming languages, the Blackboard system integrates monocular depth sensing with autonomous target detection to produce comprehensive depth and target maps. These maps are merged to generate a detailed 2D bird’s-eye view of the operational area, captured from an elevated camera perspective. A distinctive feature of the Blackboard system is its modular framework, which allows for seamless updates or replacements of its depth sensing and target detection modules.

I. INTRODUCTION

The primary objective of this research was to develop an integrated target mapping software that amalgamates spatial target data from one module with depth information from another. This integration facilitates the generation of a comprehensive 2D representation, termed the "Blackboard view," of targets within a specified area. This view is derived from data captured by an elevated camera feed. The core software of this project, named "Blackboard," was specifically engineered to enable efficient interaction between two distinct artificial intelligence modules, thereby achieving the intended mapping task. In terms of depth perception, the project utilized a commercial off-the-shelf (COTS) solution that incorporates the "High-Quality Monocular Depth Estimation via Transfer Learning" model, developed by Alhashim and Wonka [1]. Furthermore, the YOLO package [2] was utilised for object detection and the incorporation of these two packages into the Blackboard solution is a key aspect of the research, playing a vital role in enhancing the overall functionality and efficacy of the target mapping system.

II. LITERATURE REVIEW

A. Monocular Depth Sensing

Most common hardware-intensive depth sensing modules will utilise a stereo ocular setup (dual camera) or some sort of Lidar/radar system to accurately sense depth. More software-intensive modules that utilise a singular camera measure the translational and rotational data between different frames

of a video stream to predict depth. The monocular (single camera) depth sensing module used takes an entirely different approach to predicting depth. The High-Quality Monocular Depth Estimation via Transfer Learning developed by Ibraheem Alhashim and Peter Wonka utilises two CNNs in an encoder-decoder setup to leverage transfer learning for depth predictions [1]. This depth sensing model was written in Python and as such the encoder and truncated decoder shown was an implementation of a dense-net 169 CNN with regular skip connections. The model is trained with single frames as the input and ground truth data (from any reputable source e.g. radar, Kinect IR etc.). The model that was used as a module in the Blackboard program developed was trained on generic, readily available data found in the KITTI and NYU V2 depth data set. As such, this data was not optimised for the detection of sheep in a field. This led to inaccuracies in the depth of data obtained. However, as the main objective of this project was the generation of software packages these inaccuracies did not affect the overall success of the project. As shown in figure 1, this depth estimator uses CNNs with a structure similar to that used with image processing done by the target detection neural net. Here It can be seen that the dense depth team has made use of transfer learning by taking image classification encoders found in Python and utilising them in the generation of depth maps. A thought for improving the overall efficiency of the network found here would be to take the algorithm used by the dense depth team and modify the structure of a YOLO CNN to enable the generation of depth maps with the C-based YOLO module. This would be a good idea to explore in another project. This may however prove easier said than done as the dense depth module utilises libraries specific to Python. The structure and form in which the dense-net 169 CNN has been used are much the same as in the image processing techniques from which the CNN has been transferred and learned. The main difference is found within the loss function of the network. As stated by [1], the loss function is given by

$$L(y, \hat{y}) = \lambda L_{depth}(y, \hat{y}) + L_{grad}(y, \hat{y}) + L_{SSIM}(y, \hat{y}) \quad (1)$$

Where y is the ground truth depth map and \hat{y} is the prediction depth map, L_{depth} gives the difference of their depth values,

L_{grad} gives the difference of the gradient for their depth values and L_{SSIM} gives the structural similarity of the ground truth and prediction depth maps. Explanations for each of these terms can be found in [1].

This loss function is greatly contrasted to a regular CNN loss function such as that used in the YOLO framework given by (Mean squared error):

$$L(y, \hat{y}) = MSE = \frac{1}{n} \sum_{i=1}^n (y - \hat{y})^2 \quad (2)$$

Whilst the basic structure of the CNN is kept the same, some of its most basic processes (such as the loss function) have been altered in the process of transfer learning to allow this conventional image processing CNN to instead produce depth maps.

B. Languages Used - C, C++, Python, Cython

C is a low-level, general-purpose programming language that can be used on a wide variety of systems. C has been used to write operating systems (Windows, MAC/iOS - objective C, Linux etc.) interpreters (such as the Python interpreter) and various applications. C++ is an extension of the C language. C++ is a superset of C. C++ can run most of C code while C cannot run C++ code. C++ is an object-orientated programming language and as such supports such features as polymorphism, encapsulation and inheritance. C is a procedural language only and as such does not support any of these features. C++ also provides exception handling and memory allocation operators whereas C does not. More differences are listed here [8]. A C or C++ compiler will be able to generate bytecode directly executable on micro-controllers and microprocessors and as such is a good choice for any project looking to embed their program onto a simple system such as the drone being used for autonomous sheep herding. Python is an object-orientated, high-level programming level. Python is an interpreted language in that it needs a program to go from a high-level language to a form that a compiler can understand. Most Python interpreters are written in C (as most operating systems are written in C or a subsidiary of C) but there are other interpreter options such as Iron Python (based on the .NET framework) Jython (based on the Java platform) etc. The CPython (based on C) interpreter is the most efficient as C is the most universal language and therefore an interpreter based on this language will be most useful on a range of operating systems. Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language (based on Pyrex) [5]. Cython was designed to allow the user to call C or C++ code from Python natively and interact with C or C++ code from the Python interface. In this project, the Cython API along with the Python interpreter was taken and used to call Python modules from C and C++ code. pyx header files and C++ structures were used to interact over the two languages to create a bridge to pass data from a Python module back to a C++ module to be handed back to a C module.

III. METHODOLOGY

The objective of this research was to develop an integrated software system capable of determining the position of targets in three-dimensional space and rendering these positions onto a two-dimensional "bird's eye view" Blackboard. This system integrates a target detection module with a monocular depth sensing module. The YOLO framework [2] was employed for target detection due to its previous optimization for specific target types [9]. Concurrently, the dense-depth package was chosen for monocular depth sensing [1] to generate depth prediction heat maps utilising only a single frame as an input to the module's neural network. The blackboard system was developed for integration with autonomous drones, aligning with the broader mission objectives. The architecture was designed with modularity in mind, allowing for the depth-sensing and target-detection components to be replaced or upgraded as needed.

A. Target Detection

For target detection, the YOLO framework was utilized. Modifications were made to the detector, image, and demo modules of YOLO's source code to enable the transfer of detection data. These alterations were confined to the framework's higher-level functionalities, without changing the core structure or the target detection capabilities of the YOLO CNN.

B. Monocular Depth Sensing

The high-quality monocular depth estimation package by [1] was implemented for depth sensing. This package utilises transfer learning and an encoder-decoder architecture to produce depth estimation heat maps of an image. A Python-based inpainting method for depth estimation was developed and integrated into the package. Due to the absence of a target-specific depth dataset, a combined dataset from NYU and KITTI, which closely matched the environmental and target parameters, was used for initial training. Examples of the depth maps produced are presented in figure 2.

C. An Integrated System

The integrated system initialises both the models for the C-written YOLO package and Python Python-written dense-depth package. The primary Blackboard module will load the models for these components into memory to quickly access them. The primary Blackboard module will then process a given image through the detection module passing through the C model and receiving back the target positional data. The primary Blackboard module will then process the same given image through the depth sensing module passing through the Python model and receiving back a depth map in the form of a 4D depth array. Once the target data and depth data have been received, the main Blackboard module will then determine the target's depth based on its position in the depth map. This information is then processed to allow for the targets to be drawn. The target was then drawn onto the blackboard

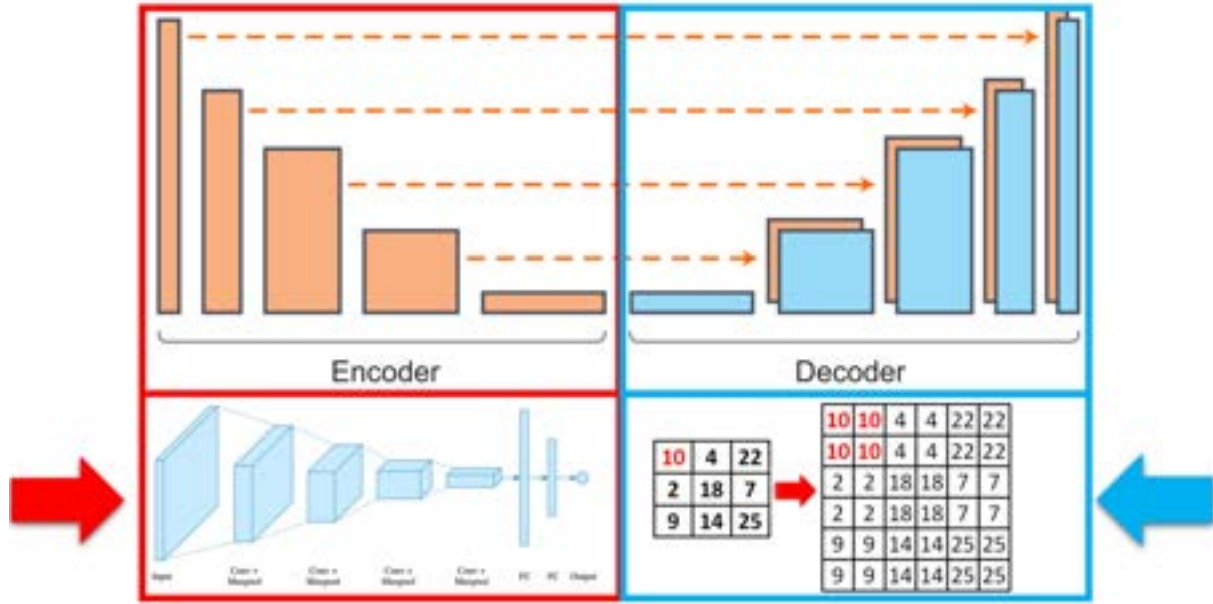


Fig. 1: The encoder-decoder setup used within this model. Pictured in red is the CNN used as an encoder. The decoder is a truncated version of the same CNN. Pictured in blue is an example of the bi-linear up sampling used in conjunction with the truncated CNN decoder.



Fig. 2: Shown on the right is the input. Shown on the left is the associated depth map processed by the depth sensing module

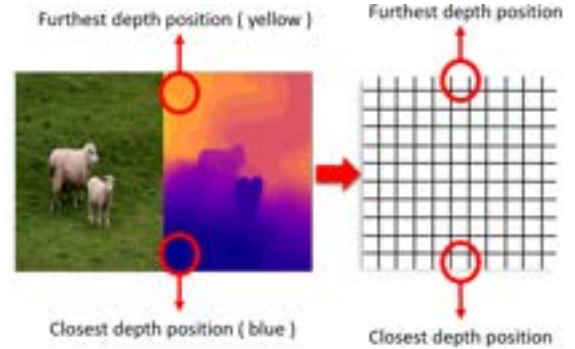


Fig. 3: Setting a frame of reference for depth

occupancy grid by finding the centre of the target through the following function:

$$X_{mid} = \frac{X_{left} + T_{width}}{2} \quad (3)$$

$$Y_{mid} = \frac{Y_{top} + T_{height}}{2} \quad (4)$$

Note that X_{left} , Y_{top} , T_{width} and T_{height} were all given by the target detection module. The depth was then read from the 4D array using these centre positions of the targets. The normalisation of the depth concerning the upper and lower bounds of the depth map as well as the window in which the frame is being drawn was done by the equation:

$$DN = \frac{(BW_{max} - BW_{min}) \times (TD - D_{min})}{D_{max} - D_{min}} + BW_{min} \quad (5)$$

where

- DN is depth normalized,
- BW_{max} is board width maximum,
- BW_{min} is board width minimum,
- TD is target depth,
- D_{min} is depth minimum, and
- D_{max} is depth maximum.

depthNorm and X_{mid} were then taken as the centre of the target on the Blackboard. The size of the targets was then normalised by the use of the `get_target_dims` function. This function would fit a found target into one of three categories, small, medium or large. The original iteration of this function used only the special dimensions of the targets and the target depth to determine what size the target should present on the Blackboard. It was apparent however that this would not functionally with the depth estimations being provided by the dense-depth module. The depth estimation given by the depth

module usually falls between 1-2 metres. This was most likely a result of depth truth data from the training data sets being collected by a Microsoft Kinect device which has a maximum range of 3.5m. To overcome this issue, an updated depth estimate was refined which made use of the size of each target seen on the frame to get a relative size ratio of each target. The height and depth values of each target were collated from the monocular depth sensing module. These estimates were then sorted into an array and the standard deviation of the data set was then determined. If the difference between each target and the median target was greater than the standard deviation then the target was assigned a small value. Similarly, if the difference was less than the negative of the standard deviation then that target was assigned a large value. The maximum and minimum depth values were then used to set a frame of reference for drawing in targets with relation to their y-coordinate with seen depth. The depth value array was traversed to determine the minimum and maximum distances produced by a depth map. A visual representation of how this was used as a frame of reference is shown in Figure 3.

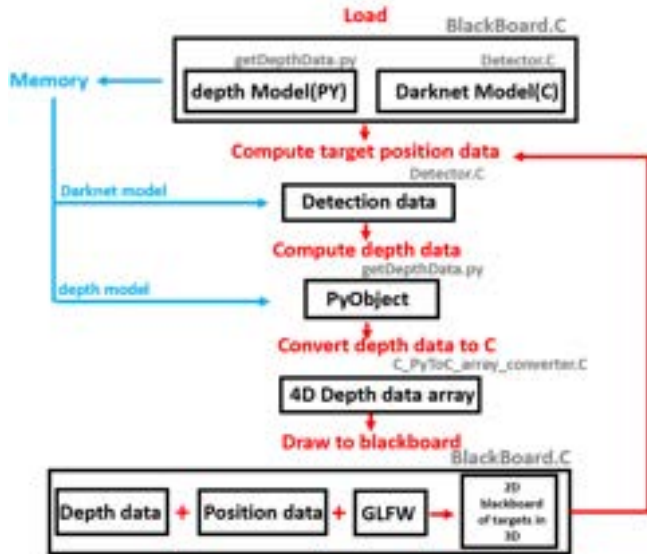


Fig. 4: The high-level function is highlighted in red and the data being processed is delineated within distinct boxes. Additionally, the specific module in use is identified in grey text. The final box in the sequence represents the amalgamation of required data to achieve the intended output

D. Using Cython to integrate C with Python modules

The majority of the modules written for the main project were to allow for the integration of the Python dense-depth package to integrate with the main module of the main package as well as the YOLO framework. Since the depth sensing module used was written in Python and the main module and YOLO were written in C a bridging program was needed to allow for the transfer of data from Python to C and C to Python. Figure 4 shows the overarching design of the system and refers to this integration of languages. The intricacies of these modules which allowed for the bridging between Python

and C are discussed below in the modules and functionality section. Note that the loading of both models only occurs once and each model can be accessed at any time from memory.

IV. RESULTS

Figures 5-12 show the full system operations across a range of tests. On the left, the object detection output from the YOLO package is displayed. Next, in the centre, a dense-depth depth map prediction is shown for the same input image. Lastly, on the right is shown the Blackboard "bird eye view" drawing of targets on a 2D occupancy grid with the depth values produced by a dense-depth depth map shown to the side. Table 1 shows The average time complexity of the blackboard submodules. It can be observed in Table 1 that the modules written in C are (of course) much faster than the dense-depth modules written in Python. Of significance is that the Python dense depth model takes 2.5 seconds to make a prediction and the Py-C bridging code takes 1 second to convert the 4D depth prediction data.

TABLE I: Largest time complexities in main software package

Modules	time complexity(ms)
drawBoard	49.8502
load_model(DenseDepth)	18636.84
getDepthData(DenseDepth)	2578.8
load_model(YOLO)	2531.9
YOLO detector	22
getDepthDataV2(C)	3521.093
Blackboard (total/Frame)	3754.12

V. DISCUSSION

A. Time complexity

As observed in Table 1, loading the dense-depth module takes almost 9x as long as loading the YOLO target detection module. Whilst this is interesting to note it is unimportant as the model for both only has to be loaded once as can be seen in figure ?? . Whilst an extended loading time on startup is not ideal it does not affect how effective the overall system is. What is important is the processing time seen thereafter. The processing time of both C modules is shown to be roughly 50 milliseconds for drawing the targets on Blackboard and roughly 22 milliseconds for target detection processing completed by YOLO. This puts the overall processing speed for both C modules at roughly 70 milliseconds per frame. I use the term roughly here as the processing speed is affected by other contributing factors such as background processes being handled by the CPU etc. so it is difficult to obtain an exact time on processing speed without a larger sample set to draw upon. The majority of the processing speed for a frame is however shown to be the dense-depth estimation and data conversion process. The python dense-depth module takes over 2.5 seconds to process a single frame into a depth map and the data conversion process takes over a second. This puts the overall time to process a single frame from a still image to a Blackboard of targets at over 3.5 seconds. It can therefore be



Fig. 5: Shown here is a good example of how the program can function given a semi-accurate depth map. The depth map produced a relatively good reading for the target which was about half of the range of the frame. This paired with the size normalisation produced a good representation of where the target should be placed.



Fig. 6: It can be observed in this depth map that the targets have skewed the gradient of the image depth. This caused the target depth estimation to come up short causing the target position to be more forward than it should.

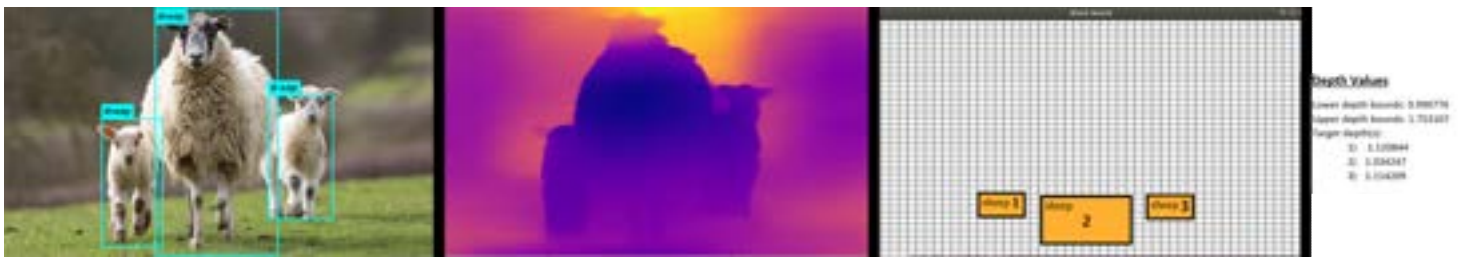


Fig. 7: The dense-depth module has picked up here that the targets are in the foreground. The resulting Blackboard frame is a good representation of where the targets would be given this image. It can also be seen that the size-normalisation module has correctly differentiated between the sizes of the three targets.

seen that the limiting factor for data processing is the python dense-depth module. Even with multi-threading processes for target detection and depth map estimation the processing time for a single frame would still be over 3.5 seconds.

B. Depth estimation accuracy

At current the depth estimation model consistently pushes the depth estimation for targets into the foreground. Most prominent in figure 9 and figure 10 but seen in all depth maps produced, it can be observed that the targets are outlined and brought directly to the front. This is due to the network not yet being trained on sheep data and is most likely caused by the colour gradient differentiation between targets and background colour information. The dense-depth model is much more accurate at characterising the depth gradient for background information. This is observed in figures 8-11 where the background has been effectively characterised. Figure 12 shows another good differentiation between near and far on the background information depth map but is disrupted by a target cluster in the top middle section. This target cluster

disrupts the depth map estimation and places a large patch of foreground where it should not be. It can therefore be observed that the dense-depth model works well for estimating the depth map for the background information but does not effectively characterise the depth of targets.

C. Current depth estimation limitations

It was observed that the depth range always fell between 0.9 and 2.5 meters. This was due to the training information used to train the dense-depth module for depth estimation. A combination of the KITTI and NYU V2 data sets was used for depth estimation training. These data sets relied on ground truth boxes collected by a Microsoft Kinect. The maximum effective range of a Microsoft Kinect is 3.5 meters so the upper limit of the depth estimation for this model will be 3.5 meters.

D. Target size normalisation

Shown well in figure 7 is an example of target size differentiation. This figure shows two small targets on either side of the



Fig. 8: The depth map has placed targets 2,3 and 4 too close. The gradient of the foreground is mapped out well though and places the rest of the targets in a relatively correct position.

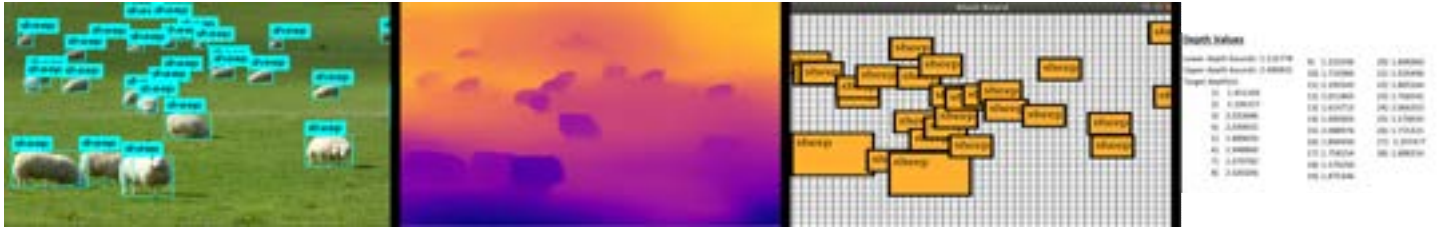


Fig. 9: The dense-depth module has misinterpreted the targets in this image for foreground gradients. This resulted in many of the target depth estimations being too close causing local clustering in the foreground as opposed to the background where the clustering should occur. The size normalisation module has also incorrectly determined the relative size of two of the targets at the front left of the image.

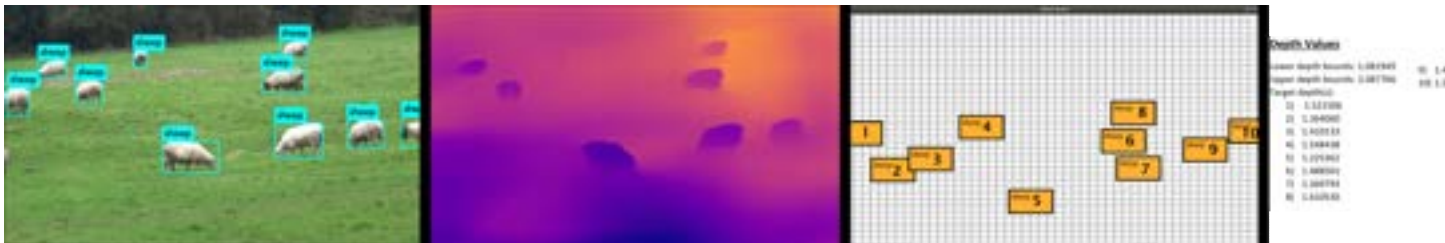


Fig. 10: The dense-depth module has again misinterpreted the rapid colour change brought upon by the targets as the foreground. This pushed all the detected targets into the foreground.



Fig. 11: The dense-depth module has much more effectively characterised the depth map of this image. A local cluster of sheep can be seen in the top right corner as well as a sparse cluster in the bottom left corner.

large target which is also observed in the original image shown to the left of the Blackboard. Figures 8, 10 and 11 show good target size normalisation with many targets. It is shown here that all the targets are actually the same size as they should be. It can however be observed that there are still bugs in the size normalisation module as shown in figures 9 and 12. Figure 9 mischaracterises two targets at the front of the frame and figure 12 mischaracterises the size of a target that is half cut out of the frame. The error in figure 9 is most likely due to the inaccuracy of in-depth data which was a contributing factor in

the size normalisation algorithm and the error in figure 12 is due to a function added that tries to account for the positioning of a target (front or side view).

E. The Blackboard System in use

The Blackboard package processes a frame producing both target information and a depth map. The YOLO module produces high precision information whilst the depth-sense module provides low precision. This information is then sent to the Blackboard program and is used to draw targets. Figure

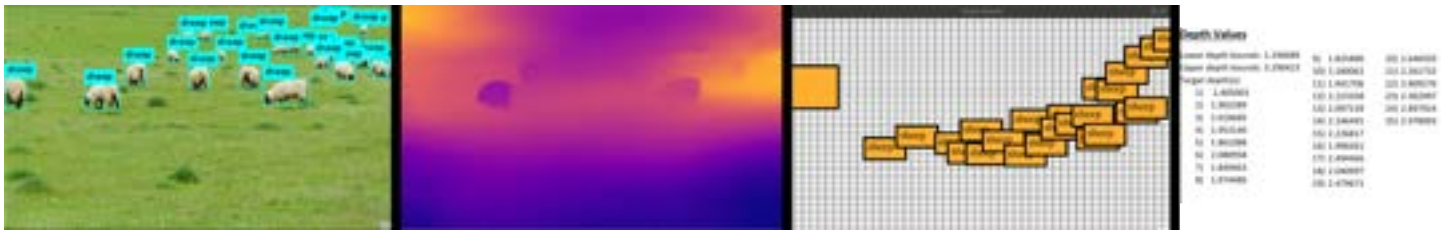


Fig. 12: The dense-depth module has incorrectly characterised the top-middle sector of this image as foreground. This has pushed all of the targets detected close to the middle closer than they should be.

9 shows well that the x and y coordinates for the middle of a target are accurately represented. Here it is seen that the target is shown to the far left of the screen which is represented in the Blackboard printout as well. The target is also observed at about half the depth range which is also accurately represented on the Blackboard printout. Extrapolating this further to many targets, figure 11 shows well that the program functions accurately and gives a good representation of targets when the depth map is semi-accurate. Local clustering of targets can be seen in the top right and bottom left quadrants. The size normalisation of targets is also shown well here.

VI. DISCUSSION

A. Time complexity

At current the program works well and transitions smoothly to produce a Blackboard rendition of a static frame/image. For this to be extended to a non-static reference like a video, major changes in the program will need to be made. For a video stream to seem continuous and smooth it needs to be presented at a minimum of 20 fps [7]. The YOLO framework for example can produce detections at 45 fps. The time needed by the main software package to produce one single Blackboard printout is 3.5 seconds. As discussed earlier the limiting factor for this processing time is the creation of depth maps from the dense-depth module. I believe this is due to the Python program interfacing with C modules. The Python interpreter is needed to do this and as such the program is slow. This could be optimised by Cythonising the Python modules further to try and convert them directly to C. It is my opinion that this will not work well however as the AI Python models used are Python-specific and as such trying to directly convert them to C will most likely cause performance issues further down the track. The dense depth model utilises two CNNs to make predictions. If the YOLO CNNs were retrofitted for transfer learning to enable them to be used as the CNNs for the depth map production process then the processing time of depth map production from a C module should be roughly 40 milliseconds. If the Blackboard module can be optimised to allow no longer than 5 milliseconds of processing time and multi-threading was used to enable the target detection and depth map predictions to occur at the same time then the time complexity of the overall program could be brought down to 49 milliseconds (depth map production as limiting factor and Blackboard program processing time). This would allow the main software package to process video feed at 20 fps. Whilst this is still not ideal it is much better than the 0.3 fps the

current package works at. Therefore, my recommendation is to rewrite the dense-depth algorithm in C using the YOLO CNNs as a basis for the module. This would also interface with the main program much better.

B. Depth estimation

The depth estimations observed by the dense-depth module were of poor accuracy as they were not yet trained on the specified targets (sheep). We do not however believe that taking the current method of collecting a data set will be effective for training this model though. Whilst the 3.5-meter range of the Kinect is more than effective for the autonomous car environment the data sets were collected for, this is not suitable for the depth target estimation environment it is now needed. The targets can be regularly observed at much further than 3.5 meters and as such a new method of collecting depth truth information is needed. A new device will need to be created that can perform the functionality of the Microsoft Kinect at distances much larger than 3.5 meters. The limiting factor for range with target detection is the camera quality used. The target detection CNN can be scaled up to take a very high-quality image. A high-quality image will show more accurate targets and as such the YOLO framework will be able to more accurately detect targets at a larger distance with higher-quality images. The limiting factor for the range of depth estimation is however the data set it is trained on. If the maximum range of the data set is 3.5 meters then the maximum depth estimation will be 3.5 meters (as was observed). Therefore, an overall limiting factor for the range of this program will be the range of the data set which tracks back to the range of the collection device used to obtain the ground truth data for the target range used for the training of the depth estimation model. This should be taken into account when designing the device that will be used for collecting this data set.

VII. CONCLUSION

In this research, an integrated software system was developed, capable of processing an elevated static feed for target detection and depth assessment. The system then utilized this information to construct a 2D "bird's eye view" of identified targets on a Blackboard module. While the software successfully met all functional requirements, it was observed that the time complexity and depth accuracy metrics did not align with the desired standards. Moving forward, we recommend

a strategic revision of the dense-depth model by transitioning its development to C. This modification could enhance the efficiency and accuracy of the depth-sensing component. Additionally, careful consideration should be given to the selection of the device used to collect ground truth data for the training dataset of the depth estimation network, as this has significant implications for the overall accuracy of the model. The software also included a feature allowing users to interact with predictions made by a trained YOLO target detection model on previously unseen images. This functionality enabled users to save these predictions or manually draw bounding boxes around targets. The intent behind this feature was to facilitate the creation of a dynamic labelling program, significantly reducing the time required for labelling images in the training of YOLO models. This aspect of the software presents a promising direction for future enhancements, particularly in improving the efficiency of model training processes.

REFERENCES

- [1] Alhashim, A. Wonka, P. (2018). *High Quality Monocular Depth Estimation via Transfer Learning*. Saudi Arabia: KAUST. Available at: <https://arxiv.org/pdf/1812.11941.pdf> (Accessed: 20 July 2019).
- [2] Redmon, J. Farhadi, A. (2018) *YOLOv3: An Incremental Improvement*. Washington: University of Washington. Available at: <https://pjreddie.com/media/files/papers/YOLOv3.pdf> (Accessed: 14 April 2019).
- [3] Alhashim, A. Wonka, P. (2019). *High Quality Monocular Depth Estimation via Transfer Learning GitHub Code*. Saudi Arabia: KAUST. Available at: <https://github.com/ialhashim/DenseDepth> (Accessed: 20 July 2019).
- [4] Wojna, Z. Ferrari, V. Guadarrama, S. Silberman, N. Chen, L.Fathi, S. Uijlings, J. (2019). *The Devil is in the Decoder: Classification, Regression and GANs*. <https://arxiv.org/pdf/1707.05847.pdf>(Accessed: 20 July 2019).
- [5] *Cython Documentation*. Available at: <https://cython.org/> (Accessed: 20 July 2019).
- [6] Krishan, K. (2016). *Difference between C and C++*. Available at: <http://cs-fundamentals.com/tech-interview/c/difference-between-c-and-cpp.php> (Accessed: 20 July 2019).
- [7] CARD, S. K., MORAN, T. P. AND NEWELL, A. (1983). *The psychology of human-computer interaction* (Accessed: 25 October 2019).
- [8] Krishan, K. (2016). *Difference between C and C++*. Available at: <http://cs-fundamentals.com/tech-interview/c/difference-between-c-and-cpp.php> (Accessed: 20 July 2019).
- [9] Colelough, B. (2019). *Optimizing Data Diversity for Accurate Prediction Models: Insights from a Multi-Class Sheep Image Data-Set Analysis towards Autonomous Shepherding*.