Ref. Ares(2023)327304 - 16/01/2023

Project Number: 101016906
Start Date of Project: 2021/01/01
Duration: 48 months

Type of document D4.3 – V1.1

VR Agronomic-Oriented Perception Component

Dissemination level	PUBLIC
Submission Date	2021-12-30
Resubmission Date	2023-01-16
Work Package	WP4
Task	T4:4
Type	Report
Version	1.1
Author	PaleBlue
Approved by	Consortium

DISCLAIMER:
The sole responsibility for the content of this deliverable rests with the authors. It does not necessarily reflect the opinion of the European Union. Neither the Directorate-General for Communications Networks, Content and Technology, Artificial Intelligence and Digital Industry nor the European Commission are responsible for any use that may be made of the information contained therein.

History of changes

Changes in original text	New version	Old Version
Paragraph 3.5, p. 16	Clarified the usage of agronomic sensors based on the "Request for Revision Comment".	N.A.
Paragraph 3.6, p. 17	Details on additional sensors have been included based on the "Request for Revision Comment".	N.A.

Executive Summary

This document describes details about the simulation of sensors in a virtual 3D environment. The communication between the Unity 3D environment and the ROS environment is described. The implementation of the available sensors is discussed with emphasis on the depth and lidar cameras including details on achieving the performance requirements for these cameras.

Table of Content

1 Introduction 6

2 Pipeline Architecture 6

2.1 Unity 6

2.2 ROS 7

3 Virtual sensors 7

3.1 RGB camera 8

3.1.1 Sensor calibration 9

3.2 Depth camera 10

3.2.1 Sensor calibration 11

3.3 LIDAR 12

3.3.1 3D ray casting using CPU physics 12

3.3.2 3D ray casting using graphics depth maps 14

3.3.3 Point clouds 15

3.4 GPS/IMU sensor 16

3.5 Grape color, ripeness, sugar level sensors 16

3.6 Additional sensors 17

4 References 18

Abbreviations and Acronyms

TS	Technical Staff
WP	Work Package
RGB	Red-Green-Blue
RGB camera	Colour camera without any depth information
LIDAR	Light detection and ranging
VR	Virtual Reality
ROS	Robot Operation System
GPU	Graphics Processing Unit
IMU	Inertial Measurement Unit
GPS	Global Positioning System
GPS-RTK	GPS with Real-Time Kinematic positioning

1 Introduction

A critical part of the Canopies simulation environment is the simulation of the sensors. These sensors are used by the partners to receive information about the environment in which the robots are moving. Decisions are made on this information to avoid collisions with humans, the vineyard structure and grapes, but also to enable cutting grape bunches from the vines, placing them in the boxes and pruning branches. The simulation environment is used to test, and possibly train, the robots for all those tasks. It is therefore important that the simulated sensors behave as closely as possible to the real sensors.

This document describes how these simulated sensors are realized. First, the connection between the 3D simulation environment in Unity and the ROS world is described. Then, the different types of simulated sensors are described in detail, including RGB cameras, Depth cameras, Lidars, IMU & GPS sensors and agronomic ripeness sensors.

2 Pipeline Architecture

2.1 Unity

The Unity 3D environment is connected with a TCP connection to the ROS environment as is shown in Figure 1. This TCP connection makes it possible to have the Unity 3D environment running on a different platform (for instance Windows) than the ROS environment which is usually running in an

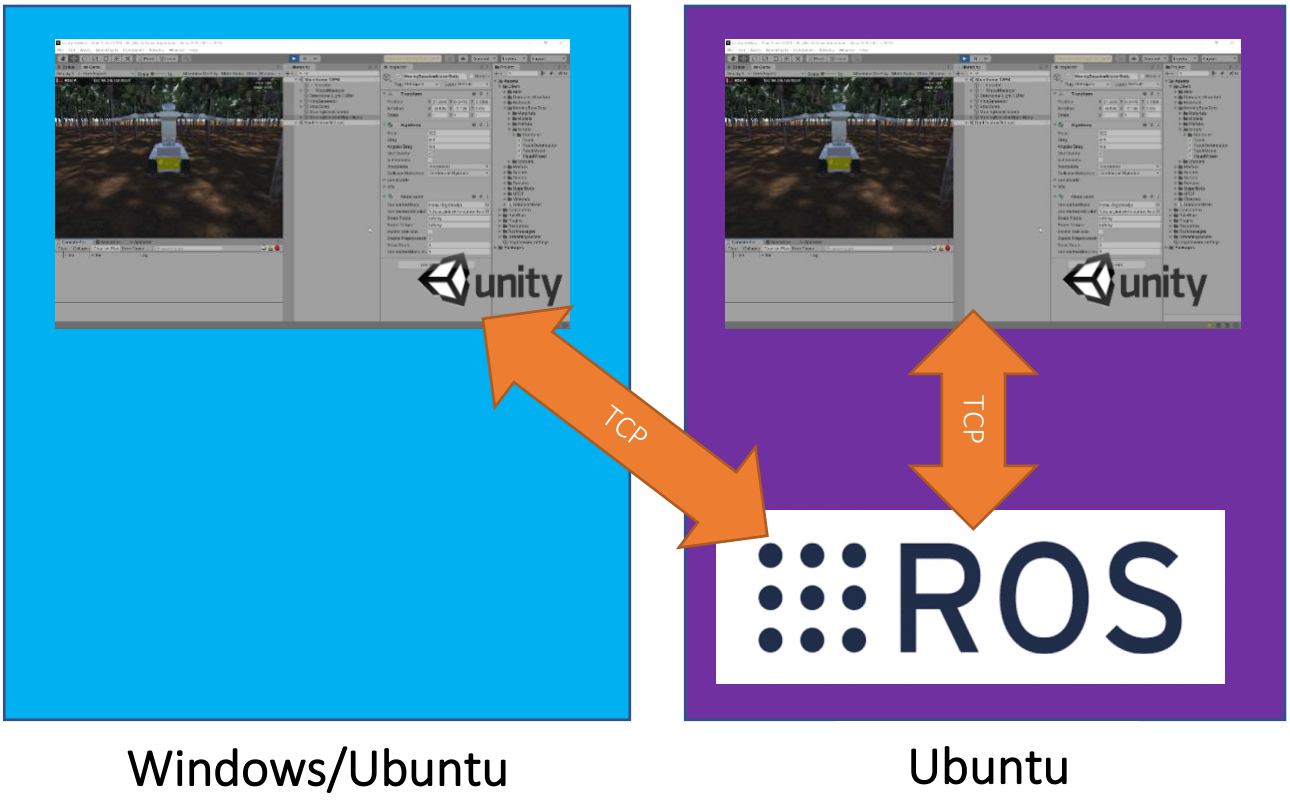

Figure 1 Data connections between Unity on Windows or Ubuntu with the ROS connector on Ubuntu.

Ubuntu 20.04 environment. Because of this design choice we have the option to run Unity on either Ubuntu or Windows according to the specific needs of the user.

2.2 ROS

A ROS component will subscribe and publish the data in the ROS environment and forward these to and from the Unity 3D environment using the TCP connection.

Values from the parameter server can be requested from the Unity environment to ensure that the parameters set for specific sensors can be used.

3 Virtual sensors

For the robotic interface, several sensors need to be simulated. Each of these sensors will sample the simulation environment in real-time and produce data on a ROS interface. Due to the simulation computational complexity, the number and type of sensors which can be included in the scene depends on the hardware specifications of the computer running the simulation. However, it is expected that multiple sensors can be used simultaneously in this setup.

With the current implementation it is possible to run multiple sensors simultaneously. We have tested the expected robot setup with 2 lidars and 2 depth cameras and one IMU on the following computer:

- Processor: AMD RX3900X
- Graphics card: AMD RX5700XT

Figure 2 shows the performance with the considered setup. We observe the performance with an update rate of 50-200 frames per second with a CPU load of around 36% and a GPU load of 50%.

We should note that there is still room for improvement in the performance so we expect that the final performance numbers will be better.

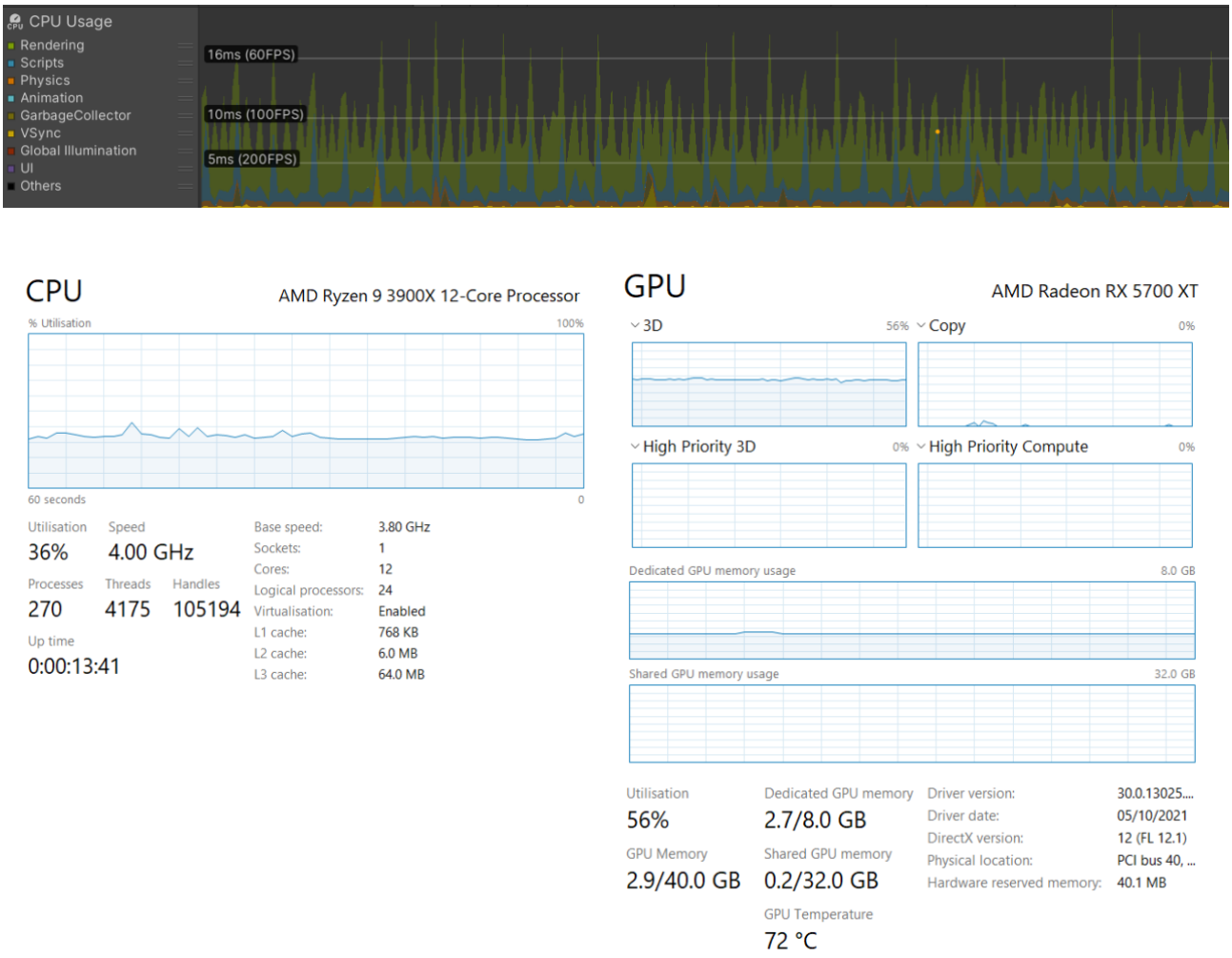

Figure 2 Performance figures for a robot with 2 Lidars, 2 depth cameras and an IMU sensor

3.1 RGB camera

A color or RGB camera is a standard component in the Unity 3D engine. It can be attached at any position in the scene and its resolution, field-of-view, color space, and related settings can be configured as needed.

Normally, the image captured by the camera is rendered on a screen, but it is possible to capture this data and process it. In this way, it is possible to generate ROS messages based on the images captured by such a camera.

The composition of the image is done on the graphics processing unit (GPU) of the computer. Generating the ROS messages can only be done on the central processing unit (CPU) of the computer. This means that we need to send the data back from the GPU to the CPU. This communication path is not very common, but with asynchronous GPU readback it is possible to do it efficiently. The downside of this method is that it creates a latency of "a few frames" which is 20ms on average.

The video stream is then published on an `image_raw` topic using `sensor_msgs/Image` type messages.

3.1.1 Sensor calibration

The color image coming from an RGB camera is different than the perception of the human eye. The Unity cameras are normally intended to be viewed on a display or a virtual reality headset. For our purpose, we need to introduce specific camera-dependent perception properties to match the limitations used in the color sensors of the camera.

The scene is rendered on a low-level using High-Dynamic Rendering (HDR) which means that the color values calculated for the pixels on the screen exceed the range of the display. With that, it is possible to model bright sunlight and dark shadows.

In the case of a simulated ROS camera, the high dynamic range can be mapped to the simulated dynamic range of the simulated camera. This means that the simulated camera may become overexposed when it is looking at very bright places like leaves directly hit by sunlight at noon. On the other hand, it may become underexposed and produce a nearly black picture in dark places under the canopy in the early morning for example.

It should be noticed that it is generally very hard to determine the exact properties of the cameras in this area as the available documentation does not provide this information. An alternative would be to have a conditioned scene both in real-life and virtual reality and compare the output of the real camera and the virtual camera in this situation. However, the problem is then in the conditioned scene: it is nearly impossible to ensure that the real and virtual world scenes are identical and that we are in fact comparing the cameras, not the scenes. Note that the same applies also to the GAZEBO simulator, where ideal sensors are simulated and do not necessarily reflect real-world conditions.

So, for now, the sensor properties are just calibrated subjectively such that the output of the camera looks as it would be expected in these situations. If this approach turns out not to be satisfactory, alternative strategies will be investigated.

3.2 Depth camera

Figure 3 Placement of sensors in an Intel RealSense camera

The simulated depth cameras actually consist of two sensors: the depth sensor and the color sensor. This matches the configuration of the real-world depth cameras in which those sensors are separate as well.

The depth sensor itself is not a stereo camera setup like in the image above but it is a single sensor placed in the middle between the stereo cameras.

In this way we can configure the depth camera component to match the specifications of different actual depth cameras:

Table 1 - Specifications of RealSense depth cameras

	Depth camera				Color Camera			
Model	Resolution	Position	Vertical FoV	Range	Resolution	Position	Vertical FoV	Range
RealSense D415	1280*720	-0.75mm	40˚	0.45-10m	1280*720	-3.5mm	42˚	0.11-10m
RealSense D435	1280*720	0.75mm	58˚	0.28-10m	1280*720	-2.2mm	42˚	0.11-10m
RealSense D455	1280*720	0mm	58˚	0.52-10m	1280*720	1.15mm	58˚	0.11-10m

Like for the RGB camera, the data of the depth camera is generated by the GPU of the computer. For the color sensor, it is the same method as described in the previous section. For the depth sensor, the image is actually generated from the depth map of the unity camera. This depth map is used internally by the unity camera to compute the occlusion of objects in the camera. We use this to generate an image with the requested resolution in which every point contains information about the distance to the closest object at the given camera pixel.

We observe that the required computing power for depth cameras is higher than for the lidars because for the depth cameras the information from two sensors has to be combined and the required bandwidth for a depth camera is higher than a lidar (depth camera: 30fps x 1280*720 vs. lidar: 10fps x 1024*64).

For the depth camera, we can produce depth and color map images and point-clouds.

It should be noted that the bandwidth requirement of the point-cloud data is such that it is not possible to run multiple depth cameras simultaneously. With a standard point-field size of 20 bytes (for X, Y, Z coordinated and a RGB value), a full Point Cloud message for a depth camera frame takes 1280*720*20 = 18Mbyte. In practice not all points will be sent because these values are outside the range of the camera. A typical value is that only 30% of the points are actually sent, resulting in point cloud messages of about 5Mbyte. With a framerate of 30fps this results in a bandwidth of 600 Mbyte/second. This value exceeds the available bandwidth of a typical 1Gbit network connection.

Note that the RealSense Gazebo plugin also advises using the depth_image_proc package to generate point clouds. We can read this in the SyrianSpock package documentation on which the PAL Realsense plugin we use is based: https://github.com/SyrianSpock/realsense_gazebo_plugin.

For that reason, it is advised to use the color and depth maps instead and generate the point cloud points where necessary using the depth_image_proc package.

3.2.1 Sensor calibration

The depth values coming from the simulated depth camera are very accurate as they are directly derived from the scene as can be seen in Figure 4. This is the same as when using Gazebo. In the real-world data, we have random and systematic errors in the data which are not simulated in this way. We will investigate the need to add specific types of noise or errors to simulate this effect.

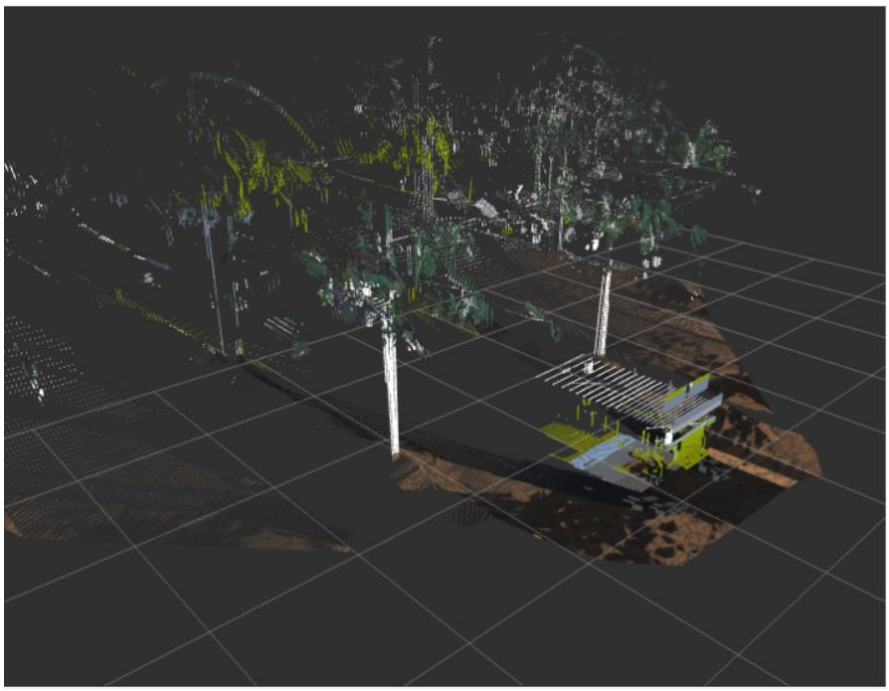

Figure 4 Simulated depth camera point cloud

3.3 LIDAR

A lidar sensor is simulated using a rotating camera sensor with a sample size equal to the real hardware lidar. For example, the Ouster OS1-64 will have a vertical resolution of 64 pixels and a horizontal 360 degrees resolution of 1024 pixels with a vertical field-of-view of 45 degrees. With a rotation frequency of 10Hz, the lidar simulation needs to compute 64*1024*10 = 655,360 points per second.

For lidar we have investigated two approaches:

1. Use 3D ray casting using physics on the CPU.
2. Use 3D ray casting using graphics depth maps on the GPU.

3.3.1 3D ray casting using CPU physics

This will perform ray-casting in a way that is similar to how the Lidar itself works: from the sensor a depth measurement is made in a specific direction which is repeated many times to produce the point cloud. So, for this implementation, we need to do this 655,360 times per second per lidar. We found that a recent processor (AMD RX39000) can do this in above 339 milliseconds.

Hierarchy | Live | Main Thread | CPU:717.78ms GPU:--ms

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	98.1%	0.0%	3	9.2 KB	704.51	0.06
Update.ScriptRunBehaviourUpdate	97.7%	0.0%	1	6.1 KB	701.43	0.00
BehaviourUpdate	97.7%	0.0%	1	6.1 KB	701.43	0.00
LidarRaycast.Update()	97.7%	50.3%	1	1.2 KB	701.28	361.61
Physics.Raycast	47.2%	47.2%	655360	0 B	339.50	339.50

Figure 5 Measurements for doing 655360 ray-casts per second

Although this looks promising, we have to realize that we have 2-3 lidars in the scene. Also, this performance figure only does the ray-casting: we do not take into account the construction of the

ROS message here. Add to that that we need to simulate depth cameras and all the rest of the simulation too; we see that performance can become an issue.

Running this code multithreaded is possible and will improve the performance. Still, when using multiple lidar sensors will take a lot of computing power.

Besides the computational cost, the physics ray cast has also limited precision. The ray cast runs against the physics colliders instead of the visible graphics. In Figure 6 you can see the visible structure and the colliders drawn with green lines. The physics colliders are often much simpler than the visible graphics to reduce the computational overhead of the physics engine. This can be seen in Figure 6 where the vines and leaves have no colliders by default.

In the Canopies environment, we also limit the number of colliders to limit the computational overhead for the dynamic physics computations. For instance, the vines and leaves of the plants do not have any collider. Only when a robot is very close (where this distance can be configured) and needs to interact with it like when pruning a branch, colliders are added.

This means that for the ray-cast the parts without colliders are invisible and will therefore not appear in the lidar's output. In practice this means that all vines will be invisible when this solution would be used.

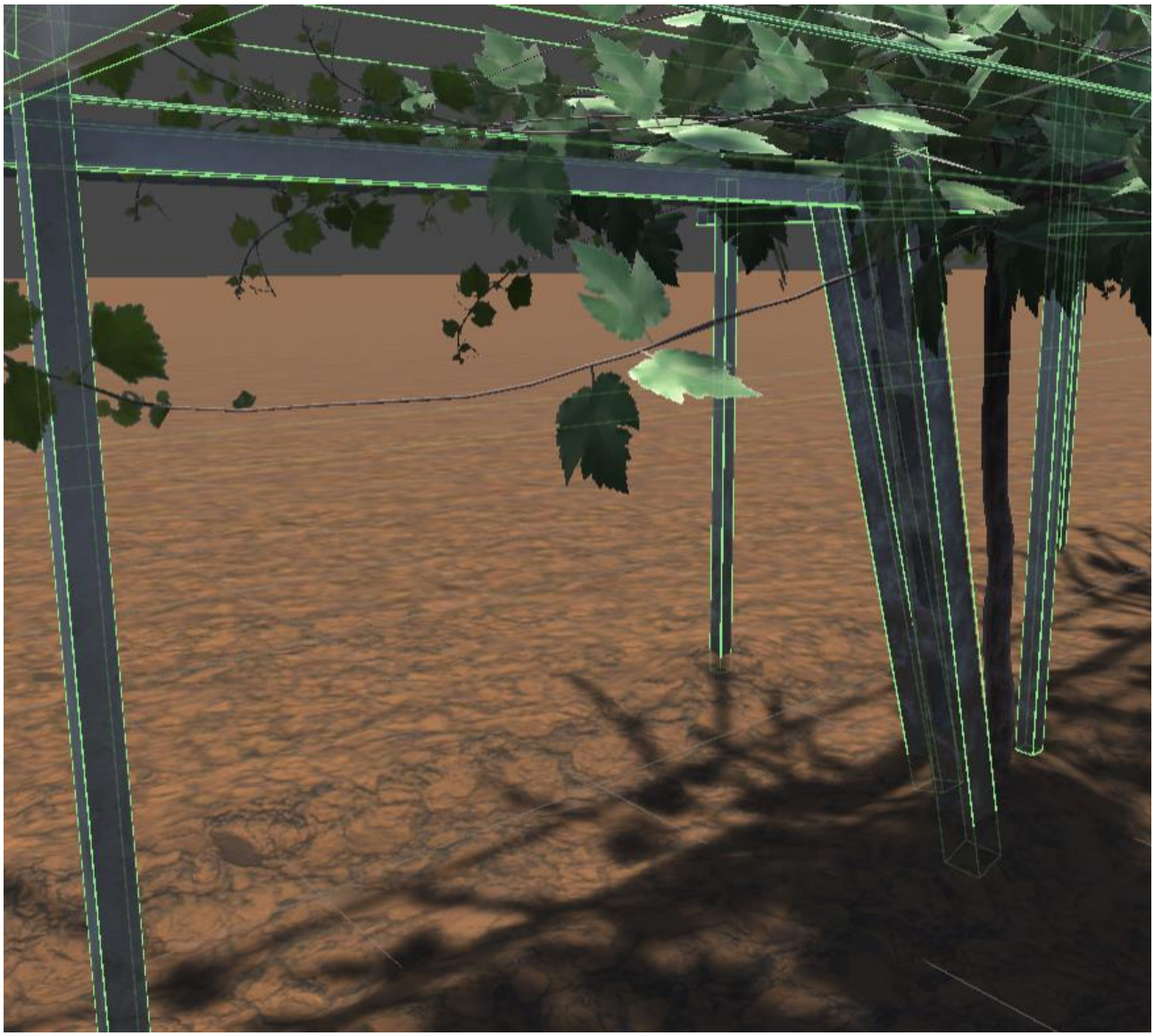

Figure 6 Canopies environment. The green lines represent colliders. Note that the branches and leaves do not have colliders.

3.3.2 3D ray casting using graphics depth maps

The alternative we chose is to use the depth maps used by the GPU to render the 3D scene. Depth maps are basically used to determine which parts of the meshes are occluded by other meshes and should therefore not be drawn. We can use that information to generate the lidar data.

This solution is very fast as it is executed by the GPU which is designed to compute lots of points in parallel plus the data is already generated and available in the GPU.

There are two complexities with the solutions: it is hard to generate a spherical depth map and we need to retrieve the data from the GPU back to the CPU.

As a workaround for the spherical depth map, we divide the circular motion of the lidar into a number of segments and use a planar depth map for each segment. This will produce a full 360 sweep of

correct data but has the drawback that the points are not sampled circularly. This is visible in the point cloud data in Figure 7 where a pentagram-like pattern of the sampled points is shown. Each point is accurate and correct. However, the places where the environment is sampled are a bit different than the real lidars.

Like with RGB and depth cameras, copying the data back from the GPU to the CPU introduced some latency compared to the ray-casting solution.

Overall, this solution proved to work good enough for our setup and we chose to use this option. The side-effect of this decision is that the computer simulating the sensors has to have a discrete graphics card (see [D2.4]).

3.3.3 Point clouds

The ROS point cloud is generated from the depth maps received from the GPU. The segments are combined to create the full 360 degrees image. The intensity value is based on the grayscale value of the texture of the mesh point of the point found by ray-cast. As this mesh point is not influenced by lighting or shading it should have a similar value as the real-world intensity value. It still needs to be evaluated whether this is sufficient for the application.

Like with the depth camera, the default data produced by the simulated lidars is error-free. Although this may be convenient for some tests, we will investigate the addition of random and systematic noise to the data to simulate more realistic data.

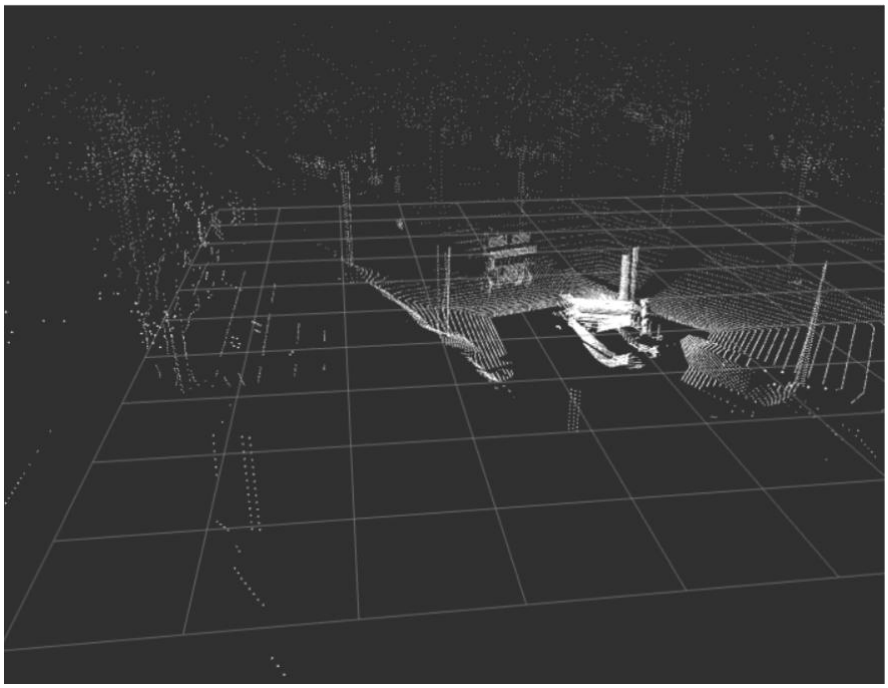

Figure 7 Simulated Lidar point cloud

Currently, the simulation just supports point clouds (sensor_msgs/PointCloud2), but more interfaces may be added later when needed.

3.4 GPS/IMU sensor

For the moving base, IMU and GPS sensors are used to get information about the position and orientation. The chosen sensors are the SBG Ellipse-E IME and an Septentrio GPS-RTK system as detailed in Deliverable 2.1 "Requirements, Specifications and Benchmarks". The SBG sensor uses GPS-RTK data to get more accurate location information.

On the ROS side, the SBG sensor outputs the combined data, so no direct interface to the Septentrio GPS sensor is needed. In the simulation, we therefore also have just one sensor which produces the same simulated data in the same way as the SBG sensor does.

The values of the GPS/IMU sensors can be derived directly from the movements of the mobile base in the simulated environment. As we use a fully dynamic model for the mobile base, information about position/rotation, velocity and acceleration are directly available. As we use dynamics for this, it already includes quite a lot of simulated noise in the measurements resulting from the uneven ground, the interaction between the tracks and the ground and the inertia of the mobile base itself. This does not include systematic errors like gyroscope velocity bias but these are compensated in the Septentrio using sensor fusion with an accelerometer, compass and the GPS-RTK sensors. We expect a low need to simulate these errors, but they can be added easily when the need for those arises.

This information is then used to produce the required messages for the ROS control of the mobile base.

3.5 Grape color, ripeness, sugar level sensors

In the simulation, grape ripening is not a dynamic process, but instead is represented as a set of pre-set grape bunch 3D objects, showing different stages of ripening. These stages have been based on input from farming partner AGRIMESSINA that provided pictures and materials, in order to ensure the highest quality of the virtual representation. The 3D objects in this set represent the level of ripeness as a texture color of the grapes, as described in Section 3.1.1 of [D.2.4].

Figure 8 Different ripeness levels represented by different 3D objects for grape bunches

The RGB-D sensors return the color that can be mapped to the sugar level. Notably, this represents the input for the development of the algorithms to be used for the assessment of the grape's ripeness. In this regard, PaleBlue together with UNIROMA1 have identified that the features of the RGB-D camera available in the Virtual Environment are sufficient for implementing the agronomic

sensing functionality. The algorithms for determining the ripeness of the grapes will be developed and tested on data taken from the simulation, as well as the real field, to ensure similar performance.

The algorithms for determining the ripeness of the grapes will be developed by exploiting real data collected from the field and then deployed in the simulator as a steppingstone for more complex human-robot interactions. Since the simulator relies on the appearance of real data, it is expected that the estimation algorithms in simulation behave in a similar way as in the experimental field. This will allow us to accurately simulate complex tasks such as harvesting decisions and grape bunch collection. With this, we consider the simulation part of the agronomic sensing to be completed.

3.6 Additional sensors

According to the current development of the research activities, it has been assessed that an RGB-D sensor is sufficient as an agronomic sensor. The use of more complex cameras, such as multispectral and hyperspectral cameras, has been considered, but in view of the hardware costs and difficult integration on the robotic platform (e.g., in addition to the cameras, a strong and multidirectional light source should be used for hyper-spectral cameras), the RGB-D sensor has been identified as the best trade-off between accuracy and efficiency.

4 References

Table 2 References

Id	Title	Version	Date
[D2.1]	Requirements, Specifications and Benchmarks	1.0	2021-04-30
[D2.2]	Specifications and KPIs for the two CANOPIES robot prototypes	1.0	2021-06-30
[D2.3]	Dimensioning of the Real-World Scenario for Final Demo	1.0	2021-10-29
[D2.4]	VR Farming Environment Specification	1.0	2021-10-29
[D3.4]	Simulated Farming Environment and Basic Robotic Components	1.0	2021-08-31