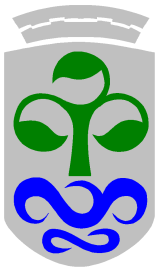


UNIVERSIDADE
DE VIGO

ESCOLA SUPERIOR DE ENXEÑARÍA INFORMÁTICA

Memoria do Traballo de Fin de Máster que presenta

Rodrigo González Linares

para a obtención do Título de Máster en Intelixencia Artificial

Sistema de vixilancia baseado na imaxe dixital e intelixencia artificial para a detección de aves no contorno de vertiportos

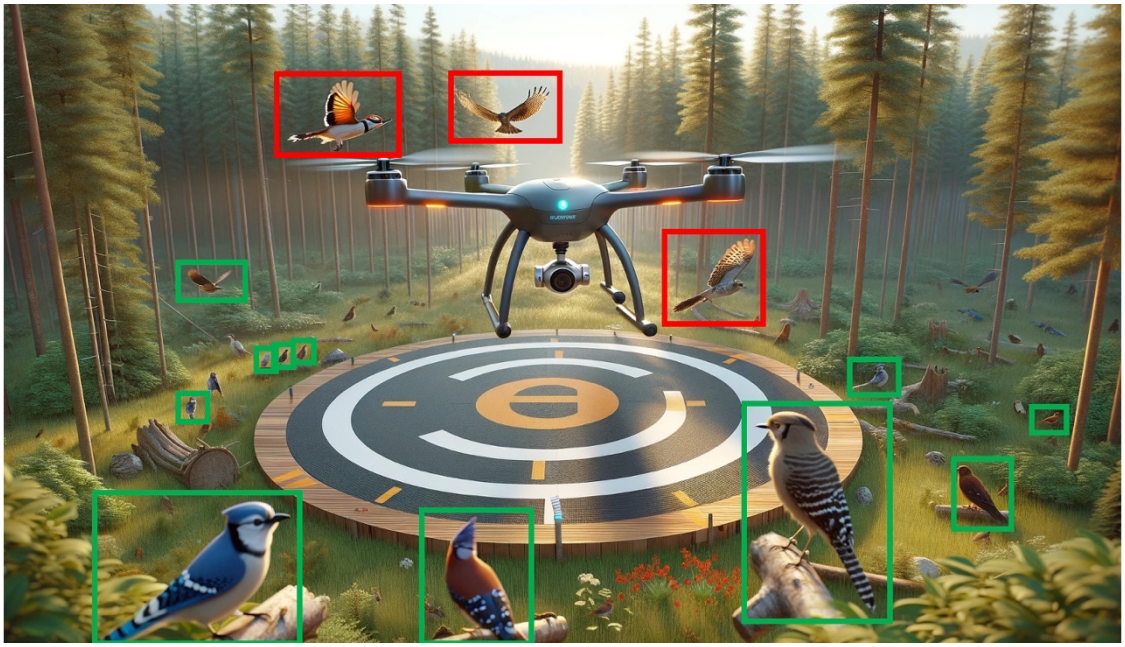
Febreiro, 2024

Traballo de Fin de Máster Nº: ASFJOBB

Titor/a: David N. Olivieri Cecchi (U. de Vigo), Higinio González Jorge (U. de Vigo), Patricia M. López de Frutos (CRIDA)
Área de coñecemento:
Ciencias da Computación e Intelixencia Artificial
Departamento: Informática

Artificial Intelligence and digital image-based monitoring system for the detection of birds in vertiport environments

By
Rodrigo González Linares



Supervisors:
David N. Olivieri Cecchi – University of Vigo
Higinio González Jorge – University of Vigo
Patricia M. López de Frutos – CRIDA

Abstract

Object detection, classification, and estimation of position are three fundamental tasks in the realm of computer vision. The development of modern deep learning architectures like convolutional neural networks and vision transformers have been instrumental in creating models capable of performing these tasks accurately and addressing important applications. Here we focus on one of such applications; a monitoring system capable of identifying and estimating the position of birds (the main hazard to vertical takeoff and landing aircraft) in three-dimensional space on vertiport environments. To accomplish this under a simple single-camera system, a pipeline including MiDaS for monocular depth estimation, YOLO for detection and classification, and SAM (or alternatively also YOLO) for an optional segmentation step was constructed. Moreover MiDair, a fine-tuned MiDaS model, was created to better estimate the depths of flying objects, whose correct localization tends to be the most challenging for the network.

Resumo

A detección, clasificación e estimación de posición de obxectos son tres tarefas fundamentais no ámbito da visión artificial. O desenvolvemento de arquitecturas modernas de aprendizaxe profunda, como redes neuronais convolucionais e transformadores de visión, foi fundamental na creación de modelos capaces de realizar estas tarefas con precisión e abordar aplicacións importantes. Aquí centrámonos nunha desas aplicacións; un sistema de vixilancia capaz de identificar e estimar a posición das aves (principal perigo para aeronaves de despegue e aterraxe vertical) no espazo tridimensional no contorno de vertiportos. Para lograr isto usando un sistema simple consistente dunha soa cámara, construíuse un fluxo de procesamento que inclúe MiDaS para a estimación de profundidade monocular, YOLO para detección e clasificación, e SAM (ou alternativamente tamén YOLO) para un paso de segmentación opcional. Ademais, MiDair, un modelo MiDaS reentrenado, foi creado para estimar mellor as profundidades dos obxectos voadores, cuxa correcta localización tende a ser a máis desafiante para a rede.

Resumen

La detección, clasificación y estimación de posición de objetos son tres tareas fundamentales en el ámbito de la visión artificial. El desarrollo de arquitecturas modernas de aprendizaje profundo, como redes neuronales convolucionales y transformadores de visión, ha sido fundamental para la creación de modelos capaces de realizar estas tareas con precisión y abordar aplicaciones importantes. Aquí nos centramos en una de esas aplicaciones; un sistema de vigilancia capaz de identificar y estimar la posición de aves (principal peligro para las aeronaves de despegue y aterrizaje vertical) en el espacio tridimensional en entornos de vertipuertos. Para lograr esto usando un sistema simple consistente de una sola cámara, se construyó un flujo de procesamiento que incluye MiDaS para la estimación de profundidad monocular, YOLO para detección y clasificación, y SAM (o alternativamente también YOLO) para un paso de segmentación opcional. Además, MiDair, un modelo MiDaS reentrenado, fue creado para estimar mejor las profundidades de los objetos voladores, cuya correcta localización tiende a ser la más desafiante para la red.

Table of contents

1. Introduction

1.1. Context and objectives

In the past few years, computer vision (a branch of informatics concerned with image understanding and object recognition) has experienced tremendous success across a broad set of tasks that were previously considered hard. This has been brought about largely by deep learning (DL), superseding many traditional algorithms.

A particular subfield that has recently garnered significant attention is object recognition, primarily due to its intimate relationship with autonomous navigation. While this has been popularized by projects like Tesla's Autopilot and Alphabet's Waymo, the reliance on object recognition for navigation is not exclusive to cars. Many sorts of aircraft, for example, increasingly utilize this technology, where the ability to accurately identify and classify objects in real-time is crucial for ensuring safe and efficient operation in dynamic environments.

In what respect aircraft (either fully autonomous or otherwise), not only are onboard systems relevant for navigation, but also air traffic control systems from ground are important to manage their takeoff and landing. While onboard systems play a crucial role in guiding aircraft during flight, the coordination and oversight provided by ground-based air traffic control systems are equally indispensable, especially during takeoff and landing maneuvers. These ground systems are essential for orchestrating the smooth flow of air traffic, guaranteeing the safety of departures and arrivals, and facilitating seamless transitions between different airspace sectors. Relating to this, and for the purposes of this thesis, it is fundamental to introduce the concept of vertiport.

The European Union Aviation Safety Agency (EASA) defines a vertiport as "an area of land, water, or structure used or intended to be used for the landing and takeoff of VTOL [vertical takeoff and landing] aircraft". [1]

According to the EASA, VTOL aircraft are a broad category of vehicles capable of vertical takeoff and making use of more than two propulsion units. Therefore, traditional quadcopter drones, as well as some manned vehicles as so-called Air Taxis fit the definition.

Different VTOL aircraft have distinct takeoff profiles. While quadcopters can trace a fully vertical takeoff trajectory, the Air Taxis are only capable of achieving a fully vertical trajectory for the first stage of takeoff, which is immediately followed by a diagonal trajectory.

To avoid any unintended collisions during takeoff, an obstacle free volume is defined over the vertiport. Whereas the obstacle free volume for a vertiport intended for quadcopters could extend only above the vertiport area, other VTOLs aircraft call for distinctly shaped volumes. For Air Taxis, for example, a funnel-shaped volume has been proposed to accommodate for its takeoff profile.

While vertiports are positioned in such a way to maintain the obstacle free volume of static objects like buildings or trees, one needs to constantly monitor this volume for dynamic objects like birds and other drones using the space. This project aims to tackle exactly this, the creation of an automatic monitoring system for dynamic objects on vertiport environments.

The system should be able to detect, classify, and position objects in three-dimensional space. Birds being the main hazard for takeoff, the focus will be specifically centered on them, although the scope of the system could be extended with relative ease to include other potentially hazardous objects like drones.

The accomplishment of the task at hand necessitates leveraging state-of-the-art deep neural networks, specifically convolutional neural networks (CNNs) and vision transformers (ViTs). The upcoming subsection outlines the fundamental working principles of these architectures, laying the groundwork for subsequent discussions on specific models. These models will form the cornerstone of the air traffic monitoring system, comprising three essential modules: detection and classification, segmentation, and depth estimation.

As it can be seen in figure 1, YOLO (a CNN) is used in both the detection and classification module and in the segmentation module, SAM (a ViT) will be included in the segmentation module exclusively, and MiDaS (another ViT) in the depth estimation module.

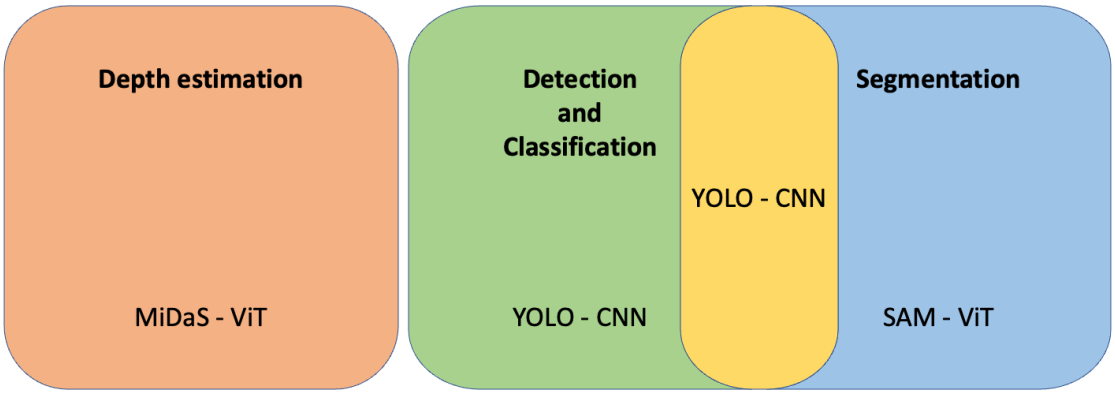


Figure 1. Modules implemented to solve the problem of bird positioning in a vertiport environment.
The modules, along with the models used for its implementation, and the type of DL architecture they are based upon.

1.2. Deep learning architectures for computer vision

While many computer vision tasks can be tackled using classical computational tools, today's most advanced systems make use of DL.

How relatively shallow neural networks (NNs) can aid in such kinds of tasks has been famously exemplified using the Modified National Institute of Standards and Technology (MNIST) dataset. MNIST is composed of a series of annotated black and white images of hand-written digits (0 through 9). The objective is then to predict the correct digit label from the image. This problem is easily solved with a simple sequential neural network, and as such has become the standard textbook introductory example, or "Hello World" of machine learning.

Thus, the traditional approach to solve this problem is by training a multilayer perceptron (MLP); a NN composed of an input layer, a few hidden layers, and an output layer. All layers in this simple architecture are fully connected (that is, each neuron is connected to every neuron in the previous layer).

To feed the image into this type of network, it must first be flattened (i.e., convert it from a 2-dimensional to a 1-dimensional array), losing some spatial information in the process. Moreover, these networks are generally unable to account for significant scale variance and translation of objects. Despite this, after training, the network is typically capable of achieving high classification accuracies due to the relative simplicity of the task.

CNNs were introduced to overcome this loss of spatial information. This type of network preserves the spatial structure of the image by using a series of convolutional layers instead of flattening the image into a one-dimensional array. These layers apply a set of learnable filters (or kernels) to the input, capturing local features such as edges and textures in the early layers, and more complex patterns like shapes or specific objects in deeper layers. A convolution is nothing more than a sliding multiplication between a section of an array and a kernel. Additionally, pooling layers are commonly intercalated between convolutional layers. Pooling operations reduce the size of an array by dividing it into sections and keeping only one unique value per section; typically the maximum (max pooling) or the average (average pooling). A graphical depiction of these two operations can be seen in figure 2-A and B, respectively.

CNNs saw great success in a myriad of different computer vision tasks such as image classification, object detection, segmentation and it is commonly applied to address problems regarding medical image analysis, autonomous driving, and video monitoring, among others. A CNN representation used for a classification problem can be seen in figure 2-C.

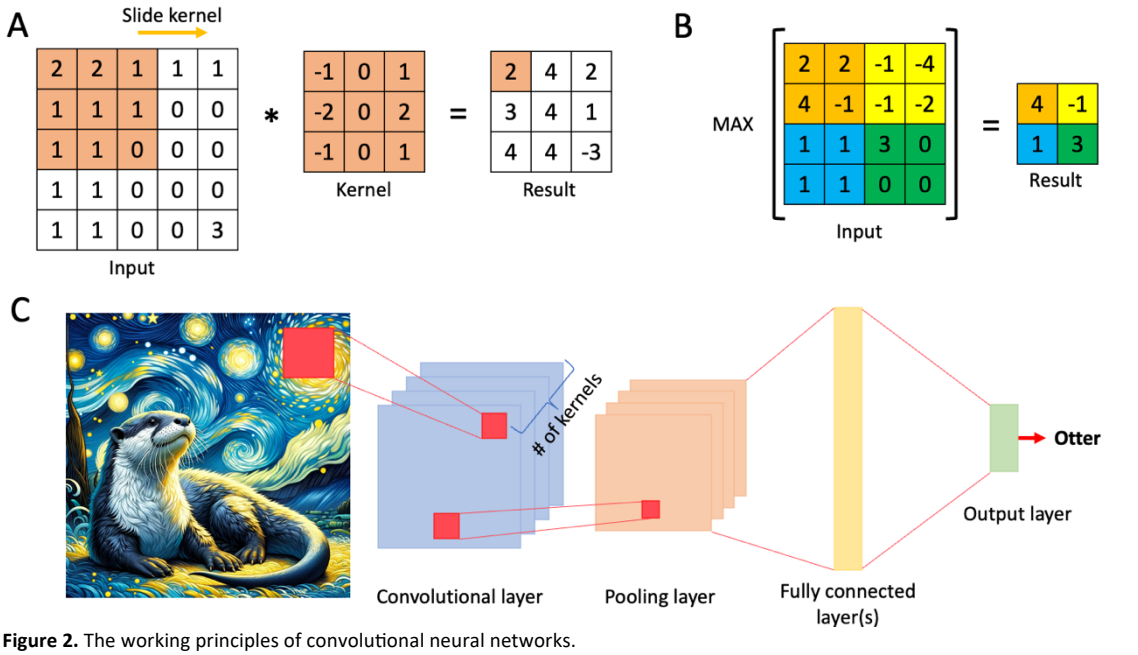


Figure 2. The working principles of convolutional neural networks.
A. A convolution is an operation where a kernel slides over an input array. When the kernel is over a section of the array, their inner product is calculated and stored as an entry in the resulting array. The region of the input array in pale orange, when operated with the kernel, produces the highlighted entry in the resulting array. The rest of entries are produced by sliding the kernel one cell at a time (i.e., with a stride of 1 for this particular example) to the right and downwards. **B.** Pooling, and more concretely max pooling in this case, is calculated by dividing an array into sections and selecting the maximum value on each section. The results are then collected into the resulting array of smaller size than the input array. **C.** Basic architecture of a CNN used for classification. A convolutional layer applies a set of kernels over an image, thereby extracting high level features from it. A pooling layer is commonly added after a convolutional one to reduce the dimensionality of the representation. Although in this case one convolutional and one pooling layer are shown, commonly many more of these layers are added, each time extracting lower level features. For classification

purposes, the output of the CNN is flattened and passed through fully connected and an output layer, in charge of producing the final prediction.

In 2017, a seemingly niche article addressing language translation introduced a novel architecture that soon took the whole field of AI by storm, even beyond the confines of natural language processing. “Attention is all you need” by Google researchers introduced the transformer model. [2]

The key innovation in the transformer architecture is the attention mechanism, which allows the model to focus on different parts of an input sequence. Attention involves three types of vectors: queries, keys, and values. Each token (i.e., a vector representation of a word, a sub-word, punctuation mark, etc.) in the input sequence broadcasts one of these vectors. A query (Q; of size d_K, for an N-word input) represents how much attention the token being currently processed should pay to other tokens in the input data. Keys represent all elements in the input data and are used along with the query to assess their relevance to each other. All keys can be collected into a key matrix (K) of size $N \times d_K$. The multiplication of Q and K generates a matrix of weights that then needs to be further multiplied by a vector of values (V), of size $N \times d_V$, which by itself contains the actual (unweighted) information the model will pass to subsequent layers. The multiplication of Q and K is scaled by $\sqrt{d_K}$ so that each row has unit variance, and a softmax is applied to convert it to a probability distribution where each row sums to 1, in the following manner:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_K}}\right)V$$

So far, as described here, because the information is not processed sequentially (i.e., attention values are calculated for each token independently of the others) a sense of position is lacking. This is where positional encodings come into play. In reality, a Q vector is constructed by summing a raw query vector containing information about the word itself (e.g., vectors representing related words would be similar to each other) with a positional encoding vector of the same dimension representing its position within a sequence.

Although the original transformer had both encoder and decoder blocks, some applications do not require the use of both. Text generation models like Generative Pretrained Transformer (GPT) only require a decoder, while language understanding models like Bidirectional Encoder Representations from Transformers (BERT) need only an encoder. This latter type of transformer model is of special interest to us, as it has been applied to computer vision in the form of ViTs. Introduced in 2021, also by Google researchers with “An image is worth 16x16 words: transformers for image recognition at scale”, it has quickly been adopted by many state-of-the-art computer vision models. [3]

As it can be seen in figure 3, ViTs use a “bag of patches” approach for image processing. Unlike CNNs, which process an image as a grid of pixels, ViTs first divides the image into a sequence of smaller image patches. Each of these patches is then transformed into a token by flattening. Hereafter the tokens are passed through a linear embedding layer

to create token embeddings, and positional embeddings are summed. One would expect the two-dimensional position of the patches would be essential for understanding the scene represented in an input image. Curiously enough, the article introducing this type of network found that one-dimensional encoding (left to right, top to bottom) worked better than two-dimensional and relative positional encodings. The embeddings are then passed through an encoder-only transformer followed by a set of fully connected, and a classification output layer.

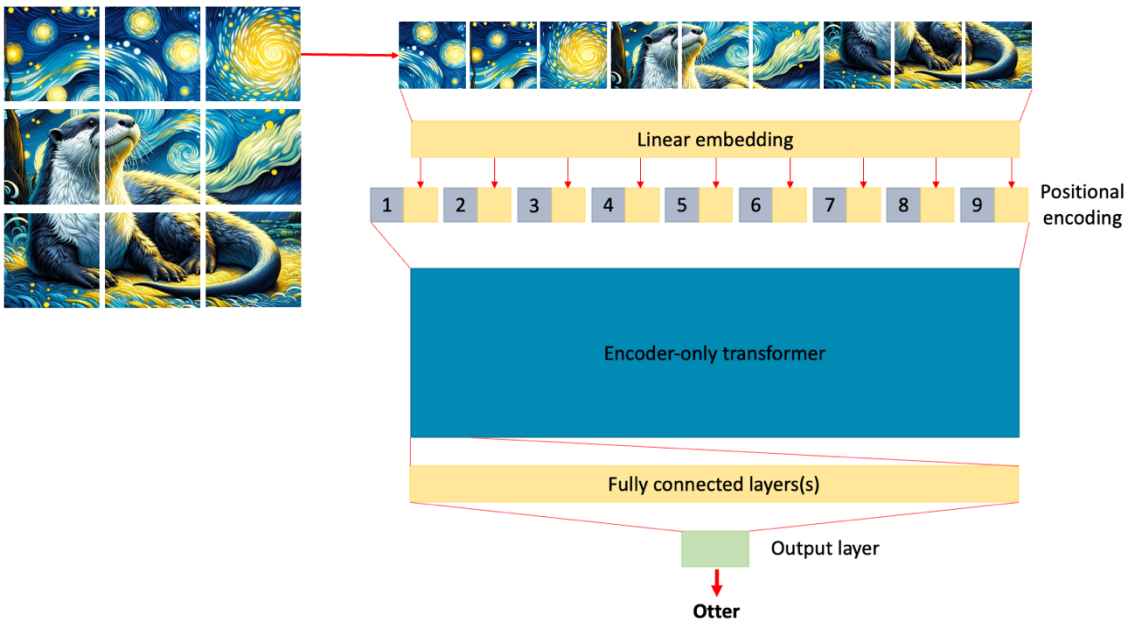


Figure 3. Basic architecture of a ViT used for classification.
An image is divided into patches and embedded via a linear embedding layer. The positional embeddings are summed to the image embeddings in a process known as positional encoding. The embeddings are then passed through an encoder-only transformer, followed by fully connected layers and an output layer to produce a final classification result.

1.3. State-of-the-art models for object detection, classification, and segmentation

In 2015 a network coined You Only Look Once (YOLO) was introduced, instantly becoming the most popular model regarding object detection and classification due to its performance. [4]

While prior methods first aimed at identifying parts of the image where objects might be located using one NN, and later a second classifier NN to label such parts, YOLO took a different approach by doing both at once with a single NN (hence the name).

As shown in figure 4, the model achieves this by dividing the image into a grid (typically $S \times S$). Each cell within this grid predicts B bounding boxes, whose center lies within the grid cell. While the bounding boxes can occupy areas outside the limits of the grid cell, only the information from such grid cell is used to construct the box, composed of an x and y coordinate, along with its width (w) and height (h), and a prediction confidence; in total five values per box. During training this confide is defined as $Pr(object) * IOU_{Pred}^{Truth}$, where $Pr(Object)$ is the probability a bounding box contains an object and IOU_{Pred}^{Truth} is the intersection over union (a measure of overlap) between the predicted

and ground truth bounding boxes. The model also predicts C conditional class probabilities per cell corresponding to each of the i categories of objects the model is capable of predicting. Finally, class specific confidence scores are calculated as follows:

$$Pr(class_i \vee object) * Pr(object) * IOU_{Pred}^{Truth} = Pr(class_i) * IOU_{Pred}^{Truth}$$

The final predictions are therefore in the shape of a $S \times S \times (B * 5 + C)$ tensor.

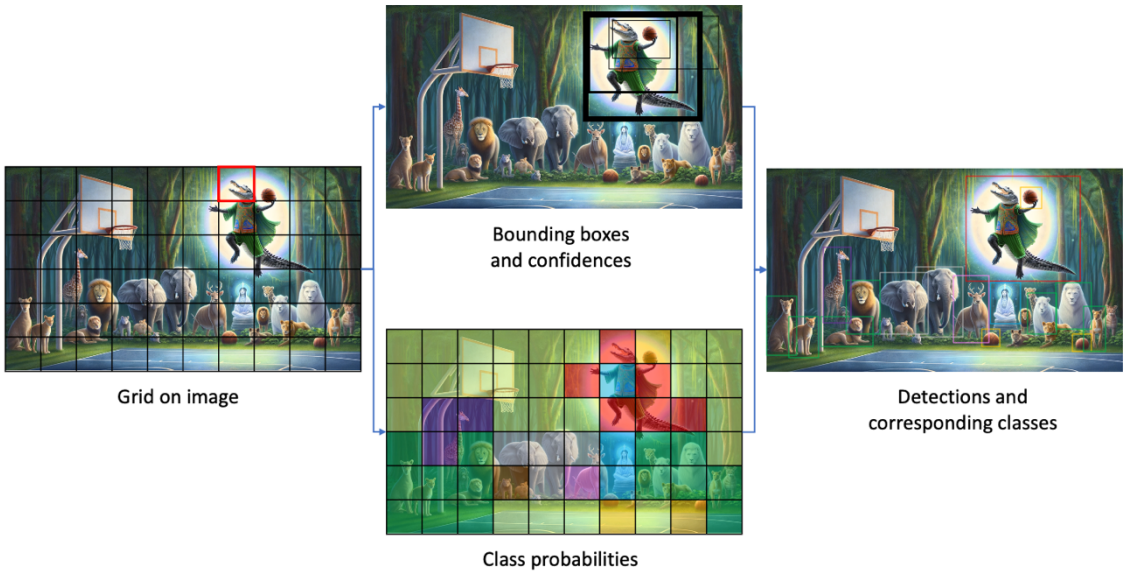


Figure 4. Working principles of YOLO models.
An image is divided into cells with grid. The YOLO model predicts B bounding boxes per cell, each with a certain confidence. A higher confidence is denoted by a thicker, while a lower confidence is denoted by a thinner line. The model also predicts C class probabilities per cell. The most probable classes are denoted by the coloring of the cells (e.g., red for crocodile, blue for human, orange for ball, etc.). Finally, bounding boxes over a certain confidence threshold are counted as detections and draw with the color of the corresponding class probability. To avoid clutter we focus on the grid cell highlighted in red and show its corresponding bounding boxes only for it.

YOLO is a convolutional model, and although further elaborations of the YOLO architecture have introduced some performance-boosting innovations, this continues to be true to this day.

Two families of YOLO models were unveiled in 2023, and are now regarded as the state-of-the-art. The first is YOLOv8, developed by a company called Ultralytics. They offer several models suitable for different tasks: classification, detection, segmentation, pose estimation, and tracking. While classification models only return a list of objects predicted to be present in the image, detection models can additionally draw bounding boxes over the objects, and on top of that, segmentation models are able to draw a segmentation mask over them. Instead of a segmentation mask, pose estimation models can extract key points from the objects and how they connect to each other. All models (except classification ones) are capable of tracking objects throughout video frames.

While, for example, segmentation models are also capable of performing detection, the precision for this task tends to be inferior to that of the equivalent size detection model.

Regarding the number of parameters, the models come in five sizes for different precision-inference speed needs.

The other type of model is YOLO-NAS (Neural Architecture Search), developed by Deci. NAS, first introduced in 2016, refers to a family of algorithms aimed at finding optimal architectures from a predetermined search space using reinforcement learning. [5]

For the creation of YOLO-NAS, Deci researchers used a proprietary NAS algorithm; Automated Neural Architecture Construction (AutoNAC).

There exist YOLO-NAS models for detection and pose estimation, each available in three sizes. These models introduce some innovations to the base YOLO architecture such as attention and quantization-aware convolutional blocks. The latter makes it possible to quantize the models from FP16 precision to INT8 for significantly faster inference for only a very minor hit in precision.

Introduced by Meta in 2023, the aptly named Segment Anything Model (SAM) made significant progress in performing image segmentation tasks, outperforming other existing models. [6]

SAM is a “promptable” model. This means that along with an image, one can include a point, a bounding box, or even text. When a point is included, SAM will try to segment the pointed object and, when a bounding box is included, it will do so for the main object within the box. Text can provide extra context to aid in the segmentation process, while when passing only an image with no additional context SAM will try to identify as many objects as it can and segment them all.

The three components that compose SAM are a simple prompt encoder, a ViT image decoder, and a transformer mask decoder that takes information from both encoders and produces masks foreground probabilities for each pixel in the image. These probabilities can then be used to construct actual masks.

1.4. Stereo vision and deep learning approaches to depth estimation

We humans and other animals can estimate the depth of objects through binocular vision. The most classical *in silico* depth estimation scheme, stereo vision, is the computational version of this.

A stereo vision setup employs two cameras positioned on the same horizontal axis, separated by a small distance. Each camera captures slightly different views of the scene, and depth information can be extracted using trigonometry, and the fact that objects closer to the cameras exhibit greater disparity compared to those farther away.

Initially, the cameras are calibrated to ascertain their intrinsic parameters (like lens distortion and focal length) and extrinsic parameters (the relative position and orientation of the cameras). Following calibration, image rectification aligns the images onto a common plane, simplifying the process of finding corresponding points between the images.

To estimate the depth of objects in the scene, one of many feature matching algorithms must be used. Once the features are matched, the depth of the objects in the scene can be estimated as a function of the disparity between corresponding features.

OpenCV provides robust functions for camera calibration, image rectification, feature matching, and disparity computation.

While stereo vision and other technologies like Light Detection and Ranging (LiDAR) can generate accurate depth estimation data, there has been a considerable interest in monocular depth estimation mainly due to the minimal setup required; as its name suggest, a single camera.

DL approaches have been at the forefront regarding this subject matter. In particular a family of models coined MiDaS (not being an acronym, nor standing for anything in particular) have become the most popular. While the original MiDaS models were convolutional in nature, the most recently introduced MiDaS family of models is based on ViT backbones. [7] The most powerful of these is the Bidirectional Encoder representation from Image Transformers (BEiT); a BERT-like ViT. [8] Additional backbones are also available; namely Swin2, Swin, Next-ViT, and LeViT.

As shown in figure 5, MiDaS uses “hooks” to extract information from intermediate levels of the ViT. Tokens are reconstructed from the output of the hooks and passed through reassemble blocks; convolutional decoders that construct an image-like representation. Reassemble blocks hooking onto early layers of the ViT reconstruct the image at higher resolutions, while those doing so onto deeper layers do so at lower resolutions. Reassemble blocks are plugged into fusion blocks that fuse and up sample the representation. Finally, the last fusion block connects to a head in charge of constructing the final relative depth map representation. [9]

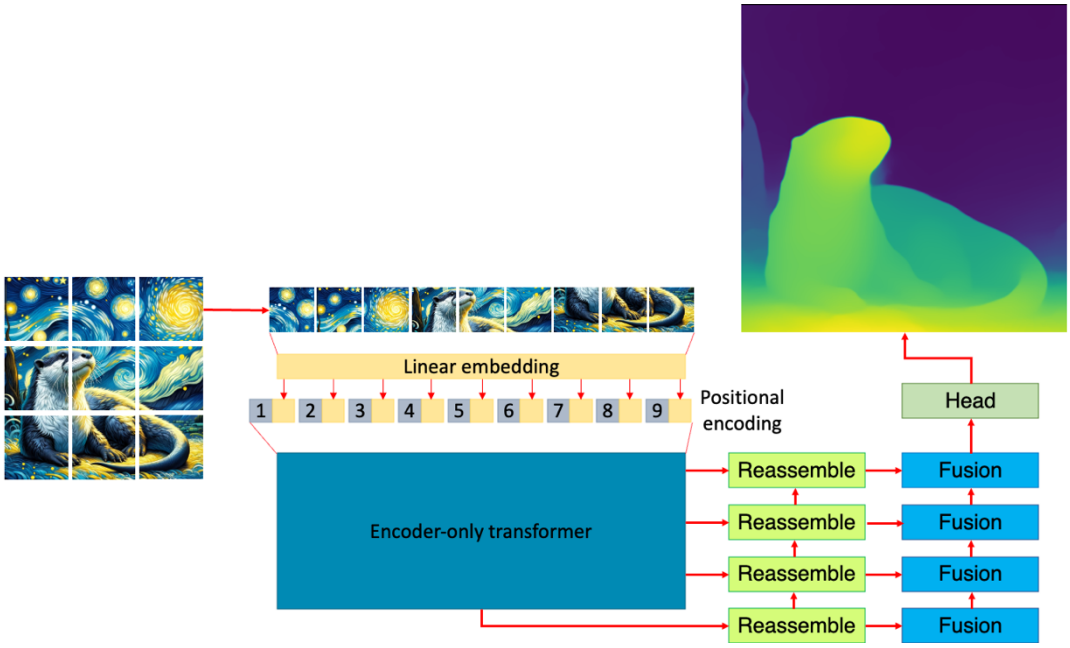


Figure 5. MiDaS architecture.
The latest iteration of MiDaS models is based on the common ViT architecture, but introduces reassemble and fusion blocks. Reassemble blocks are convolutional decoders which use hooks coming from intermediate layers of the transformer to construct an image-like representation. Fusion blocks on the other hand fuse and up sample the representation. The model’s head oversees constructing the ultimate relative depth map representation of the scene present in the input image. Regarding the relative depth map, yellow hues represent closeness, while blue ones farness, and green ones something in between.

Interestingly, MiDaS has found its way as a component in the popular image generation model Stable Diffusion, where it is used to condition the diffusion model so that it preserves depth coherency.

A network called ZoeDepth built on top of the MiDaS encoder is capable of estimating metric depth rather than relative one. It does this by plugging the hooks into a metric bin module. [10] Despite this, in our case MiDaS was preferred over ZoeDepth. For this study, fiducial markers were employed to convert the relative depth maps into metric ones, reducing the likelihood of spatial inconsistencies between adjacent video frames and ensuring that at least two points within the frame are correctly positioned relative to the camera.

2. Methods

2.1. Unity

Unity is a popular cross-platform game engine. In this thesis, it is used to implement a virtual vertiport environment, as it can be seen in figure 6. Several free assets from the Unity store were used to create the environment, with the most important one being “Living Birds”. This is a pack containing seven animated bird species able to spawn, fly around the scene, and land on user-defined spots. The included species are blue jay, cardinal, chickadee, crow, gold finch, robin, and sparrow. Moreover, the white platform in the middle is the vertiport.

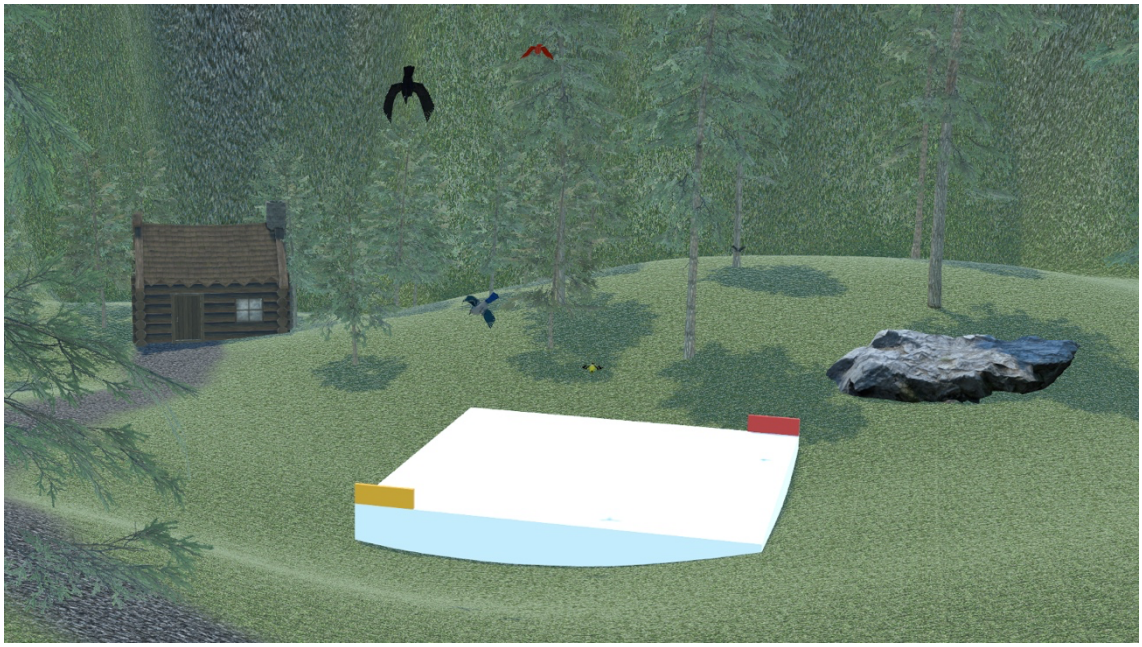


Figure 6. Aerial view of the scene constructed using Unity.
Several assets like grass and road textures, trees, a cabin, and a rock were used to construct the static elements of the scene. The white platform in the middle is the vertiport. Moreover, seven different bird species can spawn and move around the scene.

It is possible to interact with Unity using C# scripts. This capability was used to collect images and depth maps of the scene, as it will be explained in later sections. In the game engine, when the scene is running, a snapshot can be taking by pressing “s”, a video capture (comprising a series of frames taken at regular intervals, which would then need to be converted into video format) can be started by pressing “c”, and ended by pressing the same key again, and both a snapshot and a depth map of the scene can be taken by pressing “d”. These are saved to folders in the desktop that can then be relocated and/or rename as needed. Section 6.4. describes the appropriate file structure required for performing inference, synthetic data creation, and model training.

The full C# scripts can be found in sections 8.1., 8.2., and 8.3.

2.2. Pipeline

To address the problem of detecting birds on or near a the vertiport, the pipeline shown in figure 7 was created. Firstly, a frame is passed through a MiDaS model to obtain a relative depth map. This map is then converted into a metric depth map using a couple of fiducial markers.

The same original frame is fed into a YOLOv8 or YOLO-NAS model to obtain the bounding boxes of the birds in the scene. If a YOLOv8-seg model is used, the segmentation masks for each bird are additionally extracted. If a non-segmentation YOLO model was used, a SAM model can be optionally used to obtain the segmentation masks.

The boxes (and segmentation masks if available) are used to extract the corresponding areas on the metric depth map. Each of these extracted sections of the metric depth map are fed into k-means, where each pixel of the map is assigned to any of k clusters. The number of clusters, k, is chosen empirically with the objective that one cluster would represent the identified bird, while the rest of the clusters would correspond to the background and potential occlusions. After the cluster assignation, non-valid clusters are eliminated, and the smallest cluster centroid value is chosen as the depth (z_i) of the corresponding i-th bird.

Given this depth information and known intrinsic camera parameters, trigonometric operations are used to calculate the horizontal (x_i) and vertical (y_i) components of the birds’ coordinates.

Finally, a processed frame is presented where birds within the vertiport airspace are enclosed in red-colored bounding boxes, and birds outside the airspace are enclosed in green-colored bounding boxes. Moreover, the status of the vertiport is presented in the bottom left corner: “CLEAR FOR LANDING” if no bird is within the airspace, and “NOT CLEAR FOR LANDING” if at least one bird is within the airspace.

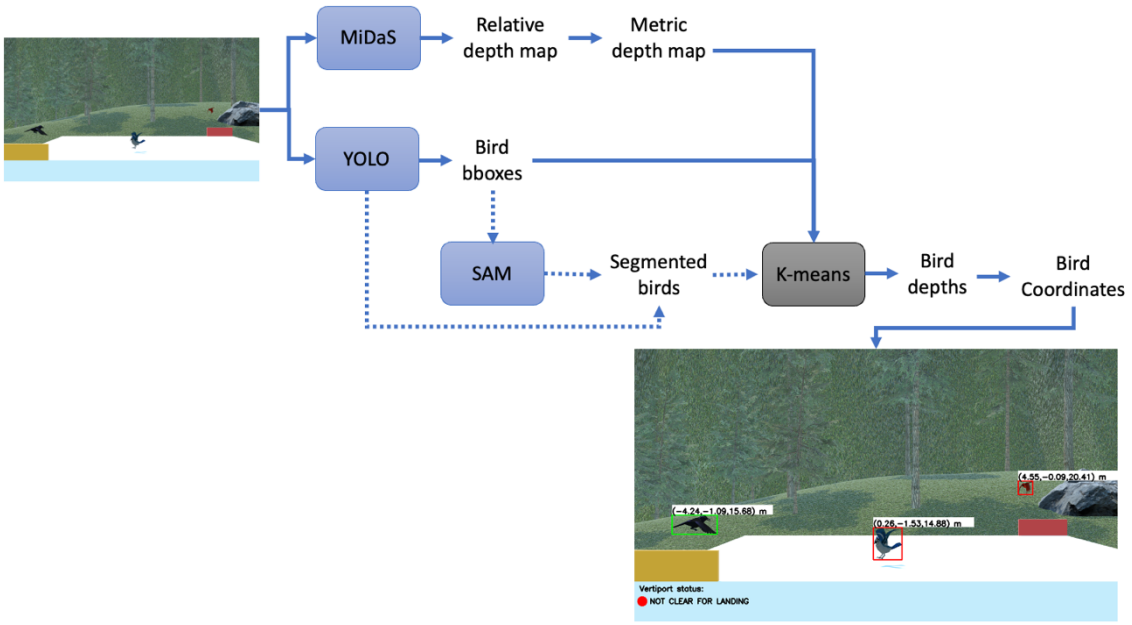


Figure 7. Monitoring system pipeline.
A frame is used to create a relative depth map using a MiDaS model. This depth map is processed to create a metric depth map. The frame is also used to detect birds and obtain their corresponding bounding boxes using a YOLO model. Optionally, is a YOLO-seg or a SAM model are used, segmentation masks of the birds are obtained. K-means is used to assign a depth value to each detected bird based on the metric depth map, the bounding boxes and segmentation masks if available. Lastly, the three-dimensional coordinates of each bird are obtained using the depths and some intrinsic camera parameters. Based on this data, a process frame representing the current state of the vertiport is presented.

2.3. Depth maps

As previously mentioned, relative depth maps are obtained by feeding a frame into a MiDaS model. Recall that "relative" refers to the fact that pixel depth values assigned by MiDaS are relative to each other within the same image, rather than being absolute measurements of distance in real-world units such as meters or centimeters. To be useful for our purposes however, the relative depth map needs to be converted into a metric depth map. This is achieved with the aid of fiducial markers. In the test scene used here, two static squares (one orange, one red) of known depth serve this purpose, as it can be seen in figure 8. The orange marker ($fiducial_1$) is positioned 10.79 m, while the red marker ($fiducial_2$) 20.79 m away from the camera.

Figure 8. Sample frame with red arrows pointing towards the two fiducial markers.
The yellow marker is positioned 10.79 m away from the camera, while the red one is positioned 20.79 m away.

Each fiducial marker gets assigned a single relative depth value. The assigned values are the minimum ones within the area corresponding to the respective marker. As the relative values produced by MiDaS are greater for closer objects and smaller for more distant objects (see figure 9-A), this ensures that partial occlusions of the marker (e.g., flying birds) are ignored.

To obtain the metric depth map, a transformation to the relative depth map is applied as follows:

$$m = \frac{RelativeDepth_{fiducial2} - RelativeDepth_{fiducial1}}{MetricDepth_{fiducial2} - MetricDepth_{fiducial1}}$$

$$b = RelativeDepth_{fiducial1} - m * MetricDepth_{fiducial1}$$

$$MetricDepthMap = \frac{RelativeDepthMap - b}{m}$$

An example of a relative depth map obtained using MiDaS BEiTLarge 512, and its corresponding metric depth map are shown in figure 9.

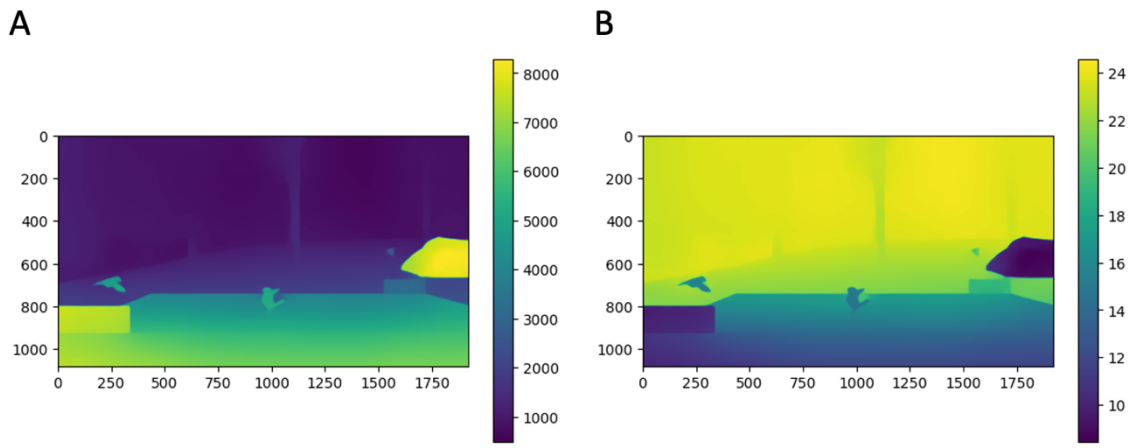


Figure 9. Relative and metric depth maps.
A. Relative depth map of the sample frame outputted by a MiDaS model. The color bar is in relative units. Closer objects tend to result in greater relative depth values. **B.** Estimated metric depth map of the sample frame. The color bar is in meter units. Closer objects tend to result in smaller metric depth values.

There is a glaring issue with the previous depth maps. Notice the rock in the right-most part of the image is predicted to be very close to the camera; less than 10 m away when it is evident it is more than 20 m away.

The reason this occurs is that MiDaS makes an implicit tradeoff regarding how much attention it pays to local versus global features. The higher the resolution of the images, the more importance is given to local features, while the lower, the more is given to global ones. Thus, by reducing image resolution (e.g., with a scaling factor of 0.5) before feeding it to MiDaS, the artifact of Figure 9 (and other similar situations) could be eliminated. Figure 10 shows the corrected metric depth map based on reducing the image resolution prior to MiDaS processing. As seen, the rock is now correctly positioned relative to other objects in the scene in accordance with our qualitative expectations.

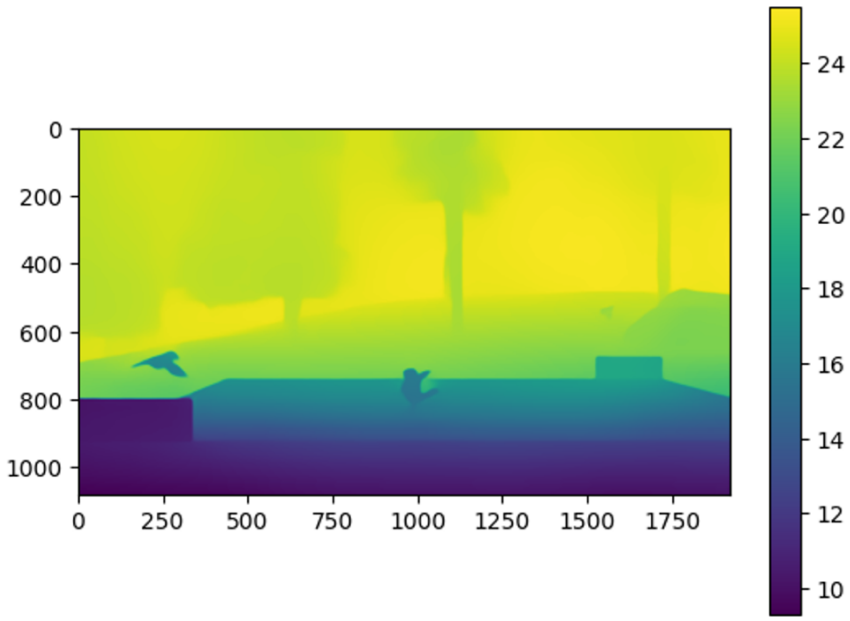


Figure 10. Impact of frame scaling on depth maps.
Metric depth map created by scaling the sample frame 0.5x. Using this scaling the qualitative consistency of the scene is preserved, the rock appearing to be relatively well positioned.

2.4. Object recognition and bounding boxes

As described previously, YOLO models can provide a list of a bounding boxes with associated class confidence scores, which is quite convenient for filtering purposes.

The YOLOv8x and YOLO-NAS-L results for confidences 0.25 and over are shown in figure 11.

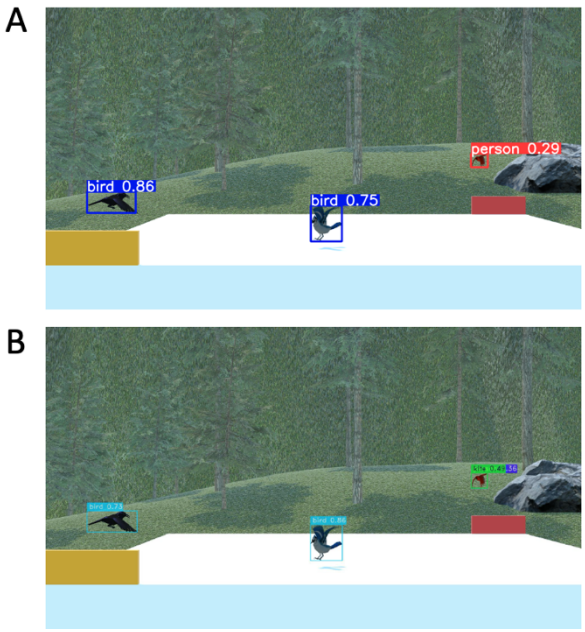


Figure 11. YOLOv8x and YOLO-NAS-L results on sample frame.
The images show instances of objects detected with a confidence of 0.25 or over **A.** YOLOv8x classified two instances as birds and one as a person. **B.** YOLO-NAS-L classified all three instances as birds. Additionally, one of the instances was identified as a person and a kite.

YOLOv8 (figure 11-A) managed to correctly identify two out of the three birds, while YOLO-NAS (figure 11-B) managed to identify all three birds in this example. The rightmost bird was additionally identified as a person and a kite by YOLO-NAS.

After inference, the bounding boxes are filtered to only keep those corresponding to birds.

2.5. Depth estimation

To estimate the depth of each individual detected bird, the bounding boxes are used to extract the corresponding sections of the metric depth map. Additionally, if a YOLO-seg or SAM was used, a segmentation mask is applied to the depth map before extraction.

As it can be seen in figure 12, each section is separated into k clusters using k-means, and the values of the cluster centroids are stored. Clusters are deemed invalid if they do not contain enough members (above a certain percentage of total pixels) and are excluded from further processing. The depth of the bird is then assigned to be the minimum value of the valid centroids.

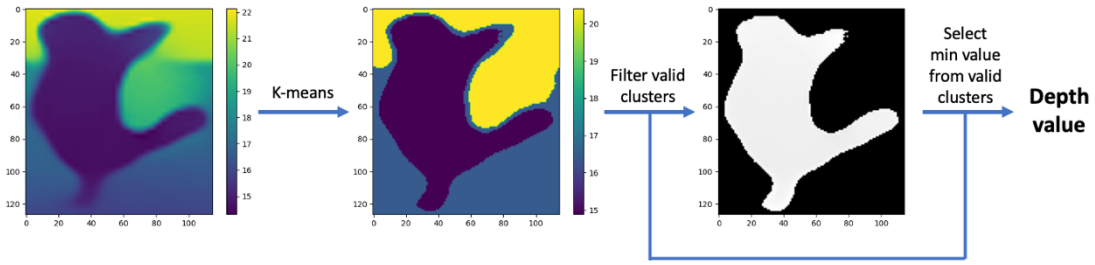


Figure 12. Depth value assignment process.
A unique metric depth value is assigned to each instance recognized as a bird in a three-step process: 1. Areas of the metric depth map corresponding to obtained bounding boxes are extracted (left plot). 2. Pixels are separated into K clusters (3 in this case). A graphical depiction of the results is presented in the central plot, where each color represents the value of the corresponding cluster centroid. 3. Only clusters containing a number of pixels over a certain threshold fraction are considered for the assignment of a depth value. The right plot shows valid clusters in white and invalid clusters in black (using a threshold value of 0.3). A depth value is finally assigned as the minimum-valued valid centroid.

The rationale behind this process is that occlusions (if present) are eliminated during invalidation, and the background when excluding centroid depth values greater than the minimum.

2.6. Coordinates calculation

Once the depths are estimated, the vertical and horizontal coordinates can be calculated using known camera parameters.

First, the sensor pixel coordinates of the bounding box centers are converted into metric coordinates. The size of the camera sensor used in our setup is 36x24 mm. The conversion from pixel coordinates to metric coordinates is performed in an analogous manner (i.e., using equivalent equations) as the relative to metric depth maps conversion. The view from the sensor is depicted in figure 13-A.

The field of view of the camera in the horizontal direction (fov_x) is 39.59775º, while in the vertical direction (fov_y) is 26.99147º. For further calculations, we will use the half-field of views (hfov). Knowing the metric coordinates of the objects and the half-field of views of the camera we can obtain the dimensions of the plane where the object is located, and using sensor dimensions we can calculate an expansion ratio, or how much bigger this plane is in relation to the sensor. Multiplying the expansion ratio with the object sensor coordinates, we obtain the horizontal (x_i) and vertical (y_i) components coordinates in three-dimensional space. This process is depicted in figure 13-B.

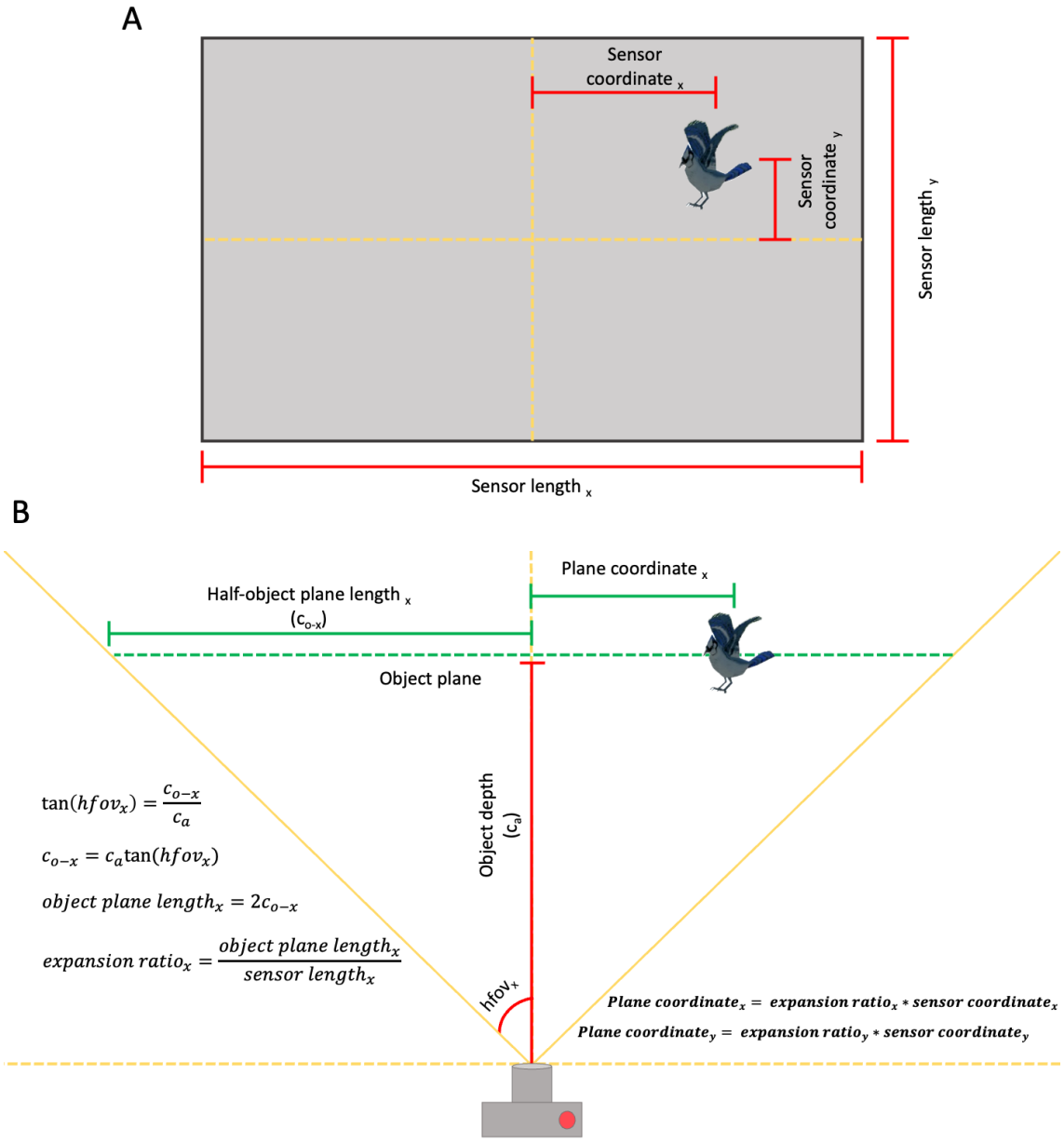


Figure 13. Calculation of vertical and horizontal coordinates.
Known distances are represented using red bars, while unknowns are represented with green bars. **A.** Camera sensor, where the origin of the coordinate system corresponds to the center. The sensor dimensions and the coordinates of an object of interest are known. **B.** xz-plane view of a scene. The inclined yellow lines represent the limits of the field of view of the camera. A right triangle can be constructed using one of these limits along with the object's depth and half the x-length of the plane the object lies in. Using trigonometry, we can solve for the x-length of the object plane, and using the sensor x-length we obtain an expansion ratio. Multiplying the sensor x-coordinate by this expansion ratio we can finally obtain the x-coordinate of the object in three-dimensional space. Analogous calculations are done in the yz-plane to obtain the y-coordinate.

Along with depth, a complete set of coordinates is constructed as a three-tuple (x_i, y_i, z_i). More precisely, the horizontal and vertical coordinates of this tuple correspond to the center of the bounding box in question.

2.7. Construction of processed frame

To construct the output frame, additional information about the limits of the vertiport airspace is required. In this scenario, we define the limits of the vertiport airspace as any space above the vertiport landing platform. The origin of the frame of reference is centered at the camera, and the limits are: x_{limits}=[-4.07,5.93] m, and z_{limits}=[10.84,20.84] m. The are no limits set for y, as the whole visible vertical space is part of the vertiport airspace.

With this information along with the coordinates, birds within the airspace are enclosed by red bounding boxes, while those outside by green boxes. Also, a coordinates tuple is displayed along with each bounding box.

Lastly, if no birds are detected in the scene or all detected ones are enclosed by green boxes, a green circle along with the message “CLEAR FOR LANDING” will be displayed. Conversely, if at least one bird is enclosed by a red box, a red circle along with the message “NOT CLEAR FOR LANDING” will appear. Frames representing both scenarios can be observed in figure 14.

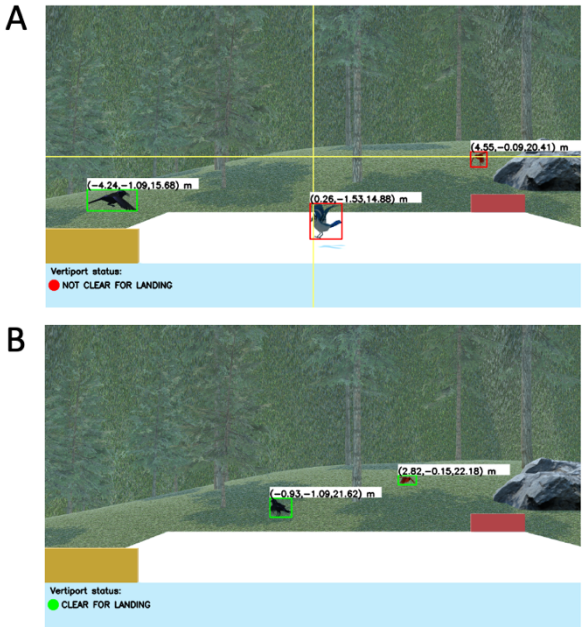


Figure 14. Processed frames.
Birds within the designated airspace are bounded by a red box, while those outside are bounded by a green box. The shown coordinates are in the form (x,y,z) and are in unit meters. **A.** Processed frame of a scenario in which the airspace is not free, with two out of three birds within the airspace. A “NOT CLEAR FOR LANDING” message appears to communicate this fact. Horizontal and vertical axes intersecting at the camera center are shown. The intersection represents the origin of the frame of reference and serve to verify all birds are in the correct quadrant as described by their corresponding sets of coordinates. **B.** Processed frame of a scenario in which the airspace is free, with all two birds in the scene are outside the designated airspace. A “CLEAR FOR LANDING” message appears to communicate this fact.

2.8. Inference

To process an image first import the vertiport surveillance module using:

```
from VertiportSurveillance import *
```

Once done, define the limits of the vertiport, the metric depth of fiducial markers, pixel positions of the fiducial markers, the size of the sensor (in mm), and the half field of view (simply referred as fov in the code). The parameters for the sample scene can be defined as follows:

```
# Define vertiport limits with respect to the origin (in m)
vertiport_x_limits = (-4.07, 5.93)
vertiport_z_limits = (10.84, 20.84)
# Define depths (in m) of fiducial markers
depth_fiducial1 = 10.79
depth_fiducial2 = 20.79
# Define pixel positions (x1,x2,y1,y2) of fiducial markers
position_fiducial1 = (0, 327, 803, 921)
position_fiducial2 = (1526, 1720, 677, 731)
# Define sensor size (in mm, (x,y))
sensor_size = (36,24)
# Define field of view (in degrees, (x,y))
fov = (39.59775/2, 26.99147/2)
```

Next define a YOLO and a MiDaS model, and optionally a SAM model. For example, to define YOLO-NAS-L, MiDaS BEiTLarge 512 and SAM-L, and send them to an NVIDIA GPU, use:

```
# Load YOLO-NAS model
model_YOLO = super_gradients.training.models.get('yolo_nas_l',
                                pretrained_weights="coco").cuda()
# Load MiDaS model
model_MiDaS, transform_MiDaS = load_MiDaS('dpt_beit_large_512', device='cuda')
# Load SAM model
model_SAM = sam_model_registry['vit_l'](checkpoint=
                               './SAM/sam_vit_l_0b3195.pth')
model_SAM.to(device='cuda')
```

YOLOv8 models are loaded in a slightly different manner. To load YOLOv8x, for example:

```
# Load YOLOv8 model
model_YOLO = YOLO('YOLO/yolov8x.pt')
```

To process a single image, define its path and run after changing the optional parameters to desired values:

```
process_image(image_path, model_YOLO, model_MiDaS, transform_MiDaS,
        vertiport_x_limits, vertiport_z_limits,
        depth_fiducial1, depth_fiducial2, position_fiducial1,
        position_fiducial2, sensor_size, fov, model_SAM = None,
        output_path = None, confidence_YOLO = 0.25,
        bounding_box_iou_threshold = 0.5, device_YOLO = 'cpu',
        device_MiDaS = 'cpu', relevant_class = 14,
        scaling_factor = 1.0, num_clusters = 3,
        fraction_of_pixels_threshold = 0.1,
        show_depth_map = False, draw_axes = False)
```

The optional arguments do the following:

- model_SAM: If provided, a SAM will be used for segmentation.
- output_path: Specifies the path where the processed image will be saved. If not provided, the image won't be saved.
- confidence_YOLO: Sets the confidence threshold for YOLO predictions. Predictions with confidence scores below this threshold will be discarded.
- bounding_box_iou_threshold: Sets the threshold for the intersection over union (IoU) metric used for filtering out overlapping bounding boxes.
- device_YOLO: Specifies the device (CPU, CUDA or MPS) on which YOLO model inference will be performed.
- device_MiDaS: Specifies the device (CPU, CUDA or MPS) on which MiDaS model inference will be performed.
- relevant_class: This parameter specifies the class label that is considered relevant for processing. It's used to filter out irrelevant bounding boxes. The default class (14) corresponds to “bird”.
- scaling_factor: Rescaling factor of the image for MiDaS processing.
- num_clusters: Specifies the number of clusters (k) for k-means.
- fraction_of_pixels_threshold: This parameter determines the threshold for the fraction of pixels that must be present within a cluster for it to be considered valid during the final depth assignation.
- show_depth_map: Specifies whether to display the depth map after processing.
- draw_axes: Specifies whether to draw vertical and horizontal axes on the processed image.

Video can be processed using the following function:

```
process_video(video_path, model_YOLO, model_MiDaS, transform_MiDaS,
        vertiport_x_limits, vertiport_z_limits,
        depth_fiducial1, depth_fiducial2, position_fiducial1,
        position_fiducial2, sensor_size, fov, model_SAM = None,
        output_path=None, fps=1, confidence_YOLO = 0.25,
        bounding_box_iou_threshold = 0.5, device_YOLO = 'cpu',
        device_MiDaS = 'cpu', relevant_class = 14,
        scaling_factor = 1, num_clusters = 3,
```

```
fraction_of_pixels_threshold = 0.1, draw_axes = False)
```

The only new optional argument is “fps”, which specifies the frames per second at which the output video will be saved if an output path is provided.

In unity there is no standard way to capture video, therefore a function to create a video from a set of frames was included:

```
create_video_from_frames(frames_path, output_video_path = None, fps = 1)
```

This function requires a path to a directory containing video frames to be provided. The directory can be created within Unity, as previously mentioned, by pressing “c” to start capturing frames, and pressing the same key again to stop.

The full code for the vertiport surveillance module can be found in section 8.4.

2.9. Construction of synthetic data

For testing, training and validation purposes, images of the scene along with corresponding synthetic metric depth maps were obtained. The depth maps were created by casting one ray per pixel, from the camera to the scene, and recording the distances at which they hit an object’s mesh. The result of this process for the sample frame is shown in figure 15. The meshes of flying birds are spherical while for birds resting in the ground are cuboid shaped. Another detail is trees only have meshes rendered in the lower part of the trunk, so the depths of the upper part of the trunks as well as the leaves and branches cannot be recorded in the depth map.

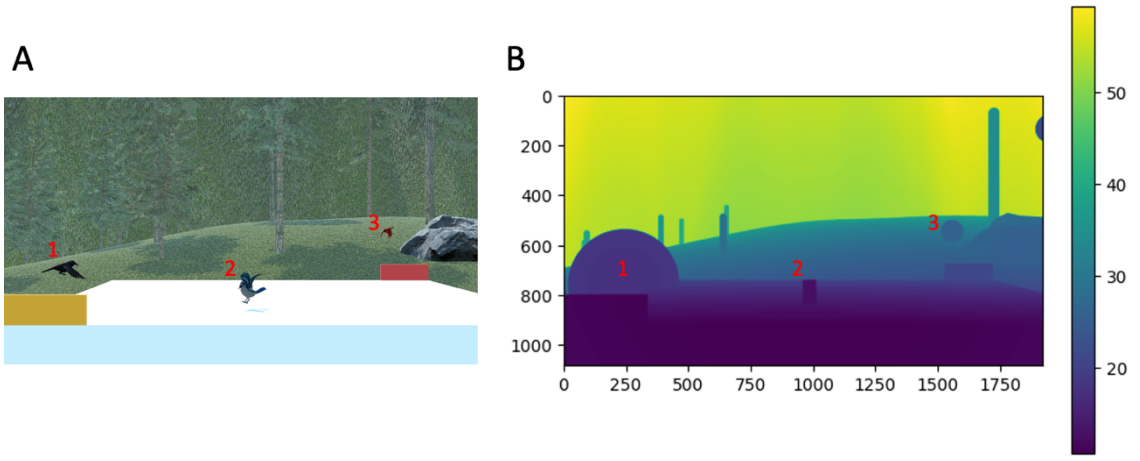


Figure 15. Sample frame and corresponding synthetic metric depth map.
Bird instances and their corresponding meshes recorded in the depth map are numbered. **A.** Sample frame. **B.** Corresponding synthetic metric depth map. Meshes of flying birds are renders as spheres while those of landing or resting birds are rendered as cuboids.

To partially address the inaccuracies of these synthetic depth maps, the following refinement process (figure 16) was implemented for the test set: 1. Bounding boxes were

manually drawn on instances of birds (using the jupyter_bbox_widget library on a jupyter notebook), 2. SAM was used on the sections corresponding to these bounding boxes to obtain a segmentation mask of each bird in the frame, 3. These masks were applied to the corresponding synthetic depth map to obtain bird-shaped sections of the spheres, 4. The most common depth value within each segmented bird was used as the depth for the whole bird.

Additionally, for validation and training sets, 5. The sections were superimposed onto a metric depth map of the background, and 6. Using the corresponding relative depth map (precomputed using MiDaS), the metric depth map of the scene was converted into a relative depth one, such that the minimum and maximum values of both relative depth maps correspond. This last step is necessary as MiDaS was trained on relative depth maps, not metric maps.

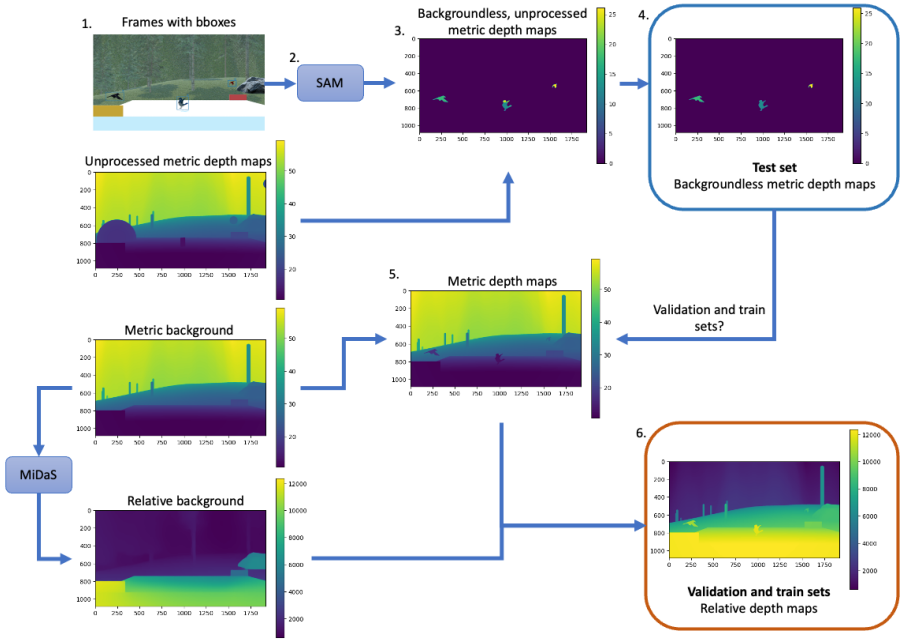


Figure 16. Construction process of synthetic test, validation, and train sets
1. Bounding boxes are manually drawn onto each frame of a set **2.** SAM is used to segment the birds enclosed by the boxes. **3.** The segmentation masks are applied to the unprocessed depth map of the scene. **4.** The most common depth value within each segmented bird is applied to all pixels, obtaining backgroundless metric depth maps. When creating a test set the process stops here, for validation and train sets it continuous. **5.** The masks are superimposed over a background metric depth map. **6.** The corresponding background relative depth map (computed from the metric one using MiDaS) is used to create relative depth maps of the frames such that the minimum and maximum values of both maps correspond.

A processed set can be created by running:

```
process_set(set_path, model_SAM, pickle_dataset = True, relative_depth = True,
                store_refined = True)
```

A set directory should contain a metric background depth map (BackgroundDepthMap.csv), a relative background depth map (RelativeBackgroundDepthMap.csv), a JSON file with the positions of the bounding boxes for each frame (annotations.json), a subdirectory containing the frames (SceneImages), and another containing the unprocessed

metric depth maps corresponding to each frame (DepthMaps). The directory and subdirectories containing the frames and depth maps are automatically created in the desktop the first time one presses “d” while running a scene in Unity. Each subsequent press will add an extra frame and depth map to the corresponding subdirectory. The JSON file can be created using jupyter_bbox_widget and added to the directory. The metric background depth map can be created as any other rough depth map by pressing “d” in unity. It should be renamed and moved to the main directory, however. Its corresponding frame should be removed from SceneImages, and be used to create the background relative depth map using MiDaS.

Once the process is finished, a new subdirectory called RefinedDepthMaps will appear in the main directory, if store_refined was set to True.

The function will return a PyTorch dataset of frames and processed depth maps. By default, this dataset will be pickled and saved to the main directory for later use. In particular, these datasets are used during model training.

The relative_depth argument should be set to True for the train and validation sets, and to False for the test set.

2.10. Pipeline evaluation

Specific pipeline setups (i.e., versions of the pipeline using distinct models and parameters) can be evaluated on a test set by importing the evaluation module and running:

```
from Evaluation import *

agregate_predictions_and_calculate_metrics(set_path, model_YOLO,
                            model_MiDaS, transform_MiDaS,
                            depth_fiducial1, depth_fiducial2,
                            position_fiducial1,
                            position_fiducial2,
                            sensor_size, fov,
                            iou_matching_threshold = 0.5,
                            model_SAM = None,
                            confidence_YOLO = 0.25,
                            bounding_box_iou_threshold = 0.5,
                            device_YOLO = 'cpu',
                            device_MiDaS = 'cpu',
                            relevant_class = 14,
                            scaling_factor = 1.0,
                            num_clusters = 3,
                            fraction_of_pixels_threshold = 0.1)
```

The only new optional argument is iou_matching_treshold, which is the minimum IoU value an obtained bounding box would need to have with a ground truth bounding box, for it to be considered as a true positive.

The function will return a dictionary of metrics including:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

$$MAE = \frac{1}{TP}\sum_{i=1}^{TP}\left|depth_{true} - depth_{predicted}\right|$$

Where TP is the number of true positive samples, FP of false positives, FN of false negatives, and MAE is the mean average error.

Additionally, it will return the time to process all frames in the set (in minutes), the average time to process a frame (in seconds), the number of frames evaluated, and the number of objects (i.e., birds) evaluated.

The evaluation module script can be found in section 8.5.

2.11. MiDaS fine-tuning

Several experiments were conducted to fine-tune the MiDaS BEiTLarge 512 model with a focus on freezing layers up to a specified depth. The fine-tuning process involved selectively immobilizing deeper layers while keeping the latter layers unfrozen. This approach was implemented to investigate the impact of layer-specific adjustments on the overall performance of the model. In all cases 250 frames and respective ground truth synthetic depth maps for a test set, and 25 for a validation set, were used.

Two different custom loss functions were implemented. The first is the reverse Hubert or BerHu loss (L_{BerHu}), which applies L1 loss for pixels in which the depth difference between the predictions and targets is under an error threshold c, defined as m times the maximum error in the batch, where m is a tunable parameter, and L2 for pixels above this threshold. The problem this loss tries to address is that of the model quickly learning to predict the depth of the background, which represents most of the pixels in a given depth map, and as such downplaying the importance of foreground objects. Using only L2 (or even worse, only L1), the loss would quickly become quite small during training even if the predictions regarding the birds remain poor. Using BerHu with a well selected m parameter, the background loss will tend to be calculated using L1, while the losses of the birds would generally be calculated using L2. The final loss is calculated as the mean of all pixel losses. L_{BerHu} was implemented for its use with PyTorch as follows:

```
class BerHuLoss(nn.Module):

    def __init__(self, m = 0.2):

        super(BerHuLoss, self).__init__()
        self.m = m

    def forward(self, predictions, targets):

        # Calculate the absolute difference
        diff = torch.abs(targets - predictions)
        # Calculate the threshold c as m times the max error in the batch
        c = self.m * torch.max(diff).item()
        # Condition for selecting L1 or L2 loss
        condition = diff <= c
        # Apply L1 loss
        l1_loss = diff
        # Apply L2 loss
        l2_loss = (diff**2 + self.m**2 * torch.max(diff).item()**2) / (2 * \
            self.m * torch.max(diff).item())
        # Combine the losses based on the condition
        loss = torch.where(condition, l1_loss, l2_loss)

        # Return the mean loss
        return torch.mean(loss)
```

The next loss was specifically created by René Ranftl *et al.* to train depth estimation models like MiDaS and address the primary sources of mismatch between datasets, such as and varying scales and baselines (i.e., the distance between the cameras in a stereo setting). [11] This loss is actually a combination of two other losses; namely, the scale- and shift-invariant loss with trimming ($L_{ssitrim}$) and gradient regularization loss (L_{reg}). Because in this scenario a dataset was produced using the same technique, there is no problem with scale and shift invariances, and therefore $L_{ssitrim}$ was simplified to L_{trim}. This would also apply for a non-synthetic dataset constructed using standardized equipment (i.e., the same type of LiDAR or stereo vision system). L_{trim} is nothing but an L1 loss where the differences below a certain tunable trim ratio are ignored. L_{reg}, the other piece of the final loss, is computed by performing different scalings on the targets and predictions (in this case 1x, 1/2 x, 1/4 x, 1/8 x, and 1/16 x), calculating the horizontal and vertical gradients using convolutions and x- and y-derivative kernels respectively, and computing the sum of horizontal and vertical gradients L1 losses. The final loss (referred to as $L_{trim} + L_{reg}$) is calculated as the sum of the losses, where L_{reg} is weighted by a tunable parameter α. Since the authors do not provide an implementation of the loss, it was implemented as follows, including a helper function to compute the gradients:

```
class LtrimLregLoss(nn.Module):

    def __init__(self, trim_ratio=0.2, alpha=0.5):

        super(LtrimLregLoss, self).__init__()
        self.trim_ratio = trim_ratio
        self.alpha = alpha

    def forward(self, predictions, targets):

        # Compute trimmed MAE loss
        diff = torch.abs(targets - predictions)
        diff_flat = diff.view(-1) # Flatten the tensor
        # Compute number of elements to trim
        num_to_trim = int(self.trim_ratio * diff_flat.size(0))
        # Sort the tensor and trim the elements
        diff_trimmed = torch.sort(diff_flat).values[num_to_trim:]
        # Compute the mean of the trimmed tensor
        trim_loss = diff_trimmed.mean()

        # Compute regularization loss
        reg_loss = 0.0
        for scale in [1, 2, 4, 8, 16]:  # Multi-scale gradients
            scaled_predictions = F.interpolate(predictions,
                    scale_factor=1/scale, mode='bilinear',
                            align_corners=False)
            scaled_targets = F.interpolate(targets, scale_factor=1/scale,
                    mode='bilinear', align_corners=False)
            pred_grad_x, pred_grad_y = compute_gradients(scaled_predictions)
            target_grad_x, target_grad_y = compute_gradients(scaled_targets)
            reg_loss += F.l1_loss(pred_grad_x, target_grad_x) + \
                                        F.l1_loss(pred_grad_y, target_grad_y)

        # Combine losses
        total_loss = trim_loss + self.alpha * reg_loss

        # Return the total loss
        return total_loss

def compute_gradients(array):

    # Assuming array is a 4D tensor: (batch_size, channels, height, width)
    # Create gradient kernel
    d = torch.tensor([[-1., 0., 1.]], device=array.device, dtype=torch.float)
    # Create a stack of kernels for each channel
    d = d.repeat(array.shape[1], 1, 1, 1)
```

```
    # Compute horizontal and vertical gradients
    horizontal_grad = F.conv2d(array, d, padding=(0, 1))
    vertical_grad = F.conv2d(array, d.transpose(2, 3), padding=(1, 0))

    # Return the gradients
    return horizontal_grad, vertical_grad
```

To fine-tune a model, first import the appropriate module as such:

```
from FineTune import *
```

Then set the desired parameters and run:

```
train_model(model, train_dataset, valid_dataset, device, optimizer,
        loss_function, batch_size = 5, num_epochs = 100, patience = 10,
        output_filename = 'MiDaS/weights/MiDair.pt',
        pickle_results = False, pickle_filename = 'MiDair_results.pkl',
        resume_training_from_checkpoint = False)
```

This will return information about training progress for each epoch (lists containing the train losses, train MAEs, validation losses, and validation MAEs). Optionally, these results can be pickled for recordkeeping. Moreover, if additional rounds of training are to be performed and there is a desire to concatenate the prior results, resume_training_from_checkpoint must be set to True.

The train and validation datasets can be passed directly if available or unpickled first. For example:

```
import pickle

train_dataset_path = 'TrainSet/dataset.pkl'
val_dataset_path = 'ValSet/dataset.pkl'

with open(train_dataset_path, 'rb') as file:
    train_dataset = pickle.load(file)

with open(val_dataset_path, 'rb') as file:
    val_dataset = pickle.load(file)
```

To load a fine-tuned model, not only the type of backbone needs to be specified, but also the path to the weights. For example, if a MiDaS BEiTLarge 512 fine-tuned model was named MiDair_large.pt, and it is stored in the standard directory, run:

```
model_MiDaS, transform_MiDaS = load_MiDaS('dpt_beit_large_512',
                    model_path = 'MiDaS/weights/MiDair_large.pt',
                    device='cuda')
```

For more details refer to the fine-tuning script provided in Section 8.6.

3. Results

3.1. Ablation study

The performance of the system was evaluated on a synthetic data test set of 49 frames and 125 instances of birds.

For these tests, a TP is only considered as such if the bounding box obtained during inference has an IoU of more than 0.5 with respect to a ground truth bounding box.

It is important to notice the precision, recall and F1 scores are affected only by the YOLO model, while the MAE is affected by all machine learning models in the pipeline.

Additionally, average execution time of the pipeline was recorded. The experiments were performed using an NVIDIA T4 GPU.

The default setup includes:

- MiDaS BEiTLarge 512
- Scaling factor (SF) of 1.0 (i.e., no scaling) for the frame when passed through MiDaS
- A threshold (TH) of 0.1 for the minimum fraction of pixels a cluster must contain to be considered for depth assignment
- k=3 for k-means
- No SAM model
- A confidence (Conf) of 0.25 for the YOLO model
- YOLO-NAS-L

An ablation study was performed by varying one or two parameters at a time relative to the default setup. The parameters to vary include SF, TH, K, SAM-L usage, Conf, and YOLO model. Based on this, the best performing parameters were selected and tested all together as an “optimal” model, both with and without the addition of SAM. These results are summarized in table 1.

Table 1. Ablation study results.
The tests were performed on 49 frames and 125 instances of birds and using an NVIDIA T4 GPU. This table presents the results of the default model and single or double-parameter modified models with respect to the default. The left-most column specifies the modified parameter. Performance metrics include precision, recall, F1 score, MAE (in meters), and average processing time (in seconds). The default model's results are highlighted in light yellow. Modifications related to MiDaS, segmentation (k-means and SAM), and YOLO are color-coded in orange, green, and blue hues, respectively. Gray highlights indicate results unaffected by the modification. The best results within each parameter category that also surpass the default model are in bold. Based on the study, the best parameters were picked, and an optimized system was evaluated. Namely the parameters that differ from the default system are the usage of YOLO-NAS-M, SF=0.75, and K=5. An additional version also using SAM was also evaluated. These results are also shown.

Modification	Precision	Recall	F1	MAE (m)	Time (s)
Default	0.7030	0.5680	0.6283	4.7173	0.9177
SF=0.25	0.7030	0.5680	0.6283	10.3969	**0.5421**
SF=0.50	0.7030	0.5680	0.6283	4.7888	0.8198
SF=0.75	0.7030	0.5680	0.6283	**4.6021**	0.8346
SF=1.50	0.7030	0.5680	0.6283	4.7524	0.8228
TH=0.2	0.7030	0.5680	0.6283	4.7447	**0.8056**
TH=0.3	0.7030	0.5680	0.6283	5.0201	0.8274
K=1	0.7030	0.5680	0.6283	5.1821	**0.7437**
K=2	0.7030	0.5680	0.6283	4.7574	0.7895
K=4	0.7030	0.5680	0.6283	4.7058	0.8547
K=5	0.7030	0.5680	0.6283	**4.6988**	0.9327
SAM, K=1	0.7030	0.5680	0.6283	4.7114	2.0547
SAM, K=2	0.7030	0.5680	0.6283	4.7154	2.0935
SAM	0.7030	0.5680	0.6283	**4.6951**	2.1253
Conf=0.20	0.6496	**0.6080**	0.6281	4.7597	0.8845
Conf=0.30	0.7143	0.5200	0.6019	4.7870	0.8005
Conf=0.35	0.7250	0.4640	0.5659	**4.6083**	0.7474
Conf=0.40	**0.7917**	0.4560	0.5787	4.6088	**0.7322**
NAS-M	**0.7500**	**0.6000**	**0.6667**	**4.6581**	0.8249
v8x	0.7500	0.3600	0.4865	4.9783	0.4785
v8l	0.7800	0.3120	0.4457	5.1431	**0.4079**
v8x-seg	0.7407	0.3200	0.4469	4.7673	0.4367
Optimal	**0.7500**	**0.6000**	**0.6667**	**4.4954**	0.9733
Optim.+SAM	**0.7500**	**0.6000**	**0.6667**	4.5262	2.1875

3.2. Qualitative analysis of the results

Looking through the processed test set frames, one cannot find a single false positive result (i.e., a non-bird object enclosed by a bounding box). Although it is standard for the field to consider obtained bounding boxes with an IoU of 0.5 or over with respect to the ground truth as TP, relaxing this threshold to 0.25 leads to a precision of 0.9208, and 0.9500 for the default and optimal settings, respectively. Using this relaxed threshold, the recall also increases in both cases to 0.7440 and 0.7600. This shows that there are no problems with other objects being mistaken with birds, but there are definitely some birds that cannot be identified as such, as it can be seen in figure 17, where birds unidentified by the optimal setup are highlighted with red arrows. There is also a clear pattern involving bird species, in which crows are almost never missed, while cardinals are commonly missed, with other species lying somewhere in between.

Figure 17. Frames where not all birds were detected using the optimal setup. Undetected birds are highlighted with a red arrow.

The MAE values are also relatively large, with the optimal model obtaining a result of 4.4954 m, considering the vertiport is 10 m in length. There are two things to keep in mind regarding these results. The first is that the synthetic ground truth maps are not perfect, as the depth values could not be recorded using the bird objects themselves, but rather the spherical meshes surrounding them. The ratio of these spheres is somewhat substantial when compared to the birds, meaning the depths are off by some amount, and therefore the obtained MAE results are an overestimation. The other observation is that some outlier MiDaS predictions are off by a great margin, significantly skewing the overall results, as it can be seen in figure 18. Here, predictions that are clearly off by a large margin are highlighted with a red arrow. Also, when present, birds are connected to the shadow they cast over the vertiport to get a better sense of their true distances.

As a remainder, the z-axis (i.e., depth) limits of the vertiport are 10.84 and 20.84 m. For instance, the top left frame predicts one of the lower birds to be inside the vertiport when it is clearly inside, while predicting the upper one to be further than the red fiducial marker, when based on size it is clearly quite close, possibly closer than the orange fiducial marker; thereby overshooting by more than 10 m. As another example, the top center image shows birds clearly inside the vertiport as being outside, further than the red fiducial marker.

Figure 18. Frames where depth predictions are off by a wide margin using the optimal setup.
Birds whose predicted depth is off by a wide margin are highlighted with a red arrow. Additionally, birds casting a shadow over the vertiport are connected to it by a red line to better gauge their true depths.

Figure 19 is representative of the majority of situations, in which the detection within the frame is correct. As can be seen, all birds are detected, and the predicted depths seem to be quite close to the actual depth values.

Figure 19. Frames presenting no glaring issues using the optimal setup.
In these frames all birds are detected, and their predicted depth values make qualitive sense.

3.3. Creation of MiDair via MiDaS fine-tuning

Several experiments were made to improve the performance of MiDaS using 250 frames and ground truth synthetic relative depth maps for a test set, and 25 for a validation set. Both BerHu and Ltrim + Lreg were used, different hyperparameter values were tested, and distinct sets of layers were frozen. Most training schemes yielded fine-tuned models that were outperformed by the base MiDaS BEiTLarge 512 model during evaluation on the test set. One scheme, however, was successful in producing models surpassing the base. This includes a first fine-tuning round using L_{trim} + L_{reg} with a relaxed trim ratio of

0.2, with the main aim of learning to correctly predict the depth of the static background. This was followed by a second fine-tuning round using a more aggressive ratio of 0.80, so that the network focused on accurately predicting the depth of the dynamic elements of the scene (i.e., the birds).

The training curves are shown in figure 20. Panel A shows the results of the first round, while B shows them for the second one. For both rounds the patch encoder and the first 20 (out of 25) transformer blocks of BEiT were frozen, while the rest of the layers were kept unfrozen. These include the last five transformer blocks, and all reassemble and fusion blocks, as well as the head, and corresponds to about 26% of the total model parameters. The model includes hooks at blocks 5, 11, 17, and 23, so only the last hook is affected under this scheme. AdamW was employed as the optimizer with the learning rate set to 1e-5. Training utilized a batch size of 5, spanning a maximum of 50 epochs, and early stopping was set with patience of 5 epochs. The α parameter for $L_{trim} + L_{reg}$ was set to 0.5.

It is important to note that, unlike the tests described in section 3.1., the MAE on the training curves represents the average absolute error per pixel across all images in the training or validation dataset, respectively. Moreover, while testing, the evaluation is performed on metric depth maps, while during fine-tuning it is done on relative depth maps.

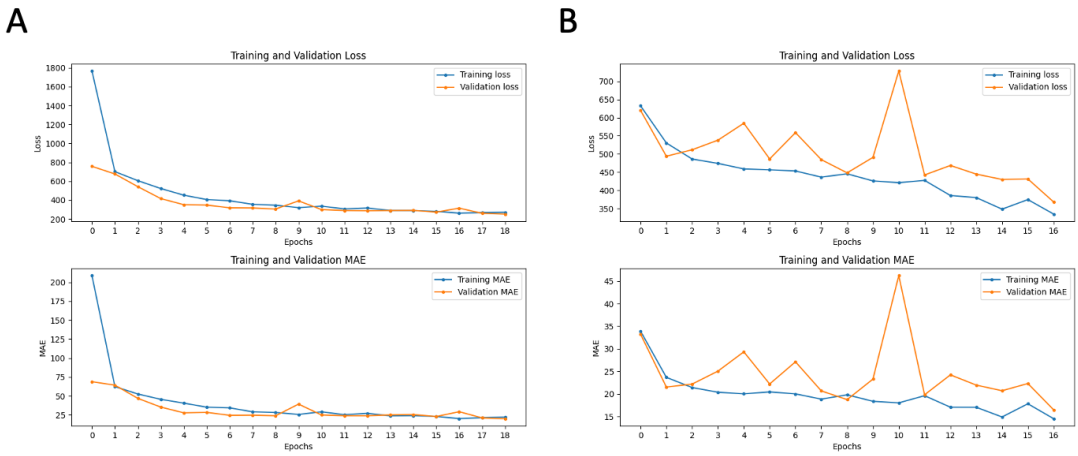


Figure 20. Training curves for both rounds of fine-tuning.
Training and validation loss and MAE up until the epoch where the checkpoint with the lowest loss was obtained for **A.** a first round of fine-tuning with a trim ratio of 0.2, and **B.** a second round of fine-tuning with a trim ratio of 0.8.

Tests analogous to those described in section 3.1. were performed using the optimal settings, only replacing the base MiDaS with this fine-tuned version coined MiDair. This setup achieved a metric MAE score of 3.8611 m over true positive samples. This is a significant improvement when compared to the same evaluation using the base MiDaS model, which again scored 4.4954 m.

4. Discussion and conclusions

Both YOLO and MiDaS models could be further optimized through fine-tuning to obtain better results. Although an improved version of MiDaS (i.e., MiDair) was obtained for dealing with the specific scenario addressed in this thesis, a richer dataset containing images from many vertiports, taken from different angles, and showing a plethora of different situations would yield more generalizable results, but would also be a massive manual endeavor and out of the scope of the project. Regardless, the fact the fine-tuning was ultimately successful suggests MiDair was able to pick contextual queues like shadows, and the relative sizes of different species of flying birds.

Base MiDaS does a decently good job estimating the depth of objects in relation to each other when partial occlusions exist. For example, if object A is partially occluding object B, MiDaS will very likely predict object A is closer to the camera than object B. However, when no partial occlusions exist, depth prediction becomes significantly more difficult. This situation is often the case when dealing with flying objects.

My recommendations for future work regarding MiDaS is to first perform fine-tuning by employing traditional techniques on a rich dataset containing not only images of flying birds in a vertiport environment, but ideally also a plethora of different flying objects, all accompanied by their respective ground truth depth maps. Perhaps it would also be beneficial to include some of the original data MiDaS was trained on (i.e., datasets like DIML Indoor and MegaDepth) to avoid catastrophic forgetting of previously learned information. In such a case, it might be worth implementing the full $L_{ssitrim} + L_{reg}$ loss.

This improved MiDair model could then be further fine-tuned to boost performance in specific environmental conditions. This might be done per vertiport basis to account for particularities, like the presence of novel bird species or drone types, very particular lighting conditions, etc. One way to do this easily and cheaply would be using low-rank adaptation (LoRA). [12] LoRA works by freezing the weights W of the NN and training a separate set of weights ΔW that will later be summed to W. Furthermore, ΔW can be decomposed into A and B arrays (i.e., $\Delta W=BA$), and these arrays can be trained instead. If ΔW is of size dxk, then B is of size dxr, while A is of size rxk; where r is the rank, a tunable parameter. The lower the value of r, the least parameters one would have to train, but also the lower performance one would obtain when compared to traditional fine-tuning. Effectively, A reduces the dimensionality of ΔW by removing linearly dependent columns, and B reestablishes the original dimensionality. The main advantage of this method when compared to traditional fine-tuning techniques is that many less parameters need to be trained, while experiencing negligible performance losses when an appropriate r value is chosen. Moreover, because the updates are simply wrapped around the original weights array (i.e., $W_{new} = W + \Delta W$), unlike other techniques, there are no increases in inference latency, which might be critical for this type of problems requiring real-time processing. Hugging Face offers a LoRA implementation as part of its Parameter-Efficient Fine-Tuning (PEFT) library. If a per vertiport-bases fine-tuning approach is going to be taken, perhaps one could even obviate the fiducial markers and use ZoeDepth instead of MiDaS.

Lastly in relation to fine-tuning, a potentially effective approach to obtain better performing models would be to define a two-term loss function that explicitly considers bounding boxes (or segmentation masks). One term would be more aggressive and deal

with in-bounding box sections, while the other would be tamer and deal with the background.

Regarding inference time, the tests were performed using an NVIDIA T4, considered to be a mid-tier GPU. Using a higher-tier GPU, or even better, more than a single GPU, would significantly speed up the process. Moreover, if models are INT8-quantized they might be able to process video at decent frame rates with a single mid-tier GPU.

In any case, the investment in GPUs might be quite low when compared to the alternative, involving expensive LiDAR sensors or stereo vision setups for depth estimation purposes. Regardless, one cannot get around DL and GPU usage regarding detection and classification anyways. These cost benefits are, for example, in part what pushed Tesla to go vision-only for their latest Autopilot iterations; a pipeline that notoriously includes monocular depth estimation.

Neither YOLO nor MiDaS are recurrent in any way; that is, they process frames one at a time, completely disregarding whatever frame or set of frames came before. This is obviously a downside when dealing with video, both because past frames can provide valuable information, and because temporal coherency is desired. If one was to get very ambitious with the system, probably the best way to achieve top performance would be to pass some information about past frames to YOLO and MiDaS. This could be as simple as including bounding boxes and associated depth values from prior frames, along with the current frame during each forward pass.

As a final note, a few relevant models have very recently emerged. On January 19th 2024, an open-source model named Depth Anything (a nod to SAM) was published by TikTok researchers and claims to be the new state-of-the-art regarding monocular depth estimation, following a ViT architecture just like MiDaS. [13] For future work it might be worth trying this model in the pipeline and build from there if noticeable improvements are noticed.

Interestingly, diffusion models are now being used for monocular depth estimation. In early December 2023, researchers from ETH Zürich presented the open-source Marigold. [14] This model is based on Stable Diffusion and was later fine-tuned on depth data to perform well regarding monocular depth estimation. Later in December Google DeepMind presented Diffusion for Metric Depth (DMD), with yet unreleased code at the time of writing. [15] While Marigold is akin to MiDaS in that it produces relative depth maps, DMD is similar to ZoeDepth, as it outputs metric depth maps directly.

Diffusion models are notoriously slow during inference when compared to ViTs as several forward passes are needed to produce the final representation, so they might not be appropriate for real-time applications like this. Also, although Marigold produces much more detailed maps than MiDaS (which is not particularly important in our case), when introduced into the pipeline, Marigold tends to severely overshoot the depth of the birds. This can be seen in figure 21, where Marigold was tried on the sample frame and on one of the problematic frames presented in section 3.2.

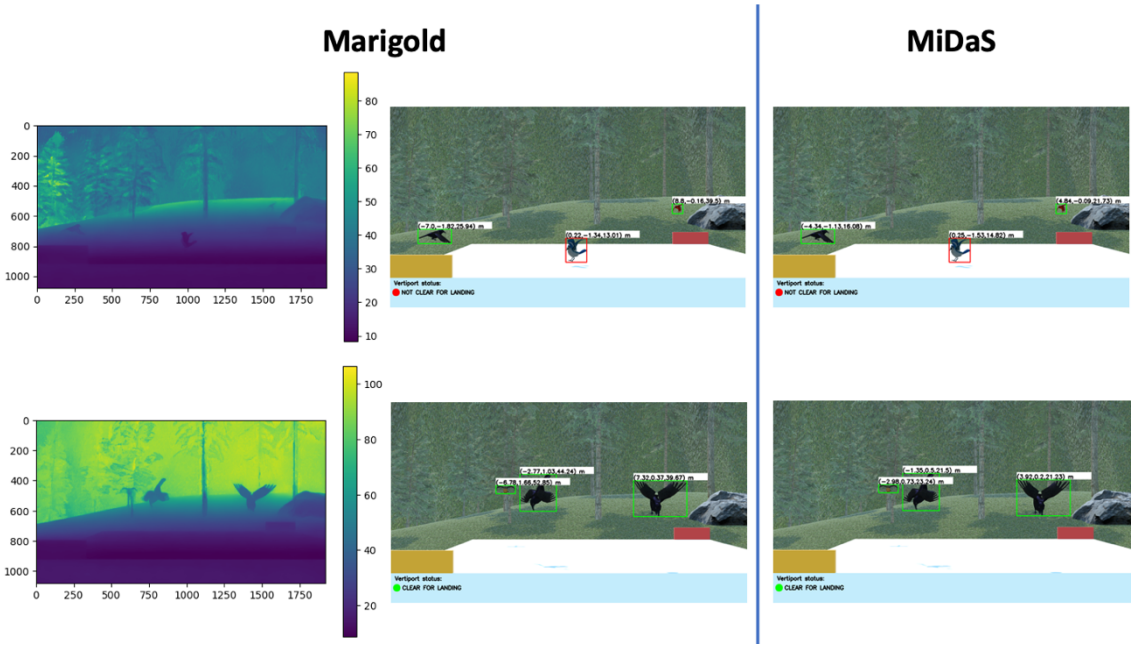


Figure 21. Metric depth map and predictions obtained with Marigold contrasted to MiDaS.
Metric depth maps obtained with Marigold, as well as the corresponding final processed frames, along with the results from MiDaS for contrast. Marigold seems to severely overestimate the depth of most birds when compared to MiDaS. The frames were processed under optimal settings as described in section 3.1.

All in all, the creation of a pipeline capable of detecting and positioning birds in three-dimensional space was achieved. Moreover, fine-tuning MiDaS to obtain MiDair resulted in an increased ability to estimate the depth of flying birds with precision under a monocular vision scheme.

5. Acknowledgments

I want to thank David Olivieri (University of Vigo), Higinio González (University of Vigo), and Patricia López (CRIDA) for acting as my supervisors. This specially goes to David, whose insightful feedback, support throughout the writing process of this thesis, and engaging discussions have been instrumental in shaping its final form. I also want to acknowledge the generosity of CRIDA for providing me with a grant to undertake this project, and to my parents for their unconditional support.

6. Notes

6.1. On model licenses

YOLOv8 models are distributed by Ultralytics under an AGPL-3.0 license, or an alternative enterprise license to bypass the AGPL-3.0 requirements. YOLO-NAS is distributed by Deci under an Apache-2.0 license for non-commercial use, and an agreement should be made with Deci for commercial purposes. MiDaS is released under an MIT license. SAM is distributed by Meta under an Apache-2.0 license. Model selection or even training

similar models from scratch should be considered when employing the pipeline for commercial applications.

6.2. On PyTorch's loading of state dictionaries

It is possible that, to load MiDaS models correctly, you must enforce PyTorch to load state dictionaries in a non-strict manner. If you encounter problems first check if PyTorch is enforcing strict loading by running:

```
# Specify the file path
file_path = "/usr/local/lib/python3.10/dist-packages/torch/ \
   nn/modules/module.py"

# Open and read the file
with open(file_path, 'r') as file:
   lines = file.readlines()

# Define the substring to search for
search_string = "def load_state_dict("

# Initialize a variable to store the line index
line_index = None

# Iterate over the lines to find the first occurrence of the substring
for i, line in enumerate(lines):
   if search_string in line:
      line_index = i
      break

# Check if the substring was found and print the relevant lines
if line_index is not None:
   for i in range(line_index, min(line_index + 4, len(lines))):
      print(f"Line {i + 1}: {lines[i]}", end='')
else:
   print(f"No line found containing '{search_string}'")
```

If this is the case, you should see that the argument "strict" within the load_state_dict function is set to True. Check on which line this argument is (in my case 2068) and set the line number accordingly. To change this to False run:

```
# Check if the file has enough lines
line_number = 2068
if len(lines) >= line_number:
   # Replace the substring in the specific line
   lines[line_number - 1] = lines[line_number - 1].replace(
```

```
        "strict: bool = True", "strict: bool = False")

    # Write the modified content back to the file
    with open(file_path, 'w') as file:
        file.writelines(lines)
else:
    print(f"The file does not have {line_number} lines.")
```

After restarting the kernel the problem should be gone.

6.3. On library versions

The following non-standard library versions were used on Python 3.9.12:

- numpy: 1.23.0
- ultralytics: 8.0.200
- segment-anything: 1.0
- supervision: 0.16.0
- torch: 2.1.0
- cv2: 4.8.1
- matplotlib: 3.8.0
- sklearn: 0.0.post1
- super-gradients: 3.3.0
- jupyter_bbox_widget: 0.5.0

In particular, it is important to install a version of super gradients older than 3.6.0, as this last version changed the way to extract bounding boxes from YOLO-NAS models, which will cause an error during inference.

6.4. On the file structure and beit.py correction

The main directory contains VertiportSurveillance.py, FineTune.py, Evaluation.py, and the TrainSet, ValSet, TestSet, MiDaS, YOLO, and SAM subdirectories. YOLO contains weights from YOLOv8 models (which can be downloaded here: https://github.com/ultralytics/ultralytics), while SAM does it for the family of models of the same name (available here: https://github.com/facebookresearch/segment-anything). MiDaS is a clone of this GitHub repository https://github.com/isl-org/MiDaS. After cloning, add model weights to the weights subdirectory, which can be downloaded from the same repository.

The MiDaS directory contains a small error that might prevent you from correctly using models with a BEiT backbone. In MiDaS/midas/backbones/beit.py change mentions of “self.drop_path” to “self.drop_path1”, these occur within the block_forward function definition.

7. References

[1] "Vertiports in the Urban Environment | EASA." Accessed: Jan. 28, 2024. [Online]. Available: https://www.easa.europa.eu/en/light/topics/vertiports-urban-environment

[2] A. Vaswani *et al.*, "Attention Is All You Need," Jun. 2017, [Online]. Available: http://arxiv.org/abs/1706.03762

[3] A. Dosovitskiy *et al.*, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," Oct. 2020, [Online]. Available: http://arxiv.org/abs/2010.11929

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," Jun. 2015, [Online]. Available: http://arxiv.org/abs/1506.02640

[5] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," Nov. 2016, [Online]. Available: http://arxiv.org/abs/1611.01578

[6] A. Kirillov *et al.*, "Segment Anything," Apr. 2023, [Online]. Available: http://arxiv.org/abs/2304.02643

[7] R. Birkl, D. Wofk, and M. Müller, "MiDaS v3.1 -- A Model Zoo for Robust Monocular Relative Depth Estimation," Jul. 2023, [Online]. Available: http://arxiv.org/abs/2307.14460

[8] H. Bao, L. Dong, S. Piao, and F. Wei, "BEiT: BERT Pre-Training of Image Transformers," Jun. 2021, [Online]. Available: http://arxiv.org/abs/2106.08254

[9] R. Ranftl, A. Bochkovskiy, and V. Koltun, "Vision Transformers for Dense Prediction," Mar. 2021, [Online]. Available: http://arxiv.org/abs/2103.13413

[10] S. F. Bhat, R. Birkl, D. Wofk, P. Wonka, and M. Müller, "ZoeDepth: Zero-shot Transfer by Combining Relative and Metric Depth," Feb. 2023, [Online]. Available: http://arxiv.org/abs/2302.12288

[11] R. Ren´, R. Ranftl, K. Lasinger, D. Hafner, K. Schindler, and V. Koltun, "Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer," 2020.

[12] E. J. Hu *et al.*, "LoRA: Low-Rank Adaptation of Large Language Models," Jun. 2021, [Online]. Available: http://arxiv.org/abs/2106.09685

[13] L. Yang, B. Kang, Z. Huang, X. Xu, J. Feng, and H. Zhao, "Depth Anything: Unleashing the Power of Large-Scale Unlabeled Data," Jan. 2024, [Online]. Available: http://arxiv.org/abs/2401.10891

[14] B. Ke, A. Obukhov, S. Huang, N. Metzger, R. C. Daudt, and K. Schindler, "Repurposing Diffusion-Based Image Generators for Monocular Depth Estimation," Dec. 2023, [Online]. Available: http://arxiv.org/abs/2312.02145

[15] S. Saxena, J. Hur, C. Herrmann, D. Sun, and D. J. Fleet, "Zero-Shot Metric Depth with a Field-of-View Conditioned Diffusion Model," Dec. 2023, [Online]. Available: http://arxiv.org/abs/2312.13252

8. Appendices

8.1. Unity C# script for capturing snapshots – CameraSnapshot.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO;

public class CameraSnapshot : MonoBehaviour
{
    public Camera camera;
    private int snapshotCount = 0;

    private void Update()
    {
        // Check if the space key is pressed
        if (Input.GetKeyDown(KeyCode.S))
        {
            // Increment the snapshot count
            snapshotCount++;

            // Capture a snapshot from the camera
            CaptureSnapshot("CameraSnapshot_" + snapshotCount);
        }
    }

    private void CaptureSnapshot(string fileName)
    {
        // Create a RenderTexture to temporarily render the camera's view
        RenderTexture renderTexture = new RenderTexture(Screen.width, Screen.height, 24);
        camera.targetTexture = renderTexture;

        // Render the camera's view
        camera.Render();

        // Create a Texture2D and read the pixels from the RenderTexture
        Texture2D texture = new Texture2D(Screen.width, Screen.height, TextureFormat.RGB24, false);
        RenderTexture.active = renderTexture;
        texture.ReadPixels(new Rect(0, 0, Screen.width, Screen.height), 0, 0);
        texture.Apply();

        // Define the folder path on the desktop
        string folderPath = System.Environment.GetFolderPath(System.Environment.SpecialFolder.Desktop);
        string folderName = "BirdSnapshots";
        string fullPath = Path.Combine(folderPath, folderName);

        // Check if the folder exists, and create it if it doesn't
        if (!Directory.Exists(fullPath))
        {
            Directory.CreateDirectory(fullPath);
        }

        // Save the Texture2D as a PNG file in the folder
        string filePath = Path.Combine(fullPath, fileName + ".png");
        File.WriteAllBytes(filePath, texture.EncodeToPNG());

        // Clean up
        camera.targetTexture = null;
        RenderTexture.active = null;
        Destroy(renderTexture);
        Destroy(texture);
    }
}
```

8.2. Unity C# script for capturing video frames – CaptureVideo.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO;

public class CaptureFrames : MonoBehaviour
{
    public Camera camera;
    private int snapshotCount = 0;
    private float timeBetweenSnapshots = 1f / 10f; // 30f; // 1/30 seconds

    private float timeSinceLastSnapshot = 0f;
    private bool isTakingSnapshots = false;

    private string subfolderName;

    private void Start()
    {
        // Initialize subfolder name with date and time in 'dd/MM/yyyy-HH.mm.ss' format
        subfolderName = System.DateTime.Now.ToString("dd-MM-yyyy_HH.mm.ss");
    }

    private void Update()
    {
        timeSinceLastSnapshot += Time.deltaTime;

        // Check if it's time to take a snapshot and isTakingSnapshots is true
        if (isTakingSnapshots && timeSinceLastSnapshot >= timeBetweenSnapshots)
        {
            // Reset the time counter
            timeSinceLastSnapshot = 0f;

            // Increment the snapshot count
            snapshotCount++;

            // Capture a snapshot from the camera with leading zeros
            CaptureSnapshot(camera, snapshotCount.ToString("D5") + ".png");
        }

        // Check if the 'C' key is pressed to toggle snapshot capture
        if (Input.GetKeyDown(KeyCode.C))
        {
            isTakingSnapshots = !isTakingSnapshots; // Toggle snapshot capture
        }
    }

    private void CaptureSnapshot(Camera camera, string fileName)
    {
        // Create a RenderTexture to temporarily render the camera's view
        RenderTexture renderTexture = new RenderTexture(Screen.width, Screen.height, 24);
        camera.targetTexture = renderTexture;

        // Render the camera's view
        camera.Render();

        // Create a Texture2D and read the pixels from the RenderTexture
        Texture2D texture = new Texture2D(Screen.width, Screen.height, TextureFormat.RGB24, false);
        RenderTexture.active = renderTexture;
        texture.ReadPixels(new Rect(0, 0, Screen.width, Screen.height), 0, 0);
        texture.Apply();

        // Define the folder path on the desktop as 'BirdVideos'
        string folderPath = System.Environment.GetFolderPath(System.Environment.SpecialFolder.Desktop);
        string mainFolderName = "BirdVideos"; // Update main folder name to 'BirdVideos'
        string fullPath = Path.Combine(folderPath, mainFolderName, subfolderName);

        // Check if the folder exists, and create it if it doesn't
        if (!Directory.Exists(fullPath))
        {
            Directory.CreateDirectory(fullPath);
        }

        // Save the Texture2D as a PNG file in the subfolder
        string filePath = Path.Combine(fullPath, fileName);
        File.WriteAllBytes(filePath, texture.EncodeToPNG());

        // Clean up
```

```
        camera.targetTexture = null;
        RenderTexture.active = null;
        Destroy(renderTexture);
        Destroy(texture);
    }
}
```

8.3. Unity C# script for capturing frames and depth maps – DepthMap.cs

```
using UnityEngine;
using System.IO;
using System;
using System.Text;

public class DepthMapCreator : MonoBehaviour
{
    public Camera targetCamera; // Assign this in the Unity Inspector
    private string depthMapFolder;
    private string sceneImageFolder;
    private int fileNumber;

    void Start()
    {
        // Initialize folder paths
        string desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
        depthMapFolder = Path.Combine(desktopPath, "TestSet", "DepthMaps");
        sceneImageFolder = Path.Combine(desktopPath, "TestSet", "SceneImages");

        // Create directories if they don't exist
        CreateDirectoryIfNotExists(depthMapFolder);
        CreateDirectoryIfNotExists(sceneImageFolder);

        // Get the starting file number
        fileNumber = GetStartingFileNumber(depthMapFolder, "DepthMap");
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.D))
        {
            ReplaceMeshColliders();
            CreateAndSaveDepthMap();
            CaptureAndSaveImage();
        }
    }

    private void ReplaceMeshColliders()
    {
        foreach (GameObject obj in FindObjectsOfType(typeof(GameObject)))
        {
            MeshFilter meshFilter = obj.GetComponent<MeshFilter>();
            if (meshFilter != null)
            {
                MeshCollider existingCollider = obj.GetComponent<MeshCollider>();
                if (existingCollider != null)
                {
                    Destroy(existingCollider);
                }
                MeshCollider meshCollider = obj.AddComponent<MeshCollider>();
                meshCollider.sharedMesh = meshFilter.mesh;
            }
        }
    }

    private void CreateAndSaveDepthMap()
    {
        if (targetCamera == null)
        {
```

```
            Debug.LogError("Target camera not assigned.");
            return;
        }

        int width = targetCamera.pixelWidth;
        int height = targetCamera.pixelHeight;
        StringBuilder csvContent = new StringBuilder();

        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                Ray ray = targetCamera.ScreenPointToRay(new Vector3(x, y, 0));
                RaycastHit hit;
                if (Physics.Raycast(ray, out hit))
                {
                    float depthValue = hit.distance;
                    csvContent.Append(depthValue.ToString());
                }
                else
                {
                    csvContent.Append("-1");
                }

                if (y < height - 1)
                    csvContent.Append(",");
            }
            csvContent.AppendLine();
        }

        string depthMapPath = Path.Combine(depthMapFolder, $"DepthMap{fileNumber:D5}.csv");
        File.WriteAllText(depthMapPath, csvContent.ToString());
    }

    private void CaptureAndSaveImage()
    {
        // Create a RenderTexture to temporarily render the camera's view
        RenderTexture renderTexture = new RenderTexture(targetCamera.pixelWidth, targetCamera.pixelHeight, 24);
        targetCamera.targetTexture = renderTexture;

        // Render the camera's view
        targetCamera.Render();

        // Create a Texture2D and read the pixels from the RenderTexture
        Texture2D image = new Texture2D(targetCamera.pixelWidth, targetCamera.pixelHeight, TextureFormat.RGB24, false);
        RenderTexture.active = renderTexture;
        image.ReadPixels(new Rect(0, 0, targetCamera.pixelWidth, targetCamera.pixelHeight), 0, 0);
        image.Apply();

        // Save the image as a PNG file
        byte[] imageBytes = image.EncodeToPNG();
        string imagePath = Path.Combine(sceneImageFolder, $"SceneImage{fileNumber:D5}.png");
        File.WriteAllBytes(imagePath, imageBytes);

        // Increment the file number for the next capture
        fileNumber++;

        // Clean up
        targetCamera.targetTexture = null;
        RenderTexture.active = null;
        Destroy(renderTexture);
        Destroy(image);
    }

    private void CreateDirectoryIfNotExists(string path)
    {
        if (!Directory.Exists(path))
        {
            Directory.CreateDirectory(path);
        }
    }
```

```
    private int GetStartingFileNumber(string folderPath, string filePrefix)
    {
        int maxNumber = 0;
        foreach (string filePath in Directory.GetFiles(folderPath))
        {
            string fileName = Path.GetFileNameWithoutExtension(filePath);
            if (fileName.StartsWith(filePrefix))
            {
                string numberStr = fileName.Substring(filePrefix.Length);
                if (int.TryParse(numberStr, out int number) && number > maxNumber)
                {
                    maxNumber = number;
                }
            }
        }
        return maxNumber + 1; // Start from the next number
    }

}
```

8.4. Main Python script – VertiportSurveillance.py

```
import math
import numpy as np
from ultralytics import YOLO
from segment_anything import SamPredictor, sam_model_registry
import supervision as sv
import torch
import os
import subprocess
import cv2
import matplotlib.pyplot as plt
from MiDaS.midas.model_loader import load_model
from sklearn.cluster import KMeans
import json
import pickle
import time
# When importing super_gradients print is disabled
# saving the current stdout before importing and
# then restoring it enables the print again
import sys
stdout = sys.stdout
import super_gradients
sys.stdout = stdout

############################ PROCESS VIDEO & IMAGE ############################

def process_image(image_path, model_YOLO, model_MiDaS, transform_MiDaS,
            vertiport_x_limits, vertiport_z_limits,
            depth_fiducial1, depth_fiducial2, position_fiducial1, position_fiducial2,
            sensor_size, fov, model_SAM = None, output_path = None,
            confidence_YOLO = 0.25, bounding_box_iou_threshold = 0.5, device_YOLO = 'cpu', device_MiDaS = 'cpu',
            relevant_class = 14, scaling_factor = 1.0, num_clusters = 3, fraction_of_pixels_threshold = 0.1,
            show_depth_map = False, draw_axes = False):

    # Load image
    image = cv2.imread(image_path)
    # Convert image from BGR to RGB
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Process image and extract 3D coordinates
    objects_coordinates_3D, relevant_boxes = process_frame(image, model_YOLO,
                                        model_MiDaS, transform_MiDaS,
                                        depth_fiducial1, depth_fiducial2,
                                        position_fiducial1, position_fiducial2,
                                        sensor_size, fov, model_SAM = model_SAM,
                                        confidence_YOLO = confidence_YOLO,
                                        bounding_box_iou_threshold = bounding_box_iou_threshold,
                                        device_YOLO = device_YOLO,
                                        device_MiDaS = device_MiDaS,
                                        relevant_class = relevant_class,
                                        scaling_factor = scaling_factor,
                                        num_clusters = num_clusters,
```

```
                                    fraction_of_pixels_threshold = fraction_of_pixels_threshold,
                                    show_depth_map = show_depth_map)

    # Draw on image
    image_processed = draw_on_frame(image, objects_coordinates_3D, relevant_boxes,
                            vertiport_x_limits, vertiport_z_limits, draw_axes = draw_axes)

    # Display the resulting image
    plt.imshow(image_processed)
    plt.show()

    # Save the image if output_path is provided
    if output_path:
        image_processed = cv2.cvtColor(image_processed, cv2.COLOR_BGR2RGB)
        cv2.imwrite(output_path, image_processed)

def process_video(video_path, model_YOLO, model_MiDaS, transform_MiDaS,
                vertiport_x_limits, vertiport_z_limits,
                depth_fiducial1, depth_fiducial2, position_fiducial1, position_fiducial2,
                sensor_size, fov, model_SAM = None, output_path=None, fps=1,
                confidence_YOLO = 0.25, bounding_box_iou_threshold = 0.5, device_YOLO = 'cpu', device_MiDaS = 'cpu',
                relevant_class = 14, scaling_factor = 1, num_clusters = 3, fraction_of_pixels_threshold = 0.1,
                draw_axes = False):

    # Open the video file
    cap = cv2.VideoCapture(video_path)

    # Check if the video file opened successfully
    if not cap.isOpened():
        print('Error: Could not open video file')
        return

    # Get video frame properties
    frame_width = int(cap.get(3))
    frame_height = int(cap.get(4))

    # Define the codec and create a VideoWriter object if output_path is provided
    if output_path:
        fourcc = cv2.VideoWriter_fourcc(*'mp4v')  # Define the codec
        out = cv2.VideoWriter(output_path, fourcc, fps, (frame_width, frame_height))

    while True:

        # Read a frame from the video
        ret, frame = cap.read()

        # Check if the frame was read successfully
        if not ret:
            break  # Video has ended

        # Convert frame from BGR to RGB
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        # Process frame and extract 3D coordinates
        objects_coordinates_3D, relevant_boxes = process_frame(frame, model_YOLO,
                                        model_MiDaS, transform_MiDaS,
                                        depth_fiducial1, depth_fiducial2,
                                        position_fiducial1, position_fiducial2,
                                        sensor_size, fov, model_SAM = model_SAM,
                                        confidence_YOLO = confidence_YOLO,
                                        bounding_box_iou_threshold = bounding_box_iou_threshold,
                                        device_YOLO = device_YOLO,
                                        device_MiDaS = device_MiDaS,
                                        relevant_class = relevant_class,
                                        scaling_factor = scaling_factor,
                                        num_clusters = num_clusters,
                                        fraction_of_pixels_threshold = fraction_of_pixels_threshold)

        # Draw on frame
        draw_on_frame(frame, objects_coordinates_3D, relevant_boxes, vertiport_x_limits, vertiport_z_limits,
                    draw_axes = draw_axes)

        # Display the resulting frame
        cv2.imshow('Frame', frame)

        # Write the frame to the output video file if provided
        if output_path:
            out.write(frame)

        # Press Q on keyboard to exit
        if cv2.waitKey(25) & 0xFF == ord('q'):
```

```
            break

    # Release the video capture object
    cap.release()

    # Release the output video file if provided
    if output_path:
        out.release()

############################## MAIN PROCESSING FUNCTION ##############################

def process_frame(frame, model_YOLO,
                  model_MiDaS, transform_MiDaS,
                  depth_fiducial1, depth_fiducial2,
                  position_fiducial1, position_fiducial2,
                  sensor_size, fov, model_SAM = None,
                  confidence_YOLO = 0.25, bounding_box_iou_threshold = 0.5, device_YOLO = 'cpu',
                  device_MiDaS = 'cpu', relevant_class = 14,
                  scaling_factor = 1.0, num_clusters = 3, fraction_of_pixels_threshold = 0.1,
                  show_depth_map = False):

    # To load a YOLO model use:
        # model_YOLO = YOLO(model_path)
        # E.g. model_YOLO = YOLO('YOLO/yolov8x.pt')
    # To load a YOLO-NAS model use:
        # model_YOLO = super_gradients.training.models.get(model_name,pretrained_weights='source').device()
        # E.g. model_YOLO = super_gradients.training.models.get('yolo_nas_l',pretrained_weights='coco').cpu()
    # To load a SAM model use:
        # model_SAM = sam_model_registry[model_type](checkpoint=model_path) AND  model_SAM.to(device=device)
        # E.g. model_SAM = sam_model_registry['vit_l'](checkpoint='sam_vit_l_16_224.pth') AND model_SAM.to(device='cpu')

    # Initialize segmentation performed flag
    segmentation_performed = False

    # Predict Bounding boxes using YOLO
    # Check if the model is YOLO-NAS model
    if 'yolo_nas' in str(type(model_YOLO)):
        # Run inference
        results_YOLO = model_YOLO.predict(frame, conf=confidence_YOLO)
        # Obtain bounding boxes
        prediction_objects = list(results_YOLO._images_prediction_lst)[0]
        boxes = prediction_objects.prediction.bboxes_xyxy
        # Obtain relevant boxes
        int_labels = prediction_objects.prediction.labels.astype(int)
        relevant_boxes = [box for box, class_name in zip(boxes, int_labels) if class_name == relevant_class]
    # Otherwise, the model is regular YOLO model
    else:
        # Run inference
        results_YOLO = model_YOLO.predict(frame, device=device_YOLO, conf=confidence_YOLO)
        # Obtain bounding boxes
        boxes = get_bounding_boxes(results_YOLO)
        # Obtain relevant boxes
        relevant_boxes = get_relevant_boxes(boxes, relevant_class)
        # check if the model is YOLO-SAM model
        if '-seg' in str(type(model_YOLO)):
            # Obtain relevant masks
            relevant_masks = get_relevant_masks_YOLO(results_YOLO, relevant_class)
            # Resize relevant masks to the original frame size
            relevant_masks = cv2.resize(relevant_masks, (frame.shape[1], frame.shape[0]),
                                        interpolation=cv2.INTER_NEAREST)
            # Set segmentation performed flag to True
            segmentation_performed = True
    # Use SAM model if segmentation has not been performed by YOLO and SAM model is provided
    if segmentation_performed == False and model_SAM != None:
        # Obtain relevant masks
        relevant_masks = get_relevant_masks_SAM(model_SAM, frame, relevant_boxes)
        # Set segmentation performed flag to True
        segmentation_performed = True

    # Check if segmentation has been performed
    if segmentation_performed == False:
        # If segmentation has not been performed by YOLO or SAM, set relevant masks to None
        relevant_masks = None

    # Check if there are relevant boxes to process, otherwise return the empty list and exit early
    if relevant_boxes != []:
        # Eliminate boxes that significantly overlap
        relevant_boxes = eliminate_overlapping_boxes(relevant_boxes, bounding_box_iou_threshold = bounding_box_iou_threshold)
        # Obtain the centers of the relevant boxes
        centers = []
        for box in relevant_boxes:
```

```
            centers.append(calculate_box_center(box))
    else:
        return [], []

    # Rescale the input frame to favour either more consistency on the scene structure (downscaling)
    # or on higher-frequency details (upscaling)
    if scaling_factor != 1.0:
        # Save the original frame size
        original_frame_size = frame.shape
        if scaling_factor < 1.0:
            # Select interpolation method suitable for downscaling
            interpolation = cv2.INTER_AREA
        else:
            # Select interpolation method suitable for upscaling
            interpolation = cv2.INTER_CUBIC
            # Upscale the frame
        frame = cv2.resize(frame, (int(frame.shape[1] * scaling_factor), int(frame.shape[0] * scaling_factor)),
                           interpolation = interpolation)

    # Get relative depth map using MiDaS
    relative_depth_map = predict_MiDaS(model_MiDaS, transform_MiDaS, frame, device = device_MiDaS)

    # Rescale the relative depth map to the original size
    if scaling_factor != 1.0:
        # If downscaling was used, use interpolation method suitable for upscaling
        if interpolation == cv2.INTER_AREA:
            interpolation = cv2.INTER_CUBIC
        # If upscaling was used, use interpolation method suitable for downscaling
        else:
            interpolation = cv2.INTER_AREA
        relative_depth_map = cv2.resize(relative_depth_map, (original_frame_size[1], original_frame_size[0]),
                                        interpolation=cv2.INTER_AREA)

    # Find the best positions for the fiducial markers
    position_fiducial1, position_fiducial2 = find_best_fiducial_positions(relative_depth_map,
                                                 position_fiducial1, position_fiducial2)

    # Obtain relative to real depth correspondace of fiducial markers
    fiducial1 = (depth_fiducial1, relative_depth_map[position_fiducial1[0], position_fiducial1[1]])
    fiducial2 = (depth_fiducial2, relative_depth_map[position_fiducial2[0], position_fiducial2[1]])

    # Construct metric depth map
    metric_depth_map = obtain_metric_depth_map(relative_depth_map, fiducial1, fiducial2)

    # Show depth map if requested
    if show_depth_map:
        plt.imshow(metric_depth_map)
        plt.colorbar()
        plt.show()

    # Obtain depths for relevant objects
    depths = get_depths(relevant_boxes, metric_depth_map, relevant_masks = relevant_masks,
                        num_clusters = num_clusters, fraction_of_pixels_threshold = fraction_of_pixels_threshold)

    # Get image size (x,y), so flip the order of the shape
    image_size = (metric_depth_map.shape[1], metric_depth_map.shape[0])

    # Obtain 3D coordinates where the origin is the sensor center
    objects_coordinates_3D = []
    for center, depth in zip(centers, depths):
        objects_coordinates_3D.append(calculate_3D_coordinates(sensor_size, image_size, fov, depth, center))

    # Return the obtained 3D coordinates and relevant boxes
    return objects_coordinates_3D, relevant_boxes

############################ ANCILLARY FUNCTIONS FOR FRAME PROCESSING ############################

def find_line_equation(point1, point2):

    # Find the equation of a line given two points
    x1, y1 = point1
    x2, y2 = point2

    # Calculate the slope (m)
    m = (y2 - y1) / (x2 - x1)

    # Calculate the y-intercept (b)
    b = y1 - m * x1

    # Return the equation of the line as a tuple (m, b)
    return m, b
```

```
def calculate_metric_depth_map(relative_depth_map, m, b):

    # Calculate the metric depth map from the relative depth map
    # and the equation of the line
    metric_depth_map = (relative_depth_map - b) / m

    # Return the metric depth map
    return metric_depth_map

def obtain_metric_depth_map(relative_depth_map, fiducial1, fiducial2):

    # Find the equation of the line where the points correspond to fiducial markers
    m, b = find_line_equation(fiducial1, fiducial2)

    # Calculate the metric depth map
    metric_depth_map = calculate_metric_depth_map(relative_depth_map, m, b)

    # Return the metric depth map
    return metric_depth_map

def pixels_to_metric_params(sensor_size, image_size):

    # Find equation to convert pixels to m
    # Sensor sizes should be in m (divede by 1000)
    point1 = (0,0)
    # Find equation for x
    point2 = (image_size[0],sensor_size[0]/1000)
    mx, bx = find_line_equation(point1, point2)
    # Find equation for y
    point2 = (image_size[1],sensor_size[1]/1000)
    my, by = find_line_equation(point1, point2)

    # Return parameters
    x_params = (mx, bx)
    y_params = (my, by)

    # Return parameters
    return x_params, y_params

def pixels_to_metric(sensor_size, image_size, coordinates):

    # Obtain parameters for pixels to metric conversion
    x_params, y_params = pixels_to_metric_params(sensor_size, image_size)

    # Extract parameters
    mx, bx = x_params
    my, by = y_params
    x, y = coordinates

    # Calculate coordinates in mm
    x_m = mx * x + bx
    y_m = my * y + by

    # Return coordinates in m
    return (x_m, y_m)

def calculate_3D_coordinates(sensor_size, image_size, fov, depth, coordinates):

    # Convert sensor size to m
    sx, sy = sensor_size[0]/1000, sensor_size[1]/1000

    # Convert coordinates from pixels to m
    coordinates = pixels_to_metric(sensor_size, image_size, coordinates)

    # Translate coordinates so that the center of the image is the origin
    coordinates_centered = (coordinates[0] - sx / 2,
                            sy / 2 - coordinates[1])

    # Convert degrees to radians
    fovx_rad = math.radians(fov[0])
    fovy_rad = math.radians(fov[1])

    # Get ratios
    rx = 2 * depth * math.tan(fovx_rad) / sx
    ry = 2 * depth * math.tan(fovy_rad) / sy

    # Calculate 3D coordinates
    coordinates_3D = (coordinates_centered[0] * rx, coordinates_centered[1] * ry, depth)

    # Return 3D coordinates
```

```
    return coordinates_3D

def calculate_box_center(box):

    # Extract coordinates
    x1, y1, x2, y2 = box

    # Calculate center
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2

    # Return center
    return (cx, cy)

def get_bounding_boxes(results_YOLO):

    # Obtain bounding boxes
    for result in results_YOLO:
        boxes = result.boxes.cpu().numpy()

    # Return bounding boxes
    return boxes

def get_relevant_boxes(boxes, class_number):

    # Initialize relevant boxes
    relevant_boxes = []

    # Check if there are any boxes
    if boxes == []:
        # If there are no boxes, return relevant_boxes = []
        return relevant_boxes

    # Check if any of the boxes correspond to the relevant class
    for box in boxes:
        if box.cls[0] == class_number:
            relevant_boxes.append(box.xyxy[0])

    # Return relevant boxes
    return relevant_boxes

def draw_on_frame(frame, objects_coordinates_3D, relevant_boxes, vertiport_x_limits, vertiport_z_limits, draw_axes = False):

    # Draw axes if requested
    if draw_axes:
        cv2.line(frame, (0, frame.shape[0] // 2), (frame.shape[1], frame.shape[0] // 2), (255, 255, 100), 4)
        cv2.line(frame, (frame.shape[1] // 2, 0), (frame.shape[1] // 2, frame.shape[0]), (255, 255, 100), 4)

    # Define parameter for drawing on frames
    # Define thickness for the bounding boxes
    box_thickness = 3
    # Define the font and scale for the text
    font = cv2.FONT_HERSHEY_SIMPLEX
    font_scale = 1
    font_thickness = 4
    font_color = (0, 0, 0)
    # Define status circle parameters
    center_coordinates = (30, 1000)
    radius = 20
    circle_thickness = -1

    # Initialize vertiport clear flag
    vertiport_clear = True

    # Loop through the bounding boxes if there exist relevant boxes
    if relevant_boxes != []:
        for box, coordinate_3D in zip(relevant_boxes, objects_coordinates_3D):
            # Extract corners of the bounding box
            x1, y1, x2, y2 = box
            # Convert corners to integers
            x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)
            # Draw a red bounding box on the frame if the object is within the vertiport space
            if (vertiport_x_limits[0] <= coordinate_3D[0] <= vertiport_x_limits[1]) and (vertiport_z_limits[0] <= coordinate_3D[2]
                                                                                         <= vertiport_z_limits[1]):
                cv2.rectangle(frame, (x1, y1), (x2, y2), (255,0,0), box_thickness)
                # Set vertiport clear flag to False
                vertiport_clear = False
            # Draw a green bounding box on the frame if the object is outside the vertiport space
            else:
                cv2.rectangle(frame, (x1, y1), (x2, y2), (0,255,0), box_thickness)
            # Add a white background to the text
```

```
            cv2.rectangle(frame, (x1, y1-40), (x1+400, y1-2), (255,255,255), -1)
            # Write the 3D coordinates of the object on the frame
            cv2.putText(frame,'('+str(round(coordinate_3D[0],2))+','+str(round(coordinate_3D[1],2))+','+
                        str(round(coordinate_3D[2],2))+')'+' m',
                        (x1, y1-5), font, font_scale, font_color, font_thickness)

    # Write 'Vertiport status:' message on the frame
    cv2.putText(frame,'Vertiport status:',(20,960), font, font_scale, font_color, font_thickness)
    # Indicate if the vertiport is clear for landing with a message and a green circle
    if vertiport_clear == True:
        # Draw a green circle on the image
        cv2.circle(frame, center_coordinates, radius, (0,255,0), circle_thickness)
        cv2.putText(frame,'CLEAR FOR LANDING',(60,1010), font, font_scale, font_color, font_thickness)
    # Indicate if the vertiport is not clear for landing with a message and a red circle
    else:
        # Draw a red circle on the image
        cv2.circle(frame, center_coordinates, radius, (255,0,0), circle_thickness)
        cv2.putText(frame,'NOT CLEAR FOR LANDING',(60,1010), font, font_scale, font_color, font_thickness)

    # Return the frame
    return frame

def find_best_fiducial_positions(relative_depth_map, position_fiducial1, position_fiducial2):

    # Extract the space covered by the fiducial markers. Images work in the (y,x) coordinate system
    # so the indices are reversed
    fiducial1 = relative_depth_map[position_fiducial1[2]:position_fiducial1[3], position_fiducial1[0]:position_fiducial1[1]]
    fiducial2 = relative_depth_map[position_fiducial2[2]:position_fiducial2[3], position_fiducial2[0]:position_fiducial2[1]]

    # Find the coordinates of the lowest value
    row1, col1 = np.where(fiducial1 == fiducial1.min())
    row2, col2 = np.where(fiducial2 == fiducial2.min())

    # Add the valid positions to the original positions to get the valid positions in the original map
    fiducial1_best_position = (position_fiducial1[2] + row1[0],
                               position_fiducial1[0] + col1[0])
    fiducial2_best_position = (position_fiducial2[2] + row2[0],
                               position_fiducial2[0] + col2[0])

    # Return the best fiducial positions (y,x)
    return fiducial1_best_position, fiducial2_best_position

def get_relevant_masks_YOLO(results_YOLO, relevant_class):

    for result in results_YOLO:
        # Check if there are any masks
        if result.masks is None:
            # Return a 2D array of zeros if there are no masks
            return np.zeros((2, 2))
        # Get the masks and bounding boxes (where the classes are stored)
        boxes = result.boxes.data
        masks = result.masks.data
        # Get the classes
        classes = boxes[:, 5]
        # Get the indices of the relevant class
        relevant_indices = torch.where(classes == relevant_class)
        # Extract the relevant boxes and masks
        relevant_masks = masks[relevant_indices]
        # Rescale mask
        relevant_masks = torch.any(relevant_masks, dim=0).int()
        # Convert the masks to numpy array
        relevant_masks = relevant_masks.cpu().numpy()

    # Return relevant boxes and masks
    return relevant_masks

def get_relevant_masks_SAM(model_SAM, frame, relevant_boxes):

    # Initialize mask predictor
    mask_predictor = SamPredictor(model_SAM)
    # Set image
    mask_predictor.set_image(frame)

    # Initialize relevant masks list
    relevant_masks_separated = []
    # Obtain relevant masks that best fit the relevant boxes
    for box in relevant_boxes:
        masks, scores, logits = mask_predictor.predict(
            box=box,
            multimask_output=True
        )
```

```
        # Get detections
        detections = sv.Detections(
            xyxy=sv.mask_to_xyxy(masks=masks),
            mask=masks
        )
        detections = detections[detections.area == np.max(detections.area)]
        # Extract mask from detections
        mask = detections.mask
        # Reshape masks to be able to operate on images and depth maps
        mask = mask.reshape(mask.shape[1], mask.shape[2])
        # Append mask to relevant masks list
        relevant_masks_separated.append(mask)

    # Combine all relevant masks into one
    relevant_masks = np.logical_or.reduce(relevant_masks_separated)

    # Return relevant masks
    return relevant_masks

def load_MiDaS(model_name, model_path = None, device = 'cpu', model_from_hub = False):

    # Load model
    # From local path
    if model_from_hub == False:
        if model_path == None:
            model_path = './MiDaS/weights/' + model_name + '.pt'
        model_MiDaS, _, _, _ = load_model(device, model_path, model_name)
    # From torch hub
    else:
        model_MiDaS = torch.hub.load('intel-isl/MiDaS', model_name)
        device = torch.device(device)
        model_MiDaS.to(device)
        model_MiDaS.eval()

    # Load transforms
    midas_transforms = torch.hub.load('intel-isl/MiDaS', 'transforms')
    # Select transforms
    # Check if the model is large or hybrid
    if 'arge' in model_name or 'ybrid' in model_name:
        transform_MiDaS = midas_transforms.dpt_transform
    else:
        transform_MiDaS = midas_transforms.small_transform

    # Return model and transform
    return model_MiDaS, transform_MiDaS

def predict_MiDaS(model, transform, image, device = 'cpu'):

    # Transform image
    input_batch = transform(image).to(device)

    # Predict
    with torch.no_grad():
        prediction = model(input_batch)

    # Transform prediction
    prediction = torch.nn.functional.interpolate(
        prediction.unsqueeze(1),
        size=image.shape[:2],
        mode='bicubic',
        align_corners=False,
    ).squeeze()

    # Prediction to numpy
    results_MiDaS = prediction.cpu().numpy()

    # Return prediction
    return results_MiDaS

def calculate_iou(box1, box2):

    # Extract coordinates
    x1A, y1A, x2A, y2A = box1
    x1B, y1B, x2B, y2B = box2

    # Calculate the coordinates of the intersection rectangle
    xA = max(x1A, x1B)
    yA = max(y1A, y1B)
    xB = min(x2A, x2B)
    yB = min(y2A, y2B)
```

```
    # Calculate the area of the intersection rectangle
    intersection_area = max(0, xB - xA) * max(0, yB - yA)

    # Calculate the areas of the individual bounding boxes
    areaA = (x2A - x1A) * (y2A - y1A)
    areaB = (x2B - x1B) * (y2B - y1B)

    # Calculate intersection over union (IoU)
    iou = intersection_area / (areaA + areaB - intersection_area)

    # Return IoU
    return iou

def eliminate_overlapping_boxes(boxes, bounding_box_iou_threshold = 0.5):

    # Sort the boxes by area (largest first)
    boxes.sort(key=lambda box: (box[2] - box[0]) * (box[3] - box[1]), reverse=True)

    # Initialize a list to store non-overlapping boxes
    non_overlapping_boxes = [boxes[0]]

    # Iterate through the sorted boxes
    for box in boxes[1:]:
        # Check IoU with the first box in the non-overlapping list
        iou = calculate_iou(non_overlapping_boxes[0], box)
        # If the IoU is below the threshold, add the box to the non-overlapping list
        if iou < bounding_box_iou_threshold:
            non_overlapping_boxes.append(box)

    # Return the non-overlapping boxes
    return non_overlapping_boxes

def get_depths(relevant_boxes, metric_depth_map, relevant_masks = None, num_clusters = 3, fraction_of_pixels_threshold = 0.1):

    # Initialize depths
    depths = []

    # Check if there are any relevant boxes
    if not relevant_boxes:
        return depths

    # Iterate through relevant boxes and calculate depths
    for box in relevant_boxes:

        # Extract the relevant region from the depth map
        relevant_region = metric_depth_map[round(box[1]):round(box[3]), round(box[0]):round(box[2])]

        if relevant_masks is not None:
            # Apply the relevant mask to the relevant region
            relevant_region = relevant_region * relevant_masks[round(box[1]):round(box[3]), round(box[0]):round(box[2])]

        # Flatten the relevant region into a 1D array and remove the zeros from the segmentation mask
        relevant_depth_values = relevant_region[relevant_region != 0].flatten()

        # Reshape the depth values for clustering
        relevant_depth_values = relevant_depth_values.reshape(-1, 1)

        # Use K-Means clustering to separate object depths
        kmeans = KMeans(n_clusters=num_clusters).fit(relevant_depth_values)

        # Obtain the cluster centers
        cluster_centers = kmeans.cluster_centers_
        # Obtain the cluster counts
        cluster_counts = np.bincount(kmeans.labels_)
        # Obtain valid clusters by checking if the fraction of pixels in the cluster is above the threshold, reducing the chance of picking
        # a cluster pertraining to an oclusion
        valid_clusters = [cluster_centers[i][0] for i in range(len(cluster_centers)) if (cluster_counts[i] / len(relevant_depth_values)) >=
                          fraction_of_pixels_threshold]
        # Obtain the object depth as the minimum of the valid clusters, therby reducing the chance of picking a cluster corresponding to
        # the background
        object_depth = min(valid_clusters)
        # Append the object depth to the depths list
        depths.append(object_depth)

    return depths

############################ CREATE VIDEO ###################################

def create_video_from_frames(frames_path, output_video_path = None, fps = 1):

    # If an output path for the video is not provided, save in the same path as the frames
```

```
    if output_video_path == None:
        output_video_path = frames_path

    # Get the resolution of the first frame
    first_frame = os.path.join(frames_path, '1.png')  # Assuming the first frame is named '1.png'
    frame = cv2.imread(first_frame)
    frame_height, frame_width, _ = frame.shape
    resolution = f"{frame_width}x{frame_height}"

    # FFMpeg command to create the video
    ffmpeg_command = [
        'ffmpeg',
        '-framerate', str(fps),
        '-pattern_type', 'glob',
        '-i', os.path.join(frames_path, '*.png'),
        '-c:v', 'libx265',
        '-s', resolution,
        output_video_path + '.mp4'
    ]

    # Run FFMpeg command to create the video
    subprocess.run(ffmpeg_command)

############################ DATASET CREATION FUNCTION ############################

def process_set(set_path, model_SAM, pickle_dataset = True, relative_depth = True, store_refined = True):

    images_path = os.path.join(set_path, 'SceneImages')
    depth_maps_path = os.path.join(set_path, 'DepthMaps')
    background_depth_map_path = os.path.join(set_path, 'BackgroundDepthMap.csv')
    relative_background_depth_map_path = os.path.join(set_path, 'RelativeBackgroundDepthMap.csv')
    ground_truth_boxes_path = os.path.join(set_path, 'annotations.json')

    # If storing refined depth maps, create the folder
    if store_refined:
        refined_depth_maps_path = os.path.join(set_path, 'RefinedDepthMaps')
        if not os.path.exists(refined_depth_maps_path):
            os.makedirs(refined_depth_maps_path)

    # Load MiDaS transforms
    midas_transforms = torch.hub.load('intel-isl/MiDaS', 'transforms')
    transform_MiDaS = midas_transforms.dpt_transform

    # Obtain Ground truth boxes in the correct format
    with open(ground_truth_boxes_path) as json_file:
        ground_truth_boxes = json.load(json_file)

    # Assuming convert_ground_truth_format is a predefined function
    bounding_boxes = convert_ground_truth_format(ground_truth_boxes)

    # Initialize lists for storing data
    images = []
    images_filenames = []
    transformed_images = []
    depth_maps = []

    # Filter out '.DS_Store' from the list of filenames
    images_filenames = [f for f in sorted(os.listdir(images_path)) if f != '.DS_Store']

    # Process each image in the folder
    for filename in images_filenames:
        if filename.startswith('SceneImage'):

            # Create the images path
            image_path = os.path.join(images_path, filename)

            # Load the image
            image = cv2.imread(image_path)
            # Convert image from BGR to RGB
            image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
            images.append(image)

            # Apply the MiDaS transform
            transformed_image = transform_MiDaS(image).to("cpu")
            transformed_images.append(transformed_image.squeeze(0))

    # Filter out '.DS_Store' from the list of filenames for depth maps
    depth_map_filenames = [f for f in sorted(os.listdir(depth_maps_path)) if f != '.DS_Store']

    # If a relative depth map is required, then we are processing a test or a validation set,
    #so we need to load the background depth map
```

```
    if relative_depth:
        # Open the background depth map and process it
        background_depth_map = np.loadtxt(background_depth_map_path, delimiter=',').T
        background_depth_map = np.flipud(np.where(background_depth_map == -1, np.nan, background_depth_map))
        # Open the relative background depth map (already pre-processed)
        relative_background_depth_map = np.loadtxt(relative_background_depth_map_path, delimiter=',')

        # Exract the minimum and maximum depth values from the relative background depth map
        d_target_min = relative_background_depth_map.min()
        d_target_max = relative_background_depth_map.max()
    else:
        background_depth_map = None

    # Process each depth map in the folder
    for filename, image, image_filename in zip(depth_map_filenames, images, images_filenames):
        if filename.startswith('DepthMap'):

            # Create the depth maps path
            depth_map_path = os.path.join(depth_maps_path, filename)

            # Open the rough depth maps and process them
            rough_depth_map = np.loadtxt(depth_map_path, delimiter=',').T
            rough_depth_map = np.flipud(np.where(rough_depth_map == -1, np.nan, rough_depth_map))

            # Create the refined depth map
            depth_map = create_refined_depth_map(image, rough_depth_map, bounding_boxes[image_filename], model_SAM,
            background_depth_map)

            # Convert the metric depth map into a relative one using the minimum and maximum depth values from the background depth map,
            # to make it comparable with the MiDaS depth maps
            if relative_depth:
                depth_map = transform_depth_map(depth_map, d_target_min, d_target_max)

            # Append the depth map to the list
            depth_maps.append(depth_map)

            # Save the refined depth map if required
            if store_refined:
                np.savetxt(set_path + "/RefinedDepthMaps/" + filename, depth_map, delimiter=',')

    # Convert images and depth maps into properly shaped tensors
    transformed_images_t = torch.stack(transformed_images)
    depth_maps_t = torch.tensor(depth_maps).unsqueeze(1) # Add a channel dimension to make it match with the images

    # Create a dataset
    dataset = torch.utils.data.TensorDataset(transformed_images_t.float(), depth_maps_t.float())

    # Save the dataset if required
    if pickle_dataset:
        dataset_filename = os.path.join(set_path, 'dataset.pkl')
        with open(dataset_filename, 'wb') as f:
            pickle.dump(dataset, f)

    # Return the dataset
    return dataset

############################ ANCILLARY FUNCTIONS FOR DATASET CREATION ############################

def calculate_transformation_coefficients(d_min, d_max, d_target_min, d_target_max):

    # Calculate the transformation coefficients
    a = (d_target_min - d_target_max) / (d_max - d_min)
    b = d_target_max - a * d_min

    # Return the coefficients
    return a, b

def transform_depth_map(depth_map, d_target_min, d_target_max):

    # Calculate the minimum and maximum depth values of the depth map
    d_min = depth_map.min()
    d_max = depth_map.max()

    # Calculate the transformation coefficients
    a, b = calculate_transformation_coefficients(d_min, d_max, d_target_min, d_target_max)

    # Transform the depth map
    transformed_depth_map = depth_map * a + b

    # Return the transformed depth map
    return transformed_depth_map
```

```
def create_refined_depth_map(frame, rough_depth_map, bounding_boxes, model_SAM, background_depth_map = None):

    # Obtain relevant masks
    relevant_masks = get_relevant_masks_SAM(model_SAM, frame, bounding_boxes)

    # Refine depth map using the segmentation masks
    refined_depth = np.where(relevant_masks, rough_depth_map, 0)

    for box in bounding_boxes:
        # Extract the box coordinates
        x1, y1, x2, y2 = box
        x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)
        # Extract the section of the depth map corresponding to the box
        section = refined_depth[y1:y2, x1:x2]
        # Filter out zeros
        non_zero_section = section[section != 0]
        # Find most common non-zero value in the section
        values, counts = np.unique(non_zero_section, return_counts=True)
        most_common_non_zero_value = values[np.argmax(counts)]
        # Replace non-zero values in the section with the most common non-zero value
        section[section != 0] = most_common_non_zero_value
        # Replace the section in the refined depth map
        refined_depth[y1:y2, x1:x2] = section

    # Show the filtered depth map
    if background_depth_map is not None:
        # Replace zeros in the filtered depth map with the background depth map values
        refined_depth = np.where(refined_depth == 0, background_depth_map, refined_depth)

    return refined_depth

def convert_ground_truth_format(dataset_ground_truth_boxes):

    # Initialize the dictionary to be returned
    converted_dataset_ground_truth_boxes = {}

    # Convert the format of ground truth boxes to a dictionary whose values are lists of Numpy arrays in xyxy format
    for image_name, boxes in dataset_ground_truth_boxes.items():
        converted_boxes = []
        for box in boxes:
            if box["width"] > 0 and box["height"] > 0:  # Filter out invalid boxes
                x1, y1 = box["x"], box["y"]
                x2, y2 = x1 + box["width"], y1 + box["height"]
                converted_box = np.array([x1, y1, x2, y2], dtype=np.float32)
                converted_boxes.append(converted_box)
        converted_dataset_ground_truth_boxes[image_name] = converted_boxes

    # Return the converted dictionary
    return converted_dataset_ground_truth_boxes
```

8.5. Evaluation Python script – Evaluation.py

```
from VertiportSurveillance import *

from collections import defaultdict

############################ MAIN EVALUATION FUNCTION ############################

def agregate_predictions_and_calculate_metrics(set_path, model_YOLO,
                                model_MiDaS, transform_MiDaS, depth_fiducial1, depth_fiducial2, position_fiducial1,
                                position_fiducial2, sensor_size, fov,
                                iou_matching_threshold = 0.5, model_SAM = None, confidence_YOLO = 0.25,
                                bounding_box_iou_threshold = 0.5, device_YOLO = 'cpu', device_MiDaS = 'cpu',
                                relevant_class = 14, scaling_factor = 1.0, num_clusters = 3,
                                fraction_of_pixels_threshold = 0.1):

    # Obtain the file names of the frames and their corresponding depth maps
    frame_names = sorted(f for f in os.listdir(set_path + "/SceneImages/") if f.startswith("SceneImage"))
    depth_map_names = sorted(f for f in os.listdir(set_path + "/RefinedDepthMaps/") if f.startswith("DepthMap"))

    # Open the json file containing the ground truth boxes
    with open(set_path + "/annotations.json") as json_file:
        dataset_ground_truth_boxes = json.load(json_file)

    # Convert the format of ground truth boxes to a dictionary whose values are lists of tuples in xyxy format
    dataset_ground_truth_boxes = convert_ground_truth_format(dataset_ground_truth_boxes)
```

```
    # Initialize counts and array for metric calulations
    total_true_positives = 0
    total_false_positives = 0
    total_false_negatives = 0
    all_depth_errors = []
    time_to_process_frames = 0

    # Initialize percent counter
    i = 0
    # Calculate the increment for each 10% to print back to the user
    increment = len(frame_names) / 10
    next_threshold = increment
    # Inform the user the process is about to begin
    print('BEGINNING EVALUATION!')

    for frame_name, depth_map_name  in  zip(frame_names, depth_map_names):

        # Load frame
        frame = cv2.imread(set_path + "/SceneImages/" + frame_name)
        # Convert frame from BGR to RGB
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        # Load the refined depth map (pre-transposed and flipped)
        refined_depth_map = np.loadtxt(set_path + "/RefinedDepthMaps/" + depth_map_name, delimiter=',')

        # Extract ground truth boxes corresponding to the current image
        ground_truth_boxes = dataset_ground_truth_boxes[frame_name]

        # Obtain the counts and error for the frame
        true_positives, false_positives, false_negatives, depth_errors, time_to_process_frame = evaluate_prediction(
            frame, refined_depth_map, ground_truth_boxes, model_YOLO,
            model_MiDaS, transform_MiDaS, depth_fiducial1, depth_fiducial2, position_fiducial1, position_fiducial2,
            sensor_size, fov, iou_matching_threshold = iou_matching_threshold, model_SAM = model_SAM,
            confidence_YOLO = confidence_YOLO, bounding_box_iou_threshold = bounding_box_iou_threshold, device_YOLO = device_YOLO,
            device_MiDaS = device_MiDaS, relevant_class = relevant_class, scaling_factor = scaling_factor, num_clusters = num_clusters,
            fraction_of_pixels_threshold = fraction_of_pixels_threshold)

        # Updates counts, append depth errors, and update processing time
        total_true_positives += true_positives
        total_false_positives += false_positives
        total_false_negatives += false_negatives
        all_depth_errors.extend(depth_errors)
        time_to_process_frames += time_to_process_frame

        # Update percent counter if the processed percentage surpasses a multiple of 10%
        if i >= next_threshold:
            percent_complete = (i) / len(frame_names) * 100
            print(f"{percent_complete:.0f}% done")
            # Update the next threshold to the next 10% increment
            next_threshold += increment
        # Update the percent counter
        i += 1

    # Calculate global precision, recall, and F1 score
    precision = total_true_positives / (total_true_positives + total_false_positives) if total_true_positives + total_false_positives > 0 else 0
    recall = total_true_positives / (total_true_positives + total_false_negatives) if total_true_positives + total_false_negatives > 0 else 0
    f1_score = 2 * (precision * recall) / (precision + recall) if precision + recall > 0 else 0

    # Calculate global mean squared error for depth estimation
    global_MAE = np.mean(all_depth_errors) if all_depth_errors else None

    # Calculate the total number of objects evaluated
    total_objects_evaluated = total_true_positives + total_false_negatives

    # Inform the user the process has finished
    print("ALL DONE!")

    # Return the metrics and the total number of objects that were evaluated
    return {
        'YOLO precision': precision,
        'YOLO recall': recall,
        'YOLO F1': f1_score,
        'MiDaS MAE (m)': global_MAE,
        'Time to process frames (min)': time_to_process_frames/60,
        'Average frame processing time (s)': time_to_process_frames/len(frame_names),
        'Number of frames evaluated': len(frame_names),
        'Number of objects evaluated': total_objects_evaluated
    }

############################ EVALUATE SINGLE PREDICTION ############################
```

```
def evaluate_prediction(frame, refined_depth_map, ground_truth_boxes, model_YOLO,
                model_MiDaS, transform_MiDaS, depth_fiducial1, depth_fiducial2, position_fiducial1, position_fiducial2,
                sensor_size, fov, iou_matching_threshold=0.5, model_SAM=None, confidence_YOLO=0.25,
                bounding_box_iou_threshold = 0.5, device_YOLO = 'cpu', device_MiDaS = 'cpu', relevant_class = 14,
                scaling_factor = 1.0, num_clusters = 3, fraction_of_pixels_threshold = 0.1):

  # Record the start time
  start_time = time.time()

  # Process frame and extract 3D coordinates
  objects_coordinates_3D, predicted_boxes = process_frame(frame, model_YOLO, model_MiDaS, transform_MiDaS,
                                    depth_fiducial1, depth_fiducial2, position_fiducial1, position_fiducial2,
                                    sensor_size, fov, model_SAM = model_SAM, confidence_YOLO = confidence_YOLO,
                                    bounding_box_iou_threshold = bounding_box_iou_threshold, device_YOLO = device_YOLO,
                                    device_MiDaS = device_MiDaS, relevant_class = relevant_class,
                                    scaling_factor = scaling_factor, num_clusters = num_clusters,
                                    fraction_of_pixels_threshold = fraction_of_pixels_threshold)

  # Record the end time
  end_time = time.time()

  # Calculate time to process frame
  time_to_process_frame = end_time - start_time

  # Match predictions with ground truth using IoU
  matches = match_predictions_with_ground_truth(predicted_boxes, ground_truth_boxes, iou_threshold = iou_matching_threshold)

  # Initialize counts
  true_positives = 0
  depth_errors = []

  for pred_idx, gt_matches in matches.items():
    if gt_matches:
      # Choose the ground truth box with the highest IoU for this prediction
      matched_gt_idx = max(gt_matches, key=gt_matches.get)
      true_positives += 1

      # Get the coordinates of the matched ground truth box
      gt_box = ground_truth_boxes[matched_gt_idx]

      # Calculate true depth value for this ground truth box
      x1, y1, x2, y2 = map(int, gt_box)  # Convert box coordinates to integers
      section = refined_depth_map[y1:y2, x1:x2]

      # Calculate the most common value in the section of the depth map corresponding to the ground truth box
      non_zero_elements = section[section != 0] # Remove zero values
      if non_zero_elements.size > 0:
        # Process ground truth depth value
        unique_elements, counts = np.unique(non_zero_elements, return_counts=True) # Count unique elements
        max_count_index = np.argmax(counts) # Find the index of the most common value
        most_common_value = unique_elements[max_count_index] # Obtain the most common value
        # Process predicted depth value
        predicted_depth = objects_coordinates_3D[pred_idx][2]
        # Calculate the error between the predicted and true depth values
        depth_errors.append(abs(most_common_value - predicted_depth))

  # Calculate false positives and false negatives
  false_positives = len(predicted_boxes) - true_positives
  false_negatives = len(ground_truth_boxes) - true_positives

  # Return the counts and errors for the frame
  return true_positives, false_positives, false_negatives, depth_errors, time_to_process_frame

########################### ANCILLARY FUNCTION ###########################

def match_predictions_with_ground_truth(predicted_boxes, ground_truth_boxes, iou_threshold = 0.5):

  # Initialize the dictionary to be returned
  matches = defaultdict(dict)

  # Return a dictionary with matches between predicted and ground truth boxes, along with their IoU
  # (i.e. {predicted box index: {ground truth box index: IoU}})
  for i, pred_box in enumerate(predicted_boxes):
    for j, gt_box in enumerate(ground_truth_boxes):
      iou = calculate_iou(pred_box, gt_box)
      if iou >= iou_threshold:
        matches[i][j] = iou

  # Return matches
  return matches
```

8.6. Model training Python script – FineTune.py

```
from VertiportSurveillance import *

import torch.nn as nn
from torch.utils.data import DataLoader
from torch.nn.functional import l1_loss
import torch.nn.functional as F

########################### FINE TUNING FUNCTION ###########################

def train_model(model, train_dataset, valid_dataset, device, optimizer,
            loss_function, batch_size = 5, num_epochs = 100, patience = 10,
            output_filename = 'MiDaS/weights/MiDair.pt', pickle_results = False,
            pickle_filename = 'MiDair_results.pkl', resume_training_from_checkpoint = False):

    # Load data
    train_loader = DataLoader(train_dataset, batch_size = batch_size, shuffle = True)
    valid_loader = DataLoader(valid_dataset, batch_size = batch_size)

    # Set model to training mode
    model.train()

    # Initialize lists to contain losses and MAE
    train_losses = []
    train_mae = []
    valid_losses = []
    valid_mae = []

    # Check if there is a pickle file to resume training
    if resume_training_from_checkpoint and os.path.isfile(pickle_filename):
        with open(pickle_filename, 'rb') as file:
            data = pickle.load(file)
            train_losses, train_mae, valid_losses, valid_mae = data.values()

    # Initialize patience counter
    patience_counter = 0

    print('Training has started!')

    # Training and validation loop
    for epoch in range(num_epochs):

        # Record start time
        start_time = time.time()

        # Initialize for training
        train_total_loss = 0.0
        train_total_mae = 0.0
        train_count = 0

        # Train over batches
        for inputs, targets in train_loader:

            inputs, targets = inputs.to(device), targets.to(device)

            # Forward pass
            outputs = model(inputs)

            # Add a channel dimension to the model output
            outputs = outputs.unsqueeze(1)

            # Resize the model's output to match the target depth map size
            outputs = F.interpolate(outputs, size = (targets.size(2), targets.size(3)), mode = 'bilinear', align_corners = False)

            # Compute the loss
            loss_value = loss_function(outputs, targets)

            # Accumulate the total loss
            train_total_loss += loss_value.item() * inputs.size(0)

            # Compute and accumulate MAE
            train_total_mae += l1_loss(outputs, targets, reduction = 'mean').item()
            train_count += inputs.size(0)

            # Backpropagate the error to change the model weights
            optimizer.zero_grad()
            loss_value.backward()
            optimizer.step()
```

```
# Calculate average train loss and MAE
train_avg_loss = train_total_loss / train_count
train_avg_mae = train_total_mae / train_count
train_losses.append(train_avg_loss)
train_mae.append(train_avg_mae)

#_____________________________________________________________________

# Initialize for validation
valid_total_loss = 0.0
valid_total_mae = 0.0
valid_count = 0

# Validate over batches
model.eval()  # Set the model to evaluation mode
with torch.no_grad():
    for inputs, targets in valid_loader:

        inputs, targets = inputs.to(device), targets.to(device)

        # Forward pass
        outputs = model(inputs)

        # Add a channel dimension to the model output
        outputs = outputs.unsqueeze(1)

        # Resize the model's output to match the target depth map size
        outputs = F.interpolate(outputs, size = (targets.size(2), targets.size(3)), mode = 'bilinear', align_corners = False)

        # Compute the loss
        loss_value = loss_function(outputs, targets)

        # Accumulate the total loss
        valid_total_loss += loss_value.item() * inputs.size(0)

        # Compute and accumulate MAE
        valid_total_mae += l1_loss(outputs, targets, reduction = 'mean').item()
        valid_count += inputs.size(0)

# Calculate average validation loss and MAE
valid_avg_loss = valid_total_loss / valid_count
valid_avg_mae = valid_total_mae / valid_count
valid_losses.append(valid_avg_loss)
valid_mae.append(valid_avg_mae)

#_____________________________________________________________________

# Record finish time
finish_time = time.time()
elapsed = finish_time - start_time

# Print epoch results
print(f'Epoch {epoch+1}/{num_epochs} - Elapsed time: {int(elapsed)} seconds')
print(f'    Train Loss: {train_avg_loss} - Train MAE: {train_avg_mae}')
print(f'    Valid Loss: {valid_avg_loss} - Valid MAE: {valid_avg_mae}')

#_____________________________________________________________________

# Save the best model
if epoch == 0 and resume_training_from_checkpoint == False or valid_avg_loss < min(valid_losses[:-1]):
    torch.save(model.state_dict(), output_filename)
    print('    Best model so far, saved!')
    patience_counter = 0

    # Pickle results if required
    if pickle_results:
        data_to_pickle = {
            "train_losses": train_losses,
            "train_mae": train_mae,
            "valid_losses": valid_losses,
            "valid_mae": valid_mae
        }
        with open(pickle_filename, 'wb') as file:
            pickle.dump(data_to_pickle, file)

else:
    patience_counter += 1

# Check patience
if patience_counter == patience:
```

```
            print(f'    The validation loss has not improved in {patience} epochs.')
            break

        #_____________________________________________________________________

    print('Finished!')

    return train_losses, train_mae, valid_losses, valid_mae

############################ Losses ############################

class BerHuLoss(nn.Module):

    def __init__(self, m = 0.2):

        super(BerHuLoss, self).__init__()
        self.m = m

    def forward(self, predictions, targets):

        # Calculate the absolute difference
        diff = torch.abs(targets - predictions)
        # Calculate the threshold c as m times the max error in the batch
        c = self.m * torch.max(diff).item()
        # Condition for selecting L1 or L2 loss
        condition = diff <= c
        # Apply L1 loss
        l1_loss = diff
        # Apply L2 loss
        l2_loss = (diff**2 + self.m**2 * torch.max(diff).item()**2) / (2 * self.m * torch.max(diff).item())
        # Combine the losses based on the condition
        loss = torch.where(condition, l1_loss, l2_loss)

        # Return the mean loss
        return torch.mean(loss)

class LtrimLregLoss(nn.Module):

    def __init__(self, trim_ratio=0.2, alpha=0.5):

        super(LtrimLregLoss, self).__init__()
        self.trim_ratio = trim_ratio
        self.alpha = alpha

    def forward(self, predictions, targets):

        # Compute trimmed MAE loss
        diff = torch.abs(targets - predictions)
        diff_flat = diff.view(-1) # Flatten the tensor
        num_to_trim = int(self.trim_ratio * diff_flat.size(0)) # Compute number of elements to trim
        diff_trimmed = torch.sort(diff_flat).values[num_to_trim:] # Sort the tensor and trim the elements
        trim_loss = diff_trimmed.mean() # Compute the mean of the trimmed tensor

        # Compute regularization loss
        reg_loss = 0.0
        for scale in [1, 2, 4, 8, 16]:  # Multi-scale gradients
            scaled_predictions = F.interpolate(predictions, scale_factor=1/scale, mode='bilinear', align_corners=False)
            scaled_targets = F.interpolate(targets, scale_factor=1/scale, mode='bilinear', align_corners=False)
            pred_grad_x, pred_grad_y = compute_gradients(scaled_predictions)
            target_grad_x, target_grad_y = compute_gradients(scaled_targets)
            reg_loss += F.l1_loss(pred_grad_x, target_grad_x) + F.l1_loss(pred_grad_y, target_grad_y)

        # Combine losses
        total_loss = trim_loss + self.alpha * reg_loss

        # Return the total loss
        return total_loss


############################ ANCILLARY FUNCTION ############################


def compute_gradients(array):

    # Assuming array is a 4D tensor: (batch_size, channels, height, width)
    # Create gradient kernel
    d = torch.tensor([[-1., 0., 1.]], device=array.device, dtype=torch.float)
    d = d.repeat(array.shape[1], 1, 1, 1) # Create a stack of kernels for each channel

    # Compute horizontal and vertical gradients
    horizontal_grad = F.conv2d(array, d, padding=(0, 1))
```

```
    vertical_grad = F.conv2d(array, d.transpose(2, 3), padding=(1, 0))

    # Return the gradients
    return horizontal_grad, vertical_grad
```

8.7. Image annotation Jupyter Notebook – ManualBbox.ipynb

```
import os
import json
import base64
import ipywidgets as widgets
from jupyter_bbox_widget import BBoxWidget
from IPython.display import display

def encode_image(filepath):
    with open(filepath, 'rb') as f:
        image_bytes = f.read()
    encoded = str(base64.b64encode(image_bytes), 'utf-8')
    return ""+encoded

path = 'a'#'TestSet/SceneImages'
files = sorted(os.listdir(path))

annotations = {}
annotations_path = 'a/annotations.json' #'TestSet/annotations.json'

# a progress bar to show how far we got
w_progress = widgets.IntProgress(value=0, max=len(files), description='Progress')
# the bbox widget
w_bbox = BBoxWidget(
    image = encode_image(os.path.join(path, files[0]))
)

# combine widgets into a container
w_container = widgets.VBox([
    w_progress,
    w_bbox,
])

# when Skip button is pressed we move on to the next file
@w_bbox.on_skip
def skip():
    w_progress.value += 1
    # open new image in the widget
    image_file = files[w_progress.value]
    w_bbox.image = encode_image(os.path.join(path, image_file))
    # here we assign an empty list to bboxes but
    # we could also run a detection model on the file
    # and use its output for creating inital bboxes
    w_bbox.bboxes = []

# when Submit button is pressed we save current annotations
# and then move on to the next file
@w_bbox.on_submit
def submit():
    image_file = files[w_progress.value]
    # save annotations for current image
    annotations[image_file] = w_bbox.bboxes
    with open(annotations_path, 'w') as f:
        json.dump(annotations, f, indent=4)
    # move on to the next file
    skip()

w_container
```