ICSC
Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

HaMMon

WP2 Contribution to Milestone **M8**
Deliverable **TAR 2.2**

Algorithm for creation of digital twin from UAV

Date: June 30th, 2024

Partners:	INAF - Osservatorio Astrofisico di Catania
Work-Package:	WP2
Milestone:	M8
Responsible partner:	INAF - Osservatorio Astrofisico di Catania
Author(s):	Leonardo Pelonero, Fabio Roberto Vitello, Mauro Imbrosciano

Finanziato dall'Unione europea
NextGenerationEU

Italiadomani
PIANO NAZIONALE DI RIPRESA E RESILIENZA

MUR

Document revisions		
Author(s)	***Revision content***	***Date***
Leonardo Pelonero	First draft of the document	25/06/2024
Mauro Imbrosciano	Codes, images and notes on monitoring contribution	28/06/2024
Fabio Roberto Vitello	Final Revision and formatting	30/06/2024

ICSC
Centro Nazionale di Ricerca in HPC,
Big Data and Quantum Computing

HaMMon

ABSTRACT

The report presents the final script release for the creation of digital twins utilising UAV (Unmanned Aerial Vehicle) photogrammetry techniques. It provides functionalities for processing aerial imagery, generating 3D models, and creating digital representations of real-world environments. This release strategically divides the fundamental process into individual tasks, facilitating more efficient project management and control.

Finanziato dall'Unione europea
NextGenerationEU

Italiadomani
PIANO NAZIONALE DI RIPRESA E RESILIENZA

MUR

TABLE OF CONTENTS

1. INTRODUCTION 6
Definition of Digital Twin 6
Photogrammetric survey process 6
Agisoft Metashape: SfM software 8
    Install external modules in Metashape built-in python environment 9
    Install Metashape stand-alone Python module 9
2. Implemented code 10
StepWorkflow 13
ProgressPrinter 14
Settings 14
Project 15
PhotoProcessor 16
PointCloudProcessor 19
MeshProcessor 21
GeographicProjection 23
Saving and exporting the results 26
SystemMonitor 26
3. Credits and References 30

Finanziato dall'Unione europea NextGenerationEU
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
MUR

LIST OF ACRONYMS and ABBREVIATIONS

CLI: Command Line Interface
CPU: Central Processing Unit
DEM: Digital Elevation Models
GIS: Geographic Information System
GPU: Graphics Processing Unit
HaMMon: Hazard Mapping and vulnerability Monitoring
INAF: Istituto Nazionale di Astrofisica
LiDAR: Light Detection and Ranging
SfM: Structure from Motion
UAV: Unmanned Aerial Vehicle
WP: Work Package

1. INTRODUCTION

The most recent innovations in the field of 3D modelling and virtual reality have led to the development of new tools and technologies to manage and interpret this particularly complex data. Above all, in the field of topographic and environmental surveys, the use of tools for the generation and visualisation of 3D models is becoming increasingly popular. The aim is to maintain the same quality guaranteed by traditional and established measurement techniques, but with shorter timeframes. This significantly cuts down the cost of a conventional site survey and lightens the specialists workload.

Definition of Digital Twin

Building on these advancements, the concept of the Digital Twin has emerged as a powerful tool that further extends the capabilities of 3D modelling. A Digital Twin is a virtual representation of a physical object, process, or system, which can be used to monitor and analyse its real counterpart's behaviour. This allows for simulations, predictions, and optimizations of the functions of the physical entity, thereby enhancing the quality of topographic and environmental surveys.

These models represent a significant step forward in the field of data analysis for various case studies and key technologies: environmental monitoring, urban planning, land management, construction, and infrastructure. They allow us to reflect the planimetry, its current condition, and its future characteristics; providing new insights and extensive applications in various industries such as manufacturing and urban planning. By analysing historical data, it is possible to assess maintenance requirements (e.g., the detection and management of possible anomalies caused by floods and weather events), thus enabling real-time monitoring and decision-making.

In the context of site surveys, a Digital Twin can provide a comprehensive, interactive 3D model of the site, enabling specialists to conduct detailed analyses without the need for time-consuming and costly physical site visits. This not only reduces costs and saves time but also allows for more frequent and detailed analyses, leading to more accurate and reliable results.

Photogrammetric survey process

Following the explanation of the digital twin, it's important to note the techniques available for creating 3D models. Based on the needs and requests of the HaMMon project, there are two most effective techniques, for geomorphological applications, that offer a comprehensive approach to

capturing and modelling the physical world in three dimensions: LiDAR and photogrammetric processes.

Light Detection and Ranging (LiDAR) technology, is a terrestrial and airborne laser scanning method that uses light in the form of a pulsed laser to measure distances to an object. These light pulses, combined with other data recorded by the airborne system, generate precise, three-dimensional information about the shape of the Earth and its surface characteristics.

On the other hand, photogrammetry is a technique that uses multi-view stereo images from different angles (along with internal calibration information) to create precise 3D models of real-world objects or scenes and camera orientations/poses in a common 3D coordinate system. Figure 1 shows a series of overlapping photos of an object, each from a slightly different viewpoint, and then using software to turn the photos into a 3D model.

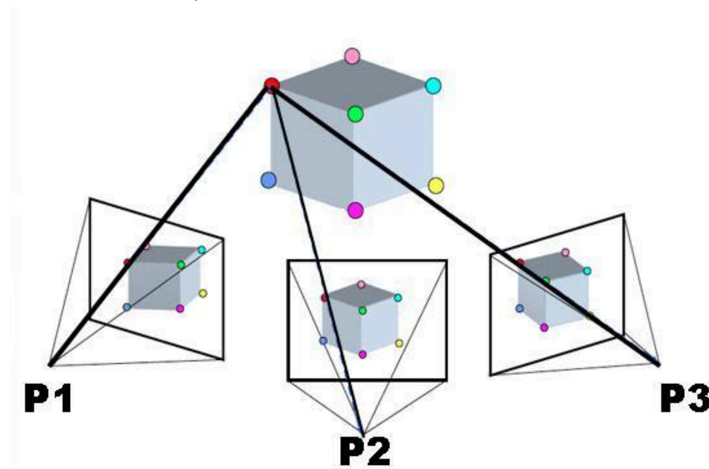

Figure 1: From point observation and intrinsic camera parameters, the 3D structure of the scene is computed from the estimated motion of the camera.

Structure from Motion (SfM) is a photogrammetric range imaging technique for estimating hyper-scale three-dimensional structures from two-dimensional image sequences that may be coupled with local motion signals. It involves the recovery of the spatial layout of visible surfaces in the scene and the camera motion with geo-referencing information.

There are numerous approaches to estimate the SfM from multiple images. Thanks to recent progress in image matching and optimization, it is now possible to compute large scale 3D reconstruction from millions of images on a reasonably sized cluster.

Most current SfM pipelines are sequential and can be briefly described as follows. In the first step features are detected in each image and matched between overlapping frames using algorithms like SIFT (Lowe, 2004). These corresponding image points are then used to iteratively reconstruct

Finanziato dall'Unione europea NextGenerationEU
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
MUR

the geometry of the image network through many bundle adjustment steps (Snavely et al. 2006). This stage involves estimating intrinsic camera parameters, which detail the camera's internal geometry (like focal length and principal point), and extrinsic parameters, which define the camera's position (three translations) and orientation (three rotations) during image capture.

These estimations have a crucial impact on the quality of end 3D reconstruction. 3D object coordinates are derived from the 2D image coordinates of these matched points, forming a sparse point cloud. Armed with the geometry of the image network, a dense point cloud is generated, assigning a 3D point to nearly every pixel in the image. The final 3D point cloud can be georeferenced during the adjustment phase, incorporating additional data to refine both intrinsic and extrinsic camera parameters, or post-adjustment through a similarity transformation, which does not allow for further optimization of the adjustment (Remondino et al. 2014).

In essence, SfM is a cost-effective method that uses only camera images to rebuild a 3D scene while also obtaining the camera poses of the monocular camera in relation to the provided scene. Despite the challenges associated with resilience, precision, completeness, and scalability, SfM is a flexible established method widely applied.

Agisoft Metashape: SfM software

Agisoft Metashape (2.1.2v), in combination with UAV (Unmanned Aerial Vehicle) systems, emerges as a suitable solution for this case study. Metashape is an advanced and versatile photogrammetry proprietary software for creating Digital Twins which can be used in Geographic Information System (GIS) applications. The application allows, starting from raw aerial images (both in controlled and uncontrolled conditions), to process: spherical panoramas, orthophotos, 3D point clouds, digital surface models, and digital elevation models (DEM).

The pros of Metashape include an intuitive GUI, a significant amount of advanced user control for increasing camera optimization (if using a reference system), editing masks, and the points clouds themselves, as well as working with multiple “chunks” of a single project. Above all, the positive aspect is that most of the processes are fully automated, operating in a ‘black box’ mode. The process involves several steps: feature extraction, feature matching, 3D reconstruction, and optional meshing and texturing from the point cloud. The goal of the reconstruction is to derive the geometrical structure of a scene from a set of photos, assuming that the camera position and internal parameters are known or can be guessed from the set of images, an example result is shown on Fig. 2.

Agisoft offers multiple licensing options (stand-alone, floating, and educational) with price-points depending on the licence choice. The professional edition, which is the full version, was used in

this research. The professional edition is also required for saving projects in PSX format, supporting headless operation, integrating the Python module, and utilising built-in Python scripting.

Install external modules in Metashape built-in python environment

Following the official guide, it is possible to install an external Python module to Metashape. It depends on the OS used and the name of the Python module to be installed.

For Linux:

```
./metashape-pro/python/bin/python3.9 -m pip install python_module_name
```

Install Metashape stand-alone Python module

The stand-alone Metashape Python module allows to integrate Metashape functionality to the custom scripts without a need of full application installation. It depends on the OS used and requires Python 3.x version installed on the system. After the download of the .whl package, it can be installed as a normal pip package.

For Linux:

```
python3 -m pip install Metashape-2.1.2-cp37.cp38.cp39.cp310.cp311-abi3-linux_x86_64.whl
```

The latest version of Metashape package is already available in the assets folder of the repository.

Figure 2: Example of digital twin obtained from the sample data provided by Metashape.

2. Implemented code

The task T2.1 "Workflow for data acquisition and creation of digital twin" is developing a workflow to produce a high-resolution 3D tiled model with centimetre accuracy of the affected area from 2D images captured along proper UAV flight paths, combining them to create a sparse and dense point cloud, creating a mesh, and generating a texture.

In the previous milestone 7, a draft of the entire execution workflow was implemented in Python. Now, we have adapted the project with an intense code refactor that has changed the structure of the code. The achieved goal is to ensure the ability to execute any step of the process without having to rerun the entire time consuming workflow. The structure of the code is represented here by the UML diagram in Fig. 3, the details of each class will be explained below.

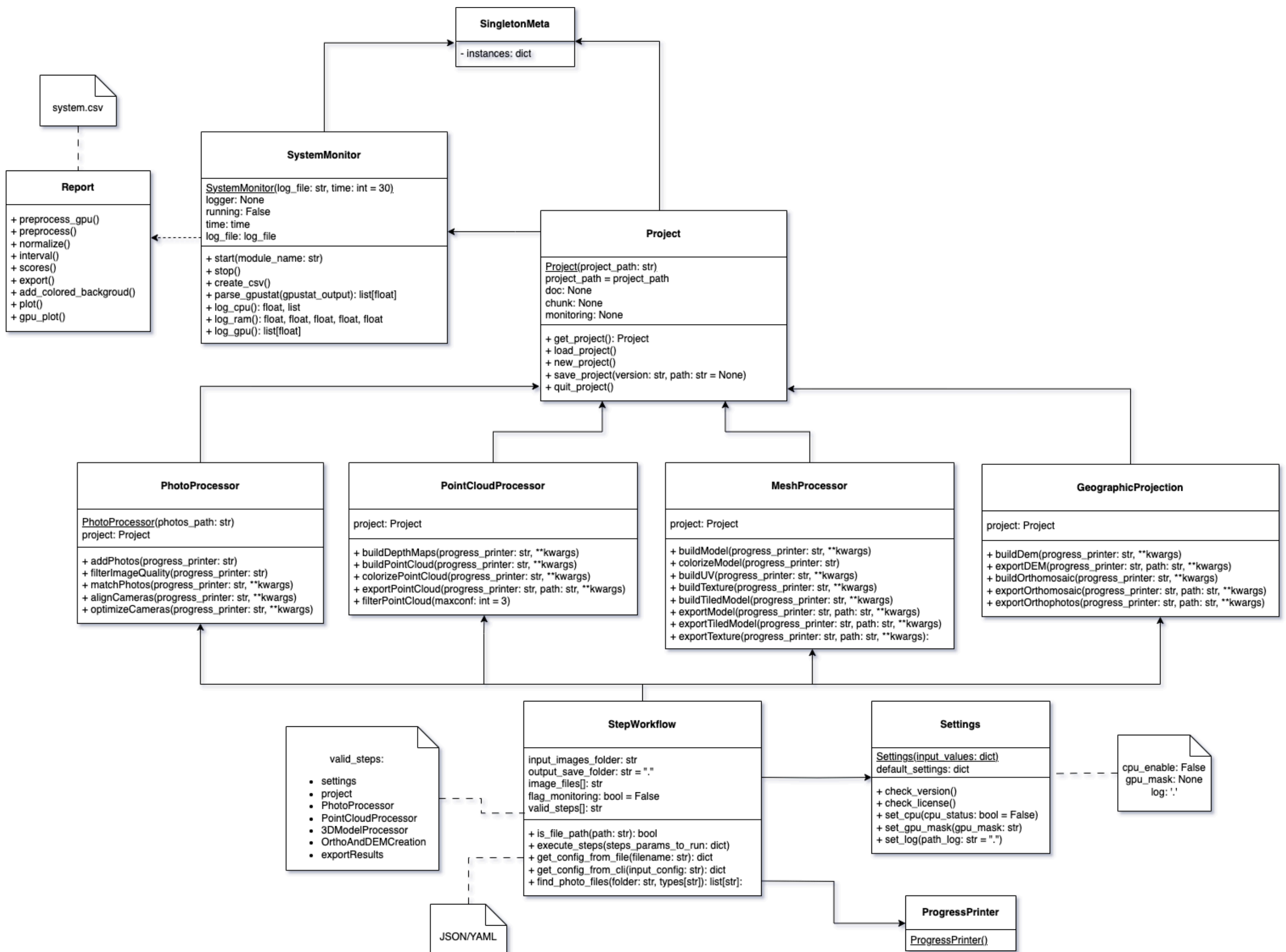

Figure 3: UML diagram of the project HaMMon-UAV-digital-twin.

In order to automate the process and avoid the use of the GUI interaction, Metashape provides a set of APIs that allow interaction with the software application. The implemented Python code processes the entire workflow of Metashape, ranging from the import of raw images to the exportation of the results obtained.
In the course of our project, we made the strategic decision to employ unit tests to validate the correctness, robustness and maintainability of our code. This approach has several advantages. Mainly, it ensures that each individual component of our software functions as expected. They serve as a form of documentation, providing future developers with insights into the intended functionality of the code. To execute the tests located in the test folder:

```
python -m unittest
python -m unittest -v  #verbose
```

Before running the scripts, requirements.txt file contains a list of all dependencies necessary for the project. To install all the Python libraries:

```
pip install -r requirements.txt
```

The instruction to run the script in headless mode from the command line is as follows:

```
python step_workflow.py <-i image_folder> <-e stepx:param1,param2 … / -c config.json>
                        [-o output_folder]
python step_workflow.py --help
```

Or using the installation folder of Metashape to run on a system without graphical user interface:

```
./metashape.sh -r step_workflow.py <-i image_folder> -platform offscreen
```

Where:

- -i: the path to the input image file
- -o: the path to the output file (it is created if it does not exist) [local position by default]
- -e: the task and parameters to execute
- -c: the path to the config file
- -m: enable monitoring

The first step of the project manages the input parameters specified at the invocation script. In particular, it is necessary to specify the folder where the photos are located to be processed and the destination directory to save the obtained results. A crucial part of this process is checking the validity of the path and ensuring there is a sufficient number of images to initiate the process.

You can choose the most convenient method to declare the steps to be executed as input parameters. There are two options: from the command line or from a configuration file (JSON,

YAML). By using the command line interface (CLI), you can list the tasks of interest and their specific input parameters.

Here is an example of a skeleton structure of a config JSON file. The main steps to be executed are specified without any particular input parameters. It is possible to be filled following the official Metashape documentation. By default, the parameters are set to their standard values.

```
{
    "workflow": {
        "settings": {
            "cpu_enable": false,
            "gpu_mask": "11",
            "log": "./path/log/"
        },
        "project":{
            "path": "../new_project/"
        },
        "PhotoProcessor":{
            "matchPhotos":{ },
            "alignCameras":{ },
            "optimizeCameras":{ }
        },
        "PointCloudProcessor":{
            "buildDepthMaps":{ },
            "buildPointCloud":{
                "maxconf": 3
            },
            "colorizePointCloud":{ }
        },
        "3DModelProcessor":{
            "buildModel":{ },
            "buildTexture":{ },
            "buildUV":{ },
            "exportTexture":{ }
        },
        "OrthoAndDEMCreation":{
            "buildDem":{ },
            "buildOrtho":{ }
        }
    }
}
```

At the end of each step the script saves the current status of the project. This approach enhances the ability to preserve the progress made, offering the flexibility to reopen the project for future modifications or changes, but also allows for the management of the project's status versioning.

Before executing any of these steps, feasibility checks are performed. This ensures that each step is viable and can be successfully completed. The script, so implemented, handles exceptions for non-existent specified paths, and saves the results in a folder with the name of the project.

StepWorkflow

StepWorkflow is the main class of the project that schedules all the steps taken in charge. It manages the input values and parameters from the CLI and the config file.

```
valid_steps = ['settings', 'project', 'PhotoProcessor', 'PointCloudProcessor', '3DModelProcessor',
               'OrthoAndDEMCreation', 'exportResults']

def execute_steps(steps_params_to_run: dict) -> None:
    # check steps_params_to_run not empty
    if not steps_params_to_run:
        raise Exception("Error: there are no processes to run.")
    # check steps_to_run are valid steps
    if any(step not in valid_steps for step in steps_params_to_run):
        raise Exception("Error: one or more specified processes are not valid.")
```

It ensures that the requested steps are valid. If any step is not in compliance with the known ones and is therefore considered invalid; the execution is immediately interrupted.

In order to facilitate image processing with Metashape, a surveying mission using a UAV drone is necessary. It's crucial to capture a series of overlapping images of the area of interest, ensuring sufficient overlap (usually 70-80%) between the images for future alignment. The number of images collected and the quality of the camera used during the surveying will affect data processing times and the quality of the final result.

After specifying the path where the photos are stored, the system verifies both the existence of the path and whether the photos in that location are in a raw format that Metashape can read, avoiding any possible extra data generated in the photo acquisition phase.

```
def find_photo_files(folder: str, types: list[str]) -> list[str]:
    files = []
    for entry in os.scandir(folder):
        if entry.is_file():
            file_extension = os.path.splitext(entry.name)[1].lower()
            if file_extension in types:
                files.append(entry.path)
    return files

# check number of images
image_files = find_photo_files(input_images_folder, [".jpg", ".jpeg", ".tif", ".tiff", ".png", ".bmp"])
if len(image_files) < 2:
    raise ValueError("Not enough images to process in the path.")
```

At the end of the project execution, a processing report is generated and saved in PDF format. It includes the characteristics, timings, and results obtained from the processing with Metashape.

```
prj.chunk.exportReport(path=output_save_folder +"/report.pdf", title="Final report")
```

```
print('Processing finished, results saved to ' + output_save_folder + '.')
prj.quit_project()
```

ProgressPrinter

To provide feedback on the progress timing and to keep track of which different process is underway, the ProgressPrinter class offers a series of functions that, when invoked periodically during the execution of long or intensive operations, allows for constant monitoring and adjustment of the process's progress.

```
class ProgressPrinter:
        def __init__( self, name ):
                self.name = name   # process name
        def __call__( self, percent ):
                print("{} progress: {:.2f}%".format(self.name, percent), end="\r", flush=True)
```

By integrating the commented methods in the code, it will be possible to choose whether to estimate the percentage of completion of the individual process or to integrate a live countdown timer that estimates the time remaining until the end of the single process execution.

Settings

The Settings class manages the configuration preferences for Agisoft. It handles settings that are useful for the upstream execution of all other processes. If these are not specified, it enables default integrated options.

```
class Settings:
   def __init__(self, input_values: dict = None) -> None:
       self.default_settings = {
           'cpu_enable': False,
           'gpu_mask': None,    # enable all available GPUs
           'log': '.'
       }
```

'Gpu_mask' sets the GPU device bit mask. It's a binary mask that enables and disables the gpu devices (e.g., 7 into bits: 111 - it means that three GPU devices will be used). Naturally, everything will depend on the availability of video cards present in the machine. Set by default all gpu available.

It is also possible to enable the CPU flag ('cpu_enable') to allow calculations both on CPU and GPU for GPU-supported tasks. However, if at least one powerful discrete GPU is used it is recommended to disable CPU flag for stable, rapid processing and optimal performance.

The system checks if the Metashape licence is activated and if the current version of the application is a compatible required version.

```
def check_version(self) -> None:
       compatible_major_version = "2.1"
       found_major_version = ".".join(Metashape.app.version.split('.')[:2])
       if found_major_version != compatible_major_version:
           raise Exception("Incompatible Metashape version: {} != {}".format(found_major_version,
                                                                  compatible_major_version))

def check_license(self) -> None:
       if Metashape.app.activated:
           print("-- Metashape is activated: ", Metashape.app.activated)
       else:
           raise Exception("No licence found.")
```

Due to the fact that in our current setup, we are executing Metashape in a non-interactive mode, Settings gives the possibility to enable logging. This becomes a valuable resource for retrospectively analysing and diagnosing any issues or anomalies that might occur during the execution of the process.

```
def set_log(self, path_log: str = ".") -> None:
        # check path and log file
        if not isinstance(path_log, str):
            raise ValueError("Error: specify a suitable path to save log file")
        if not os.path.exists(path_log):
            os.makedirs(path_log)
        if not os.path.exists(path_log + 'log.txt'):
            with open(path_log + 'log.txt', 'w'):
                pass
        Metashape.Application.Settings.log_enable = True
        Metashape.Application.Settings.log_path = path_log + 'log.txt'
```

Project

The Project class manages the correct project references. It keeps track of the documentation, the project's work chunk, and the project's absolute path.

```
class Project(metaclass=SingletonMeta):
   def __init__(self, project_path: str = None, enable_monitoring: bool = False) -> None:
       self.project_path = project_path
       self.doc = None
       self.chunk = None
       self.monitoring = None

       if enable_monitoring:
           directory_path = os.path.dirname(project_path)
           self.monitoring = SystemMonitor(directory_path + "/monitor.csv")

   @classmethod
```

```
    def get_project(cls):
        return cls()
```

Given the importance and role of this class, we have decided to designate and implement Project as a singleton. In this situation we ensure a unique instance of the class in the system, and provide a global access point to the instance. This way, we avoid the unintentionally instantiation of multiple projects.

The class supports saving/loading project files. It includes methods for creating a new project, loading an existing one, saving the project to a specific version, and closing the project.

```
def load_project(self) -> None:
        self.doc = Metashape.Document()
        self.doc.open(path=self.project_path, read_only=False)
        self.chunk = self.doc.chunk

    def new_project(self) -> None:
        self.doc = Metashape.Document()
        self.save_project(path=self.project_path, version="New project")
        self.chunk = self.doc.addChunk()

    # project version to save
    def save_project(self, version: str, path: str = None) -> None:
        if path == None:
            self.doc.save(version=version)
        else:
            # init project case
            self.doc.save(path=path, version=version)

    def quit_project(self) -> None:
        Metashape.app.quit()
```

The procedure, from the alignment of the photographic material to the generation of the 3D model, is included in four main classes that are explained below. These steps handle most of the data processing needs, with most operations being executed automatically based on the user's parameters. These classes have been implemented as an adapter design pattern to allow the interaction with Metashape modules. This is particularly important for better managing the types of parameters that the Metashape APIs require.

PhotoProcessor

Once the project is created and the photos are imported in Metashape, they will need to be aligned. In this phase, the position of the camera and the orientation of each photo are determined, and the model is built with a sparse point cloud. The position and orientation of the camera are calibrated by internal and external orientation parameters, which are based on the

focal length of the camera, coordinates and position of the main point of the frame, and lens distortion coefficients. The PhotoProcessor class is responsible for this process.

```
def addPhotos(self, progress_printer: str) -> None:
       if self.project.monitoring is not None:
           thread = threading.Thread(target=self.project.monitoring.start, args=('addPhotos',))
           thread.start()
       self.project.chunk.addPhotos(filenames=self.photos_path,
                                    progress=progress_printer)
       if self.project.monitoring is not None:
           self.project.monitoring.stop()
       self.project.save_project(version="addPhotos")

def matchPhotos(self, progress_printer: str, **kwargs) -> None:
       default_params = {
           'downscale': 1,
           'keypoint_limit': 40000,
           'tiepoint_limit': 10000,
           'generic_preselection': True,
           'reference_preselection': False,
           'filter_stationary_points': True,
           'guided_matching': False,
           'subdivide_task': True
       }
       default_params.update(kwargs)
       if self.project.monitoring is not None:
           thread = threading.Thread(target=self.project.monitoring.start, args=('matchPhotos',))
           thread.start()
       self.project.chunk.matchPhotos(progress=progress_printer, **default_params)
       if self.project.monitoring is not None:
           self.project.monitoring.stop()
       self.project.save_project(version="matchPhotos")
```

We can mention some of the settings parameters for image alignment:

- downscale: set the accuracy level. The lower the level, the more precise the estimate (0 - Highest, 1 - High, 2 - Medium, 4 - Low, 8 - Lowest) the more time-consuming it is.
- generic preselection: favours the alignment process when the set of photos is very high;
- reference preselection: it adapts better in the case where only a few reference points are detected, for example, during surveys of wooded areas or cultivated fields;
- keypoint limit: upper limit of feature points on every image;
- tie point limit: upper limit of matching points for every image. It is recommended to set the value as 10,000. A limit that is too high or low could compromise areas in the generation of the point cloud.

Once the cameras are aligned, it is also possible to optimise their positioning with the optimizeCameras function, specifying the required parameters.

```
def alignCameras(self, progress_printer: str, **kwargs) -> None:
        default_params = {
            'adaptive_fitting': False,
            'reset_alignment': False,
            'subdivide_task': True
        }
        # update default params with the input
        default_params.update(kwargs)
        if self.project.monitoring is not None:
            thread = threading.Thread(target=self.project.monitoring.start, args=('alignCameras',))
            thread.start()
        self.project.chunk.alignCameras(progress=progress_printer, **default_params)
        if self.project.monitoring is not None:
            self.project.monitoring.stop()
        self.project.save_project(version="alignCameras")

def filterImageQuality(self, progress_printer: str) -> None:
        if self.project.monitoring is not None:
            thread = threading.Thread(target=self.project.monitoring.start, args=('filterImageQuality',))
            thread.start()
        self.project.chunk.analyzeImages(cameras=self.project.chunk.cameras,
                                         progress=progress_printer)

        num_disable_photos = 0
        for camera in self.project.chunk.cameras:
            if float(camera.meta['Image/Quality']) < 0.5:
                camera.enabled = False
                num_disable_photos += 1

        if self.project.monitoring is not None:
            self.project.monitoring.stop()
        self.project.save_project(version="filterImageQuality")
```

The presence of poor quality, grainy or blurred photos can negatively affect the results of camera alignment as well as the processing of the final texture. For this reason, all photos are filtered based on their image quality. Images with a quality lower than 0.5 are excluded from the process. The image quality value is calculated based on the sharpness level of the image.

As a result in Fig. 4, a sparse point cloud and a set of camera positions are generated. The sparse point cloud represents the alignment results of the photos and will not be used directly. On the contrary, the set of camera positions is necessary for subsequent processing of the 3D model in Metashape.

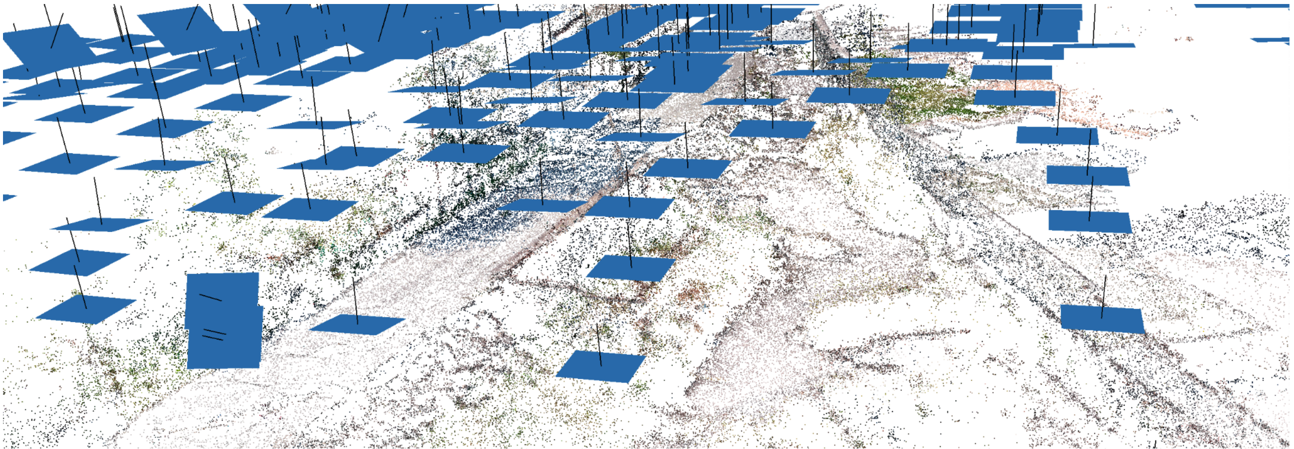

Figure 4: Sparse cloud and camera position (Fleri Sicilia 2018).

PointCloudProcessor

Metashape provides the ability to create and view the model in the form of a dense point cloud. This is generated using the initial sparse point cloud, accurate camera alignments, and point interpolation. This process fills the spaces between the sparse points, resulting in a high-resolution, detailed, and complete 3D representation of the captured scene. The class responsible for this process is the PointCloudProcessor class.

The dense cloud can be produced by choosing the source data: depth maps or tiled model. Once this is selected, it will then be possible to determine the quality of the point cloud; higher settings always imply a longer processing time (1 - Ultra high, 2 - High, 4 - Medium, 8 - Low, 16 - Lowest). In this case, the depth maps are calculated on each image. Enabling reuse depth in the chunk allows it to be utilised in the creation of the point cloud.

```
def buildDepthMaps(self, progress_printer: str, **kwargs) -> None:
        default_params = {
            'downscale': 2,
            'filter_mode': Metashape.MildFiltering,
            'reuse_depth': False,
            'subdivide_task': True
        }
        default_params = update_existing_keys(default_params, kwargs)
        if self.project.monitoring is not None:
            thread = threading.Thread(target=self.project.monitoring.start, args=('buildDepthMaps',))
            thread.start()
        self.project.chunk.buildDepthMaps(progress=progress_printer, **default_params)
        if self.project.monitoring is not None:
            self.project.monitoring.stop()
```


Finanziato dall'Unione europea NextGenerationEU
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
MUR

```
        self.project.save_project(version="buildDepthMaps")
```

A depth map is an image channel that contains information relating to the distance of the surfaces of scene objects from a viewpoint. Each pixel in a depth map is assigned a value to represent the distance of that pixel from a reference point, creating a 3D representation of the scene for its RGB image or virtual scene. The result is a series of depth maps that help fill in the spaces between keypoints, thereby improving the accuracy of the 3D model.

```
def buildPointCloud(self, progress_printer: str, **kwargs) -> None:
        default_params = {
            'source_data': Metashape.DepthMapsData,
            'point_colors': True,
            'point_confidence': True,
            'keep_depth': True,
            'subdivide_task': True
        }
        default_params = update_existing_keys(default_params, kwargs)
        if self.project.monitoring is not None:
            thread = threading.Thread(target=self.project.monitoring.start, args=('buildPointCloud',))
            thread.start()
        self.project.chunk.buildPointCloud(progress=progress_printer, **default_params)
        if self.project.monitoring is not None:
            self.project.monitoring.stop()
        self.project.save_project(version="buildPointCloud")
```

For the creation of the point cloud, the most important parameters are ‘point_colors’, ‘point_confidence’, and ‘keep_depth’. The ‘point_colors’ parameter includes the points colour information. It can be set to False to reduce the process time. The ‘point_confidence’ parameter filters the dense cloud point, which can help remove any remaining outliers. The ‘keep_depth’ parameter maintains the depth maps, useful for subsequent calculation of DEM.

```
def filterPointCloud(self, maxconf: int = 3) -> None:
        if self.project.monitoring is not None:
            thread = threading.Thread(target=self.project.monitoring.start, args=('filterPointCloud',))
            thread.start()
        for chunk in self.project.doc.chunks:
            chunk.point_cloud.setConfidenceFilter(0, maxconf)
            all_points_classes = list(range(128))
            chunk.point_cloud.removePoints(all_points_classes)
            chunk.point_cloud.resetFilters()
        if self.project.monitoring is not None:
            self.project.monitoring.stop()
```

To improve the quality of the resulting models, it can be useful to remove small artefacts and impurities from the cloud model. filterPointCloud can remove the filtered low confidence points. Given the ‘maxconf’ threshold, the function selects only those points with confidence levels below

this threshold from the point cloud of each chunk in the project. Due to the fact that the selection is non-linear scale, we advise that the observed confidence variations are within the range of 1-5. Deleting the selected points effectively removes all low-confidence points from the point cloud, leaving only high-confidence points active.

MeshProcessor

The class MeshProcessor is involved in the creation of 3D models. The depth maps created on PointCloudProcessor are designated as source data, which guarantees higher quality results compared to models with sparse cloud source data. If the area of interest to be processed is very large, it is possible to subtask the creation of the 3D model into blocks, specifying the size of the blocks in metres and the reference coordinate system (blocks_size, blocks_crs).

```
def buildModel(self, progress_printer: str, **kwargs) -> None:
       default_params = {
           'surface_type': Metashape.HeightField,
           'source_data': Metashape.DepthMapsData,
           'interpolation': Metashape.EnabledInterpolation,
           'face_count': Metashape.HighFaceCount,
           'vertex_colors': True,
           'vertex_confidence': True,
           'keep_depth': True,
           'split_in_blocks': False,
           'blocks_crs': Metashape.CoordinateSystem("WGS 84"),
           'blocks_size': 250,
           'build_texture': True,
           'subdivide_task': True
       }
       default_params.update(kwargs)
       if self.project.monitoring is not None:
           thread = threading.Thread(target=self.project.monitoring.start, args=('buildModel',))
           thread.start()
       self.project.chunk.buildModel(progress=progress_printer, **default_params)
       if self.project.monitoring is not None:
           self.project.monitoring.stop()
       self.project.save_project(version="buildModel")
```

With enabled interpolation mode, the software generates a geometric model without empty holes based on refined level interpolations. The problem is that this process risks generating large additional geometric areas. By disabling the interpolation, it leads to accurate reconstruction results since only areas corresponding to point cloud points are reconstructed.

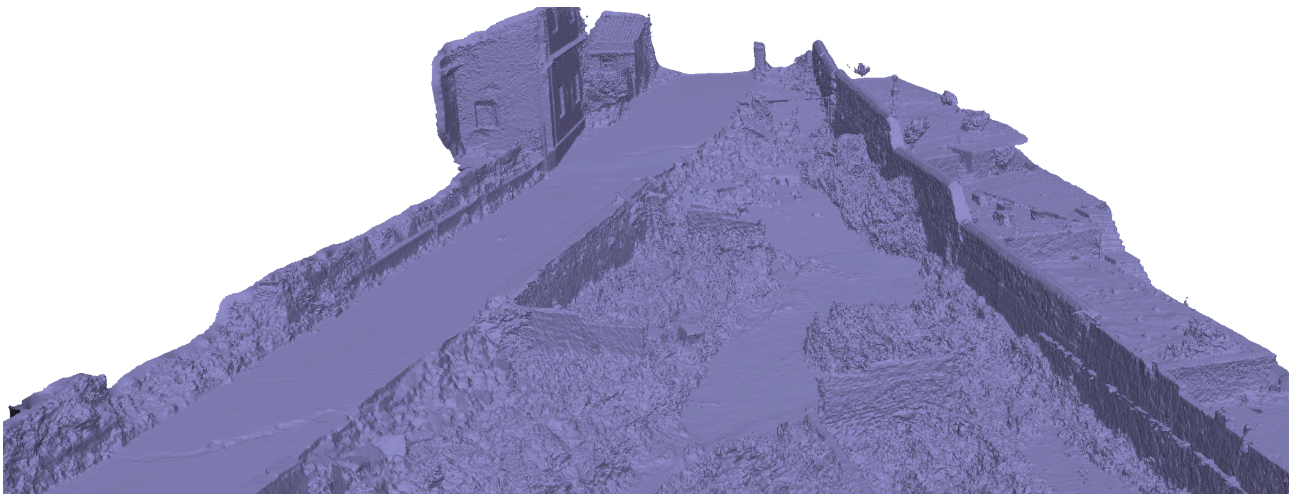

Figure 5: 3D Mesh model (Fleri Sicilia 2018).

The specified surface type (HeightField) is optimised for modelling on planar surfaces such as terrains or basereliefs. This parameter is excellent for processing aerial photographs as it requires less memory and allows the processing of larger data sets. This setting can be changed to Arbitrary as needed to allow the processing of closed objects like buildings, at the cost of higher memory consumption. A result is shown in Figure 5.

Once the 3D model is created, it will be possible to cover it by applying the texture. In the texture building process, it is possible to specify the size of the pixel structure.

```
def buildUV(self, progress_printer: str, **kwargs) -> None:
        default_params = {
            'mapping_mode': Metashape.GenericMapping,
            'page_count': 1,
            'texture_size': 8192
        }
        default_params.update(kwargs)
        if self.project.monitoring is not None:
            thread = threading.Thread(target=self.project.monitoring.start, args=('buildUV',))
            thread.start()
        self.project.chunk.buildUV(progress=progress_printer, **default_params)
        if self.project.monitoring is not None:
            self.project.monitoring.stop()
        self.project.save_project(version="buildUV")

    def buildTexture(self, progress_printer: str, **kwargs) -> None:
        default_params = {
            'blending_mode': Metashape.MosaicBlending,
            'texture_size': 8192,
            'fill_holes': True,
            'ghosting_filter': True
```

```
    }
    default_params.update(kwargs)
    if self.project.monitoring is not None:
        thread = threading.Thread(target=self.project.monitoring.start, args=('buildTexture',))
        thread.start()
    self.project.chunk.buildTexture(progress=progress_printer, **default_params)
    if self.project.monitoring is not None:
        self.project.monitoring.stop()
    self.project.save_project(version="buildTexture")
```

UV mapping is a highly effective technique for applying image textures. It significantly increases the level of detail of textures on a three-dimensional model with high precision and calculation speed. This process involves laying out a 3D model's surface in a two-dimensional space, enabling textures to wrap around the model accurately. Without proper UV mapping, textures would appear distorted, stretched, or misaligned, detracting from the overall realism of the scene. A final result is presented in Figure 6.

Figure 6: Textured Mesh (Fleri Sicilia 2018).

GeographicProjection

The resulting project can be structured and used for the generation of the Orthomosaic model and the Digital Elevation Model (DEM).

DEM represents a set of measurements that record the elevation of the earth's surface and also contain information about the spatial relationships between these measurements. Figure 7 represents the DEM obtained from the Fleri model. In Metashape, the DEM can be rasterized from

the point cloud, depth maps, or the 3D model. The most accurate results are obtained by setting the dense point cloud as source data. The orthomosaic instead, is generated based on mesh directly.

It is possible to generate both the Digital Terrain Model (DTM), which represents the earth's surface without objects on top like buildings or plants, and the Digital Surface Model (DSM), which is the surface with all elements on top.

```
def buildDem(self, progress_printer: str, **kwargs) -> None:
        default_params = {
            'source_data': Metashape.PointCloudData,
            'interpolation': Metashape.EnabledInterpolation,
            'subdivide_task': True
        }
        default_params.update(kwargs)
        if self.project.monitoring is not None:
            thread = threading.Thread(target=self.project.monitoring.start, args=('buildDem',))
            thread.start()
        self.project.chunk.buildDem(progress=progress_printer, **default_params)
        if self.project.monitoring is not None:
            self.project.monitoring.stop()
        self.project.save_project(version="buildDem")

def buildOrthomosaic(self, progress_printer: str, **kwargs) -> None:
        default_params = {
            'surface_data': Metashape.ElevationData,
            'fill_holes': True,
            'subdivide_task': True
        }
        default_params.update(kwargs)
        if self.project.monitoring is not None:
            thread = threading.Thread(target=self.project.monitoring.start, args=('buildOrthomosaic',))
            thread.start()
        self.project.chunk.buildOrthomosaic(progress=progress_printer, **default_params)
        if self.project.monitoring is not None:
            self.project.monitoring.stop()
        self.project.save_project(version="buildOrthomosaic")
```


Finanziato dall'Unione europea NextGenerationEU
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
MUR

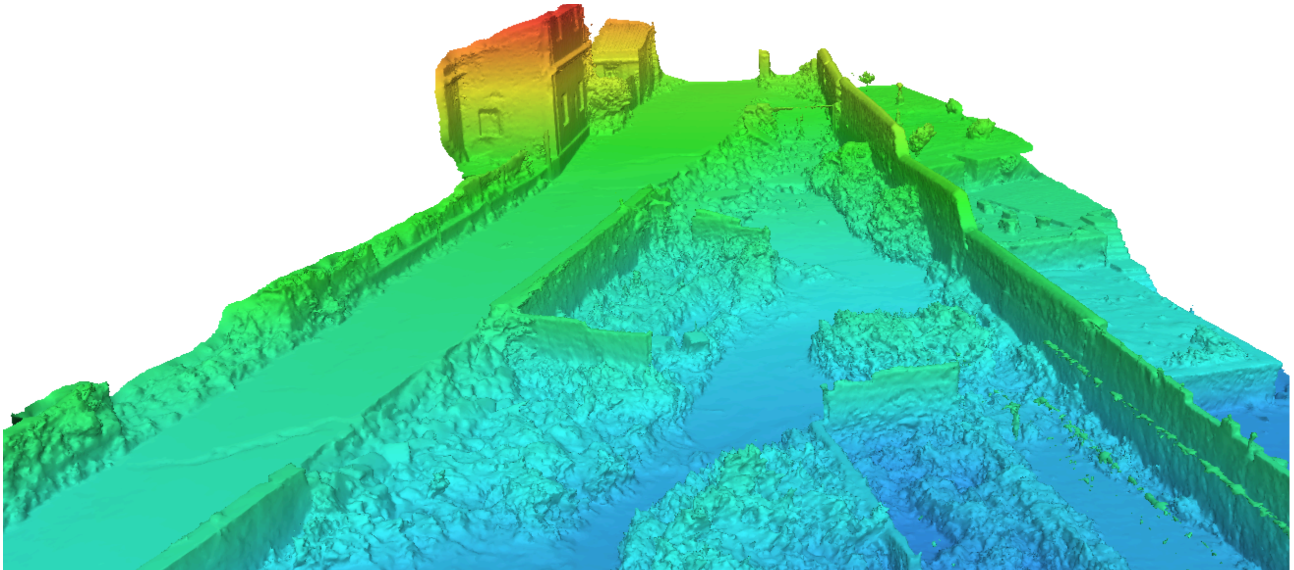

Figure 7: DEM (Fleri Sicilia 2018).

In aerial photographic survey data processing, exporting high quality orthomosaic models is a common task. A final result is presented in Figure 8. The process consists of combining the images (orthophotos) projected onto the surface based on the preferred x, y, or z axis.

Figure 8: Exported orthomosaic (Fleri Sicilia 2018).

Finanziato dall'Unione europea
NextGenerationEU
Italiadomani
PIANO NAZIONALE DI RIPRESA E RESILIENZA
MUR

Saving and exporting the results

The various processing stages involved in the reconstruction of a 3D model can be time-consuming. The Metashape software allows you to save the current project in the *.psx format, which preserves the references along with the results of each process that has been carried out. Each class implemented has an export function of the obtained result.

Here is an example of a skeleton structure of a config YAML file. The indicated steps execute the export of the obtained results in dedicated subfolders. Note how the project is loaded, taking the .psx file format as a reference. This structure allows for organised storage and easy access to the results of each step.

```
workflow:
  settings:
    cpu_enable: false
    gpu_mask: '11'
    log: './path/log/'
  project:
    path: '../new_project/new_project.psx'
  exportResults:
    exportDEM: {}
    exportOrthomosaic: {}
    exportModel: {}
    exportPointCloud: {}
    exportTiledModel: {}
    exportTexture: {}
    exportOrthophotos: {}
```

SystemMonitor

The SystemMonitor class manages the monitoring of the computational resources on which the program runs. It is capable of recording the computational resource values consumed during the execution of the Python script. This is accomplished through a logging system and external libraries (gpustat and psutils) that monitor the periodic status of the utilised system: CPU, GPU, and RAM. If enabled at the beginning of the script execution, the resource check is conducted every 30 seconds.

The decision to integrate monitoring into our system is primarily driven by the need for performance tracking and troubleshooting. Monitoring the performance tracking allows us to follow the system's performance over time. This is crucial for identifying any bottlenecks or issues that could be impacting performance. Additionally, it enables us to measure the effectiveness of

our optimizations and modifications, providing valuable feedback that can guide our development process.

In addition to this, having a troubleshooting tool allows us to resolve issues more quickly and prevent the onset of future problems. This proactive approach not only helps in maintaining the smooth operation of our system but also in avoiding potential disruptions that could impact the system's performance and reliability. In a similar way, the availability of a troubleshooting tool expedites the resolution of issues and helps ward off future problems. This proactive strategy not only aids in maintaining the seamless operation of our system but also helps circumvent potential disruptions that could impair the system's performance and reliability.

The monitoring will be strictly limited to the tracking of resources during the execution of tasks of interest. Once the current task is completed, the monitoring is stopped, only to be resumed in the next step. The outcome obtained is saved in a tabular format in the system.csv file after each iteration.

Before integrating the monitoring into the final version of the project, we conducted tests on it using a demo, which can be found in the demo folder. We have produced a script, "demo_process.py," which sequentially launches all the necessary steps for the creation of the digital twin (with a standard set of parameters), where the monitoring library has been integrated.

We have launched demo_process.py, in synergy with WP1, in the Kubernetes Cluster of the HaMMon PoCI (the related docker image is in docker folder of the git project):

- Node: "juju-a498f0-hammon-88"
- Processor: Intel Xeon Processor (Skylake, IBRS) with 32 cores
- RAM: 62.79 Gib
- GPUs: 2 x TeslaV100-SXM2-32GB with 32 Gib of memory

The data detected by the installed libraries are appropriately parsed to extract only the information of our interest. Below is a tabular excerpt from the system.csv file obtained by monitoring the Kubernetes Cluster system on a large Zurich dataset (10.2GB) with 1207 photos of 6000x4000 resolution using a single GPU.

Finanziato dall'Unione europea NextGenerationEU
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
MUR

Module	Time	CPU%	Cores %	RAM %	RAM active	RAM Available	RAM Used	GPUs ID: 0	model: TeslaV100-SXM2 -32GB	'mem_total': '32768MB
Start	171542	8.0	[0.0, 0.0, 0.0, 0.0, 2.8, 0.0 ..] x 32	3.5	20.61 GB	60.621 GB	1.249 GB	'temp': '35C'	'cpu_usage': '0%'	'mem_used': '0'
New Project	171542	6.8	[0.7, 0.4, 0.4, 0.4, 2.4, 0.6 ...] x 32	3.5	20.61 GB	60.620 GB	1.250 GB	'temp': '35C'	'cpu_usage': '0%'	'mem_used': '0'
addPhotos	171542	6.4	[0.2, 0.2, 0.2, 0.4, 0.6, 0.4 ...] x 32	3.5	20.61 GB	60.618 GB	1.251 GB	'temp': '35C'	'cpu_usage': '0%'	'mem_used': '0'

Table 1: Excerpt of the first entries from a system.csv file.

The Table 1 is composed as follows: Module reports the name of the task under analysis, Time is the current time, expressed in seconds since the epoch, CPU usage reports the percentage of CPU usage and its individual cores, RAM usage active, available and used currently, and GPU characteristics such temperature, cores and RAM usage.

Once all the data from a workflow has been gathered, it is important to analyse the collected data. We have implemented 'report.py' that analyses the previously described data to generate a JSON file containing extremal values and an Excel document detailing time-averaged values for both the overall process and each individual step. Additionally, the library provides two methods to graphically represent the results. This allows us to visualise the trend of resources used by Metashape over time. Samples usage of its most notable method are shown in the './reports/report.ipynb' notebook.

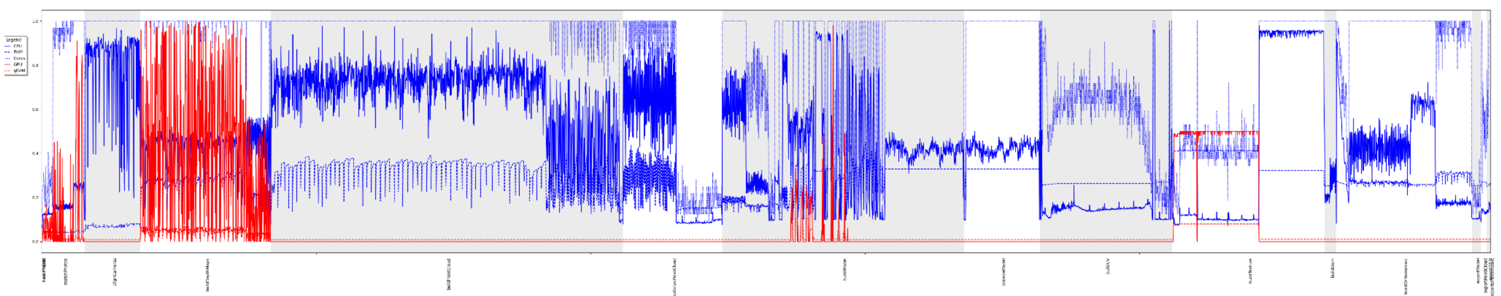

Figure 9: Hardware Resource usage over the entire workflow process.

As indicated in the legend of Fig. 9, GPU resources are shown in red, CPU resources in blue, and the dot line represents the amount of RAM used. As we expected the use of GPU acceleration is supported mostly on specific steps: image matching, depth maps generation, mesh generation based on depth maps and mesh refinement.

Finanziato dall'Unione europea NextGenerationEU
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
MUR

The data represented in the graph are normalised into a single range, with peaks reported as their 100% maximum resource usage. Another point to consider is the use of a node with two GPUs.

Given the enormous amount of time for the execution of an entire workflow, the results are very compact and not optimal in their visualisation, as shown in Fig. 9. It is possible with the PLOT method present in 'report.py' to select and graph only the steps of interest (see Fig. 10).

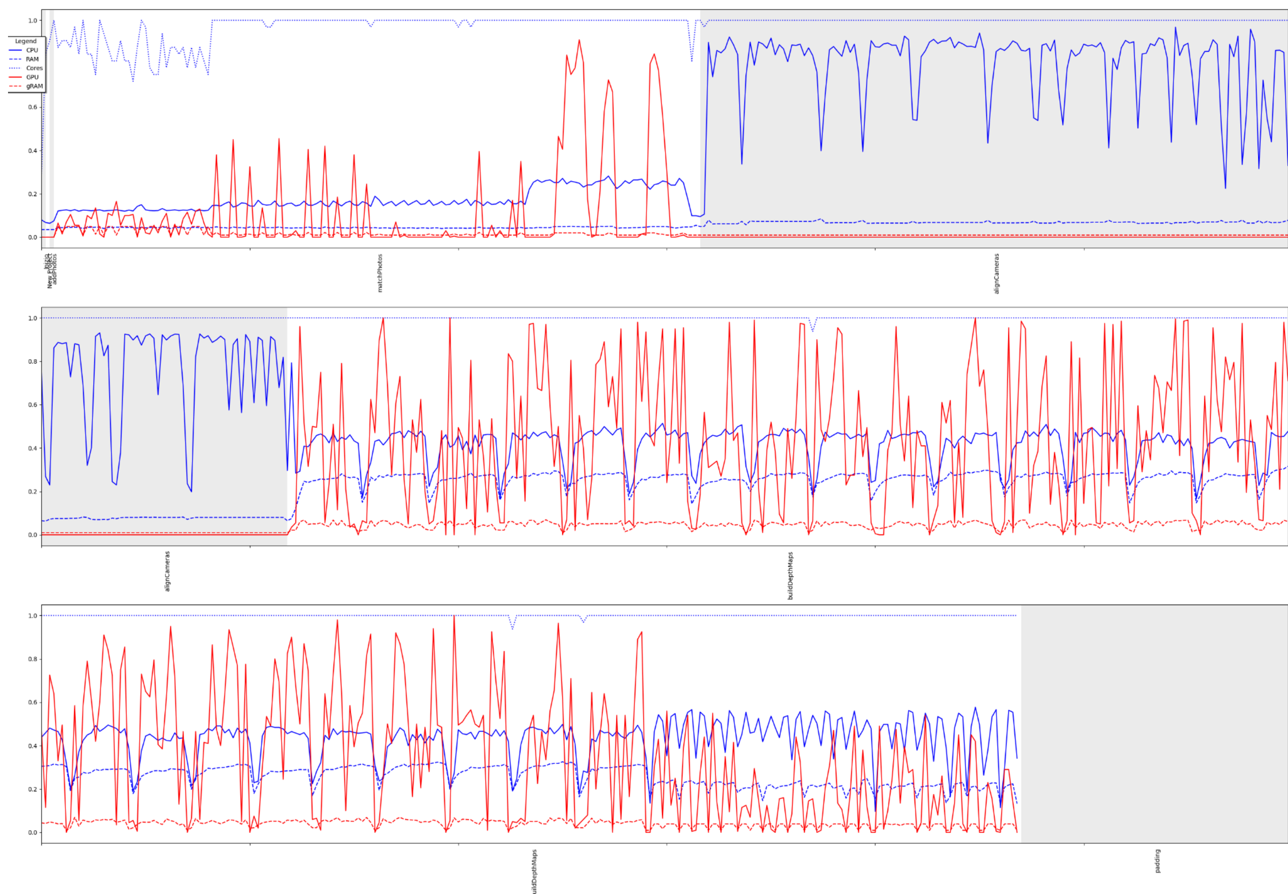

Figure 10: Hardware resource highlight of selected steps: matchPhotos, alignCameras, buildDepthMaps.

With the TWO_GPU_PLOT method, it is possible to highlight and compare the performance of the two video cards. It shows the contribution of the individual video cards compared to the total of their calculation (see Fig. 11).

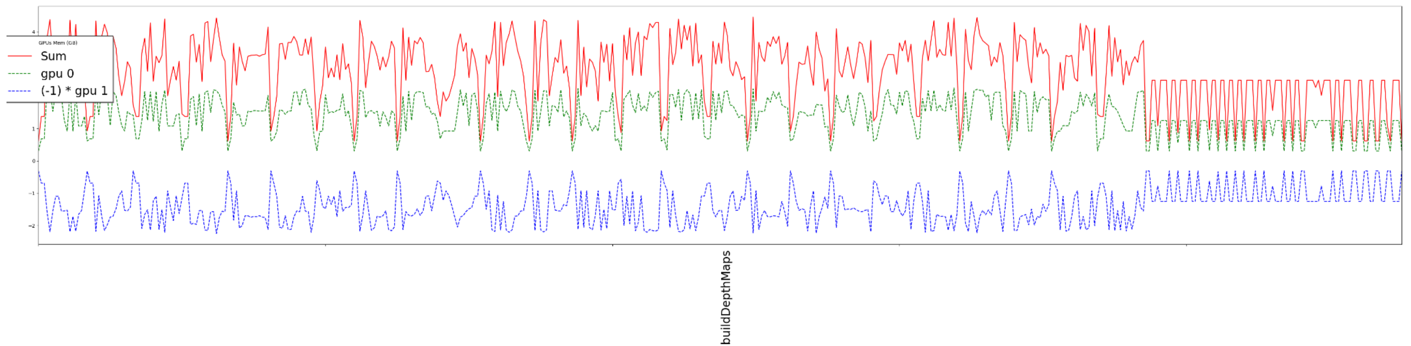

Figure 11: GPU memory usage for a selected process step: green for the first video card, blue for the second (negative values plotted for clarity), and red for the combined total.

3. Credits and References

The image processing and 3D model creation were performed with the software Agisoft Metashape Professional (https://www.agisoft.com).

The photos used for the project were sourced from Survey Bonali Fabio: Fleri Sicilia 2018.

The photos of Zurich used for benchmark monitoring were sourced from Wingtra dataset (https://wingtra.com).

Regarding the script mentioned, the source code can be found in the respective repository (https://github.com/ICSC-Spoke3/HaMMon-UAV-digital-twin).

Please note that all images and materials used are for research and study purposes only, and all rights are reserved to their respective owners.

A survey of Digital Twin techniques in smart manufacturing and management of energy applications (https://doi.org/10.1016/j.geits.2022.100014)

Adaptive structure from motion with a contrario model estimation (2012) (DOI:10.1007/978-3-642-37447-0_20)

Photogrammetry is for everyone: Structure-from-motion software user experiences in archaeology (DOI:10.1016/j.jasrep.2020.102261)

Structure from motion photogrammetric technique (https://doi.org/10.1016/B978-0-444-64177-9.00001-1)