

**ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA**

---

**SCHOOL OF ENGINEERING AND ARCHITECTURE**

DEPARTMENT OF ELECTRICAL, ELECTRONIC  
AND INFORMATION ENGINEERING  
"GUGLIELMO MARCONI" - DEI

**Master's Degree**  
in  
AUTOMATION ENGINEERING

**DEVELOPMENT OF AN INDUSTRIAL LINE  
PARAMETRIC EDITOR IN VIRTUAL REALITY  
VIA AUTODESK VRED**

**Master Thesis**  
in  
MECHATRONICS SYSTEMS MODELING AND CONTROL M

Candidate:  
**Gian Marco Selleri**

Supervisor:  
**Prof. Ing. Alessandro Macchelli**

Advisors:  
**Ing. Davide Barchi**  
**Sig. Gildo Bosi**  
**Ing. Edoardo Reggiani**

Academic Year 2019/20  
Session I



# Abstract

This thesis describes the development of a parametric virtual reality editor for an industrial line for quality inspection, packaging and palletizing of ceramic tiles.

This project has been developed entirely within the R&D laboratory of Sacmi S.C., and includes a complete study of the implementation of an application of this kind, from the initial software and hardware selection phase, through the design of algorithms and code programming - in Python language - to the creation of interactive tools and widgets specifically designed for the exploration of the environment, interaction with the scene and automatic production of outputs, with the aim of framing this editor within the production process of automatic lines for the realization of ceramic materials.

The editor is defined as "parametric" due to the fact that it requires the user to provide as input a series of values necessary for the configuration of the lines - up to a maximum of three, each with a set of independent parameters - and then it automatically proceeds to compute the geometries and animate the scene, allowing the user to enter in an immersive and realistic context, in order to make a choice among the proposed lines.

Communication protocols have also been developed between different scenes for the transfer of data from one to the other, and for the transfer of the user between the different environments through commands given directly in virtual reality.

Subsequently, a phase aimed at optimizing the simulation was addressed, in order to increase its stability and decrease the possible stress induced during the presence in VR.

Finally, an early study was carried out on multi-user experience - the presence of several users in the same scene - with the aim of testing possible developments in the field of trade fairs and employee training.



# Contents

<b>Introduction</b>	<b>11</b>
<b>1 Virtual Reality - State Of The Art Hardware: Headset Gear</b>	<b>13</b>
1.1 HTC Vive Pro . . . . .	13
1.2 VRgineers XTAL . . . . .	16
1.3 HTC Vive Cosmos . . . . .	18
<b>2 Virtual Reality - Software Choice</b>	<b>21</b>
2.1 Unity . . . . .	21
2.2 Autodesk Vred . . . . .	22
<b>3 Single Line Editor</b>	<b>25</b>
3.1 EkoSort Building Alhorithm . . . . .	25
3.1.1 EkoSort - Automatic Stacker . . . . .	25
3.1.2 EkoSort Editor . . . . .	26
3.1.2.1 Preliminary Steps . . . . .	26
3.1.2.2 Library Realization . . . . .	26
3.1.2.3 Algorithm I – “Naïve” Solution . . . . .	28
3.1.2.4 Algorithm II – “Fast” Solution . . . . .	32
3.1.2.5 Compenetration Recovery . . . . .	34
3.1.2.6 Final Solution . . . . .	36
3.2 ExtraPack Building Algorithm . . . . .	37
3.2.1 ExtraPack - Palletizer . . . . .	37
3.2.2 ExtraPack Editor . . . . .	37
3.2.2.1 Library Realization . . . . .	37
3.2.2.2 Editor Algorithm . . . . .	38
3.3 Other Machines in the Line . . . . .	41
3.3.1 Flawmaster - Tile Inspector System . . . . .	41
3.3.2 Advancheck - Tile Inspector System . . . . .	42
3.3.3 EkoWrap - Packager . . . . .	42
3.3.4 Transport and Package Handling . . . . .	43
3.4 Animation . . . . .	44
3.5 G.U.I. Implementation . . . . .	49
3.5.1 Qt And PySide2 . . . . .	49
3.5.2 G.U.I. With Qt Designer . . . . .	50
3.5.2.1 Layout Choice . . . . .	50
3.5.2.2 Signal Management . . . . .	52
3.5.2.3 Code Implementation . . . . .	53
3.6 Final Line Editor . . . . .	58

<b>4</b>	<b>Three Lines Editor</b>	<b>61</b>
4.1	Editor Evolution . . . . .	61
4.1.1	Algorithm Adjustment . . . . .	61
4.1.2	Control Room . . . . .	62
4.1.2.1	Version I - "Compact" Solution . . . . .	62
4.1.2.2	Version II - "Split" Solution . . . . .	64
4.2	VR Tools Implementation . . . . .	69
4.2.1	Vred Variant Sets . . . . .	69
4.2.2	Widget Tools . . . . .	70
4.2.2.1	Building Widgets With PySide2 . . . . .	70
4.2.2.2	Editor Parameters Choice . . . . .	74
4.2.2.3	Parameters Display . . . . .	92
4.2.2.4	Output Creation . . . . .	93
4.2.3	Immersive Tools . . . . .	98
4.3	Model Optimization . . . . .	106
4.3.1	Performed Steps . . . . .	106
4.3.2	Final Results . . . . .	108
	<b>Conclusions</b>	<b>109</b>
	<b>Appendices</b>	<b>115</b>
A	EkoSort Building Code	115
B	ExtraPack Building Code	119
C	Animation Code	127
D	Final Editor Code: Main Section	151
	<b>List Of References</b>	<b>155</b>

# List of Tables

1.1	<b>Vive Pro</b> - Headset specs . . . . .	14
1.2	<b>Vive Pro</b> - Controller specs . . . . .	14
1.3	<b>Vive Pro</b> - Tracked area requirements . . . . .	14
1.4	<b>Vive Pro</b> - Hardware requirements . . . . .	15
1.5	<b>VRgineers XTAL</b> - Headset specs . . . . .	17
1.6	<b>VRgineers XTAL</b> - Software specs . . . . .	17
1.7	<b>Vive Cosmos</b> - Headset specs . . . . .	19
1.8	<b>Vive Cosmos</b> - Controller specs . . . . .	19
1.9	<b>Vive Cosmos</b> - Tracked area requirements . . . . .	19
1.10	<b>Vive Cosmos</b> - Hardware requirements . . . . .	19
3.1	<b>EkoSort: Parameters</b> . . . . .	28
3.2	<b>Parameters Boundaries: Configuration dependency</b> . . . . .	52
3.3	<b>Python code: GUI implementation</b> . . . . .	55
3.4	<b>Python code: Widget functions</b> . . . . .	57
4.1	<b>Batch file: Open Editor Scenes</b> . . . . .	66
4.2	<b>Python code: Control Room - Send Data Lists</b> . . . . .	68
4.3	<b>Variant Sets module</b> - Context menu options . . . . .	70
4.4	<b>Variant Sets module</b> - Tabs . . . . .	71
4.5	<b>Python code: Open parameters widget (1)</b> . . . . .	76
4.6	<b>Python code: Open parameters widget (2)</b> . . . . .	77
4.7	<b>Python code: Update lineEdit</b> . . . . .	78
4.8	<b>Python code: Update expected n° of stations</b> . . . . .	80
4.9	<b>Python code: Show configuration choice elements</b> . . . . .	81
4.10	<b>Python code: Round parameter value</b> . . . . .	82
4.11	<b>Python code: Enable parameter editing</b> . . . . .	83
4.12	<b>Python code: Type a character</b> . . . . .	84
4.13	<b>Python code: Delete a character</b> . . . . .	85
4.14	<b>Python code: Confirm value</b> . . . . .	86
4.15	<b>Python code: Next line widget (updating lists)</b> . . . . .	87
4.16	<b>Python code: Next line widget (preparing widget)</b> . . . . .	88
4.17	<b>Python code: Back to selection of number of lines</b> . . . . .	89
4.18	<b>Python code: Back to previous line</b> . . . . .	91
4.19	<b>Python code: Open first line description</b> . . . . .	92
4.20	<b>Python code: Type a character</b> . . . . .	95
4.21	<b>Python code: Create Excel file (1)</b> . . . . .	96
4.22	<b>Python code: Create Excel file (2)</b> . . . . .	97
4.23	<b>Python code: Switch between scenes</b> . . . . .	99

4.24 **Python code:** Toggle annotations . . . . . 100

4.25 **Python code:** Hide all annotations . . . . . 100

4.26 **Python code:** Create hidden VR menus . . . . . 101

4.27 **Python code:** Toggle calculator tool . . . . . 102

4.28 **Collaboration:** Connection Requirements . . . . . 104

4.29 **Collaboration:** Hardware Requirements . . . . . 105

4.30 **Collaboration:** Software Requirements . . . . . 105



# List of Figures

3.1	EkoSort Machine (courtesy of SACMI Imola S.C.) . . . . .	26
3.2	Automotive Rendering Tests . . . . .	26
3.3	EkoSort's scenegraph in Vreds . . . . .	27
3.4	EkoSort Parts Library . . . . .	28
3.5	Compenetration warnings . . . . .	35
3.6	Examples of possible EkoSort building . . . . .	36
3.7	Extrapack Parts Library . . . . .	38
3.8	Examples of possible EkoSort building . . . . .	39
3.9	Flawmaster Machine (courtesy of SACMI Imola S.C.) . . . . .	42
3.10	EkoWrap Machine (courtesy of SACMI Imola S.C.) . . . . .	43
3.12	Curves for Translation, Rotation and Visibility values of the stack of tiles on the first station of the EkoSort . . . . .	45
3.13	Example of the tree organization of geometries in Vred . . . . .	45
3.14	Qt Designer page: main widget (editor parameters) - Widget Editing . . .	50
3.15	Qt Designer page: main widget (editor parameters) - Signals/Slots Editing	53
3.16	Examples of signal connections . . . . .	54
3.17	Final appearance of the widget . . . . .	58
3.18	Final render of the line . . . . .	59
4.1	Control room rendering . . . . .	63
4.2	Control room rendering (secondary scene) . . . . .	64
4.3	Balcony rendering (main scene) . . . . .	64
4.4	Editor's Variant Sets module . . . . .	69
4.5	QPushButton . . . . .	72
4.6	QLineEdit . . . . .	72
4.7	QRadioButton . . . . .	73
4.8	QProgressBar . . . . .	73
4.9	Editor parameters widget . . . . .	74
4.10	Parameters display widget . . . . .	92
4.11	Output creation widget . . . . .	93
4.12	Excel output example . . . . .	94
4.13	VR interactive menu . . . . .	98
4.14	html Calculator . . . . .	101
4.15	Material selectors . . . . .	103
4.16	Virtual Reality User Avatars . . . . .	104



# Introduction

In recent years, the so-called "immersive reality" has become increasingly popular in everyday life. Not only with respect to the video game and entertainment sector, but also and above all in the industrial and commercial field. Just think, for example, of fair simulations, 3D visualizations to create preventive configurations, or those extremely powerful tools that are remote assistance or training of personnel in simulation. And these are just some of the many applications that can be used as examples.

In cases like this one can really say that the only limit is the imagination and skill of the developers (and only secondarily the hardware, which is developing extremely quickly in order to follow the needs of the market).

The aim of this thesis is to investigate the potential and possible developments that these developing technologies can offer in the context of a company workflow in the field of ceramic production lines, of which Sacmi is a global leader.

In order to provide a context for this, first of all a research was carried out on what is the current state of the art with regard to the headsets available on the market, in order to make a targeted purchase according to the needs of the project being presented here.

But precisely, what is the project on which this thesis is focused?

As mentioned earlier, before going on to definitively define the direction of this thesis, a preliminary question was posed: in which part of the company workflow (so in this case the realization of a production line for the production of ceramic tiles) can the virtual reality be inserted, enhancing the established technologies?

There are two main and most immediate answers: at the beginning and at the end of the production process. That is to say, in the initial configuration and final testing phases, both for a mere static visualization of the final line, and for an effective immersive check of the simulations carried out in the design phase.

The main focus of this thesis was therefore addressed to the first point of this list: the idea was to exploit the immersion provided by virtual reality, and the strong and immediate perception of a fictional scene that this instrument can provide, to create a configurator for a line of sorting of ceramic tiles, through the software Autodesk Vred, already widely used in applications of immersive reality.

The real power of this instrument lies in being able to make parametric models of machines that, in the original files supplied by the technical office, would be fixed and unchangeable. Instead, using the chosen program, which allows to modify and manage the geometries for a large number of file types, these models can be automatically adapted (through a script) to those that are the needs of the customer, quickly and effectively, and

then assembled in a line, and finally be displayed in a realistic and lifelike scene, all in a matter of minutes.

Now, in what could be defined as an ideal world (at least as far as industry is concerned), each production phase should be able to be interfaced in an almost immediate and automatic way with those immediately preceding and following, in such a way as to reduce to a minimum the problems of interfacing between different hardware and software, but also between different people and offices or departments (always referring to a business context), up to that almost utopian concept that could be summed up in the phrase "data must be entered only once"; that is to say: from the initial configuration it must be possible to obtain all the data useful for the following phases, without them being re-entered, over and over again.

Therefore, with this premise, the practical contribution of this application concerning the line configurator - beyond the immersive representation offered to a potential customer - lies in the creation of geometries that already reflect the final requirements, and therefore can be exported and used (even as a simple reference) in the subsequent stages of design, bypassing the ex novo creation of these models.

But now, without further ado, let's dive into the world of virtual reality.

# Chapter 1

## Virtual Reality - State Of The Art Hardware: Headset Gear

For an initial development of the Virtual Reality application, a cooperation was initially established with Protesa, a subsidiary of the Sacmi group, which provided its knowledge and experience in the field, both for the software and hardware parts.

### 1.1 HTC Vive Pro

For the hardware part, Protesa makes use of the VIVE Pro headset. It has been designed to optimize VR experience, starting from the study of comfortable ergonomics for the user [17]: it has a simple structure, in which the weight of the device is distributed in order to achieve an optimal center of gravity, allowing him/her not to strain his/her neck. This can be useful especially for long sessions in sales environment: the client can use our application for long periods of time without feeling uncomfortable.

Moreover, even if the standard version of this headset involves the use of cables, it is already predisposed for a wireless solution using an adapter (always by VIVE), already used in Protesa laboratory. An interesting development for this case is the predisposition for multiuser virtual reality.

But the main advantage of the VIVE Pro is the optimal resolution that it can provide with dual-AMOLED displays with a combined resolution of 2880 x 1600 pixels (combining the two lens) and 615 PPI, allowing the user to achieve full immersion in the digital environment created by our application. Moreover, it is equipped with a Pro Eye VR system, designed to improve simulations and processing with a precision eye-tracking technology, that enables the developer to see what the user is seeing in session, and act accordingly. This technology can lead to a big improvement in training and sales scenarios, allowing deeper data analysis. Moreover, in this way input and navigation are greatly simplified and kept more intuitive, implementing gaze-oriented menu navigation, and enabling more natural movements and gesture control in virtual reality.

Another innovation of the VIVE Pro visor is the “Foveated Rendering”: a technique that allows intelligent allocation of GPU workload, improving visualization quality and performance while optimizing graphic fidelity in users’ line of sight. Basically, according to this technique, the field of view is divided into three main areas: the Foveal area (in direct

line of sight in front of the pupil, around the focal point) in which the visor gives the highest resolution, then the Blend and the Peripheral areas, with mid and low resolution, respectively, as they pull away from the main one.

Specifications are shown in tables 1.1, 1.2, 1.3 and 1.4.

Table 1.1: **Vive Pro** - Headset specs

---

<b>Screen:</b>	Dual AMOLED 3.5" diagonal
<b>Resolution:</b>	1440 x 1600 pixels per eye (2880 x 1600 pixels combined)
<b>Refresh Rate:</b>	90 Hz
<b>Field of View:</b>	110 degrees
<b>Audio:</b>	<ul style="list-style-type: none"><li>• Hi-Res certificate headset</li><li>• Hi-Res certificate headphone (removable)</li><li>• High impedance headphone support</li></ul>
<b>Input:</b>	Integrated microphones
<b>Connections:</b>	USB-C 3.0, DP 1.2, Bluetooth
<b>Sensors:</b>	SteamVR Tracking, G-sensor, gyroscope, proximity, IPD sensor
<b>Ergonomics:</b>	<ul style="list-style-type: none"><li>• Eye relief with lens distance adjustment</li><li>• Adjustable IPD</li><li>• Adjustable headphone</li><li>• Adjustable headstrap</li></ul>

---

Table 1.2: **Vive Pro** - Controller specs

---

<b>Sensors:</b>	SteamVR Tracking 2.0
<b>Input:</b>	Multifunction trackpad, Grip buttons, Dual-stage trigger, System button, Menu button
<b>Connections:</b>	Micro-USB charging port

---

Table 1.3: **Vive Pro** - Tracked area requirements

---

<b>Standing / Seated:</b>	No min. space requirements
<b>Room-scale:</b>	A minimum play area of 6'6" x 4'11" is required, while the maximum size is 22'11" x 22'11"

---

Table 1.4: **Vive Pro** - Hardware requirements

---

<b>Processor:</b>	Intel® Core™ i5-4590 or AMD FX™ 8350 equivalent or better
<b>Graphics:</b>	NVIDIA® GeForce® GTX 970 or AMD Radeon™ R9 290 equivalent or better
<b>Memory:</b>	4 GB RAM or more
<b>Video out:</b>	DisplayPort 1.2 or newer
<b>USB Ports:</b>	1x USB 3.0 or newer
<b>Operating system:</b>	Windows® 10

---

## 1.2 VRgineers XTAL

A direct competitor in the field of virtual reality visors for professional engineering applications is the VRgineers XTAL visor, which is used to develop applications regarding virtual design evaluation, virtual prototyping, surgeon and employee training, virtual product configuration [18].

Its main advantage is given by the really high resolution that it can achieve, since this visor can render 5k images, thanks to high-density OLED displays installed together with custom-built lenses.

Like the VIVE Pro, the XTAL presents an easy-to-adjust ergonomics, with improved comfort due to the artificial leather face cushion. Moreover, it is designed for usage with eyeglasses, with an advanced adjustable focus allowing compensation for dioptries. Another interesting feature is Autoeye, an automatic lens setting, based on the user's interpupillary distance (IPD) using integrated eye cameras, which adjust the lenses for optimal image quality.

All these features are useful in a case like ours, where the user is changed frequently.

In addition, it is already embedded a leap motion hand-tracking sensor, that allows users to interact with a VR scene naturally, using their own hands instead of controllers with great stability and reliability, including finger fidelity.

An additional advantage for our application is the direct built-in integration with the software Vred (already used in Sacmi's Innovation Lab).

Specifications are shown in tables 1.5 and 1.6.



Table 1.5: **VRgineers XTAL** - Headset specs

<b>Screen:</b>	Dual OLED with low latency, low persistence, fast color switching, no blurring with fast movement
<b>Resolution:</b>	5120 x 1440 (2xQuad HD, i.e. 2560 x 1440 per eye)
<b>Refresh Rate:</b>	70 Hz
<b>Field of View:</b>	Up to 180 degrees (based on focus setting)
<b>Audio:</b>	Built-in soundcard with audio jack
<b>Input:</b>	Integrated microphone with voice commands recognition system (accessible via Windows native drivers)
<b>Optics:</b>	Patented custom-designed aspherical non-Fresnel VR lenses
<b>Ergonomics:</b>	Advanced hard head strap with artificial leather cushioning Replaceable face cushions made of artificial leather for high hygiene level
<b>Dimensions:</b>	<ul style="list-style-type: none"> <li>• Height: 299 mm</li> <li>• Width: 123 mm</li> <li>• Depth: 140 mm</li> </ul>
<b>Weight:</b>	<ul style="list-style-type: none"> <li>• Headset: 770 g (without head strap)</li> <li>• Head mount incl. balance counterweight: 440 g</li> </ul>

Table 1.6: **VRgineers XTAL** - Software specs

<b>Software Included:</b>	<ul style="list-style-type: none"> <li>• Headset Configuration Utility</li> <li>• Unreal Engine SDK</li> <li>• Unity 3D SDK</li> <li>• XTAL C++ Libraries for proprietary render engine</li> <li>• Integration (optional)</li> </ul>
<b>Software Compatibility:</b>	<ul style="list-style-type: none"> <li>• SteamVR (OpenVR) support</li> <li>• Autodesk VRED</li> <li>• Dassault Systèmes solutions</li> <li>• ESI IC.IDO</li> <li>• OS Windows 7/10</li> </ul>

## 1.3 HTC Vive Cosmos

The newest proposal in the field of virtual reality visors by Htc is the Vive Cosmos, first distributed on 3rd October 2019 [16]. It is characterized by the biggest resolution offered by Vive yet (2880 x 1700 pixels, combining the two lenses), using two lenses of 3.4” each.

The main advantage offered by this headset is its portability: it does not require fixed stations to delimit the perimeter of the play area and to compute movements, but instead these functions are based on the use of six camera sensors, providing an accurate inside-out tracking via wide field of view (FOV) and six-degree-of-freedom (6DoF) support. In this way, every movement in real space is brought up in virtual space based on the change in perspective with respect to the physical surroundings. Which means that inside-out tracking enables plug-and-play portability: it can be used everywhere, simply plugging it in a VR-ready PC desktop or laptop.

Regarding ergonomics, the forefront of the headset has been redesigned with respect to previous models, suiting a wide range of face shapes, and also supporting the usage of glasses. The interpupillary dial, however, still needs to be adjusted manually. Moreover, in order to facilitate the switching between reality and virtual reality, a flip-up design has been introduced: in any moment, the user can exit from the simulation by flipping up the front part (almost like a motorcycle helmet), without having to remove the entire headset.

Finally, the Vive Cosmos is prepared to be expanded with a suite of modular options (some still in development). Between these modules, the most important ones are the wireless adapter (in order to get rid of cables connecting the headset to the workstation) and the external tracking mod, used to combine the tracking methods of the Cosmos with the precision derived from the fixed stations (useful for example in fares, where the area of usage of the visor is fixed for long periods of time).

All these advantages (together with the excellent relationship between quality and price) have led to converge on this headset as the final choice of hardware to be purchased for the application presented in this thesis.

Specifications are shown in tables 1.7, 1.8, 1.9 and 1.10.

Table 1.7: **Vive Cosmos** - Headset specs

<b>Screen:</b>	Dual 3.4" diagonal
<b>Resolution:</b>	1440 x 1700 pixels per eye (2880 x 1700 pixels combined)
<b>Refresh Rate:</b>	90 Hz
<b>Field of View:</b>	Maximum 110 degrees
<b>Audio:</b>	Stereo Headphone
<b>Input:</b>	Integrated microphones, Headset button
<b>Connections:</b>	USB-C 3.0, DP 1.2, Proprietary Connection to Mods
<b>Sensors:</b>	G-sensor, gyroscope, IPD sensor
<b>Ergonomics:</b>	<ul style="list-style-type: none"> <li>• Flip-up visor;</li> <li>• Adjustable IPD;</li> <li>• Adjustable headstrap.</li> </ul>

Table 1.8: **Vive Cosmos** - Controller specs

<b>Sensors:</b>	Built-in sensors: <ul style="list-style-type: none"> <li>• Gyro and G sensors;</li> <li>• Hall sensor;</li> <li>• Touch sensors.</li> </ul>
<b>Input:</b>	<ul style="list-style-type: none"> <li>• System button;</li> <li>• 2 Application buttons;</li> <li>• Trigger;</li> <li>• Bumper;</li> <li>• Joystick;</li> <li>• Grip button.</li> </ul>
<b>Connections:</b>	Micro-USB charging port

Table 1.9: **Vive Cosmos** - Tracked area requirements

<b>Standing / Seated:</b>	No min. space requirements
<b>Room-scale:</b>	Minimum is 2m x 1.5m for room-scale mode

Table 1.10: **Vive Cosmos** - Hardware requirements

<b>Processor:</b>	Intel® Core™ i5-4590 or AMD FX™ 8350 equivalent or better
<b>Graphics:</b>	NVIDIA® GeForce® GTX 970 4GB, AMD Radeon™ R9 290 4GB equivalent or better VR Ready graphics card
<b>Memory:</b>	4 GB RAM or more
<b>Video out:</b>	DisplayPort 1.2 or newer
<b>USB Ports:</b>	1x USB 3.0 or newer
<b>Operating system:</b>	Windows® 10



## Chapter 2

# Virtual Reality - Software Choice

### 2.1 Unity

Unity is a multi-platform graphics engine developed by Unity Technologies that allows the development of video games and other interactive content, such as architectural visualizations or real-time 3D animations.

It is therefore particularly flexible, as it allows the writing of a wide range of programs with different intentions. The engine can be used to create three-dimensional, two-dimensional, virtual reality, and augmented reality games, as well as simulations and other experiences. The engine has been adopted by industries outside video gaming, such as film, automotive, architecture, engineering and construction.

It also provides assistance to the programmer, being an integrated development environment, i.e. during the writing of the source code, the software provides assistance, directly reporting syntax errors in the code, and providing debugging support tools. It therefore presents an environment in which users can directly view the object hierarchy, a visual editor, a detailed property viewer and a live preview of the game.

The engine offers a primary scripting API in *C#*, for both the Unity editor in the form of plugins, and games themselves, as well as drag and drop functionality.

For 3D games, Unity allows specification of texture compression, mipmaps, and resolution settings for each platform that the game engine supports, and provides support for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects.

However, despite this set of advantages, this software has been discarded for the purposes of the project presented here, as the acquisition of the know-how necessary for the development of a virtual reality application with this tool would have taken too long to achieve concrete results within the time horizon set for this thesis. Therefore the focus was shifted to a software already widely used by Protesa: Autodesk Vred.

## 2.2 Autodesk Vred

Vred is a software developed by Autodesk, used for 3D visualization and virtual prototyping, mainly used by automotive designers, since it can create product presentations, design reviews and virtual prototypes in real time.

It is particularly useful since it focuses only on geometries and materials, without the weight of a physical simulation, which is not mandatory in a mostly graphic task like the realization of an editor, with the aim of building a representation of a machine or a whole line to be inserted in a virtual reality context, in order to have a reference on how it would look and “feel” in a real environment.

Therefore, Vred comes in handy with its ability to manage geometries, by importing parts that can be realized with almost every existing CAD (since it is compatible with a great number of formats - see list below), converting them in native files, and manipulating their positioning (rototranslations). Moreover, it is able to create animations using curves in time for values of translation and rotation.

Vred supports loading (import) of 3D geometry and scenes from the following file formats:

- 3ds Max (.3ds)
- Alias (.wire)
- ASC Dental (.asc)
- AutoCAD (.dwg, .dxf)
- CATIA (.catpart, .catproduct, .cgr, .dlv, .dlv3, .dlv4, .exp, .mdl, .model, .session)
- Autodesk Inventor (.ipt, .iam)
- Cinema 4D (.c4d)
- Cosmo3D (.csb)
- Deltagen (.rtx)
- FBX (.fbx)
- FHS (.fhs)
- GeomView (.off)
- IGES (.igs)
- JT (.jt)
- Maya (.ma, .mb)
- OpenSG (.osb)
- Open Inventor (.iv)

- PLM XML (.plmxml)
- PLY (.ply)
- Pro/E Granite (.g)
- Pro/E Neutral (.neu)
- Pro/E Render (.slp)
- Python Script (.py)
- Rhinoceros (.3dm)
- Showcase (.apf)
- SolidWorks (.sldprt, .sldasm, .prt, .asm)
- STEP (.stp)
- Stereolithography (.stl)
- VRED (.vpe, .vpb, .vpf)
- VRML (.wrl)
- Wavefront (.obj)

Moreover, Vred is optimized for the realization of virtual reality scenes and environments by using Python scripts.





# Chapter 3

## Single Line Editor

Focusing on the side of pre-selling, the field of virtual reality shows its true power in showing to clients an early idea of how a new line would look like, on its dimensions and on its layout possibilities.

That's the reason why, as a first approach, it has been developed an automatic parametric graphic editor which, taking info from the client about some parameters, can develop a render of the new line and make it explorable in VR. If the result is not satisfactory, the editor can be restarted quickly to easily build a new line with a change in the parameters.

As a first step, this editor has been developed for a single machine (Nuova Sima's EkoSort), and then expanded to a whole sorting line, including quality control, packaging and package handling, up to storing packages on pallets.

### 3.1 EkoSort Building Alhorithm

#### 3.1.1 EkoSort - Automatic Stacker

The EkoSort machine is a sorting line for ceramic tiles, and one of the main products of Sacmi's consociate Nuova Sima [11].

It is a simplified sorting system that can be used, thanks to special suction cups, whatever the tile size or thickness, with consequent minimization of changeover times in case of flexible productions.

It is also extremely compact, as a result of a circular structure that makes full use of all the effectively available space, thus allowing efficient handling of even the largest tiles.

EkoSort handles products gently during the stacking phase: it limits the shock undergone by tiles during stacking, thus minimizing environmental impact, noise levels and production line complexity, as well as reducing wastes of products. All of this is possible thanks to the innovative pick-up system which ensures that, if a tile "falls", it is placed gently on the underlying stack.

The machine is exceptionally reliable thanks to extremely limited maintenance requirements: on one hand, this has been achieved by reducing the number of parts subject to wear (e.g. by eliminating belts) and, on the other, by the circular turret geometry that provides an easy access to all machine parts.

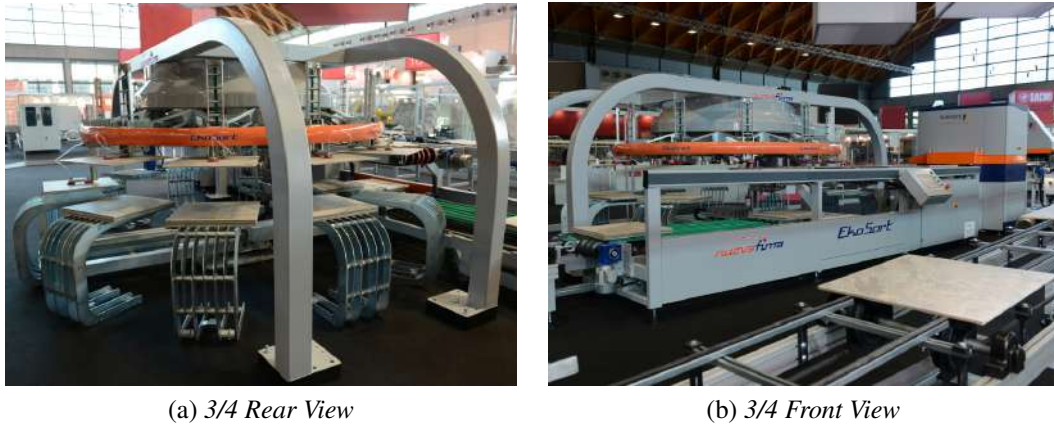


Figure 3.1: EkoSort Machine (courtesy of SACMI Imola S.C.)

### 3.1.2 EkoSort Editor

#### 3.1.2.1 Preliminary Steps

In order to get a preliminary knowledge of the software, a full online tutorial, present in the official Autodesk website, has been watched and followed, thus permitting to learn all the basics of handling geometries and the creation of scenes, from lighting, to animating, to rendering.

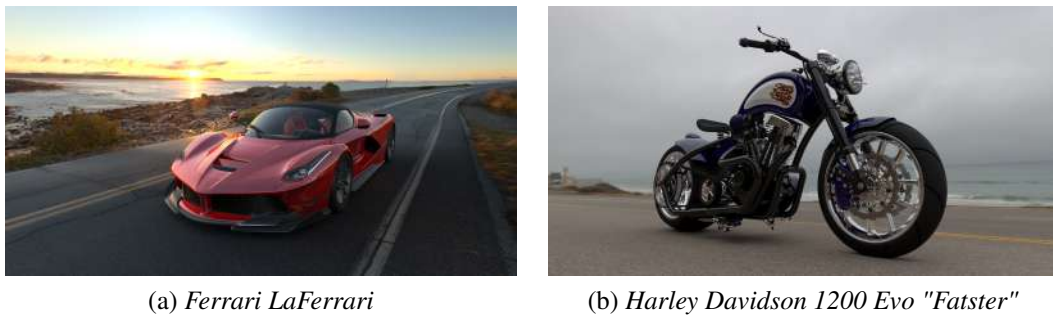


Figure 3.2: Automotive Rendering Tests

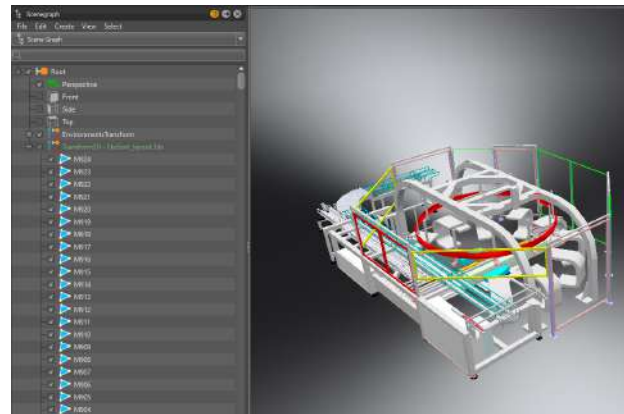
After this first tutorial, a few different models (not linked to the sorting line) have been realized, in order to get a first hands-on experience of the software (figure 3.2).

Then, in order to have a powerful and flexible tool to create an automatic method to realize the editor, and since Vred uses Python as main programming language, a couple online courses on the basics of this language have been attended on the web platform Udemy.

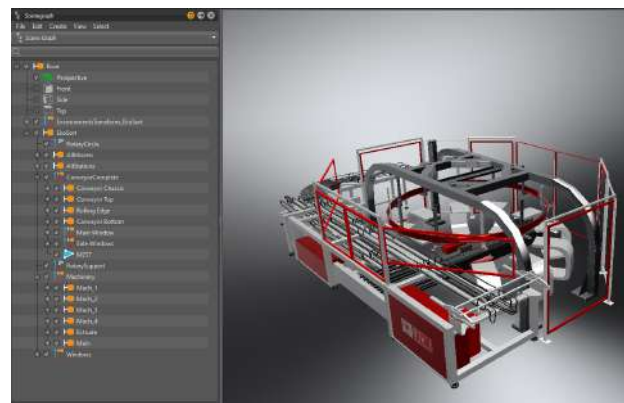
#### 3.1.2.2 Library Realization

As a first step in realizing the editor, a library of parts needed to be created. In order to do so, the original CAD file (3D object: .3ds) has been opened in Vred.

As seen in figure 3.3a, the machine is nothing but a group of single geometries, without a particular order, nor organization.



(a) *Original*



(b) *Reorganized*

Figure 3.3: EkoSort's scenegraph in Vreds

So, the next step has been to group all of them into parts, in order to better organize everything and to make it easier to isolate every object later. More realistic materials have also been applied. The result is shown in figure 3.3b.

Then, every part has been saved separately, in order to be loaded time by time and reassembled by the algorithm developed later.

The complete library of parts resulting (shown in figure 3.4) is then:

- (a) Mover 0
- (b) Station 0
- (c) Rotary Circle
- (d) Rotary Support
- (e) Conveyor Complete
- (f) Machinery
- (g) Windows

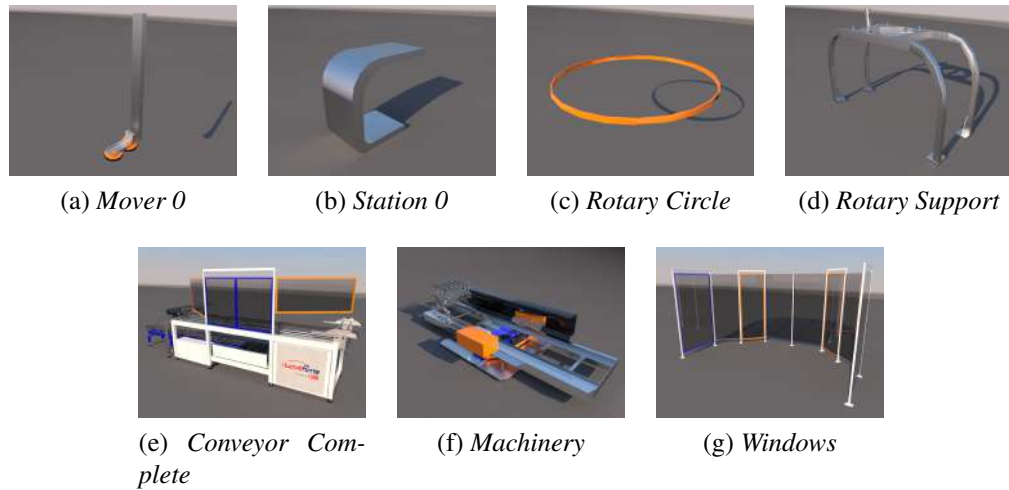


Figure 3.4: EkoSort Parts Library

### 3.1.2.3 Algorithm I – “Naïve” Solution

The main challenges in this scenario have been the construction of the carousel (with the number of movers and the diameter selected by user), and the placement of the stations under the movers, and, in particular, how many of them to actually put in place, making sure that they were not touching the conveyor belt.

As a first approach, the carousel’s realization had been faced by using Vred’s function “clone”, which creates a copy of an existing part which is linked to it, i.e., if the original part is scaled or moved, the clone is modified as well.

Obviously the parameters to be passed to the editor needed limitations (dictated by the catalogue of possible machine configurations).

The possible choices for the parameters are shown in Table 3.1.

Table 3.1: EkoSort: Parameters

Parameter	Min. Value	Max. Value
Total N° of Movers	4	16
Carousel Diameter [mm]	2000	4000

The first algorithm developed (in pseudo-code) is shown in Algorithm 1.

**Algorithm 1** “Naïve” Solution for EkoSort editor

---

- 1:  $n$  = desired number of movers
- 2:  $h$  = mover’s suction cup height
- 3:  $d$  = carousel’s diameter
- 4:  $r = d/2$

**Rotary Circle**

- 5: Import (RotaryCircle)
- 6: Translate along  $Z$  axis
- 7: Scale on  $X$  and  $Y$  dimensions

**Movers**

- 8: **if**  $n = \text{even number}$  **then**
- 9:      $limit = n/2$
- 10:    **for** ( $i = 0, i < limit$ ) **do**
- 11:       Import (mover)
- 12:       Translate in position
- 13:       Set rotation pivot to world center
- 14:       Clone (mover) along  $Y$  axis
- 15:       Group mover and clone
- 16:       Rotate around  $Z$  axis of  $i \cdot (360^\circ)/n$
- 17:        $i++$
- 18:    **end for**
- 19: **else**
- 20:      $limit = \lceil n/2 \rceil$
- 21:    **for** ( $i = 0, i < limit$ ) **do**
- 22:       Import (mover)
- 23:       Translate in position
- 24:       Set rotation pivot to world center
- 25:       **if**  $i = 0$  **then**
- 26:          Do nothing
- 27:       **else**
- 28:          Rotate around  $Z$  axis of  $i \cdot (360^\circ)/n$
- 29:          Clone mover along  $X$  axis
- 30:       **end if**
- 31:    **end for**
- 32: **end if**
- 33: Group all movers together

**Stations**

- 34: **if**  $n = \text{even number}$  **then**
  - 35:    **if**  $n \leq 6$  **then**
  - 36:       **for** ( $i = 1, i < limit$ ) **do**
  - 37:          **if**  $i = limit/2$  **then**
  - 38:            Import(station)
-

---

```
39:          Scale dimensions
40:          Translate in position
41:          Rotate around  $Z$  axis of  $180^\circ$ 
42:           $i++$ 
43:      else
44:          Import (station)
45:          Scale dimensions
46:          Translate in position
47:          Set rotation pivot to world center
48:          Clone (station) along  $Y$  axis
49:          Group station and clone
50:          Rotate group around  $Z$  axis of  $i * (360^\circ)/n$ 
51:           $i++$ 
52:      end if
53:  end for
54: else
55:  for ( $i = 2, i < limit$ ) do
56:    if  $i = limit/2$  then
57:      Import(station)
58:      Scale dimensions
59:      Translate in position
60:      Rotate around  $Z$  axis of  $180^\circ$ 
61:       $i++$ 
62:    else
63:      Import (station)
64:      Scale dimensions
65:      Translate in position
66:      Set rotation pivot to world center
67:      Clone (station) along  $Y$  axis
68:      Group station and clone
69:      Rotate group around  $Z$  axis of  $i * (360^\circ)/n$ 
70:       $i++$ 
71:    end if
72:  end for
73: end if
74: else
75:  if  $n = 5$  then
76:    for ( $i = 1, i < limit$ ) do
77:      Import (station)
78:      Scale dimensions
79:      Translate in position
80:      Set rotation pivot to world center
81:      Rotate group around  $Z$  axis of  $i * (360^\circ)/n$ 
82:      Clone station along  $X$  axis
83:      Group station and clone
84:       $i++$ 
85:    end for
```

---

---

```
86:     else
87:         for ( $i = 2, i < limit - 1$ ) do
88:             Import (station)
89:             Scale dimensions
90:             Translate in position
91:             Set rotation pivot to world center
92:             Rotate group around  $Z$  axis of  $i * (360^\circ)/n$ 
93:             Clone station along  $X$  axis
94:             Group station and clone
95:              $i++$ 
96:         end for
97:     end if
98: end if
99: Group all stations together
```

### **Rotary Support**

```
100: Import (RotarySupport)
101: Scale dimensions
```

### **Conveyor**

```
102: Import (ConveyorComplete)
103: Translate along Y axis
104: Scale dimensions
```

### **Machinery**

```
105: Import (Machinery)
106: Scale dimensions
```

### **Windows**

```
107: Import (Windows)
108: Scale dimensions
```

### **EkoSort**

```
109: Group everything
```

---

First of all, at a first glance it can easily be seen that it wasn't well organized: it was all in one block, and it didn't present any modularity. Hence, it greatly lost readability, especially when switching to effective Python code.

That's the reason why a new version of this code has been written, organizing everything in functions and function calls in the "main" part of the code. These effort had the result of producing a greatly readable and shorter code.

But the main flaw of this code was to be found in the great usage of the function “clone” to realize couples of movers and stations. The choice of relying on this procedure was made starting on the fact that every clone can inherit the transformation of the original object (called “node” in Vred), so, while handling movers and stations using mouse and visual interface on-screen could really benefit of this connection, the same thing couldn’t be said regarding manipulation of nodes via script, which required lots of renaming of previous loaded nodes (in order to select them properly). So, in the switching between direct (visual) and scripted building of the scene, almost every positive aspect of the usage of clones was lost.

Moreover, most of the time used by the script execution was due to loading time, since the other computations have been proven to be really fast.

With this script, since the smallest EkoSort constructible has 5 movers, a minimum of 10 loadings is required (3 movers, 2 stations, 5 other parts). But when the user asked for 16 movers, the number of loading procedures went up to 20 (8 movers, 7 stations, 5 other parts). This has led to a great lengthening of the computation time for the final construction of the machine..

These problems were found to be too limiting, and the need to develop a new, more efficient algorithm arose.

#### **3.1.2.4 Algorithm II – “Fast” Solution**

With the problems previously underlined in mind, a second algorithm has been developed. The main idea is simple, but effective: instead of relying on so many loadings for movers and stations, only the first one should be loaded and translated in position. Then it should be duplicated inside Vred, and the clone rotated with respect to the central axis of the carousel. Repeating the process  $n$  times brings to the complete filling of the carousel. Same thing for the stations, with the difference that, based on the number of movers and the carousel diameter, the code must skip the placing of 1, 3, or even 5 stations, in order to avoid compenetrations (in the editor viualization, in reality these stations simply wouldn’t fit in the configuration) between the conveyor belt and the nearest stations.

The selection of the number of station placings to skip has been done empirically.

So, the algorithm has then changed to what is shown in Algorithm 2.



---

**Algorithm 2** “Fast” Solution for EkoSort editor

---

- 1:  $n$  = desired number of movers
- 2:  $h$  = mover’s suction cup height
- 3:  $d$  = carousel’s diameter
- 4:  $r = d/2$

**Rotary Circle**

- 5: Import (RotaryCircle)
- 6: Translate along  $Z$  axis
- 7: Scale on  $X$  and  $Y$  dimensions

**Movers**

- 8:  $limit = n$
- 9: Import (mover)
- 10: Translate in position
- 11: **for** ( $i = 1, i < limit$ ) **do**
- 12:     Duplicate mover
- 13:     Set duplicate’s rotation pivot to world center
- 14:     Rotate the duplicate around  $Z$  axis of  $i * (360^\circ)/n$
- 15: **end for**
- 16: Group all movers together

**Stations**

- 17: **if**  $n \leq 6$  **then**
  - 18:      $start = 1$
  - 19:      $limit = n$
  - 20: **else if** ( $n \geq 13$ ) and ( $d \leq 3000mm$ ) **then**
  - 21:      $start = 3$
  - 22:      $limit = n - 2$
  - 23: **else**
  - 24:      $start = 2$
  - 25:      $limit = n - 1$
  - 26: **end if**
  - 27: Import (station)
  - 28: Scale dimensions
  - 29: Translate in position
  - 30: **for** ( $i = 1, i < limit$ ) **do**
  - 31:     Duplicate mover
  - 32:     Set duplicate’s rotation pivot to world center
  - 33:     Rotate the duplicate around  $Z$  axis of  $(i \cdot 360^\circ)/n$
  - 34: **end for**
  - 35: Delete first station loaded
  - 36: Group all movers together
-

---

### **Rotary Support**

- 37: Import (RotarySupport)
- 38: Scale dimensions

### **Conveyor**

- 39: Import (ConveyorComplete)
- 40: Translate along Y axis
- 41: Scale dimensions

### **Machinery**

- 42: Import (Machinery)
- 43: Scale dimensions

### **Windows**

- 44: Import (Windows)
- 45: Scale dimensions

### **EkoSort**

- 46: Group everything
- 

It can easily be seen how the whole procedure turned out to be shorter and more readable (thanks also to a phase of revision and improvement of the algorithm).

#### **3.1.2.5 Compenetration Recovery**

On the algorithm side, the empirical choice on the number of stations to be inserted in the machine configuration was unsatisfactory, because in many cases the cases provided for by the code were not sufficient to avoid the compenetration of the stations with the conveyor belt.

Therefore a method has been developed to find a remedy to this situation, without making substantial changes to the procedure already described, but instead going to solve a posteriori any problems of interpenetration (or excessive proximity) between the polygons of the stations and those of the conveyor belt.

According to the previous procedure, the algorithm could skip autonomously a maximum of three station placements. After the introduction of the new feature, once the machine was built, the algorithm performed a check on the distance between the last station placed and the closest part of the conveyor belt. If the distance was below a predetermined threshold (compliance was considered reached at a minimum distance of 200 mm), or worse, negative (meaning geometries compenetration is occurring), the code displayed a warning message on the terminal (reporting how many stations were placed and

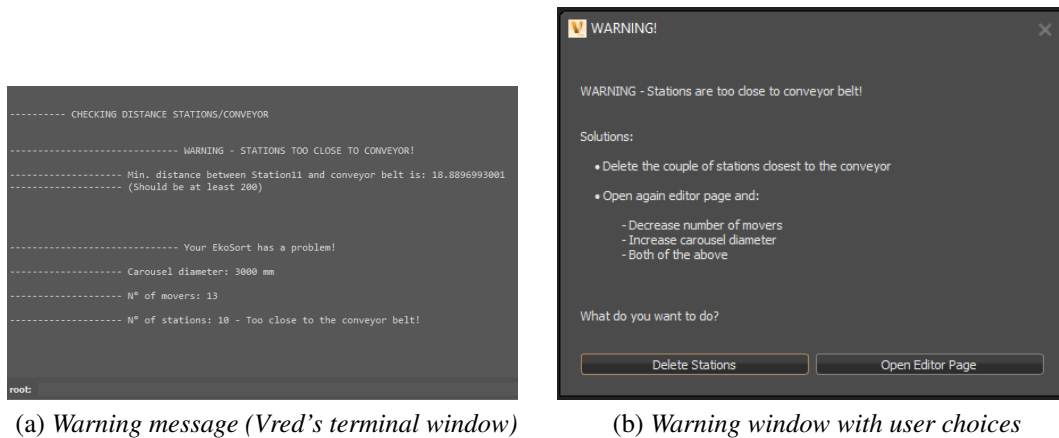


Figure 3.5: Compenetration warnings

what was the distance between the geometries concerned - figure 3.5a) and opened a window proposing two possible solutions to the user: delete an extra couple of stations, or open again the editor page, recommending to decrease the number of movers, to increase the diameter of the carousel, or both (figure 3.5b). When opened again, the editor page showed the last choice of parameters, so that it was easier for the user to apply a different choice, following the advice provided on-screen.

The implementation of dialog pages and the connection of on-screen buttons with functions in the script will be discussed in more detail later on (section 3.5).

There have been some problems regarding the measurement between station and conveyor: while it is present in Vred an instrument to take measures (both in 3D and in single directions), those values cannot be read in any way via script, since the Python documentation lacks functions in this sense.

So, a new solution has been implemented. For each node in the scene, Vred defines 6 values representing the boundaries on the object. These values define a so-called *Bounding Box*, that is a kind of boundary representing the space occupied by the geometry in question.

In Python, through the command `getBoundingBox()`, the values describing the bounding box of a specified geometry can be retrieved, stored in a vector according to the convention:  $[x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]$ .

So, since the conveyor belt was in the first quadrant, it could have been sufficient, theoretically, to use the difference between  $y_{min}$  of the conveyor and  $y_{max}$  of the station.

However, bounding boxes have a lower limit regarding their dimensions (in order to be visible in the scene). So, for thin objects like the geometries of the conveyor belts, the error between the effective dimension and the lowest value possible for the bounding box is really not negligible: along  $Y$  direction, the bounding box is 600% of the actual node dimension.

That's the reason why, in order to take a satisfactory measurement, the formula to be applied was:

$$y_{diff} = y_{conveyor}^{min} + \left( \frac{(y_{conveyor}^{max} - y_{conveyor}^{min}) - \frac{y_{conveyor}^{max} - y_{conveyor}^{min}}{6}}{2} \right) - y_{station}^{max} \quad (3.1)$$

This formula provided a good approximation of the measurement required, with an error of less than 1.2%. It has been used in the final code under the function *checkDistanceStationsConveyor()*.

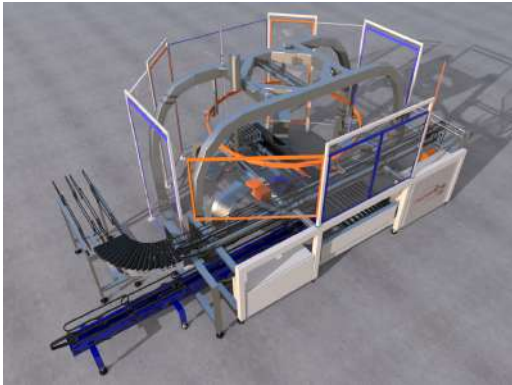
### 3.1.2.6 Final Solution

Finally, a further improvement applicable to the algorithm has been identified. In fact, although the number of loadings of the different parts had been minimized in the transition from version I to version II of the algorithm, these were basically time wasted in the realization of the scene.

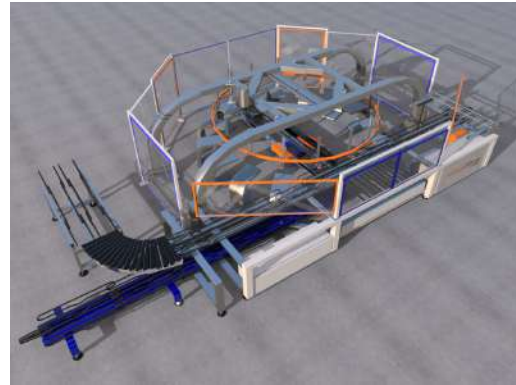
Just consider this case: what would have happened if the editor had been launched several times in a row to create different lines? At each run, the parts would have been deleted and then loaded again, and it would have been a waste of time and computation. It was therefore thought to load all the parts only at the start of the scene, and, at the algorithm execution, to simply change position and eventually scaling.

The result was more time needed at startup (in fact extremely tolerable), but much less time needed to execute each instance of the editor.

A couple of examples of the results that could be obtained with the final instance of the editor can be seen in figure 3.6, while the final code for the EkoSort's building is presented in Appendix A.



(a) Smallest buildable EkoSort: 5 movers, 4 stations, 2 m carousel



(b) Biggest buildable EkoSort: 16 movers, 11 stations, 3 m carousel

Figure 3.6: Examples of possible EkoSort building

## 3.2 ExtraPack Building Algorithm

### 3.2.1 ExtraPack - Palletizer

ExtraPack is a “portal” palletizer designed and constructed in accordance with the latest mechanical and electrical codes and standards [13]. Using an independently controlled 4-axis controller, it offers many advantages:

- positioning accuracy and repeatability;
- high operating speed;
- suitable to handle the most demanding work loads (its load capacity goes up to 250 kg, depending on the configurations).

With Extrapack it is possible to arrange the pallets independently: the layout and product code can be set individually for each single pallet.

There are two possible configurations for this machine: 2 or 3 pallets for each arch of the palletizer frame (in general each arch is dedicated to a tile variety).

Extrapack can load up to 20 pallets for the couple-pallet configuration, 21 for the triplet one. It can be equipped with a device to automatically pick up empty pallets and suction cups to pick up slipsheets. It can be programmed to handle different products whose sizes vary arriving from different lines.

### 3.2.2 ExtraPack Editor

#### 3.2.2.1 Library Realization

In a similar way to what was done for the EkoSort, the first step was to import the initial model (STEP file: .stp). This type of file is characterized by great precision (it gets to details like washers, screws and bolts), and the model already had a pretty good division of parts.

So in the first instance the model was visually improved, applying appropriate textures and materials, to achieve realism in VR. To make the editor, we then proceeded to divide it into parts, creating a library of components to be recalled within the algorithm.

The greatest difficulty has been to divide the support frame of the robot from the initial model: in fact, in order to make the editor parametric, it has been necessary to be able to chain the positioning of modules in order to create a frame with variable dimensions and number of arches. Vred does not offer specific tools, so it has been necessary to divide the geometries into the primitive polygons, select them manually and finally reposition them in the various groups. After this operation, the frame was divided into 3 parts: the initial and final arches, and a central arch (which has only two vertical pillars) to be cloned several times as the desired number of pallets increases.

Then, in order to concatenate the parts in an appropriate way, making the frame a unique and aligned part, some spheres have been inserted in these parts, called "links", whose position is read at each positioning of the part, so that the next one can be positioned in

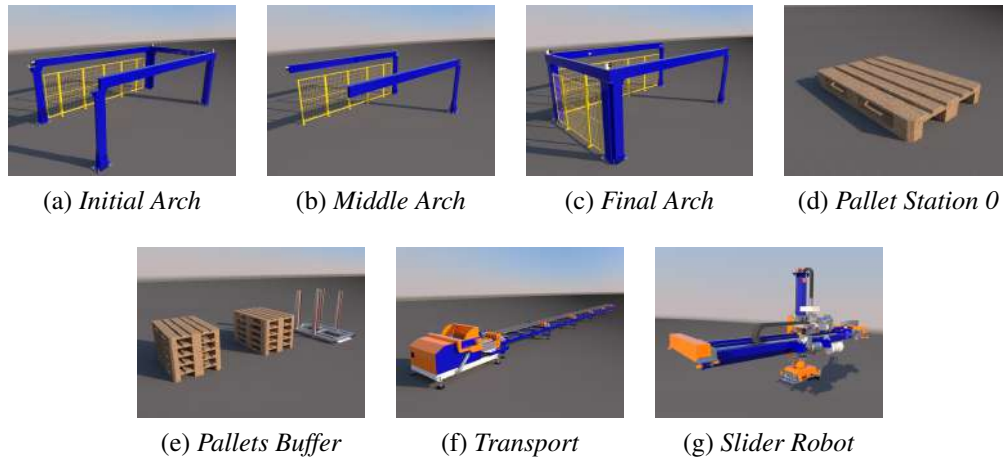


Figure 3.7: Extrapack Parts Library

its correspondence, thus aligning the geometries and giving the illusion of having a continuity.

The resulting parts at the end of this operation (shown in figure 3.7) are:

- (a) Initial Arch;
- (b) Middle Arch;
- (c) Final Arch;
- (d) Pallet Station 0;
- (e) Pallets Buffer;
- (f) Transport;
- (g) Slider Robot.

### 3.2.2.2 Editor Algorithm

As for the EkoSort, also in this case the editor waits for a series of user input parameters:

- Pallet stations for each arch: 2 or 3;
- Total number of stations: from 4 to 20, or from 6 to 21, respectively;
- Distance between pallets (within the same arch): from 500 mm to 2000 mm, or from 300 mm to 600 mm, respectively.

Now the functioning of the algorithm realized for the construction of this palletizing machine in the scene is described, analyzing function by function.

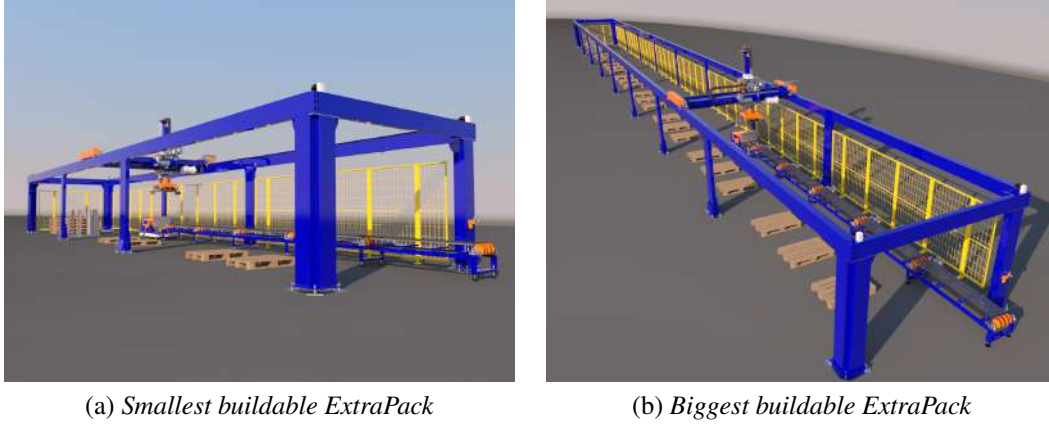


Figure 3.8: Examples of possible EkoSort building

### Reset ExtraPack

First of all, the previous environment must be cleaned up: this function takes care of erasing arches and clones of existing pallet stations. It also makes the central arch visible again (in case it was hidden). Finally, it saves the coordinates of the link sphere of the previous part of the line (i.e. the package handling one).

### Load Transport

The *Transport* part is selected and placed in position (thanks to the link sphere just considered).

### Build Slider Support

This function takes care of building the support structure for the *Slider Robot* part. First, it calculates the number of arches required, based on the desired configuration and the number of pallet stations required.

Then, it sets the coordinates of the first part of the support (*Initial Arch*) so that it is aligned with the rest, again using the link sphere already mentioned. Then it calculates the number of central arches required as  $n_{arcs} - 2$ .

If the configuration needs more than one central arch between the first and the last, it saves the final coordinates of the first arch (i.e. the point where its geometries end), and places *Middle Arch* there, so that it gives the illusion of continuity in the part. After that, it proceeds to clone (if necessary) the first central arch and to position all the copies, always leaving the impression that they are a unique and continuous group. Finally, it also positions the final part of the support (*Final Arch*), thus completing it.

If no central arches are needed, only the first and the last one are positioned, while the central one is hidden.

### Load Slider Robot

The *Slider Robot* is selected and positioned, aligning it with the other components, so that it appears to be installed on the support.

### **Load Pallet Stations**

Depending on the required configuration, 2 or 3 pallet stations can be inserted for each arch. The two methods are similar, but for convenience they have been saved in two different functions.

The first step is the declaration of a list containing all the clones of *Pallet Station 0*. After that, the function locates the original station in the first position of the first arch and proceeds to clone and position all the copies (thanks to the spheres links marking the middle of the first and last arch), at the distance specified by the user, adding them every time to the list of clones.

Finally, it groups all the copies included in the list under the "ClonesPallet" group, making it easier to delete them when the environment is reset.

The code implementing the developed algorithm is shown in Appendix B.



### 3.3 Other Machines in the Line

#### 3.3.1 Flawmaster - Tile Inspector System

The Flawmaster machines are designed to inspect ceramic tiles for mechanical, glaze and decoration defects and classify them according to their quality and shade [14].

These machines permit constant and repeatable inspection independent from the work shift. Tile inspection takes place at various angles detecting different defects such as:

- Surface defects;
- Mechanical defects (corner, edge);
- Reflection and decoration defects;
- Contamination along with shade and gloss.

Connected to any sorting machine, the Flawmaster replaces human inspection. It works with a wide set of algorithms to identify defects in a broad range of floor and wall tiles, and it is able to inspect tiles, with dark or complex decorations, at speeds of up to  $1000m^2/hour$  (normal output calculated for sizes ranging from 40x40 to 120x120 cm). Network connection is possible, in order to manage data collection and production reports.

Automatic final tile inspection allows productivity and performance to be increased while obtaining numerous benefits:

- High inspection rate;
- Optimized tile flow;
- Fewer shutdowns;
- Quality benefits;
- Uniform inspection;
- Reduction in claims;
- Continuous monitoring of production defects: this allows targeted improvements to be made to the production process in order to increase quality.

As far as the software is concerned, it is possible to obtain real-time monitoring of production, alarm signalling and production reports. In addition, it is possible to detect a wide range of defects such as:

- Decoration;
- Irregular edges;
- Cracks.

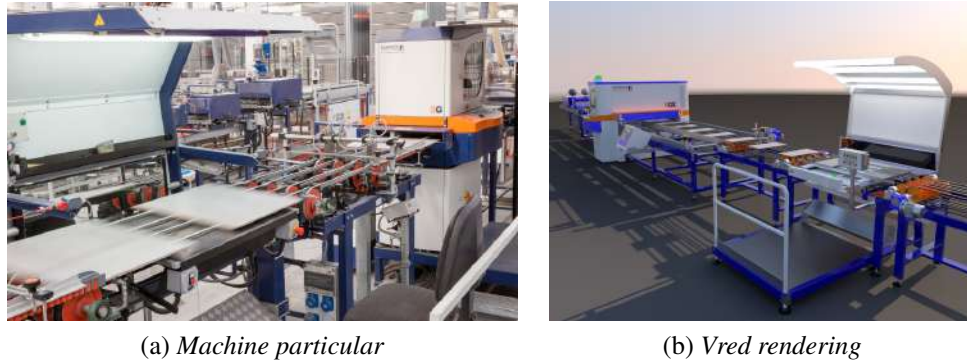


Figure 3.9: Flawmaster Machine (courtesy of SACMI Imola S.C.)

### 3.3.2 Advancheck - Tile Inspector System

Advancheck technology is used to precisely check squareness of the tiles and not only size and flatness, for example for rectified and squared products [14].

It is a fully automatic machine that detects ceramic tile size and flatness defects. The user interface consists of an industrial PC with Windows operating system. The industrial PC has a “frame grabber” card that processes the signals received from the cameras sampled every tenth of a millimetre. Detection of size/shape defects along with the possibility to connect the device to on-line control systems permits “real time” analysis.

As a result, it is possible to go back through the production process and take any corrective actions required to remedy faults found, thereby remarkably improving overall quality and saving energy.

The control system is able to:

- Calculate the size class of the tiles and determine if they are to be downgraded due to tolerance, parallelism, curvature and squareness (measurement of the two diagonals) defects;
- Set six size thresholds and three thresholds for each type of defect;
- Acquire the areas of the four corners and central measures of the sides;
- The PC can be connected to the network to manage data collection, sorting statistics and set the configuration parameters.

### 3.3.3 EkoWrap - Packager

EkoWrap is a system designed to simplify the packaging of both small-to-medium and medium-to-large sizes. This packaging technique uses two corrugated cardboard blanks and a patented closure system to package even large product sizes [12].

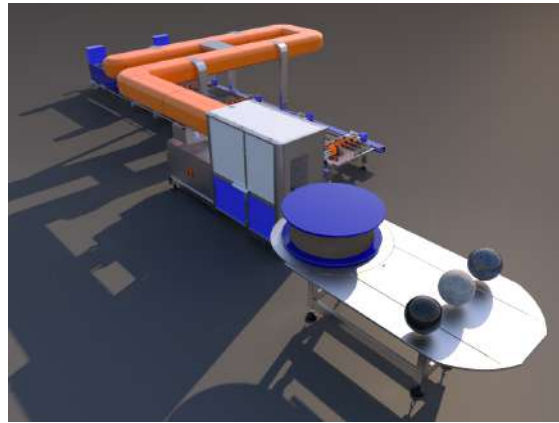
This machine allows the entire perimeter of a tile stack to be protected, corners included: the cardboard blanks overlap in the middle of two opposite sides of the box, ensuring that corners are fully wrapped, where the greatest protection is needed.

EkoWrap is cost-effective and environmentally friendly with its reduced cardboard requirements, lower quantity of packaging waste and raw material/energy savings during cardboard production ensure minimised environmental impact.

It can be used in a number of different ways. The system can be fed with either pre-cut



(a) Machine particular



(b) Vred rendering

Figure 3.10: EkoWrap Machine (courtesy of SACMI Imola S.C.)

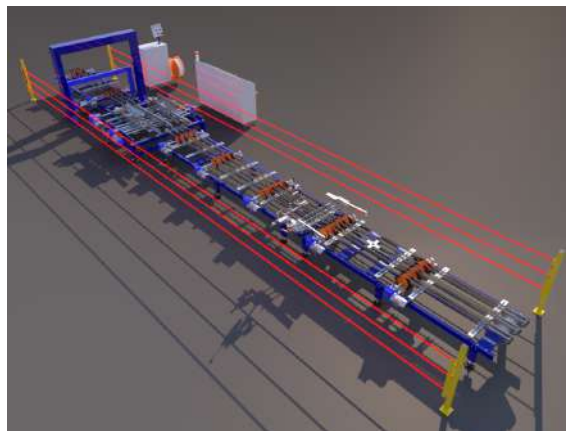


Figure 3.11: Transport and package handling

corrugated cardboard blanks, pre-printed with specific trademarks and logos, or plain corrugated cardboard blanks. In both cases, a shell is obtained, on which all the information necessary for product identification is then printed in real time.

EkoWrap allows products to be processed on both feed sides, from a minimum of 200 mm up to a maximum of 1200 mm. Systems can also, on request, be personalised to give a minimum of 150 mm on one side and a maximum of 1500 or 1800 mm on the other.

### 3.3.4 Transport and Package Handling

Between the packaging part (in this case carried out by EkoWrap) and the preparation part of the pallet (by means of the ExtraPack), a series of machines are placed which are responsible for wrapping the packages with plastic film, centring the package and positioning it precisely on the conveyor, so that it can then be effectively grasped by the robot in the last part of the line.

### 3.4 Animation

After having completely built the various parts of the sorting line through the algorithms presented in the previous sections, and having aligned them in a continuous line (for more details see section 3.6), it has been decided to further develop the scene in virtual reality, in order to really make it "alive", and thus take full advantage of the immersive features offered by this powerful tool.

After all, a static scene may be realistic and pleasing to the eye, but it fails to express the potential that can be expected from an industrial sorting line. Putting oneself in the shoes of a potential customer observing the scene, one can easily imagine how this customer would want to see the line effectively in motion.

It was therefore decided to animate a batch of 10 tiles measuring 600 mm x 300 mm x 10 mm. These tiles will flow conveyed by the belts at a speed specified by the user (with values between 0.7 m/s and 2.0 m/s, in conformity with what actually happens in Nuova Sima lines of this type), passing through the machines of the first part of quality control, and then being lifted by the EkoSort carousel and placed on the first station, where the stack of tiles will be grabbed and placed on a dedicated conveyor, which will take it through the EkoWrap, where it will take the form of a real package (including the packaging). The newly created package will then continue its journey until it reaches the Extrapack, where it will be grabbed by the Slider Robot, rotated and placed on the first available pallet.

Vred provides a range of tools to help creating animations. First of all, the animations are organized according to a timeline that uses frames as units of measurement according to the PAL system, so that:

$$1 \text{ sec} = 24 \text{ frames} \quad (3.2)$$

$$1 \text{ frame} = 0.041\bar{6} \text{ sec} \quad (3.3)$$

The animations have then been designed according to the operation of the line, and afterwards all the conventional units of measurement have been converted into the equivalent in frames so that they could be defined inside Vred.

But how does an animation fit into this software?

Vred, as a tool mainly focused on the geometries of the objects present in the scene, associates two frames to each element in the scene: one for the rototranslation axes, and one for the scaling axes. Then it is possible to assign to each object the values of rotation, translation and scaling on each desired axis. Combining to these values also the possibility to associate to the object a boolean variable to make it visible or not, it is immediately possible to see how to create even complex animations with a bit of study (see figure 3.12).

It is therefore necessary to create curves over time for these values, then they can be saved by Vred in a format called *animation block*, which will enclose the entire animation of the geometry in question. These blocks can then all be started at the same time (or saved in animation clips) and eventually be looped in order to have a movement that is

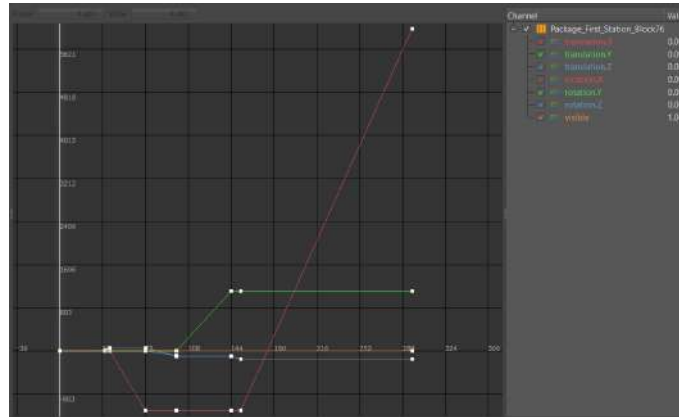


Figure 3.12: Curves for Translation, Rotation and Visibility values of the stack of tiles on the first station of the EkoSort

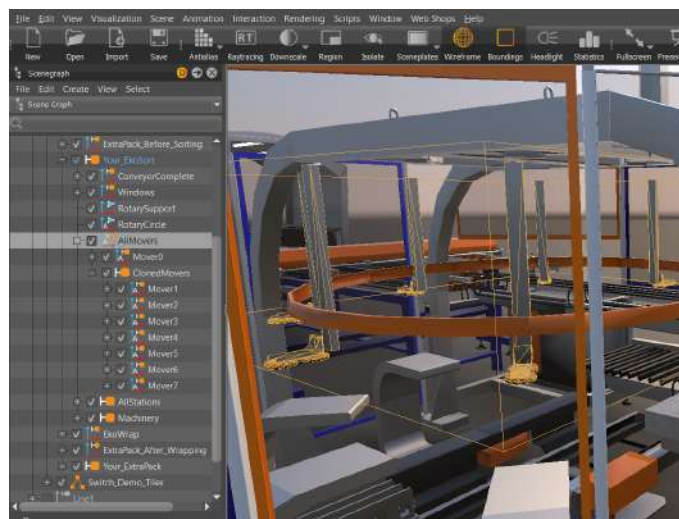


Figure 3.13: Example of the tree organization of geometries in Vred

perceived as continuous.

These curves (and along with them the animation blocks that contain them) are a very flexible and powerful tool, combined with the tree organization of the geometries in Vred, because they allow a "cascading" animation. This term refers to the fact that the animation of a *groupnode* (geometric entity representing a group of geometries located at a lower level of the tree) is spread to entities that are within that group.

Considering figure 3.13 as an example, examining the EkoSort carousel, it can be seen that the movers (i.e. the tile grabbers) are placed inside the group node "AllMovers". The rotary movement of the carousel during the animation is assigned to this last group (and will then be replicated by the RotaryCircle), and it therefore propagates to all the movers. To each of them will be imposed only the translatory motion along the z axis, without the need for rotation.

Now, the real problem in animating a scene like the one presented in this project is the fact that the editor built up to this point is parametric, i.e. it responds to user specifications, and can therefore create a large number of possible configurations for the same line. Obviously the animation must be adapted to the configuration represented. The problem

is that the animations created through curves (and then looped) are extremely "static", meaning that it is very difficult to modify them.

In addition, the differences in animations between different configurations are multiple and not trivial. If, as an example, one considers the configurations at the opposite extremes:

1. Line with EkoSort with minimum dimensions (5 movers, carousel diameter 2000 mm) and minimum conveyor speed (0.7 m/s);
2. Line with EkoSort with maximum dimensions (16 movers, carousel diameter 4000 mm) and maximum conveyor speed (2 m/s).

The changes that should be imposed in the transition from animation 1 to animation 2 would be:

- **Translation of the entire animation along the y-axis:** since in the constructing of the line, the center of the world corresponds to the center of the circular EkoSort carousel (as shown in section 3.1.2.4), as the diameter of the latter varies, the center of the conveyor belt will move along the y-axis (since the line develops in length along the x-axis). Note, however, that the exact centre of the carousel does not move, therefore the tiles will have to modify the arc trajectory they must follow accordingly, so as to perfectly match the trajectory of the movers;
- **Moving of the starting and ending points of the animation along the x-axis:** in the construction of the EkoSort (section 3.1.2.4), in order to maintain a correct balance in the dimensions of the machine as the diameter of the carousel varies, a scaling of all parts with respect to the x and y axes is carried out. The scaling with respect to the x-axis will result in an elongation of the conveyor portion of the machine model, and since the origin of the reference system is in the exact centre of the carousel, this translates into the distancing (passing from case 1 to case 2) of the starting and ending animation points;
- **Variation of animation times:** obviously changing the speed of the conveyor belts leads to a variation of the instants in which the tiles (and all the elements that interact with them) reach the various checkpoints of the animation;
- **Variation of the arrangement of the tiles on the conveyor belt:** since, as already mentioned, Vred's animations are extremely "static", and therefore not linked to the dynamics of the machine as could happen in a simulative software, it is not trivial to adapt it to the variation of key elements such as the number of movers and the diameter of the carousel. In fact, the greatest difficulty is to have a standard animation to be respected, i.e. all the tiles grabbed in sequence and placed on the same station. Logically, to do this, the tiles must be at a well-defined distance from each other (step), calculated according to the equations:

$$\beta = \frac{360}{n_{movers}} \quad (3.4)$$

$$\beta_{rad} = \beta \cdot \frac{\pi}{180} \quad (3.5)$$

$$step_{tiles} = arc = \beta_{rad} \cdot \frac{diameter}{2} \quad (3.6)$$

In addition, in case the speed is too high compared to the normal step, and therefore the rotation of the carousel is not able to guarantee the rendez-vous between mover and consecutive tiles, this step between the tiles should be doubled. In the latter case, however, the tiles may be too far apart, and therefore some of them may be even beyond the grasp point in the initial arrangement of the animation. In these cases it is therefore necessary to eliminate them, effectively animating a batch of 9 or 8 tiles;

- **Variation of the type of curves of movers and tiles:** since the EkoSort moves the tiles at fixed angles (i.e. once a tile is lifted, the carousel performs a rotation equal to the angle calculated in equation 3.4 and, if possible, drops the tile on the first available station, otherwise it keeps it lifted until the next rotations allow its release), the animation will have to be adapted to this behavior, depending on the size of the machine and the number of stations that are present.

It can therefore be noticed that it would be extremely difficult to modify an existing animation in such a way as to adapt it to the configuration created by the editor. A "tabula rasa" approach has therefore been adopted, so each time the editor is launched, existing animations are deleted, and the curves of each animated object are computed from scratch again.

Moreover, the final goal of this animation is obviously to give the impression of being in the presence of a real and working line, therefore it needs to be looped, just to give the illusion of continuity.

But how to achieve this?

In fact, without any expedient, at the end of the animation time one would see the tiles and packages disappear and then reappear in the initial positions, creating an extremely unpleasant effect for the user.

The solution has been to add duplicate elements (called "bis") to overcome this inconvenience. The bis elements are used to fill the gap left by the main animation, basically representing the elements coming from the next batch (in the case of tiles) or the previous one (in the case of the pack) that in the final moment of the animation will be at the exact point where the main elements will be at the beginning of the animation. In this way, in the loop of the animation the impression is that the overall movement is perfectly continuous.

As a first approach, in order to get a first familiarity with the problem of creating curves, and above all to understand how to synchronize all the animated elements in an exact way, a fixed animation (i.e. without being calculated according to configuration-dependent variables) was created in order to understand what should be the basis from which to start. As a basic case, the line with minimum parameters was chosen, with conveyor belt speed of 1 m/s.

After the creation of this reference animation, the next step has been to understand how to introduce the parametrism in its creation within the editor code, and how to make

the implementation of the above listed variations automatic from one code launch to another.

In the algorithm for the creation of the animation, the units of measurement should be millimeters for length and frames for time, so that they can be congruent with the methodology used in the software. It is therefore advisable to convert the speed chosen by the user during the first phase of parameter selection from m/s to mm/frame. This is done according to the equation:

$$v_{[mm/frame]} = \frac{1000}{24} v_{[m/s]} \quad (3.7)$$

During the construction of the animation of the tiles, it can be noticed that, once established the movement that the carousel must have (based on the number of movers selected) and the step that the tiles themselves must keep between them, the movements performed by the entire batch will be exactly the same, only separated in time by a certain number of frames, calculated through the relationship:

$$frame_{gap} = \frac{step_{tiles}}{v_{[mm/frame]}} \quad (3.8)$$

Then, the key reasoning has been the choice to refer all the animation to a series of key instants, essential for the synchronism of the various moving parts, while the rest of the time values set in the curves would be nothing but fixed offsets from these reference instants.

These fundamental frames are computed as follows:

- Grasping of the first tile (note that  $x_{spawn} < 0$ ):

$$frame_{grasp\ tile\ 0} = -\frac{x_{spawn}}{v_{[mm/frame]}} \quad (3.9)$$

- Arrival of nearest tile of the batch in grasping position:

$$frame_{nearest\ tile\ arrived} = frame_{grasp\ tile\ 0} - (frame_{gap} \cdot (n_{tiles} - 1)) \quad (3.10)$$

- Time horizon of the animation (time-limit).

$$time\ horizon = frame_{all\ tiles\ put\ down} + (24 \cdot 3) \quad (3.11)$$

After all these considerations and after several attempts, the final code for the creation of the animation was completed (shown in Appendix C).



## 3.5 G.U.I. Implementation

So far the methodology used for the construction of the complete line and its animation has been presented, based on what are the specifications declared by the user before the launch of the editor. But how are these input parameters actually collected?

In fact, it is necessary to have a functional G.U.I. (Graphical User Interface), so that the user can enter the desired values for the configuration in an intuitive way. It is also equally important to show in a clear way the limits within which to choose these values: in the Nuova Sima catalogue, in fact, there are only a certain number of configurations for the machines present in the line (EkoSort and ExtraPack), and it is therefore appropriate to limit the choice of parameters only within these allowed configurations.

For the creation of the GUI at this stage, the program Qt Designer has been used. This program, developed by Qt Company, provides a variety of intuitive tools for creating dialog boxes starting from a blank page.

### 3.5.1 Qt And PySide2

Qt Designer is based on the PySide2 library, which is nothing more than a port in Python language of the Qt platform.

As reported in [5], Qt is a cross-platform application development framework for desktop, embedded and mobile. Supported Platforms include Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, Sailfish OS and others.

Qt is not a programming language on its own. It is a framework written in C++. A preprocessor, the MOC (Meta-Object Compiler), is used to extend the C++ language with features like signals and slots. Before the compilation step, the MOC parses the source files written in Qt-extended C++ and generates standard compliant C++ sources from them. Thus the framework itself and applications/libraries using it can be compiled by any standard compliant C++ compiler.

In order to use the tools made available by Qt within a programming using the Python language, the "Qt for Python" project was developed by the Qt Company, offering the official Python bindings for Qt (under the library PySide2), so that Qt5 APIs can be used in Python applications, and a binding generator tool (Shiboken2) which can be used to expose C++ projects into Python.

PySide2 is organized in modules [7], which permit to realize the most diversified functions, such as the realization of charts and diagrams, up to audio, video and hardware interactions.

The main modules contained in this library that can be used to build a Widget-based UI (and so are the ones used in this application) are:

- **Qt Core:** Provides core non-GUI functionality, like signal and slots, properties, base classes of item models, serialization, and more;
- **Qt GUI:** Extends QtCore with GUI functionality: Events, windows and screens, OpenGL and raster-based 2D painting, as well as images;



more complex configurations.

Once the key role that layouts have in creating widgets has been understood, the next step has been to populate the main dialog page.

The ultimate goal of this widget was, as already mentioned, to allow the user to go and choose the desired parameters for the line. Before thinking about the layout, therefore, it was appropriate to think about what these parameters were, in consideration of all the procedures done so far. The final list is then:

- **Conveyor:** speed [m/s];
- **EkoSort:**
  - number of movers;
  - carousel diameter [mm];
- **Extrapack:**
  - configuration (number of pallet stations per arch);
  - total number of pallet stations;
  - distance between pallet stations [mm].

Once the parameters were known, it was then considered how to make it intuitive for the user to choose the parameters only within the boundaries specified in the catalog, and how to present these limitations in a clear and readable way. To do this, the best and most visually pleasing solution was to use *QSliders* and associated *QSpinBoxes* (the classes coming from Qt library will be presented in section 3.5.1).

The choice of *QSliders* is due to the fact that first of all they allow to show immediately what are the boundaries for the value associated with them (to simplify this, text *QLabels* have been inserted below the extremes of each slider in order to make the limit values explicit), and secondly it is impossible to choose values outside the permitted ones, as the slider cannot move beyond the limits. It is also very intuitive to choose the value by scrolling the slider with the mouse, and the insertion of graphical tick marks above and below the slider helps to get an idea of the possible intermediate values.

However, it is of fundamental importance to be able to read exactly the value chosen by scrolling the sliders, as well as the possibility of entering the value directly via keyboard, if desired. therefore, in a complementary way to the sliders, spinboxes have been inserted, allowing both these operations to be carried out easily.

As previously written, one of the choice parameters is given by the configuration of the Extrapack, based on how many pallet stations one wishes to have for each arch of the machine (in general this choice is linked to the quantity of tile formats to be managed by the line). Then, the other two parameters related to the Extrapack will see their limit values change according to this choice (table 3.2):

Table 3.2: **Parameters Boundaries:** Configuration dependency

Configuration	Parameter	Min. Value	Max. Value
<b>2 Pallets per Arch</b>	Total N° of Pallets	4	20
	Pallets Distance [mm]	500	2000
<b>3 Pallets per Arch</b>	Total N° of Pallets	6	21
	Pallets Distance [mm]	300	600

In order to manage this variation in the boundaries, the solution adopted was to use four sliders and four spinboxes (with the limit values specified in table 3.2), and to alternate between the two possible configurations using two *QRadioButton*.

Radio buttons are nothing but selectable buttons, which by default are mutually exclusive (i.e. selecting one, the other - or the others - will be automatically deselected, so that only one button at a time is checked). The user will then have to specify only the two parameters related to the chosen configuration, while the other two will be irrelevant. The code will then take care of the input, retaining only the values related to the selected configuration.

After that, three *QPushButtons* (i.e. simple pressable buttons) were added at the bottom of the page. They had then been associated with the launch of certain functions within the code:

- **Create:** confirms parameters choice and launches scene editor;
- **Go Back:** returns to an eventual previous page (e.g. a welcome message);
- **Close:** closes current window, editor must then be restarted again.

### 3.5.2.2 Signal Management

After completing the layout of the widget, it has been necessary to go and connect the input and output signals of the related elements in order to cooperate correctly when viewing the page.

Qt Designer provides a section of its program dedicated exclusively to the editing of signals and slots (figure 3.15). By simply dragging and dropping from one element to another, it is possible to create connections between the signals that control them (graphically represented by arrows).

First of all, the initial connections established have been those between the values of the sliders and the values shown in the respective spinboxes. Since they are separate elements, this connection had to be imposed externally. An example is given in figure 3.16a, where it can be seen that the *valueChanged(int)* output signal (which returns the assigned value as an integer), relative to the *horizontalSlider\_n* slider (i.e. the slider representing the number of movers), is associated to the input signal *setValue(int)*, which imposes the

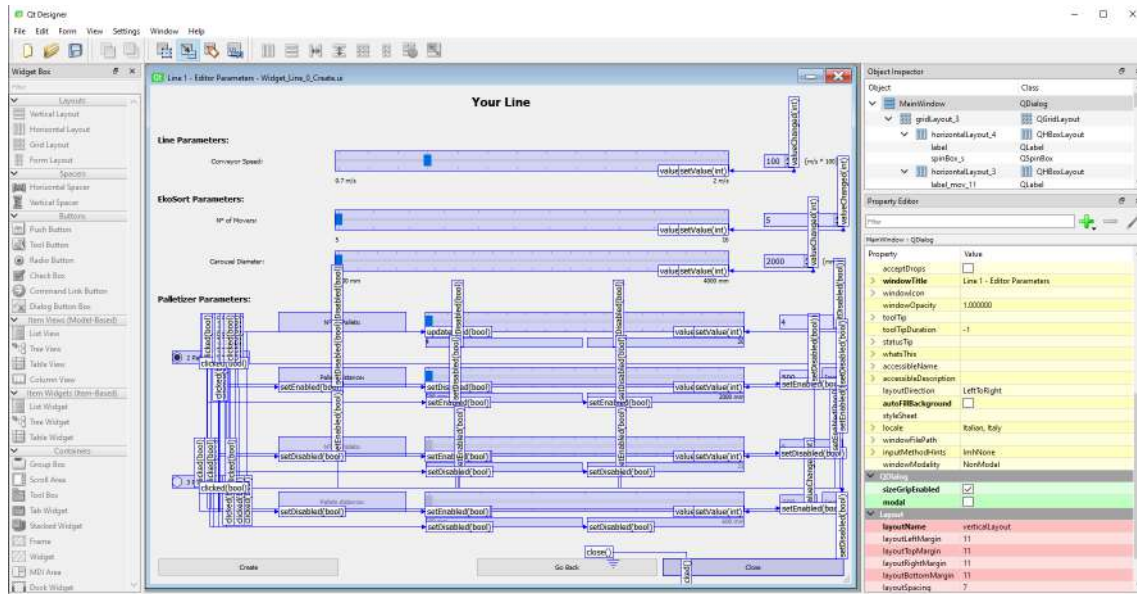


Figure 3.15: Qt Designer page: main widget (editor parameters) - Signals/Slots Editing

integer value specified to the relative spinbox (*spinBox\_s*).

After that, the connections inherent to the choice of the palletizer configuration have been set. Indeed, in order to make it extremely clear which are the limit values of the parameters that can be chosen in both cases, it is advisable that when the configuration is chosen by checking the corresponding radio button, the sliders, spinboxes and all the text labels related to it are enabled, while those related to the other configuration are disabled, making it impossible for the user to interact with them.

An example can be found in figure 3.16c, where the output value *clicked(bool)* (which returns a true boolean in case the button is clicked) related to *radioButton\_2arc* (the radio button associated to the configuration with 2 pallets per arch) is associated to the input signal *setEnabled(bool)*, which requires the activation of the object *horizontalSlider\_p\_2arc* (i.e. the slider representing the number of pallet stations in the respective configuration). The dual reasoning was then applied in figure 3.16d, where the *clicked(bool)* signal of *radioButton\_2arc* was associated with the *setDisabled(bool)* signal, which requires the deactivation of *horizontalSlider\_p\_2arc* (which represents the number of pallet stations in the other configuration).

Finally, the last interaction that can be implemented at this level is the closing of the page by clicking the *Close* push button. This is shown in figure 3.16b, where the output value *clicked()* (which returns true value each time the button is pressed) is associated with the *close()* variable of the page.

### 3.5.2.3 Code Implementation

Vred's Python library (divided into VRED Python API v1 + v2) provides a series of modules and classes for widget management. In this case, for the management of .ui files, it has been used the class *vrWidget*.

In particular, the commands used were:

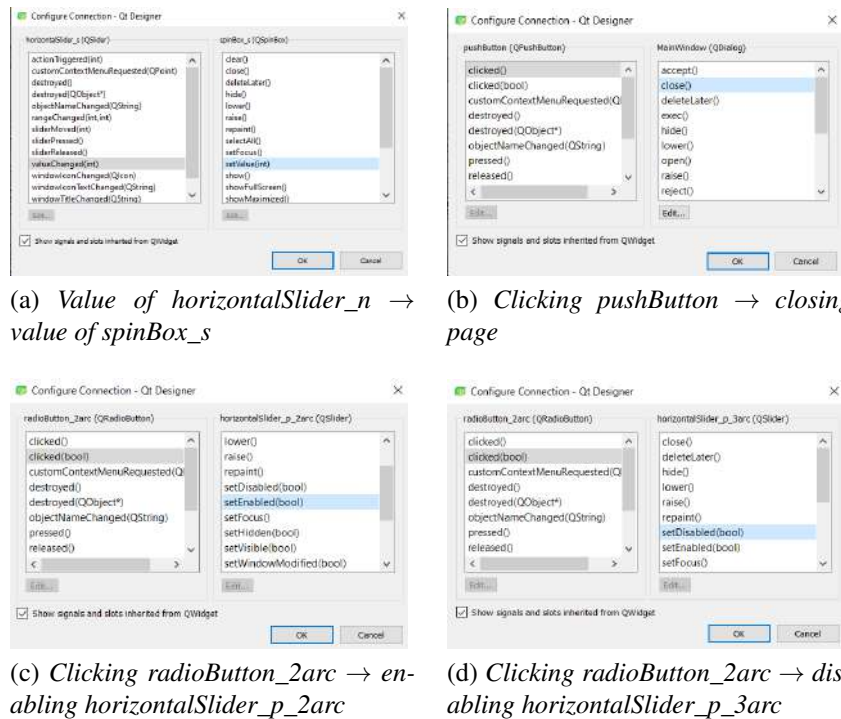


Figure 3.16: Examples of signal connections

- **`__init__(object, uifile, className, parent, name):`**

The constructor of the `vrVideoGrab` class (inherited by `vrWidget`). Among the valid sets of possible parameters, the application presented here uses `vrWidget(uifile)`, that creates a `vrWidget` object from a .ui xml file, created with Qt Designer.

Parameters:

- **object:** the QObject (*type = vrQObject*)
- **uifile:** the name of the .ui xml file (*type = string*)
- **className:** the name of the QT class (*type = string*)
- **parent:** the parent QObject (*type = vrQObject*)
- **name:** the name of the newly created widget (*type = string*)

- **`connect(sender, signal, function):`**

Connects widget signals with Python functions.

Parameters:

- **sender:** the name of the sender (*type = string*)
- **signal:** the name of the signal (*type = string*)
- **function:** the python function (*type = boost::python::object*)

- **`show(state):`**

Shows the widget.

Parameters:

- **state:** the state (*type = integer*): 1 = Show, 0 = Hide, -1 = Toggle

Therefore, in order to insert the widget in the editor it was necessary to modify its entire behavior. In fact, as already specified, inside the main widget a push button had been prepared to start the construction of the scene when pressed, while inside the widget that notifies the interpenetration of the stations (already presented in section 3.1.2.5, fig. 3.5b) it has been inserted a push button to delete the stations in excess. This means that the entire editor had to be enclosed within a function, and then connected to the pressure of the related button.

Therefore, the "main" - a term used with freedom of expression, since programs in Python do not have a real main, as happens for example with the C language - consists only in the declaration of the default values of the editor parameters (which must match those shown when the widget is first opened), the declaration of the widget, and finally the connection of the functions to its elements.

Note that all widgets must be loaded immediately (and not only at the moment when they will be used), and then the unnecessary ones must be hidden. They will then be shown (and the others hidden) when appropriate, using the *show(state)* command.

So, the first part of the code became:

Table 3.3: **Python code:** GUI implementation

---

```

1 # ----- Default (global) variables:
2 conveyor_velocity = 1 # conveyor speed [m/s]
3 n = 5 # movers
4 d = 2000 # carousel diameter [mm]
5 three_per_arc = False # 2 pallet stations per arc
6 p = 4 # pallet stations
7 palletDistance = 500 # distance between pallet stations [mm]
8
9 # ----- Create the GUI using the file .ui
10 widget = vrWidget("file_path/MainWidget.ui")
11 widget2 = vrWidget("file_path/Warning_EkoSort.ui")
12
13 # ----- Connect python functions to the GUI elements
14 widget.connect("pushButton_Create", "clicked()", mainEditor)
15 widget.connect("horizontalSlider_n", "valueChanged(int)", nMovers_Changed)
16 widget.connect("horizontalSlider_p_2arc", "valueChanged(int)", nPallets_Changed)
17 widget.connect("horizontalSlider_p_3arc", "valueChanged(int)", nPallets_Changed)
18 widget.connect("horizontalSlider_d", "valueChanged(int)", diameter_Changed)
19 widget.connect("horizontalSlider_s", "valueChanged(int)", speed_Changed)
20 widget.connect("horizontalSlider_d_2arc", "valueChanged(int)", distance_Changed)
21 widget.connect("horizontalSlider_d_3arc", "valueChanged(int)", distance_Changed)
22 widget.connect("radioButton_2arc", "clicked(bool)",
23               arc_pallets_changed_2perArc_pressed)
24 widget.connect("radioButton_3arc", "clicked(bool)",
25               arc_pallets_changed_3perArc_pressed)
26
27 widget2.connect("pushButton_delete", "clicked()", delete_ExcessStations)
28 widget2.connect("pushButton_reopenEditor", "clicked()", reopen_EditorWidget)
29 widget2.show(0) # hide warning widget

```

---

Finally, a series of functions have been created to impose the values specified within the sliders or spinboxes. First of all, since it has already been specified that in the .ui file the corresponding sliders and spinboxes should show the same value (as written in section 3.5.2.2), the values associated with the functions in the script (shown in table 3.3) are only



the ones of the sliders, in order to make the code lighter and better readable.

The reasoning followed was this: the global variables corresponding to the 6 parameters of the line have been initialized to a default (corresponding to the smallest configuration), and these values also correspond to those initially shown in the parameter selection widget when loading it. Every time the value of a slider is changed (event notified by the *valueChanged(int)* signal, which also returns the corresponding integer value), the related variable is updated. When a radio button is pressed (event signalled by *clicked(bool)*), the variables related to the number of pallet stations and their distance (referring to the selected configuration) are reinitialized to the default values.

The implementation of these functions in the code is shown inside table 3.4.



Table 3.4: **Python code:** Widget functions

---

```

1 def arc_pallets_changed.3perArc_pressed(value):
2     global p, palletDistance, three_per_arc
3     value_bool = bool(value) # value_bool == True when selected 3 pallets per arc
4     if not value_bool:
5         p = 4
6         palletDistance = 500
7     else: # should never enter here, just in case
8         p = 6
9         palletDistance = 300
10    three_per_arc = value_bool
11
12 def arc_pallets_changed.2perArc_pressed(value):
13     global p, palletDistance, three_per_arc
14     value_bool = bool(value) # value_bool == True when selected 2 pallets per arc
15     if value_bool:
16         p = 4
17         palletDistance = 500
18     else: # should never enter here, just in case
19         p = 6
20         palletDistance = 300
21     three_per_arc = not value_bool
22
23 def diameter_Changed(value):
24     global d
25     d = value
26
27 def distance_Changed(value):
28     global palletDistance
29     palletDistance = value
30
31 def nMovers_Changed(value):
32     global n
33     n = value
34
35 def nPallets_Changed(value):
36     global p
37     p = value
38
39 def reopen_EditorWidget():
40     widget2.show(0) # hide warning widget
41     widget.show(1) # show again editor widget (with old values saved)
42
43 def speed_Changed(value):
44     global conveyor_velocity
45     value_float = float(value)
46     conveyor_velocity = value_float / 100

```

---

A useful feature of .ui files is the fact that they inherit the graphical style of the program in which they are loaded (unless otherwise specified through the declaration of custom *styleSheets*), which makes their use extremely versatile and convenient. So, once the editor is launched, the final widgets will look like Vred's pages (figure 3.17).

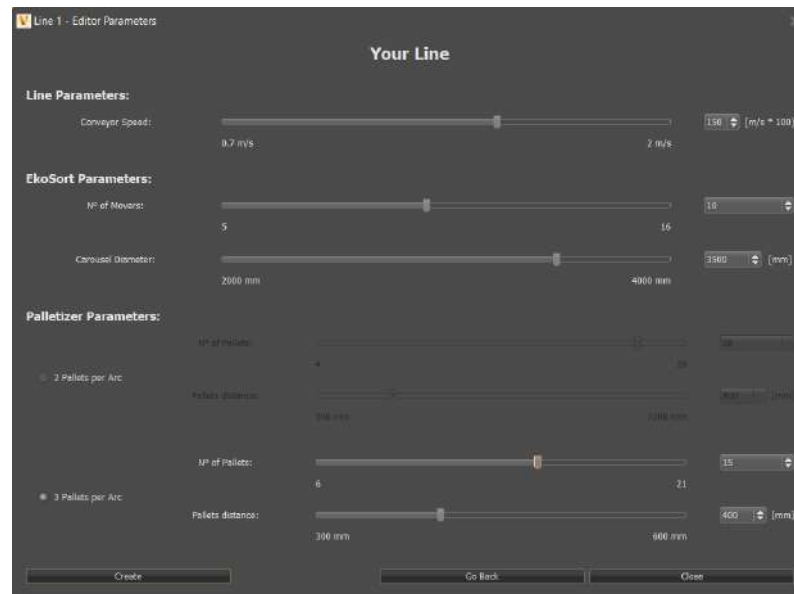


Figure 3.17: Final appearance of the widget

### 3.6 Final Line Editor

At this point, what was obtained was a code able to load an externally created dialog page, let the user choose the parameters of the line through it, to save them as variables, and then to build the scene based on them. In fact, the script first builds the EkoSort, then places in line the previous part (tile quality control) and the parts immediately following it (EkoWrap and package transport). Then it builds the ExtraPack palletizer at the end of the line.

The alignment of all the parts is done through the use of so-called "Link Spheres" that act as hooking nodes for the various modules of which the line is composed.

After the construction of the line itself, the script takes care of making it "alive", creating an animated demo that allows the user to have an idea of how the entire line works.

But how does the code launch actually happen? Vred provides the so-called *Variant Sets* (presented in detail in section 4.2.1), i.e. entities that can enclose all types of transformations and interactions of the scene, so that they can be called up easily and quickly. It is also possible to assign a hotkey (keyboard shortcut) to each variant set, which allows them to be launched at the simple pressure of a key (or combination of keys).

In particular, the variant sets allow to insert inside them scripts in Python, which will be executed at each launch. It is therefore within a special variant set, called *Line Editor*, that the final code of the editor has been inserted, since the idea behind it is precisely that of the reusability of the scene: the code has been designed to be launched several times in a row, always creating new configurations for the user who wants to touch the wide range of customization of the line configuration, without having to open a new Vred scene each time. Thanks to this powerful tool, a new configuration can be created at any moment, simply by pressing a button and entering the necessary parameters.

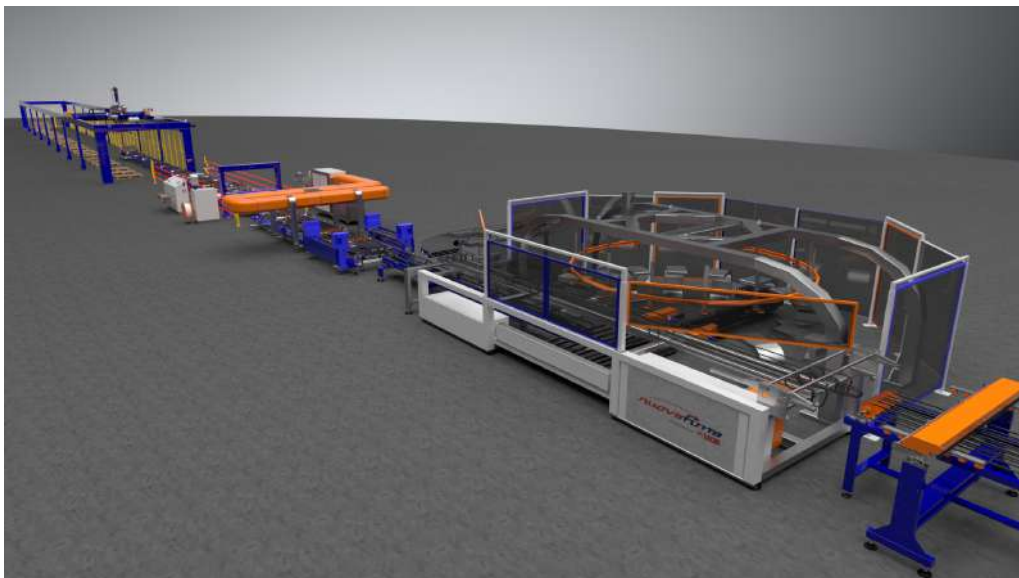


Figure 3.18: Final render of the line



# Chapter 4

## Three Lines Editor

### 4.1 Editor Evolution

After completing the development of the single line editor, it has been thought about how to expand the scene, simultaneously making it more suitable for the set goal (i.e. in the field of preselling). In addition to this, however, a second objective has been outlined: that of being able to fully explore the possibilities offered by this tool, especially in the field of interactions present in the scene, so as to have an idea of what could be possible future developments, even for purposes other than those initially set.

In practice, the new objective has become that of carrying out a real exercise in style.

#### 4.1.1 Algorithm Adjustment

The first problem faced in switching to a three-line editor was undoubtedly to adapt the code to manage the parameters of each line, and to iterate its execution for a number of times equal to the number of lines required by the user.

To do this, it has been decided to use global lists in Python to store parameters, effectively replacing the global variables of the single line case. For the choice of the configuration, a special virtual reality tool has been created (section 4.2.2.2), which provides an intuitive interface to operate the selection, and then takes care of saving the lists in the form of global variables (section 4.1.2.2).

Secondly, during its execution, the editor uses other global auxiliary lists to pass the salient data from one function to another within the code.

Analyzing the main section of the final code (presented in Appendix D), it can be seen how, in the first instance, it cleans up the scene, preparing it for reconstruction. First, it resets the tools that involve the use of widgets (section 4.2.2), and deletes the menu utilities (section 4.2.3 - Utilities Management), and then rebuilds them from scratch, keeping them hidden (it will then be the user to open them if and when necessary). Then it deletes all pre-existing animations and hides all annotations about the machines (section 4.2.3 - Machine Annotations Management).

Once the preliminary cleaning phase is finished, according to the number of lines to be created (saved in the global variable *nLines*), the code shows or hides the corresponding lines (using the *showNode* and *hideNode* commands), then runs the single line code within a for loop, whose iterations range from 1 to *nLines* (in the case of a single line, the

code will exit the loop after one iteration).

In particular, the main cycle of the editor has been separated into two parts: in the first cycle the entire part concerning the positioning and management of geometries takes place (including shadow computation to increase the realism of the simulation), as well as the first part of the animation, the one that could be defined as main, within which the time horizon of the animation is defined.

The second part of the editor cycle needs to have this value specified: it is in fact the section about the animation of the so-called "bis elements", those inserted to give the illusion of continuity in the animated loop (already shown in section 3.4). As the dimensions and velocities of the lines vary, the time horizon of the animations related to each one will obviously be different; however, the scene is of course unique, as therefore the global animation must be unique. To overcome this problem, first the lines are created and all the first parts of the animations are calculated. After that, all time horizons will be saved in the *all\_time\_horizons* list. It will then be sufficient to take the highest value among them, and use that as the reference time horizon for the global animation (second cycle of the editor).

Finally, in the last part of the main, the code takes care of calculating the shadows projected on the ground by the lines present, updating the position of the annotations (so that they always point to the referenced machines), and showing the widget tools. As a last action, it calls the user in the main scene (in virtual reality), placing him/her on the balcony.

## **4.1.2 Control Room**

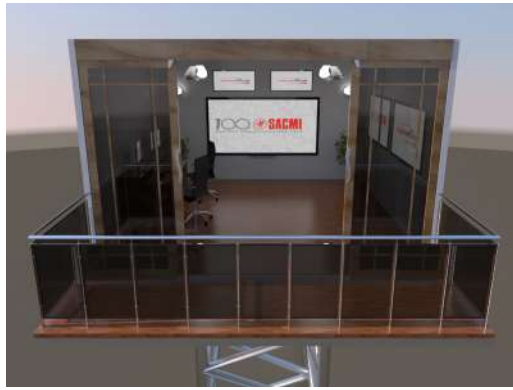
Before actually expanding the editor by increasing the number of existing lines, it has been decided to insert a control room from which the exploration of the scene could have been started. This room would have had the dual purpose of putting at ease an inexperienced user, providing a familiar environment (similar to that of an office) where he could take his first steps in virtual reality, becoming familiar with the headset and joysticks (not to be underestimated in case of lack of previous experience), in addition to a display of statistics regarding the selected lines, and a series of interactive tools for exploring online catalogs (directly in VR) and creating the final output (see sections 4.2.3 and 4.2.2.4, respectively).

### **4.1.2.1 Version I - "Compact" Solution**

In the first instance, it was decided to keep a "compact" version of the editor, by simply inserting the control room inside the same scene used for the creation of the lines. So the room was placed about 15 meters from the side of the first line.

In order to have a wider view of the functioning of the lines created by the editor, as well as a more effective visual comparison between them, it has been thought to place the control room high up, so that the user could enjoy a bird's eye view of the rest of the scene (figure 4.1b).

The control room has therefore been built: it is a room of 5 m x 5 m of walking floor, and 3 meters in height. To make it pleasant to the eyes of the user, it has been modelled



(a) Front view



(b) 3/4 view

Figure 4.1: Control room rendering

like a real office, with furniture and equipment (computers, chairs, stationery...), as shown in figure 4.1a. In addition to this, two interactive screens have been inserted: one showing the configuration chosen for the lines present, the other for the creation of the output by the user (discussed in detail in section 4.2).

Once the scene has been created from data specified on the PC screen, the user will then wear the headset and access virtual reality. Once the simulation is started, he/she will be located in the center of the control room, with his back to the lines. Once inside, the user can freely explore the room. It will be immediately possible to notice the only window facing "outside": by touching it (or "shooting" on it through the joystick pointer) the doors will open and the user will be free to move on the balcony, from which he/she will enjoy a view from the top of the lines.

From the balcony, the user can then descend freely through the teleport managed by joystick, or by using the views in the interactive menu attached to the hand.

The problem with this "compact" solution, however, resided in the fact that it was not possible to launch the editor directly in VR, but it was necessary to remove the viewer every time you wanted to recompute the lines, relaunch the program from the computer, and only then return to virtual reality once the editor had finished creating the scene. As is easily guessed, this made the use of this application extremely uncomfortable for the user.

This problem was due to the fact that, during the computations and reconstruction of the scene, if the headset was maintained on, all that could be seen would have been a fixed frame flickering in an uncontrollable way, which would have led to headaches and nausea for the user.

So the solution subsequently sought was to split the editor into two parts: on the one hand the control room itself, from which to start the editor with a special interface (see section 4.2.2.2) and on the other the main scene with the lines. The secondary scene with the control room would also have been used as a waiting room when the lines were being created in the main scene.

In this way, it would have been possible to do everything directly in virtual reality, without ever having to remove the headset.



Figure 4.2: Control room rendering (secondary scene)

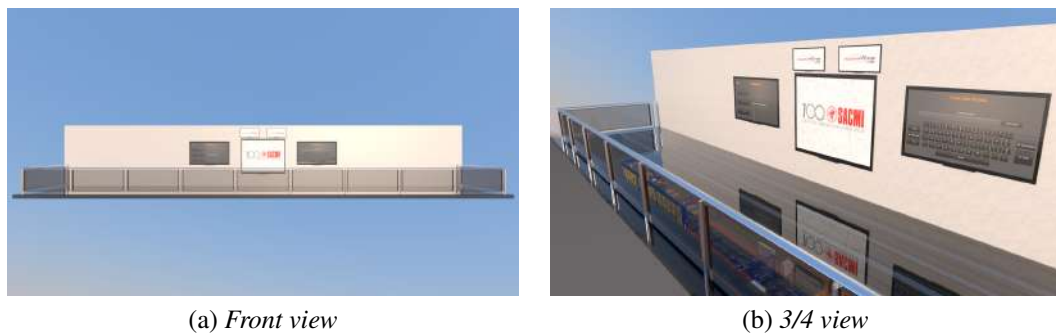


Figure 4.3: Balcony rendering (main scene)

#### 4.1.2.2 Version II - "Split" Solution

In order to move to the next version already introduced, the secondary scene had to be created first. The result (shown in figure 4.2) has been a small scene, with a limited size environment, hosting only the control room.

To reduce the sense of suffocation due to the small size of the space in which the user could actually move, the walls of the room were made of "glass" (simply modifying the walls through the library of materials provided by Vred) and the dome that encloses the scene was replaced with a natural landscape (in this case a forest). These features also provide a certain aesthetic pleasure, useful for a waiting room.

Subsequently, an element had to be introduced that would replace the control room in the main scene: something that could show the interactive screens already set up in the previous version, as well as permitting the already mentioned top view on the lines.

The solution found was to insert a balcony of 20 m x 3.5 m where the control room was previously located. The screens have then been inserted along the wall that closes the balcony. To facilitate the immersiveness of the scene and the view of the lines, the floor was also made of glass (and the metal support underneath removed), so that the view of the lines could be enjoyed at all times (figure 4.3).

Apart from the simple construction of the scenes, the next problem has been to create a communication channel between one environment and another. Ideally, in fact, the



operational flow would be the following:

1. Start simulation in virtual reality in the control room;
2. Selection of line parameters via interactive screen;
3. Transmission of line parameters to the scene containing the editor;
4. Line creation;
5. Notification by the main scene of completion of the lines;
6. Switching from one scene to another in virtual reality for the user.

As already seen in section 4.1.1, the only information to be exchanged is the number of lines to be created and the lists concerning the respective parameters. The interactive widget (shown in section 4.2.2.2) effectively takes care of the creation of the lists, so the only problem is the actual transmission of this data.

The final solution makes use of a specific function provided by Vred. It is possible, in fact, to go and enable a web interface within each scene, making it possible to govern it through a web server. This function must be enabled, specifying the port to which the scene in question is interfaced (the default port is 8888). Then, opening any web browser and going to localhost:XXXX (specifying the exact port) the user will connect to Vred through the internal web server.

If *Python API* is selected, a text box will appear where Python commands can be entered. If *Apply* is selected, they will be sent to the open scene, basically controlling it externally.

A way was then sought to automatically write the lists created by the widget in form of a character string, and with it form a series of commands (always in the form of a string) to be inserted in the text box under the Python API for the main scene containing the editor.

In order to do this, however, it was first necessary to find a method to automatically open the two scenes to a predefined port, in order to make the whole process automatic (otherwise it would be necessary to set everything manually each time).

The final method developed makes use of the commands provided by Vred for the Windows command prompt [4].

In general, two methods of command execution are made available:

- ***-prepython “command1(); command2()”*** will execute any semicolon-separated python command prior to loading the scene;
- ***-postpython “command1(); command2()”*** will execute any semicolon-separated python command after to loading the scene.

In our case, the mode chosen was the second.  
The available Python commands are:

- ***-wport “port”*** sets the port for the web interface;
- ***-screen “id”*** sets the id for the display:

Table 4.1: **Batch file:** Open Editor Scenes

---

```

1 :: Open Lines part listening to port 63340
2 start "" "C:\Program Files\Autodesk\VREDPro-12.3\Bin\WIN64\VREDPro.exe" -console
   -wport 63340 -postpython "load('C:\Users\SelleGia\Documents\LINE EDITOR\
   Editor_Linee_Multiple_ONLY_LINES.vpb')"
3
4 :: Open Control Room part listening to port 63342
5 start "" "C:\Program Files\Autodesk\VREDPro-12.3\Bin\WIN64\VREDPro.exe" -console
   -wport 63342 -postpython "load('C:\Users\SelleGia\Documents\LINE EDITOR\
   Control_Room_EDITOR.vpb')"
6
7 PAUSE

```

---

- -1: default;
- 0: primary;
- 1: secondary;

- **-software\_opengl** uses software gl for compatibility purposes;
- **-no\_opengl** disables opengl and uses Ray Tracing;
- **-nobanner** disables the startup video banner;
- **-fast\_start** defers some startup initializations to the first call of the respective function;
- **-console** displays console output.

In order to open the scenes at the desired ports, a batch file has been created to perform the scene setting operations sequentially.

A batch file is a script file in DOS, OS/2 and Microsoft Windows. It consists of a series of commands to be executed by the command-line interpreter, stored in a plain text file. A batch file may contain any command the interpreter accepts interactively and use constructs that enable conditional branching and looping within the batch file, such as *IF*, *FOR*, and *GOTO* labels. The term "batch" is from batch processing, meaning "non-interactive execution", though a batch file may not process a batch of multiple data.

The final batch file is shown in table 4.1.

So in order to open the scenes correctly and start using them, simply open the .bat file, and it will automatically open everything that is needed.

Now, the next step was to access the web server page that governed the two scenes. To do this, the Python *threading* and *requests* libraries have been used.

*Requests* is an Apache2 licensed HTTP library written in *Python for Human Beings*. Requests does all the work to implement HTTP/1.1 on Python - making it easy to integrate applications with web services. There is no need to manually add query strings to URLs, or form-encoding POST data. Keep-alive and pooling HTTP connections are

100% automatic, all thanks to `urllib3`, which is contained within `Requests` [8].

Thanks to this package, an `http_request` function has been created in Python. This function consists of a `requests.get` that sets the string contained in the text box (initially empty) under the Python API relative to the desired scene, drawing from the lists populated by the widget when selecting the parameters.

Then an instance of the `Thread` class (part of the `threading` package) is invoked. As stated in the package documentation [9], The `Thread` class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the `run()` method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, only override the `__init__()` and `run()` methods of this class. Once a thread object is created, its activity must be started by calling the thread's `start()` method. This invokes the `run()` method in a separate thread of control. Once the thread's activity is started, the thread is considered "alive". It stops being alive when its `run()` method terminates – either normally, or by raising an unhandled exception.

A thread can be flagged as a “daemon thread”. The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the `daemon` property or the `daemon` constructor argument.

The class constructor and its arguments are defined as:

```
class threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)
```

- **group**: should be *None*; reserved for future extension when a *ThreadGroup* class is implemented.
- **target**: is the callable object to be invoked by the `run()` method. Defaults to *None*, meaning nothing is called.
- **name**: is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.
- **args**: is the argument tuple for the target invocation. Defaults to `()`.
- **kwargs**: is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.
- **daemon**: daemon A boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`). This must be set before `start()` is called, otherwise *RuntimeError* is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to `daemon = False`. The entire Python program exits when no alive non-daemon threads are left.

The instance of the `Thread` class that is created will have the previously declared `http_request` function as its target.

After making this statement, it is specified that the thread will be a daemon, and it is

started using the *start()* command.

By doing so, the lists of variables (with their correct values) will be declared in the editor scene as global variables, effectively transmitting them from one scene to another.

The final Python code is shown in table 4.2.

Table 4.2: **Python code:** Control Room - Send Data Lists

---

```
1 from threading import Thread
2
3 import requests
4
5
6 def http_request():
7     # ----- Write Commands to set lists and START EDITOR in localhost at port
8     # 63340 (port of the editor part)
9     requests.get(
10         "http://localhost:63340/pythonapi?value=nLines = " + str(nLines) + "\nlist_conveyor_velocity = [" + str(list_conveyor_velocity[0]) + ", " + str(list_conveyor_velocity[1]) + ", " + str(list_conveyor_velocity[2]) + "]\nlist_d = [" + str(list_d[0]) + ", " + str(list_d[1]) + ", " + str(list_d[2]) + "]\nlist_n = [" + str(list_n[0]) + ", " + str(list_n[1]) + ", " + str(list_n[2]) + "]\nlist_n_stat = [" + str(list_n_stat[0]) + ", " + str(list_n_stat[1]) + ", " + str(list_n_stat[2]) + "]\nlist_three_per_arc = [" + str(list_three_per_arc[0]) + ", " + str(list_three_per_arc[1]) + ", " + str(list_three_per_arc[2]) + "]\nlist_p = [" + str(list_p[0]) + ", " + str(list_p[1]) + ", " + str(list_p[2]) + "]\nlist_palletDistance = [" + str(list_palletDistance[0]) + ", " + str(list_palletDistance[1]) + ", " + str(list_palletDistance[2]) + "]\nlist_bool_stations_cancelled = [" + str(list_bool_stations_cancelled[0]) + ", " + str(list_bool_stations_cancelled[1]) + ", " + str(list_bool_stations_cancelled[2]) + "]\nexecutePython('selectVariantSet(\"VR - Line Editor\")')")
11
12
13
14
15
16
17 thread = Thread(
18     target=http_request
19 )
20
21 thread.daemon = True
22
23 thread.start()
```

---

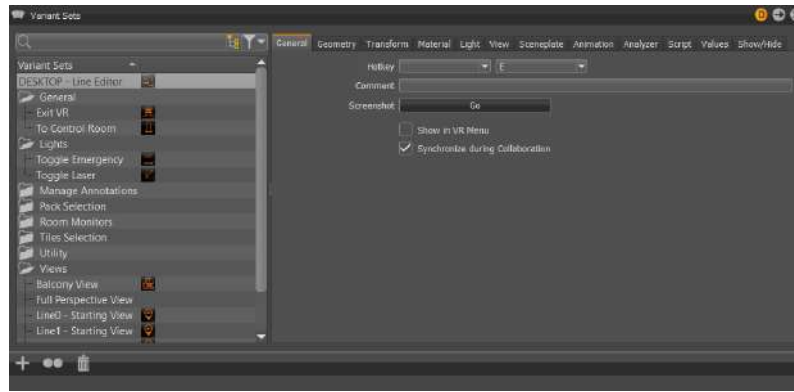


Figure 4.4: Editor's Variant Sets module

## 4.2 VR Tools Implementation

### 4.2.1 Vred Variant Sets

In order to create programmed variations within the scene, Vred provides the "Variants" entities. However, in many cases, a variant alone is not enough for managing complex models; for example, where many different switch nodes are needed to represent a specific version of the loaded model. Variant Sets provides more functionality, allowing the activation many states simultaneously. For configurators, logical connections can be generated [3].

In these cases, it is more convenient to use Variant Sets, or Vsets, to define multiple states for different properties at same time.

The Variant Sets module is used to create and modify variant sets, as well as add hotkeys [1]. As an example, the editor's Variant Set module is shown in figure 4.4. This module has a section listing every variant set of the scene on the left, tabs on the right, and an Icon Bar at the bottom, which contains shortcut icons for creating a new variant, duplicating the currently selected variant, and deleting the currently selected variant.

The list in the left section contains entries of already-defined variant sets. The context menu that can be recalled with the right click of the mouse is used to create new sets, as well as to perform other actions on its nodes. The tabs can be used to switch between the different types. Different options are then available for variant sets (table 4.3).

Finally, several tabs are present allowing the Variant Set to be associated with a wide range of actions (table 4.4).

Basically, the variant sets are the real key for all the tools that can be implemented in the scene to interact with it. That's why a lot of use has been made of them inside the editor (consider that the main code of the editor itself is enclosed within the variant set *DESKTOP - Line Editor* and *VR - Line Editor*, under the "script" tab). The tools that involve the use of these entities will be presented in section 4.2.3.

Table 4.3: **Variant Sets module** - Context menu options

<b>New Set:</b>	Creates a variant set.
<b>New Group:</b>	Creates groups to organize all variant sets that behave like folders. Variant sets can be dragged into any group or removed from a group by dragging it on its parent folder. It is not possible to recursively store folders within folders.
<b>Close Groups:</b>	Collapses the list of groups in the Variant Sets dialog box.
<b>Select:</b>	Selects and applies the current variant sets.
<b>Select Defaults:</b>	Reverts to the previously applied variant sets.
<b>Duplicate:</b>	Creates a copy of the selected variant sets.
<b>Rename:</b>	Enables the renaming of the selected variant sets.
<b>Delete:</b>	Deletes the selected variant sets.
<b>Clear:</b>	Removes all Variant Sets and groups.
<b>Optimize Sets:</b>	Optimizes and deletes all missing connections within the variant sets.
<b>Remove Empty Sets:</b>	Removes all links (parts) of the variant sets not assigned to anything in the current scene.
<b>Create Preview:</b>	Renders a preview and automatically assigns an icon to the selected variant set.
<b>Show/Hide Geometry:</b>	Shows or hides nodes dragged from the Scene Graph into the Show/Hide tab.
<b>Show Connected Variants:</b>	Selects the variants used in the variant set.

## 4.2.2 Widget Tools

### 4.2.2.1 Building Widgets With PySide2

For the realization of the interactive tools inside the scene in virtual reality that require the use of interactive screens, it has been necessary to use the functions and classes provided by the Python PySide2 library (already introduced in section 3.5.2.1).

Unlike the previous single line case, where interactions were done through keyboard and pc screen, and where widgets could be realized externally through Qt Designer, in this new case widgets had to be necessarily created at the opening of the scene (and only at that moment) via a script ad-hoc.

To do this, the script containing the creation of these widgets was inserted and saved inside the script editor of the scene containing the control room. This is because all the code contained within it is executed automatically when the part is opened. To execute it again, the operation should be executed by hand.

First of all, the construction of each widget starts from the declaration of an instance of the *QtWidget* class [6]. The widget is the atom of the user interface: it receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen. Every widget is rectangular, and they are sorted in a Z-order. A widget is clipped by its parent and by the widgets in front of it.

Table 4.4: **Variant Sets module** - Tabs

<b>General:</b>	<p><b>Hotkey</b> - Where hotkeys for switching between variant sets can be defined. The first box is for a modifier key, such as Alt or Ctrl, while the second is for selecting an alphanumeric character for the shortcut. The second field is required to assign the hotkey (for example, a letter or sign);</p> <p><b>Comment</b> - Adds a custom comment to each variant set.</p> <p><b>Screenshot (Go button)</b> - Creates a quick rendering of the Render Window, using the currently selected variant set and render settings. VRED asks for a location and format to save the image, which is an 800x600 bitmap image, by default.</p>
<b>Geometry:</b>	Creates or modifies the geometry variants. To create a geometry variant, drag a geometry node or group node to the right section in the Geometry tab.
<b>Transform:</b>	Creates or modifies the transform variants. To add a transform variant to the variant set, drag a node with transform variants to the right section in the Transform tab. You can also drag a variant from the Transform dialog or from the Variants dialog to that section.
<b>Material:</b>	Creates or modifies the material variants. To create a material variant, drag a material node to the right section in the Material tab.
<b>Light:</b>	Creates or modifies the light variants. To create a light variant set, drag a light variant to the right section in the Light tab.
<b>View:</b>	Creates or modifies the Viewport variants. Viewport variants can be set within the View tab. Drag a previously created view into the Camera Editor to the right side of the View tab.
<b>Sceneplate:</b>	Creates or modifies the sceneplate (frontplates and backplates) variants. To create a new sceneplate variant set, a sceneplate can be dragged from the Sceneplate Editor to the right side of the Sceneplate tab. If this variant set is selected, the sceneplate will be set to active. If another variant set is activated, the new sceneplate will be set to active. It is possible to add multiple sceneplates to one variant set. It is also possible to add one sceneplate to multiple variant sets.
<b>Animation:</b>	Creates or modifies the animation variants. To create an animation variant set, drag an animation from the Clip Maker (Clips icon) to the right side of the Animation tab.
<b>Analyzer:</b>	Lets the user enable, disable, or modify clipping planes within a variant set.
<b>Script:</b>	For advanced users, each variant set can contain a Python script, executed automatically, each time the variant set is activated, whether through the Variant Set module, a hotkey, or another script.
<b>Values:</b>	Add generic key/value pairs to any variant set.
<b>Show/Hide:</b>	Shows or hides a list of Scene Graph nodes, based on whether Show Geometry or Hide Geometry was selected in the Variant Set context menu.

Once a widget has been declared, it must be initialized. First, a style can be declared using the `setStyleSheet` command, to dictate the general lines that are inherited from all the elements inside the widget (except specific declarations, made at a lower level), such as the chosen font, and the size of the text elements.

After that, it is possible to declare all the elements internal to the widgets to be created.

In general, the first (and likewise the most immediate) elements inserted are the purely textual elements, created through the *QLabel* class. These labels are used for displaying text or an image. No user interaction functionality is provided.

By default, labels display left-aligned, vertically-centered text and images, where any tabs in the text to be displayed are automatically expanded. However, the look of a *QLabel* can be adjusted and fine-tuned in several ways. In particular, the positioning of the content within the *QLabel* widget area can be tuned with *setAlignment()* and *setIndent()*.

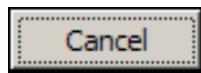


Figure 4.5: QPushButton

Right after the labels, the *QPushButton* (figure 4.5), or command button, is perhaps the most commonly used widget in any graphical user interface. Push (click) a button to command the computer to perform some action, or to answer a question. A command button is rectangular and typically displays a text label describing its action.

If the button is disabled, the appearance of the text and icon will be manipulated with respect to the GUI style to make the button look “disabled”.

A push button emits the signal *clicked()* when it is activated by the mouse, the Spacebar or by a keyboard shortcut. Connecting a function to this signal means performing it as soon as it is clicked (all specific functions performed by the widgets in the editor are shown in the immediately following sections). Push buttons also provide less commonly used signals, for example *pressed()* and *released()*.

Command buttons in dialogs are by default auto-default buttons, i.e., they become the default push button automatically when they receive the keyboard input focus. A default button is a push button that is activated when the user presses the Enter or Return key in a dialog. This can be changed by the command *setAutoDefault()*.



Figure 4.6: QLineEdit

To receive and display a text input, *QLineEdit* entities are used (fig. 4.6).

The text can be changed with *setText()* or *insert()*, and then it can be retrieved with *text()*; the displayed text (which may be different) is retrieved with *displayText()*. Text can be selected with *setSelection()* or *selectAll()*, and the selection can be *cut()*, *copy()*ied and *paste()*d. The text can be aligned with *setAlignment()*. In case the lineEdit requires to be brought into read mode, simply use the command *setReadOnly(bool)*

When the text changes the *textChanged()* signal is emitted; when the text changes other



than by calling *setText()*, the *textEdited()* signal is emitted; when the cursor is moved the *cursorPositionChanged()* signal is emitted; and when the Return or Enter key is pressed the *returnPressed()* signal is emitted.

When editing is finished, either because the line edit lost focus or Return/Enter is pressed the *editingFinished()* signal is emitted.



Figure 4.7: QRadioButton

For applications where the user is expected to make a choice between several possible options, radio buttons becomes of fundamental importance (fig. 4.7).

A *QRadioButton* is an option button that can be switched on (checked) or off (unchecked). Radio buttons typically present the user with a “one of many” choice. In a group of radio buttons, only one radio button at a time can be checked; if the user selects another button, the previously selected button is switched off.

Radio buttons are auto-exclusive by default. If *autoExclusive* is enabled, radio buttons that belong to the same parent widget behave as if they were part of the same exclusive button group. If multiple exclusive button groups for radio buttons that belong to the same parent widget are needed, they have to be put into a *QButtonGroup*.

Whenever a button is switched on or off, it emits the *toggled()* signal. Connect to this signal if an action has to be triggered each time the button changes state. Use *isChecked()* to see if a particular button is selected.

Just like *QPushButton*, a radio button displays text, and optionally a small icon. The icon is set with *setIcon()*. The text can be set in the constructor or with *setText()*.



Figure 4.8: QProgressBar

A progress bar is used to give the user an indication of the progress of an operation. The progress bar uses the concept of steps. It is set up by specifying the minimum and maximum possible step values, and it will display the percentage of steps that have been completed when it is later given the current step value. The percentage is calculated by dividing the progress ( $value() - minimum()$ ) divided by  $maximum() - minimum()$ . The minimum and maximum number of steps can be specified with *setMinimum()* and *setMaximum()*. The current number of steps is set with *setValue()*. The progress bar can be rewound to the beginning with *reset()*.

It can therefore be seen how the PySide2 library offers a great number of possibilities for the creation of widgets (those presented here are just some of the countless elements made available), and is therefore an essential tool for any application that provides a Graphic User Interface.

Now the widgets specifically created in the editor will be shown more closely.

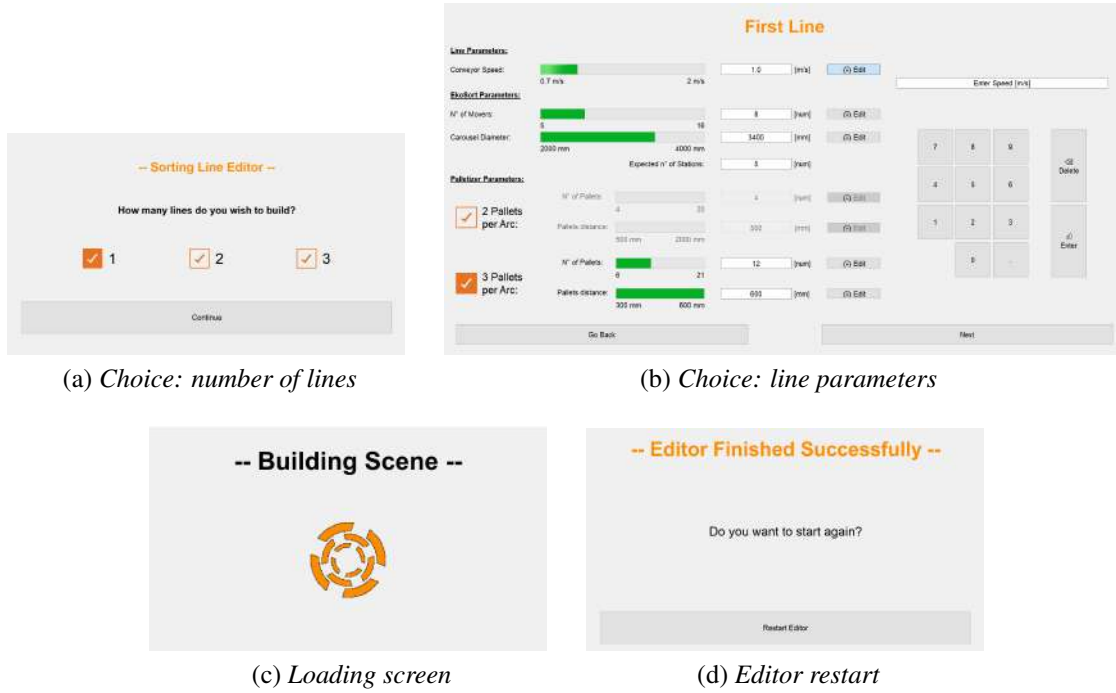


Figure 4.9: Editor parameters widget

#### 4.2.2.2 Editor Parameters Choice

The final widget for the selection of parameters within the editor is shown in figure 4.9. First, it can be noticed that it consists of two secondary widgets. The first (figure 4.9a) is used to operate the choice on the number of lines that the user would like to see represented in the scene, while the second (figure 4.9b) is used to select the parameters of each single line.

To achieve this result, it has been necessary to set the two separate widgets, and then merge them into a widget on the upper level. Then, the second widget is initially hidden to leave only the first one visible. During the selection process, the reverse operation is performed to let the second one appear. It is important to notice that only the second sub widget is used for each line: when switching to the selection of the next (or previous) lines, the elements concerning one or the other line are shown or hidden. For instance, note the label that acts as title for the widget ("First Line" in this case): there are actually four labels in that layout cell: "Your Line", "First Line", "Second Line" and "Third Line". Depending on the case, only one of the four will be shown (hiding the others), thus giving the illusion of scrolling between different widgets. Similarly, between the buttons at the bottom, the same method is applied to the pushButton "Next" (in figure) and "Create", which is shown (to the detriment of the other) during the selection of the parameters of the last line.

Lastly, to facilitate the use of the control room separated from the main scene, two additional widgets have been inserted: one that remains active during the loading of the lines in the main scene (figure 4.9c), and the other to allow the editor to be reset if necessary (figure 4.9d).

It can be easily noticed that the widget has changed with respect to the one realized

through Qt Designer (see section 3.5.1, figure 3.17): this is because, after various tests, it has been observed that in virtual reality the selection of parameters using sliders is inaccurate to say the least and difficult to manage. This is why it has been decided to perform the selection only through a virtual keypad prepared ad hoc. In order to be able to give a graphic idea of the boundaries for the various possibilities of choice of parameters, some *progressBars* have been inserted and they are updated at each change of the parameter.

Finally, it can be observed the addition of the "expected number of stations" section below the carousel diameter selection. In this lineEdit over which the user has no direct control, a preview of the number of tile stacking stations is automatically written according to the carousel diameter and the number of movers selected. This method has been designed to replace the resolution method used to solve the interpenetration between the geometry of the stations and the conveyor belt (introduced in section 3.1.2.5), making it more immediate and free from any further intervention by the user.

Now it is opportune to go and analyze the various functions that are implemented within the widget to make the operation of this tool possible.

## Open Parameters Widget

As a first step, the user must select how many lines he wants the editor to build. So the first widget shown is the one in figure 4.9a. The choice is made through the selection of one of the three pushButton present.

When pressing the Continue button, the chosen value must be saved in the global variable nLines.

Table 4.5: **Python code:** Open parameters widget (1)

---

```

1 # ----- SIGNALS AND SLOTS -----
2 self.line_selector_widget.button_continue.clicked.connect(self.
    open_parameter_widget)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def open_parameter_widget(self):
7     global nLines
8
9     if self.line_selector_widget.radio_1.isChecked():
10         nLines = 1
11     elif self.line_selector_widget.radio_2.isChecked():
12         nLines = 2
13     elif self.line_selector_widget.radio_3.isChecked():
14         nLines = 3
15     else: # Should never enter here
16         print("----- ERROR: nLines is not set properly -----")
17
18     # ----- Re-declare default GLOBAL lists
19     global list_conveyor_velocity, list_n, list_d, list_three_per_arc, list_p,
20         list_palletDistance, list_n_stat, list_bool_stations_cancelled
21
22     list_conveyor_velocity = [1.0, 1.0, 1.0] # conveyor speed [m/s]
23     list_n = [5, 5, 5] # movers
24     list_d = [2000, 2000, 2000] # carousel diameter [mm]
25     list_three_per_arc = [False, False, False] # 2 pallet stations per arc
26     list_p = [4, 4, 4] # number of pallet stations
27     list_palletDistance = [500, 500, 500] # distance between pallet stations [mm]
28     list_n_stat = [0, 0, 0]
29     list_bool_stations_cancelled = [False, False, False] # default values

```

---

After that, it is necessary to initialize the starting lists already mentioned in section 4.1.1 to the default values.

Finally, the choice of the number of lines is hidden and the parameter selection widget is shown, setting the right title and the visibility of the Next or Create button, based on the value of the nLines variable. The values stored within the progressBar are also reset.

Table 4.6: **Python code:** Open parameters widget (2)

---

```

1  # ----- Prepare parameters selection widget
2  if nLines > 1: # Need to build more than 1 line
3      self.parameters_selector_widget.label_title.setText("First Line")
4
5  # nLines - 1 is used to give the boundary!
6  # EXAMPLE: nLines == 2 (must build 2 lines) ----> (nLines - 1) == 1
7  # If current line number is 0, activate "NEXT" button
8  # If current line number is 1 (equal to nLines - 1), activate "CREATE" button
   (last parameter widget to be shown)
9  if self.parameters_selector_widget.line_number < (nLines - 1):
10     self.parameters_selector_widget.button_next.setVisible(True)
11     self.parameters_selector_widget.button_create.setVisible(False)
12  else:
13     self.parameters_selector_widget.button_next.setVisible(False)
14     self.parameters_selector_widget.button_create.setVisible(True)
15
16  self.parameters_selector_widget.progressBar_s.setValue(100)
17  self.parameters_selector_widget.progressBar_n.setValue(5)
18  self.parameters_selector_widget.progressBar_d.setValue(2000)
19  self.parameters_selector_widget.radioButton_2arc.setChecked(True)
20  self.parameters_selector_widget.enable_2arc_choice()
21  self.parameters_selector_widget.progressBar_p_2arc.setValue(4)
22  self.parameters_selector_widget.progressBar_dist_2arc.setValue(500)
23  self.parameters_selector_widget.progressBar_p_3arc.setValue(6)
24  self.parameters_selector_widget.progressBar_dist_3arc.setValue(300)
25
26  # ----- Show parameters selection widget
27  self.line_selector_widget.setVisible(False)
28  self.parameters_selector_widget.setVisible(True)

```

---

## Update lineEdit

Each time the value inside the progress bar is changed (since that is the numerical value that is then saved in the parameter lists), immediately the text string shown inside the lineEdit corresponding to the considered parameter must be updated so that the value and string are consistent.

Table 4.7: **Python code:** Update lineEdit

---

```
1 class WidgetLineParameters(QWidget):
2     def __init__(self):
3         QWidget.__init__(self)
4
5         # ----- Progress Bar
6         self.progressBar_s = QProgressBar()
7
8         # ----- lineEdit + Groups
9         self.lineEdit_s = QLineEdit("1.0")
10
11        # ----- SIGNALS AND SLOTS -----
12        self.progressBar_s.valueChanged[int].connect(self.update_lineEdit_s)
13
14        # ----- FUNCTIONS -----
15        @Slot()
16        def update_lineEdit_s(self, value):
17            float.value = float(value)
18            self.lineEdit_s.setText(str(float.value / 100))
```

---

### Update Expected Number of Stations

In order to give an estimate of the number of stations - based on the number of movers and the diameter of the carousel inserted in the choice of configuration - an empirical study was first carried out with the old version of the on-screen editor, observing in which cases it was actually appropriate to remove two or more stations from the initial configuration. The results were then included in a global list of tuples (*limit\_diameter\_list*). The index of the list elements refers to the number of movers selected: e.g. index 6 - the 7th element of the list, since it starts counting from 0 - refers to the choice of 6 movers.

The tuples included in the list are made of four elements, organized in this way:

0. the default number of stations for that number of movers;
1. the limit diameters for which, if the carousel diameter is **under** the value, 2 additional stations must be cancelled;
2. the default *Starting Value* (assuming selected diameter is **above** the 2nd element - index 1 - of the tuple). E.g: if it's 2, the first station to be cloned will be *Station2*;
3. the default *Limit Stat* value (assuming selected diameter is **above** the 2nd element - index 1 - of the tuple). E.g: if it's 5, the last station to be cloned will be *Station4* (*LimitStat* - 1).

Whenever the values of lineEdit concerning the number of movers or the diameter of the carousel are updated, the function saves both these values (variables *n\_mov* and *diam*) and performs a check within the list of tuples. It is necessary to access the list at the index given by *n\_mov*, and check the second element (given by index 1), using the command *limit\_diameter\_list[n\_mov][1]*. If this value is not zero, and if the selected diameter is less than the same value, then 2 more stations must be eliminated than those provided in the tuple (in first position, with index 0). In all other cases, instead, the expected number of stations is equal to the first element of the tuple.

Finally, the value of the lineEdit is set by transforming the number of stations into a string.

Table 4.8: **Python code:** Update expected n° of stations

---

```

1 # ----- LIMIT LIST -----
2 limit_diameter_list = [(0, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0),
3                        (0, 0, 0, 0), (4, 0, 1, 5), (5, 2400, 1, 6),
4                        (4, 0, 2, 6), (5, 0, 2, 7), (6, 0, 2, 8), (7, 0, 2, 9),
5                        (8, 0, 2, 10), (9, 3000, 2, 11), (10, 3500, 2, 12),
6                        (9, 0, 3, 9, 12), (10, 0, 3, 13), (11, 0, 3, 14)]
7
8 # ----- SIGNALS AND SLOTS -----
9 self.progressBar_n.valueChanged[int].connect(self.update_line_edit_stations)
10 self.progressBar_d.valueChanged[int].connect(self.update_line_edit_stations)
11
12 # ----- FUNCTIONS -----
13 @Slot()
14 def update_line_edit_stations(self):
15     global limit_diameter_list, list_bool_stations_cancelled
16
17     n_mov = self.progressBar_n.value()
18     diam = self.progressBar_d.value()
19
20     # checking the SECOND (1) element in the n-th tuple of the list (limit
21     # diameter)
22     # If it is != 0 (which means there is a limit diameter) and the diameter
23     # selected is BELOW this limit, 2 more stations must be cancelled
24     if limit_diameter_list[n_mov][1] != 0 and diam < limit_diameter_list[n_mov][1]:
25         stations_expected = limit_diameter_list[n_mov][0] - 2 # using the FIRST
26         (0) element in the n-th tuple of the list (expected number of stations)
27         list_bool_stations_cancelled[self.line.number] = True
28     else:
29         stations_expected = limit_diameter_list[n_mov][0]
30         list_bool_stations_cancelled[self.line.number] = False
31
32     self.line_edit.setText(str(stations_expected))

```

---



## Show Configuration Choice Elements

Every time the desired configuration for the palletizing machine is chosen (2 or 3 pallets per arch) through the two push buttons on the bottom left, the elements related to the corresponding choice must be enabled (and will therefore become interactive), while the others will be disabled (in figure 4.9a it can be seen how the elements related to the first configuration are disabled).

To do this, when declaring the elements of the widget, those related to one or the other configuration are saved in two lists: *list\_widgets\_2\_for\_arc* and *list\_widgets\_3\_for\_arc*. When selecting the first configuration, for example, for each element of the first list the command *setEnabled(True)* is given, while for each element of the second *setEnabled(False)* is given.

Table 4.9: **Python code:** Show configuration choice elements

---

```

1 # ----- SIGNALS AND SLOTS -----
2 self.radioButton_2arc.clicked.connect(self.enable_2arc_choice)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def enable_2arc_choice(self):
7     for item in self.list_widgets_2_per_arc:
8         item.setEnabled(True)
9
10    for item in self.list_widgets_3_per_arc:
11        item.setEnabled(False)
12
13    global list_three_per_arc
14    list_three_per_arc[self.line_number] = False

```

---

## Round Parameter Value

When the value of a parameter is updated, it must be rounded accordingly (e.g. the diameter of the carousel is rounded to hundreds of centimetres).

To do this, firstly the new value set for the corresponding *QProgressBar* must be read through the *value()* command, then the rounding is done through the *round()* command: the value and the number of positions to be rounded are passed to the command (for example: 2 = rounding to tenths, -2 = rounding to hundreds).

Then it is sufficient to set the rounded value in the progress bar using the *setValue()* command.

Table 4.10: **Python code:** Round parameter value

---

```
1 # ----- SIGNALS AND SLOTS -----
2 self.horizontalSlider.s.valueChanged.connect(self.round_speed)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def round_diameter(self):
7     current_value = self.lineEdit.d.value()
8     rounded_value = round(current_value, -2)
9     self.lineEdit.d.setValue(rounded_value)
```

---

## Enable Parameter Editing

As already said, in the passage from the selection of the parameters on screen to the one in virtual reality, the only mode allowed for the modification of the parameters has become the use of the numeric keypad implemented inside the widget.

By pressing the Edit button next to the parameter concerned (if it is not already checked), it will be selected, deselecting all the others. At this point, all the elements of the numeric keypad (which was previously disabled and therefore not interactive) will be activated. The "screen" of the keypad will display a message regarding the selected parameter (for example, in the case of the conveyor speed, the message will be "Enter Speed [m/s]").

If, on the other hand, the Edit key had already been selected, a press would simply deselect it and erase the message shown by the keypad screen.

Table 4.11: **Python code:** Enable parameter editing

---

```

1 # ----- SIGNALS AND SLOTS -----
2 self.button_edit_s.clicked.connect(self.enable_edit_speed)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def enable_edit_speed(self):
7     if self.button_edit_s.isChecked():
8         self.display_line_edit.setDisabled(False)
9
10    # ----- Uncheck other edit buttons
11    self.button_edit_n.setChecked(False)
12    self.button_edit_d.setChecked(False)
13    self.button_edit_p_2arc.setChecked(False)
14    self.button_edit_p_3arc.setChecked(False)
15    self.button_edit_dist_2arc.setChecked(False)
16    self.button_edit_dist_3arc.setChecked(False)
17
18    self.display_line_edit.setText("Enter Speed [m/s]")
19 else: # button_edit_s is not checked
20     self.display_line_edit.setDisabled(True)
21     self.display_line_edit.setText("")

```

---

## Type a Character

In order to write a character on the keyboard screen, this must first be enabled (an Edit button must therefore be checked).

If the screen is still showing one of the default messages (all saved in the *list\_initial\_messages* list in order to cross-check them), when a key containing a character is pressed, the message must be deleted and replaced with the character in question.

If there are other characters typed previously, the text string must be saved (in the variable *current\_text*), the character must be added, and the merge must be shown on screen using the command *setText(current\_text + "char")*.

Table 4.12: **Python code:** Type a character

---

```
1 # ----- SIGNALS AND SLOTS -----
2 self.button_1.clicked.connect(self.type_1)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def type_1(self):
7     current_text = self.display_line_edit.text()
8     if self.display_line_edit.isEnabled():
9         if current_text in self.list_initial_messages: # Initial message
10             self.display_line_edit.setText("1")
11         else:
12             self.display_line_edit.setText(current_text + "1")
13     else:
14         pass
```

---

## Delete a Character

In order to delete a character, the procedure is dual to the writing. If one of the pre-defined messages is present on the screen, it must simply become free of text when the backspace key is pressed. If a text string is already present, only the last character must be deleted each time the key is pressed. This is possible by using the command *next\_text = current\_text[:-1]*.

Table 4.13: **Python code:** Delete a character

---

```
1 # ----- SIGNALS AND SLOTS -----
2 self.button.backspace.clicked.connect(self.delete_char)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def delete_char(self):
7     current_text = self.display_line_edit.text()
8     if current_text in self.list_initial_messages: # Initial message
9         self.display_line_edit.setText("")
10    else:
11        next_text = current_text[:-1]
12        self.display_line_edit.setText(next_text)
```

---

## Confirm Value

After writing the desired value on the keypad screen, obviously it is necessary to make this change effective by sending the value to the progress bar of the parameter in question (this element is the one that keeps the value, and is consulted at the time of filling the lists for the editor code).

To do this, first of all it is necessary to save the text string stored inside the display using the *text()* command. Then, only when the keypad is enabled (i.e. any Edit key is checked) and the string is not one of the initial messages, the string is converted into the corresponding numeric value. Then a second check is done: if the value is lower or higher than the maximum limit allowed for the parameter in question, the corresponding progress bar will be assigned the lower or upper limit value, respectively. If the value is within the limits, it is assigned directly to the progress bar via the *setValue()* command. Finally, the Edit button for the parameter is deselected.

To know which parameter corresponds to the choice made, as many cases as the number of *Edit* buttons have been entered. In table 4.14 only the case of the conveyor speed is presented as an example.

Table 4.14: **Python code:** Confirm value

---

```
1 # ----- SIGNALS AND SLOTS -----
2 self.button_enter.clicked.connect(self.confirm_value)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def confirm_value(self):
7     value_chosen = self.display_line_edit.text()
8     if self.display_line_edit.isEnabled() and value_chosen not in self.
       list_initial_messages:
9         float_value_chosen = float(value_chosen)
10
11         if self.button_edit.s.isChecked():
12             speed_value_converted = int(round(float_value_chosen * 100, 0))
13
14             if speed_value_converted <= self.progressBar.s.minimum():
15                 self.progressBar.s.setValue(self.progressBar.s.minimum())
16             elif speed_value_converted >= self.progressBar.s.maximum():
17                 self.progressBar.s.setValue(self.progressBar.s.maximum())
18             else: # Value ok
19                 self.progressBar.s.setValue(speed_value_converted)
20
21                 self.button_edit.s.setChecked(False)
22         else:
23             pass
24
25         self.display_line_edit.setText("")
26         self.display_line_edit.setDisabled(True)
27     else:
28         pass
```

---

## Next Line Widget

After the selection of the parameters, when the *Next* button is pressed, the lists related to the line considered in the widget must be updated. To do this, it will simply read the value stored in the progress bar and save it in the lists, in the position regarding the considered line (0 = first line, 1 = second line, 2 = third line). For this purpose, each widget line shift updates the *line\_number* value, initialized by default to 0.

Table 4.15: **Python code:** Next line widget (updating lists)

---

```

1 # ----- SIGNALS AND SLOTS -----
2 self.button_next.clicked.connect(self.next_line_widget)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def next_line_widget(self):
7     global nLines, list_conveyor_velocity, list_n, list_d, list_three_per_arc,
8         list_p, list_palletDistance, list_n_stat, list_bool_stations_cancelled
9     # ----- Update Lists
10    list_conveyor_velocity[self.line_number] = float(self.progressBar_s.value() /
11        100)
12    list_n[self.line_number] = self.progressBar_n.value()
13    list_d[self.line_number] = self.progressBar_d.value()
14    # ----- three_per_arc is updated when selecting radiobuttons
15    if list_three_per_arc[self.line_number]: # 3 per arc chosen
16        list_p[self.line_number] = self.progressBar_p_3arc.value()
17        list_palletDistance[self.line_number] = self.progressBar_dist_3arc.value()
18    else: # 2 per arc chosen
19        list_p[self.line_number] = self.progressBar_p_2arc.value()
20        list_palletDistance[self.line_number] = self.progressBar_dist_2arc.value()
21    list_n_stat[self.line_number] = int(self.line_edit.text()) # Number of
22    stations expected

```

---

First, it updates the *line\_number* value, increasing it by 1. If the resulting value is the same as the last line (specified in *nLines*), the *Next* button must be hidden and the *Create* button must be shown, as the one that will be shown will be the last widget before the actual launch of the editor.

After that, the title will need to be updated according to the line number, using the *setText()* command.

Finally, a value reset procedure has been inserted: in case the user goes back and forth between the widgets, the program should remember the values already inserted. That's why when loading the widget of the next line, instead of resetting the default values, the values coming from the lists in the *line\_number* position are read. They are in fact initialized to the default values, and differ from them only if a choice has already been made and the user has returned back and forth through the widgets.

Table 4.16: **Python code:** Next line widget (preparing widget)

---

```

1      # ----- Prepare next "fictitious" widget: increase value of line number (to
      initialise the widget to represent next line)
2      self.line_number = self.line_number + 1
3      if self.line_number < (nLines - 1):
4          self.button_next.setVisible(True)
5          self.button_create.setVisible(False)
6      else:
7          self.button_next.setVisible(False)
8          self.button_create.setVisible(True)
9      # ----- Adjust title
10     if self.line_number == 1: # Second line widget
11         self.label_title.setText("Second Line")
12     elif self.line_number == 2: # Third line widget
13         self.label_title.setText("Third Line")
14     # ----- Set values (useful if user goes back and forth between widgets)
15     self.progressBar_s.setValue(list_conveyor_velocity[self.line_number] * 100)
16     self.progressBar_n.setValue(list_n[self.line_number])
17     self.progressBar_d.setValue(list_d[self.line_number])
18     if list_three_per_arc[self.line_number]: # Selected 3 pallets per arc
19         self.radioButton_3arc.setChecked(True)
20         self.enable_3arc.choice() # Because it is triggered only when
radioButton_3arc is CLICKED, not checked --> Call it explicitly
21         self.progressBar_p_2arc.setValue(4) # Default value
22         self.progressBar_p_3arc.setValue(list_p[self.line_number])
23         self.progressBar_dist_2arc.setValue(500) # Default value
24         self.progressBar_dist_3arc.setValue(list_palletDistance[self.line_number])
25     else: # Selected 2 pallets per arc
26         self.radioButton_2arc.setChecked(True)
27         self.enable_2arc.choice() # Because it is triggered only when
radioButton_2arc is CLICKED, not checked --> Call it explicitly
28         self.progressBar_p_2arc.setValue(list_p[self.line_number])
29         self.progressBar_p_3arc.setValue(6) # Default value
30         self.progressBar_dist_2arc.setValue(list_palletDistance[self.line_number])
31         self.progressBar_dist_3arc.setValue(300) # Default value

```

---

Analogous and dual is then the procedure followed by the **Go Back** function, associated with pressing the *Back* button.



### Back to Selection of Number of Lines

In case the user changes his mind about the number of lines to insert, he can always go back to the first widget. To do so, simply press the *Back* button inside the first line parameter widget.

When pressed, if the line number is 0, the second widget is hidden and the first widget is shown again.

Table 4.17: **Python code:** Back to selection of number of lines

---

```
1 # ----- SIGNALS AND SLOTS -----
2 self.parameters_selector_widget.button_back.clicked.connect(self.
    go_back_to_nlines_selection)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def go_back_to_nlines_selection(self):
7     if self.parameters_selector_widget.line_number >= 1:
8         pass
9     else:
10        # ----- Show line number selection widget
11        self.parameters_selector_widget.setVisible(False)
12        self.line_selector_widget.setVisible(True)
```

---

### **Back to Previous Line**

To return to the selection of the parameters of the previous line, go to the widget concerning the last line and press the Create button. When pressed, the lists must be updated with the inserted parameters and then the widget must be restored to the default values for a possible reopening of the editor.

Then, the communication protocol to pass lists from one scene to another is launched (as shown in section 4.1.2.2). The script to do this is placed inside the Variant Set *Send Data List to Lines Part + START EDITOR*, which is called using the command *executePython('selectVariantSet("set name")')*.

The other tools inside the control room are then updated to be consistent with the parameters chosen by the user. Again, the code is inserted in a Variant Set, called *Update VR Tools*.

Finally the widgets are hidden, and a loading page is shown, to indicate that the other scene is computing the lines.

### **Reset Editor Widget**

At the end of the loading of the lines, the main scene sends a signal to the control room. This signal teleports the user to the other scene, hides the loading widget, showing the one in figure 4.9d. If the user wants to re-launch the editor, it will be enough for him to return to the control room and press the Restart Editor button, which will simply display again the widget to select the number of lines.

Table 4.18: Python code: Back to previous line

---

```

1 # ----- SIGNALS AND SLOTS -----
2 self.parameters_selector_widget.button_create.clicked.connect(self.
    show_loading_widget_and_create_lines)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def show_loading_widget_and_create_lines(self):
7     global nLines, list_conveyor_velocity, list_n, list_d, list_three_per_arc,
        list_p, list_palletDistance, list_n_stat, list_bool_stations_cancelled
8
9     # ----- Update Lists
10    index = self.parameters_selector_widget.line_number
11    list_conveyor_velocity[index] = float(self.parameters_selector_widget.
        progressBar_s.value() / 100) # needs to be divided by 100 to obtain m/s
12    list_n[index] = self.parameters_selector_widget.progressBar_n.value()
13    list_d[index] = self.parameters_selector_widget.progressBar_d.value()
14    if list_three_per_arc[self.parameters_selector_widget.line_number]:
15        list_p[index] = self.parameters_selector_widget.progressBar_p_3arc.value()
16        list_palletDistance[index] = self.parameters_selector_widget.
            progressBar_dist_3arc.value()
17    else: # 2 per arc chosen
18        list_p[index] = self.parameters_selector_widget.progressBar_p_2arc.value()
19        list_palletDistance[index] = self.parameters_selector_widget.
            progressBar_dist_2arc.value()
20    list_n_stat[index] = int(self.parameters_selector_widget.line_edit.text())
21
22    # ----- Restore Default Values inside the Widget
23    self.parameters_selector_widget.radioButton_2arc.setChecked(True)
24    self.parameters_selector_widget.enable_2arc_choice()
25    self.parameters_selector_widget.progressBar_s.setValue(100)
26    self.parameters_selector_widget.progressBar_n.setValue(5)
27    self.parameters_selector_widget.progressBar_d.setValue(2000)
28    self.parameters_selector_widget.progressBar_p_3arc.setValue(6)
29    self.parameters_selector_widget.progressBar_dist_3arc.setValue(300)
30    self.parameters_selector_widget.progressBar_p_2arc.setValue(4)
31    self.parameters_selector_widget.progressBar_dist_2arc.setValue(500)
32    self.parameters_selector_widget.line_edit.setText("4")
33    self.parameters_selector_widget.label_title.setText("Your Line")
34
35    # ----- Restore default value of line number
36    self.parameters_selector_widget.line_number = 0
37
38    # ----- Send lists filled to LINES PART, starting the build of the scene
39    executePython('selectVariantSet("Send Data List to Lines Part + START EDITOR
        ")')
40
41    # ----- Update VR Tools inside the room
42    executePython('selectVariantSet("Update VR Tools")')
43
44    # ----- Hide parameters_selector_widget and show loading_widget
45    self.parameters_selector_widget.setVisible(False)
46    self.loading_widget.setVisible(True)

```

---

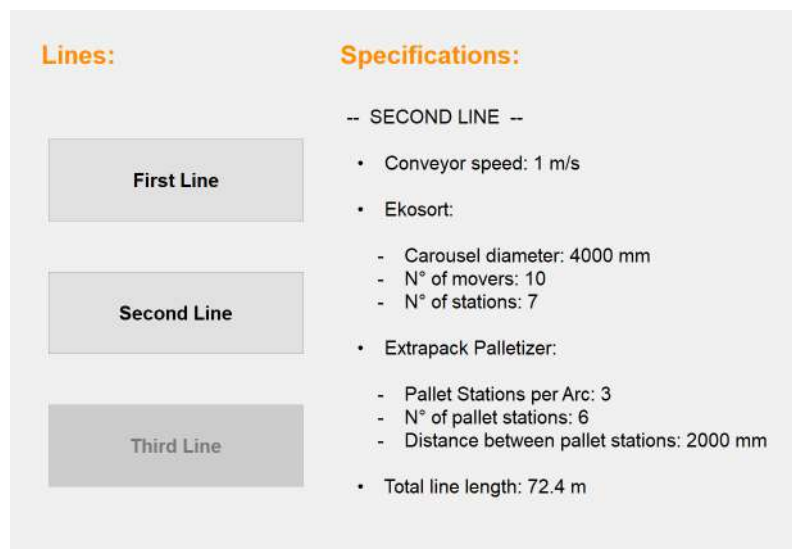


Figure 4.10: Parameters display widget

#### 4.2.2.3 Parameters Display

In order to be able to show the chosen parameters also inside the simulation (as information for the user), a special widget has been inserted in the form of an interactive screen.

It consists of three buttons on the left side (figure 4.10) and a label containing information about the selected line on the right side. The buttons on the left side are enabled only if the corresponding line is present: in this case, there are only two lines, that's why the Third line button is obscured.

As far as the right side is concerned, there are actually three different labels, which are hidden and shown appropriately according to the button that is pressed. In table 4.19 is represented the case in which the button of the first line is pressed.

Table 4.19: **Python code:** Open first line description

```

1 # ----- SIGNALS AND SLOTS -----
2 self.button1.clicked.connect(self.add_first_description)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def add_first_description(self):
7     self.temp.description.setVisible(False)
8     self.second.description.setVisible(False)
9     self.third.description.setVisible(False)
10    self.first.description.setVisible(True)

```

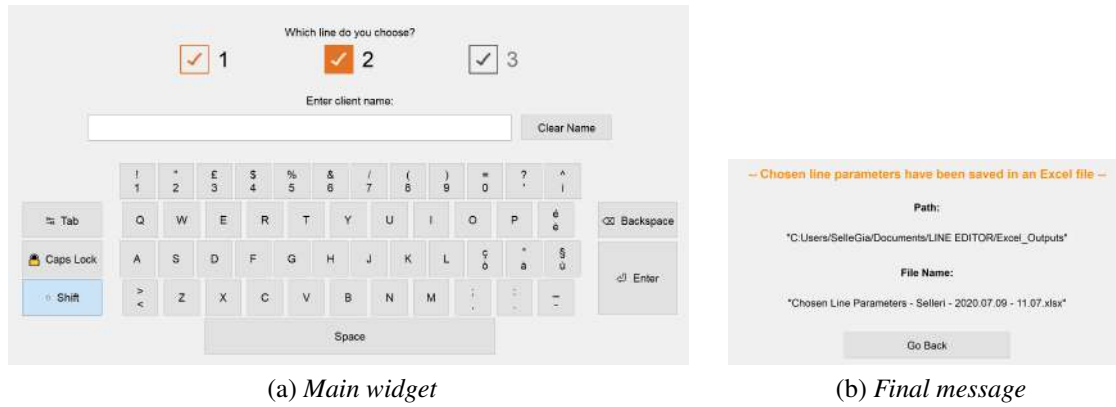


Figure 4.11: Output creation widget

#### 4.2.2.4 Output Creation

In order to ideally insert the editor in a business context, it is imperative to find a method to integrate it into the workflow. To do this, it is therefore advisable that, in addition to receiving input, it can also create an output of some kind, so that it can then be delivered to the subsequent portions of the production process.

Imagining the use of the editor in the pre-selling field, it was decided to create as output an Excel sheet containing the parameters of the line chosen by the user among those available in the scene.

To do this, it has been used the XlsxWriter library, which is a Python module for creating Excel XLSX files [19].

It can be used to write text, numbers, formulas and hyperlinks to multiple worksheets in an Excel 2007+ XLSX file. It supports features such as formatting and many more, including:

- 100% compatible Excel XLSX files;
- Full formatting;
- Merged cells;
- Defined names;
- Charts;
- Autofilters;
- Data validation and drop down lists;
- Conditional formatting;
- Worksheet PNG/JPEG/BMP/WMF/EMF images;
- Rich multi-format strings;
- Cell comments;

	A	B	C	D	E	F
1	Client:	Selleri				
2	Date and Time:	18/01/2020 - 17:07:06				
3						
4	Ekosort:					
5	Carousel Diameter [mm]:	4000				
6	N° of Movers:	16				
7	N° of Stations:	11				
8						
9	Extrapack Palletizer:					
10	Pallet Stations per Arc:	3				
11	N° of Pallet Stations:	21				
12	Pallet Stations Distance [mm]:	500				
13						
14	Conveyors:					
15	Speed [m/s]:	1				
16						

Figure 4.12: Excel output example

- Textboxes;
- Integration with Pandas;
- Memory optimization mode for writing large files.

It supports Python 2.7 (the version used by Vred), 3.4+ and PyPy and uses standard libraries only.

the use of the library has therefore been implemented within a widget used to select the line among those available (figure 4.11).

The primary widget (fig. 4.11a) is composed of two main parts: in the upper one, the line is selected by checking one of the push buttons available (note how, in this example, the choice of the third line is obscured because it is not represented in the scene), while in the lower one a keyboard has been created to enter the name of the user, so that the output can be customized.

Once the output is created, a secondary widget appears instead of the previous one (fig. 4.11b). Inside it, it is specified the final name of the output file and its location in the PC memory.

To better understand the operation of the widget, it is appropriate to analyze the main functions implemented in the program.

## Type a Character

The function used to write a character is very similar to the one already presented in section 4.2.2.2, but in this case, being to all intents and purposes a keyboard, different characters are entered depending on whether the *Shift* or *Caps Lock* key is active. Therefore, it is necessary to make a check on them and then write the right character accordingly.

Table 4.20: **Python code:** Type a character

---

```
1 # ----- SIGNALS AND SLOTS -----
2 self.button_a.clicked.connect(self.type_a)
3
4 # ----- FUNCTIONS -----
5 @Slot()
6 def type_a(self):
7     current_text = self.lineEdit.text()
8     if self.button_caps_lock.isChecked() or self.button_shift.isChecked(): # Caps
9         Lock OR Shift is active
10         self.lineEdit.setText(current_text + "A")
11         self.button_shift.setChecked(False) # Uncheck Shift in case it was
12         checked
13     else: # Shift and Caps Lock are NOT activated
14         self.lineEdit.setText(current_text + "a")
```

---

## Create Excel File

By pressing the Enter key on the virtual keyboard, the string inside the keyboard *lineEdit* is saved as the customer's name. The current date and time is then saved in a string in the format *y.m.d - h.m* (e.g. 2020.07.09 - 16.28).

The code then creates a workbook using the *xlsxwriter.Workbook()* command, which takes as argument the path of the file to be created, including its name. A worksheet must then be added to the workbook using the *add\_worksheet()* command.

Then the cell styles will be set afterwards, as well as the column width adjustment.

Table 4.21: **Python code:** Create Excel file (1)

---

```
1 # ----- IMPORTS -----
2 from datetime import datetime
3 import xlsxwriter
4
5 # ----- SIGNALS AND SLOTS -----
6 self.button_enter.clicked.connect(self.create_excel_file)
7
8 # ----- FUNCTIONS -----
9 @Slot()
10 def create_excel_file(self):
11     global client_name, line_choice
12
13     # ----- Get the client's name from the lineEdit object's text
14     client_name = self.lineEdit.displayText()
15
16     # ----- Get current day's date
17     now = datetime.now()
18     dt_filename = now.strftime("%Y.%m.%d - %H.%M")
19     dt_string = now.strftime("%d/%m/%Y - %H:%M:%S")
20
21     # ----- Create a workbook and add a worksheet
22     path_string = "D:\\Users\\Username\\Directory"
23
24     if client_name != "": # if line_edit is NOT EMPTY
25         workbook = xlsxwriter.Workbook(path_string + "\\Excel_Outputs\\Chosen Line
26         Parameters - " + str(client_name) + " - " + str(dt_filename) + ".xlsx")
27     else: # line_edit is EMPTY
28         workbook = xlsxwriter.Workbook(path_string + "\\Excel_Outputs\\Chosen Line
29         Parameters - Unknown - " + str(dt_filename) + ".xlsx")
30
31     worksheet = workbook.add_worksheet()
32
33     # ----- Add formats to use to highlight cells.
34     bold = workbook.add_format({'bold': True})
35
36     centered = workbook.add_format()
37     centered.set_align('center')
38
39     # ----- Adjust the column width.
40     worksheet.set_column(0, 0, 30)
41     worksheet.set_column(1, 1, 25)
```

---



Finally, it begins the actual filling of the cells with the elements of the interested lists, through the command *write(cell, string)*. To determine the cell, it is necessary to insert a string whose first element is the letter denoting the column, and the second the line number.

As a last action, the secondary widget is made visible (and its content updated), hiding the primary widget.

Table 4.22: **Python code:** Create Excel file (2)

---

```

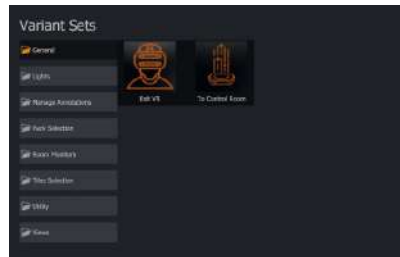
1
2  # ----- Fill output Excel file with data
3  worksheet.write('A1', 'Client:', bold)
4  worksheet.write('B1', str(client.name), centered)
5
6  worksheet.write('A2', 'Date and Time:', bold)
7  worksheet.write('B2', str(dt_string), centered)
8
9  worksheet.write('A4', 'Ekosort:', bold)
10 worksheet.write('A5', 'Carousel Diameter [mm]:')
11 worksheet.write('B5', str(list_d[line.choice]), centered)
12 worksheet.write('A6', 'N of Movers:')
13 worksheet.write('B6', str(list_n[line.choice]), centered)
14 worksheet.write('A7', 'N of Stations:')
15 worksheet.write('B7', str(list_n_stat[line.choice]), centered)
16
17 worksheet.write('A9', 'Extrapack Palletizer:', bold)
18 worksheet.write('A10', 'Pallet Stations per Arc:')
19 if list_three_per_arc[line.choice]: # == True
20     worksheet.write('B10', '3', centered)
21 else:
22     worksheet.write('B10', '2', centered)
23 worksheet.write('A11', 'N of Pallet Stations:')
24 worksheet.write('B11', str(list_p[line.choice]), centered)
25 worksheet.write('A12', 'Pallet Stations Distance [mm]:')
26 worksheet.write('B12', str(list_palletDistance[line.choice]), centered)
27
28 worksheet.write('A14', 'Conveyors:', bold)
29 worksheet.write('A15', 'Speed [m/s]:')
30 worksheet.write('B15', str(list_conveyor_velocity[line.choice]), centered)
31
32 workbook.close()
33
34 if client.name != "": # if line_edit is NOT EMPTY
35     self.file_name_label_2.setText("\Chosen Line Parameters - " + str(
client.name) + " - " + str(dt_filename) + ".xlsx\")
36 else: # if line_edit is EMPTY
37     self.file_name_label_2.setText("\Chosen Line Parameters - Unknown - " +
str(dt_filename) + ".xlsx\")
38
39 self.main_widget.setVisible(False)
40 self.message_widget.setVisible(True)

```

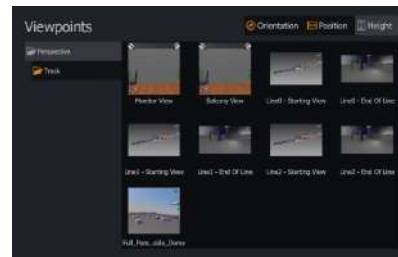
---



(a) Main page



(b) Variant Sets



(c) Viewpoints

Figure 4.13: VR interactive menu

### 4.2.3 Immersive Tools

In order to make the experience in virtual reality more pleasant and useful, a series of immersive tools involving interaction with the environment of the scene have been inserted into the editor.

These tools are implemented through the use of Variant Sets, and can be activated from the interactive menu in virtual reality (opened by pressing a special button on the joystick), which once it has been opened remains attached to the hand (figure 4.13). It is possible to interact with it in two ways:

- **Firing:** when the joystick trigger is pressed, a laser pointer will come out of the joystick. By pressing the trigger all the way down, the user will interact with what he or she is aiming at;
- **Touch:** it is possible in vr to represent the joysticks as virtual hands. In this mode, the tip of the index finger allows the user to interact with the elements by simply touching them.

## Scenes Switching

As already seen, the editor was therefore divided into two main scenes. In order to allow the user to jump at will from one to the other, it has been written a small code that uses the same communication protocol between the scenes already used for data transfer. As an example, in order to pass from the lines to the control room it is first necessary to deactivate virtual reality in the main scene (at port 63340) using the *setDisplayMode(0)* command, and then reactivate it in the other scene (port 63342), using the *setDisplayMode(4)* command.

Table 4.23: **Python code:** Switch between scenes

---

```

1 from threading import Thread
2
3 import requests
4
5
6 def http_request():
7     # ----- Exit VR in LINES --> Command used: setDisplayMode(0)
8     requests.get("http://localhost:63340/pythonapi?value=setDisplayMode%280%29")
9     # Sets displaymode = 0 (Standard Display) using VRED's menu in localhost at
    port 63340 (port of the editor part)
10
11     # ----- Open VR in CONTROL ROOM --> Command used: setDisplayMode(4);
12     requests.get("http://localhost:63342/pythonapi?value=setDisplayMode%284%29\
    nexecutePython('selectVariantSet(\"Center Room\")')")
13     # Sets displaymode = 4 (OpenVR HMD) using VRED's menu in localhost at port
    63342 (port of the waiting room part)
14
15
16 thread = Thread(
17     target=http_request
18 )
19
20 thread.daemon = True
21
22 thread.start()

```

---

## Machine Annotations Management

By framing the editor in a possible training area, virtual reality annotations have been inserted which appear above the machine if the user "shoots" at them. To do this, a small script has been implemented within a variant set, which has then been associated with a touch sensor, so the script is activated every time the user interacts with the geometry of a machine.

The labels are generated by the main editor, which also takes care of repositioning them so that they always point to the machines they describe.

Table 4.24: **Python code:** Toggle annotations

---

```
1 annotation_cent_pneum_L1 = findAnnotation("Accoppiatore Regolabile L0")
2
3 if annotation_cent_pneum_L1.isVisible():
4     annotation_cent_pneum_L1.setVisible(False)
5 else: # annotation_cent_pneum_L1.isVisible() == False
6     annotation_cent_pneum_L1.setVisible(True)
```

---

Other variant sets have been provided to close and open all the annotations at the same time. Below is the case of hiding of all annotations, the opening one is dual.

Table 4.25: **Python code:** Hide all annotations

---

```
1 list_annotations = getAnnotations()
2
3 for annotation in list_annotations:
4     if annotation.isVisible():
5         annotation.setVisible(False)
6     else:
7         pass
```

---

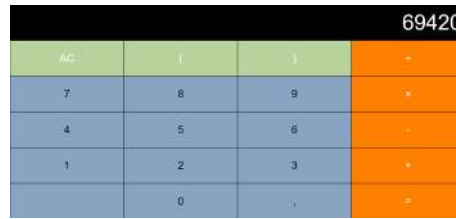


Figure 4.14: html Calculator

## Utilities Management

Still with the aim of exploring all the possibilities offered by virtual reality, some tools have been implemented that show pages that can be useful to the user. The tools considered are a calculator made in html (figure 4.14) and the web page of Sacmi's catalogue of automatic tiles sorting, packaging and palletizing lines [10].

To do this, the Vred *vrImmersiveUiService* class must be used: first, a menu is created (with a name), then the content is set, be it a file or a Url. Next, the menu must be positioned. First define an origin: in this case the middle point of the menu located in the left hand (can be opened with a joystick button). Then position it by specifying translation and rotation with respect to it, and finally set its size. As soon as it is created, it becomes invisible. It will become visible when a special Variant Set is activated.

Table 4.26: **Python code:** Create hidden VR menus

```

1 def create_hidden_vr_menus():
2     # ----- Calculator
3     hand_calculator = vrImmersiveUiService.createMenu("hand_calculator")
4     hand_calculator.setContent(path_string + "\Script.Editor.Multiple.Lines\
5     Examples\Calculator.html")
6     hand_calculator.setOrigin(vrdImmersiveMenu.ORIGIN_LEFTHAND)
7     hand_calculator.setTranslation(265, 150, -195)
8     hand_calculator.setRotation(180 + 45, 0, 0)
9     hand_calculator.setWidth(400)
10    hand_calculator.setDepth(5)
11    hand_calculator.setVisible(False)
12
13    # ----- Catalog
14    hand_catalog = vrImmersiveUiService.createMenu("hand_catalog")
15    hand_catalog.setContent("https://www.sacmi.it/it-it/ceramics/Piastrelle/Linee
16    -di-scelta,-confezionamento-e-pallettizzazione-automatiche-piastrelle")
17    hand_catalog.setOrigin(vrdImmersiveMenu.ORIGIN_LEFTHAND)
18    hand_catalog.setTranslation(280, 170, -230)
19    hand_catalog.setRotation(180 + 45, 0, 0)
20    hand_catalog.setVisible(False)
21    hand_catalog.setWidth(500)
22    hand_catalog.setDepth(5)

```

To open one of these tools, it is necessary to get information about which hand the menu is open, so that it appears on the correct hand. First, the main editor defines a boolean variable - *hand\_calculator\_opened* - initialized to *False*. This variable indicates whether the tool is open or not.

If the tool is open, launching the Variant Set will close it. If it was closed, the first thing to do is to check the origin of the interactive menu (i.e. which hand it will be open on), and then set the tool's origin at that point. Offset and orientation are set, and finally it is shown.

In the end, the value of the boolean variable is updated.

Table 4.27: **Python code:** Toggle calculator tool

---

```
1 hand_calculator = vrImmersiveUiService.findMenu("hand_calculator")
2
3 if hand_calculator_opened: # Variable initialized at False, declared in Script
4     Editor
5     hand_calculator.setVisible(False)
6     hand_calculator_opened = False
7 else: # hand calculator closed
8     main_vr_menu = vrImmersiveUiService.findMenu("ToolsMenu")
9     origin_main_vr_menu = main_vr_menu.getOrigin()
10    hand_calculator.setOrigin(origin_main_vr_menu)
11
12    if origin_main_vr_menu == vrKernelServices.vrImmersiveMenu.MenuOrigin.
13    ORIGIN_RIGHTHAND:
14        hand_calculator.setTranslation(-265, 150, -195)
15    else: # Main menu on LEFT HAND
16        hand_calculator.setTranslation(265, 150, -195)
17
18    hand_calculator.setRotation(180 + 45, 0, 0)
19    hand_calculator.setVisible(True)
20    hand_calculator_opened = True
```

---

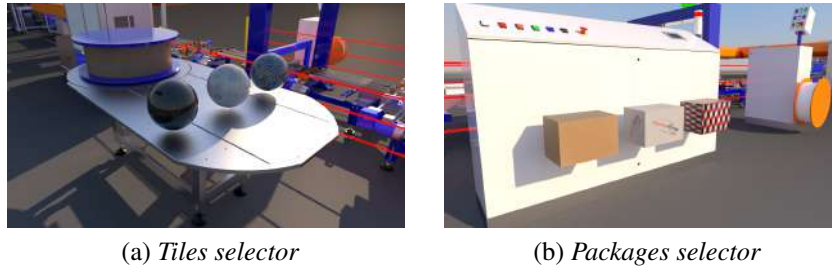


Figure 4.15: Material selectors

### Tiles / Package Selection

To entertain the user, selectors have been inserted in each line to display a decoration of choice (among three available per type) for tiles and packages. To do this, a *switch* material has been created, which allows a series of different materials to be inserted inside it, which can then be activated in an exclusive way (meaning that only one at a time can be visible).

The selectors have the shape of three spheres of the corresponding material for tiles (fig. 4.15a), and three boxes with the corresponding packaging for packages (fig. 4.15b).

A variant set has then been created for each material-line pair (i.e. 6 for each line, for a total of 18) which is in charge of activating the specific material when selected, invoking it from the *switch* material list. Each variant set has then been associated with a touch sensor matched to the selectors geometries. Therefore, at the interaction in the scene (shot or touch) with each of these balls/boxes, the corresponding material will be represented in the line.

### Scene Exploration

In order to move around the scene, Vred has two main methodologies. The first is also the simplest and most natural: the headset, thanks to its 6 cameras, is able to map the physical movements in the virtual reality scene by detecting the variations in the perspective of the elements of the room around the user; therefore, in simple terms, walking and moving physically is equivalent to doing so in the simulation.

Secondly, Vred also provides for the agile travel of large "virtual" distances by teleportation. At the contact of the thumb with the analog stick, a preview appears: a circle on the ground (or on the geometries present) with an arrow above it. Moving the hand moves the circle, rotating the wrist changes the direction indicated by the arrow. If the circle is green, when the stick is pressed the user will be teleported to the position of the circle, facing the direction of the arrow. If the circle is red, it means that the user cannot teleport to that point, and therefore nothing will happen when the analog stick is pressed.

In order to move more easily, however, special viewpoints have been created: they are fixed views created ad hoc to show the main points of the scenes. The main views (center of the balcony, beginning and end of the lines) when selected in virtual reality (through the menu section represented in figure 4.13c), effectively teleport the user to the precise point, allowing immediate navigation within the scene.

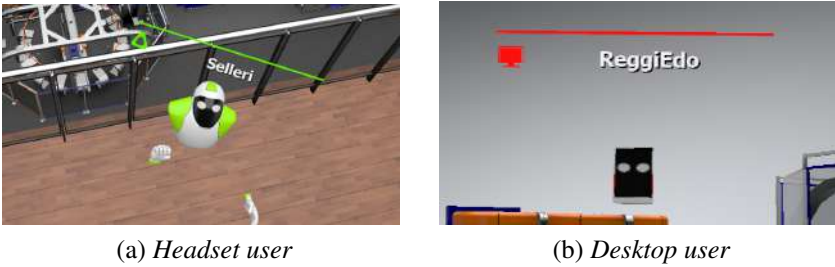


Figure 4.16: Virtual Reality User Avatars

### Machines Inspection

Among the tools provided by Vred, the flashlight is particularly useful: it is a real 3d model of a lamp, which remains attached to one of the two joysticks, and projects a beam of light. In itself it is an instrument that seems quite useless, but it has great potential. In fact, the geometries in Vred are not "solid": it is therefore possible to actually enter the machines, observing how they are made inside by illuminating everything with the torch. Obviously the outcome depends on the precision of the model represented, but Vred is able to represent an entire industrial line with a detail level that reaches up to the screw. This is an extremely important aspect as far as staff training is concerned: in fact, internal details of the machine can be seen which, in order to be observed in reality, would require almost complete disassembly of the parts.

### Multiuser Experience

During the last period of realization of this project, it has also been addressed, although in an embryonic way, the topic of multi-user experience, i.e. the entry of several people into the same scene in virtual reality. This type of shared experience is particularly useful in the areas of possible use of this editor (pre-selling, staff training, exhibition fairs), as it allows presentations and "guided tours", as well as real lessons concerning the lines and their layout and functioning.

In order to set up a shared scene, a collaboration must first be set [2]. To do this, a number of requirements must be met (tables 4.28, 4.29 and 4.30).

Table 4.28: **Collaboration:** Connection Requirements

<b>Network connection:</b>	To access the collaboration session and VRED file.
<b>Internet connection:</b>	To send/receive invites.

In this case, having had only one viewer available at the time of approaching this type of experience, tests were made on two users, one with the viewer, and one in desktop mode. Each user is represented by an avatar within the scene. While the one in desktop mode is seen simply as a tablet (figure 4.16b), the user with a viewer is represented as a robotic humanoid, with hands representing the joysticks (figure 4.16a). By pressing the keys of the joystick - in addition to activating the associated commands - the hands make gestures (close the fingers, raise the thumb).



Table 4.29: **Collaboration:** Hardware Requirements

<b>VRED system requirements:</b>	To guarantee that the machine meets the VRED requirements, as well as the additional VR specific requirements. This ensures that graphics card and GPU meet the demands of the content.
<b>VR Devices:</b>	Only a set of VR devices are supported.
<b>Audio:</b>	To minimize outside noise and potential disruptions, a noise-cancelling mic and headphones is recommended.

Table 4.30: **Collaboration:** Software Requirements

<b>Matching version of VRED:</b>	To participate, all participants must use the same version of VRED. This is defined by the first session participant.
<b>Calendar application:</b>	To send/receive meeting invites.
<b>Email application:</b>	To send/receive invites.
<b>Audio application:</b>	To communicate with others. The session leader will set this up using a service, such as Zoom, Skype, or Mumble.

Participants to the collaboration scenes can trigger things like Variant Sets and animation. When this happens, VRED uploads only these actions to participants, not the entire scene. It only syncs the changed information. This is why it's stressed that all participants use the same scene file during the collaboration session. This guarantees everyone sees the same thing.

Participants not using the same scene will experience issues with information syncing, resulting in errors and not seeing changes. There are limitations to the information synced. Though Variant Sets, animations, and transformations are synced, new geometry added to the scene is not. However, if participants upload the scene after changes have been made, even newly created geometry, variants, variant sets, and animation are synced.

## 4.3 Model Optimization

### 4.3.1 Performed Steps

At the end of the project, and after some testing, it was found that unfortunately the scene in virtual reality had obvious defects of fluidity: jumps, slow loading times, freeze frames. In general it seemed as if the headset was struggling to satisfy the computational load of all the polygons. This was especially the case when the user moved his head quickly: the viewer could not compute the geometries quickly enough, so that annoying black edges appeared at the edges of the user's field of view.

In the end, to avoid these inconveniences, a phase of optimization of the model was performed.

#### 1 - Hide Unnecessary Objects

As a first solution, it was decided to hide all the nodes considered unnecessary: the scene includes lines from fifty to a hundred meters long, so the lack of small elements of few millimeters (such as screws, nuts, bolts and washers) in positions hidden from the eye would be perfectly fine. However, assuming the case of a single line scene for staff training, this lightening would not be acceptable, as accuracy and fidelity to the original model would be of central importance.

After completing this action and carrying out some tests, it was found that this method is unfortunately ineffective. In fact, the headset in any case calculates the nodes, even if they are hidden. This is because they are still part of the scene, and could be made visible at any time. This first phase was therefore considered inconclusive for optimization purposes.

#### 2 - Delete Unnecessary Objects

Given the failure of the first solution, it was then thought to go one step further: the unnecessary nodes were not only hidden, but directly eliminated from the scene. In doing so, they no longer have to be computed by the headset, and therefore the scene turned out to be generally lighter. The editor file was 100 Mb lighter, with about 3 million less polygons for each line (for a total of 9 millions).

#### 3 - Delete Empty Subgroups

To try to further lighten the simulation, it was decided to act on the scene's tree structure (already introduced in section 3.1.2.2).

The actual parts, i.e. the geometries of the various elements, are grouped into several levels of subgroups (e.g. foot group → subgroup 1 → subgroup 2 → screw → actual geometry). Vred - and consequently the headset - must then calculate the positioning and relative translations of each of these elements: the position of each subgroup is indeed computed from the position of the group at the higher level. Therefore, in order to calculate the actual position of all geometries, Vred must first calculate a huge amount of translations and relative rotations.

In order to try to lighten the overall computational load - in an attempt to obtain better performance - a *transformation flush* was first carried out: this procedure resets the rotation value to zero by keeping the workpiece in position, in practice taking the position at the time of flush as a new zero position.

In any case, zero translations were still computed by the hardware. That's why it was decided to maintain a maximum of 4 levels of subgroups (e.g. Line0 → Transport before sorting → pneumatic centrer → foot group), removing those below (unless there were relative transformations different than 0). To do so, a small Python script was used, which selected all the nodes of the geometries (which in the model were named "Geometry", in fact), allowing to extract them and insert them in the highest level subgroup, and making it possible to remove empty groups by using the *Optimize Module*, through the *Remove empty group nodes* option.

After performing these actions, the tests showed an improvement in simulation behavior in virtual reality. Everything was smoother and more stable, even if with the maximum number of lines there was still some fatigue, when the user suddenly framed the lines. In any case, loading times have been greatly reduced. In addition, about 4000 - 5000 empty groups per line have been eliminated.

#### **4 - Geometries Conversion**

The 3D models used for the realization of this application were taken directly from CAD files derived from the designs. For this reason, the imported geometries are NURBS type, which are in fact used by most 3D CAD.

NURBS stands for Non-Uniform Rational B-Splines, a convenient algorithmic method for building curves and free-form surfaces [15]. They are generated from portions of curves whose polynomial coefficients depend on some points known as "control" points, therefore similar to vector images.

The advantage of having curves of this type, is mainly due to the fact that they are invariant with respect to the graphic transformation operations of these points. This means that if transformations of the curve are to be generated, it is sufficient to apply transformations to the control points only, without this changes the geometry of the original curve. The use of non-constant parameterized curves is the optimal solution to ensure the creation of all possible shapes.

Vred, however, prefers the use of Mesh type curves. Meshes (also called polygonal mesh surfaces) are used for the three-dimensional representation of complex reliefs, such as those of statues or terrains. Geometrically they are composed of a set of vertices joined together by many small flat surfaces, usually triangular in shape, one adjacent to the other. The description of a surface using a mesh polygonal is usually an approximation process. The degree of approximation of the shape can be managed by the number of individual mesh subelements. The higher the number of sub-elements the more detailed the surface will be.

If a comparison between these two types of surfaces is to be made, it can be said that NURBS, from a mathematical point of view, are very complex and require a high amount

of computer resources to manage them in the best way. At the same time, however, as advantages they have that they are able to represent optimally any form and that they are made up of a very limited number of elements. Mesh surfaces however, thanks to their geometric characteristics, are more easily manageable by different 3D modeling software, thus allowing maximum ductility of shapes and ease of transformation. However, they always implement a simplification of the geometry of the object of study.

It was therefore decided to convert all the geometries present from NURBS to mesh, to try to lighten the overall computational load for the headset during navigation in virtual reality. To do this, it was necessary to once again use the script to select the geometrical elements, and then use a dedicated command provided by Vred.

### **4.3.2 Final Results**

At the end of all these steps, the situation has improved significantly (especially thanks to the final conversion of geometries). The result was a generalized smoothing: the problems manifested at the beginning have become practically absent. The only time when - in the final version - the simulation shows some uncertainty is when all three lines are present, and when the user frames them all at once and suddenly. However, it's just a momentary situation, which stabilizes immediately.

The meshes also turned out to be lighter from the point of view of file size: the memory space occupied by the line scene went from 1.2 Gb - before all the optimization steps presented - to only 825 Mb, effectively showing a decrease of about 33% in the memory occupied.

# Conclusions

This project was born first and foremost as an exercise in style as it was extremely innovative - at least as far as the use of virtual reality within Sacmi S.C. was concerned - and there were no previous approaches.

This is why the application has been expanded more and more during the course of the project: in fact, it has evolved from a single line to several lines, and then a whole series of different interfaces for interaction with the scene have been developed.

In a first phase in which the focus was mainly on the perfect execution of the lines by the editor - involving parameterization and manipulation of the geometries - a set of Graphic User Interfaces on-screen to be interacted with via mouse and keyboard have been realized through the use of dedicated programs.

But once a stable version of the code was reached, it was decided to aim at a complete experience in virtual reality: tools for the insertion of information and instruments for navigation, exploration and interaction with geometries were then designed directly within the scene.

Methodologies and protocols for communication between different scenes have been studied, for the passage of data and commands to "remotely" control scenes external to the one explored by the user, and finally automatic methodologies for the creation of outputs have been studied and implemented.

It is thanks to these last aspects that one of the main objectives that had been set during the project definition phase has been reached: the capability of effectively integrating an application of this kind into the corporate workflow of a company like Sacmi.

This is why, in the perspective of incorporating the developed editor in the phase of quote realization, the utopian concept of inserting the required data only once in the whole production process is - at least in this phase - fully satisfied: a potential customer can in fact configure the most suitable line to his/her needs within the simulation in virtual reality and trigger the confirmation of the final choice, and the output data - in this case the desired parameters - can be sent in the form of Excel spreadsheet directly to the following production phases, which can automatically read them directly from the file, whenever necessary.

So what are the advantages brought by virtual reality compared to the more usual and consolidated methodologies in this field?

First of all, the real power of this technology is undoubtedly - as already mentioned - the immersion it can provide: just think about the fact that the editor allows you to see, explore, and in fact "touch" an industrial plant that in reality does not yet exist at the time

of its usage: it is like effectively making a leap into a potential future. So that's the way to overcome - if not completely, at least partially - the old conception of quotes based only on two-dimensional planimetries: no effort of imagination is needed, the future is already here, within reach of joysticks.

Always referring to the possible use in this particular phase of production, another huge advantage of the editor is undoubtedly its speed: it is able to create a single line of minimum size in just 30 seconds, while for the case limit of 3 lines of maximum size it takes about 2 minutes. So within this short period of time, the user can already explore a complete and animated scene.

It is also necessary to mention the extreme versatility and portability of a solution like the one presented in this project. In fact, as already discussed, the headset does not need fixed stations to function, since it is based on the perceived variations in the perspective of the objects within the room in which it is used, and secondly, it can be easily adapted to wireless use, eliminating the hassle of the cable connecting the headset to the computer. This means that, potentially, it can be used anywhere: all that is needed is a laptop with appropriate specifications, and a briefcase with which to transport the headset and the joysticks, and that's it. In this way, the editor could be taken directly to the customer, anywhere in the world.

Moreover, during the realization of this project was also introduced the multiuser experience, i.e. the co-presence in the same scene of several users, who can interact with it and among each other, thanks to the use of specific humanoid avatars. This connection can be established through the web, and so potentially people located thousands of kilometers from each other could gather in the same scene. Just consider the countless uses to which this solution can be adapted: trade shows, employee training, and many more.

Another point in favour of this application is undoubtedly the high reusability of the geometries offered by Vred. First of all, Vred is able to accept as input an impressive number of different CAD files, undoubtedly every main type of model used in the design phase. Therefore, the editor can be easily expanded and adapted to any type of line, and immediately inserted into any company workflow: it is not necessary to create ad hoc models to make it work, but it easily adapts to existing files.

Finally, Vred allows to modify the geometries in an easy and intuitive way: the parametrized models elaborated during the editor's operation can be exported, possibly converted, and reused later by other design phases, without the need for them to be created from scratch.

Finally, it was considered what the possible future developments of this particular application could be.

First of all, as it is logical to expect, a further extension of the editor: adding more and more parts to the considered line, until eventually representing an entire system. After that, it would be possible to increase more and more interactivity with and for the user, with the design and implementation of new tools. Finally, the performance of the simulation in virtual reality could be optimized more and more: every year Autodesk releases a new version of Vred, more stable and with added functionality. Features that could be leveraged to increasingly improve an experience of this kind.

So, to sum up, the realization of this project has really brought to the surface what are the countless possibilities of an instrument like virtual reality. This world is evolving at a vertiginous pace, and the possibilities and advantages that offers in every field, industrial and non-industrial, are now absolutely undeniable. Now more than ever it's clear that the immersive reality instruments - i.e. virtual and augmented reality - are the future.





# **Appendices**



# Appendix A

## EkoSort Building Code

---

```
1 # ----- EDITOR FUNCTIONS -----
2
3 # ----- B
4
5 def build_EkoSort():
6     reset_EkoSort()
7
8     h = 1150 # (h >= 1100 + tile thickness + tolerance) AND (h <= 1200)
9     r = d / 2
10
11     load_RotaryCircle(r, h)
12
13     load_Movers(n, r, h)
14
15     load_Stations(n, r)
16
17     load_Conveyor(r)
18
19     load_RotarySupport(r)
20
21     load_Machinery(r)
22
23     load_Windows(r)
24
25
26 # ----- L
27
28 def load_Conveyor(radius):
29     scaling_z_conveyor = 1
30     scaling_x_conveyor = radius * 0.0007 # same as rotary circle
31     scaling_y_conveyor = scaling_x_conveyor
32
33     conveyor = findNodePath("Your_Line/Your_EkoSort/ConveyorComplete", True)
34     conveyor.setWorldTranslation(0, radius, 0)
35     setTransformNodeScale(conveyor, scaling_x_conveyor, scaling_y_conveyor,
36                             scaling_z_conveyor) # scale X, Y, Z dimensions
37
38 def load_Machinery(radius):
39     global scaling_y_machinery
40
41     if radius <= 1500:
```

```
42     scaling_y_machinery = radius * 0.000645
43     if radius > 1500 and radius <= 1750:
44         scaling_y_machinery = radius * 0.00062
45     else:
46         scaling_y_machinery = radius * 0.000606
47
48     scaling_z_machinery = 1
49
50     mach = findNodePath("Your_Line/Your_EkoSort/Machinery/Machinery_mov", True)
51     setTransformNodeScale(mach, 1, scaling_y_machinery, scaling_z_machinery)
52
53
54 def load_Movers(nMovers, radius, height):
55     global limit_movers
56     limit_movers = nMovers
57
58     # select the mover by using path in Scenegraph
59     starting_mover = findNodePath("Your_Line/Your_EkoSort/AllMovers/Mover0", True)
60     # translate the mover
61     starting_mover.setWorldTranslation(0, radius, height)
62     # sets Mover0's rotation pivot to world center
63     setTransformNodeRotatePivot(starting_mover, 0, 0, 0, True)
64
65     for i in range(1, limit_movers):
66         # this is the syntax to DUPLICATE. The duplicate is saved as Mover(i)
        # automatically
67         mov_clone_temp = cloneNode(starting_mover, True)
68
69         # sets the duplicate's rotation pivot to world center
70         setTransformNodeRotatePivot(mov_clone_temp, 0, 0, 0, True)
71         # rotate the duplicate
72         setTransformNodeRotation(mov_clone_temp, 0, 0, i * 360 / nMovers)
73
74     cloned_movers_list_temp = []
75
76     for i in range(1, limit_movers): # fill the list
77         a = findNodePath("Your_Line/Your_EkoSort/AllMovers/Mover" + str(i), True)
78         cloned_movers_list_temp.append(a)
79
80     selectNodes(cloned_movers_list_temp)
81     groupSelection() # group all movers
82
83     rename_temp_movers = findNodePath("Your_Line/Your_EkoSort/AllMovers"
84                                         "/Grouped_Nodes1", True)
85     rename_temp_movers.setName("ClonedMovers")
86
87
88 def load_RotaryCircle(radius, height):
89     scaling_x_circle = (radius * 0.0007) / 0.7
90     scaling_y_circle = scaling_x_circle
91     scaling_z_circle = 0.7
92
93     circle = findNodePath("Your_Line/Your_EkoSort/RotaryCircle", True)
94     # translate RotaryCircle
95     circle.setWorldTranslation(0, 0, height + 150)
96     setTransformNodeScale(circle, scaling_x_circle, scaling_y_circle,
97                             scaling_z_circle) # scale RotaryCircle
```

```

98
99 def load_RotarySupport(radius):
100     scaling_y_rotary_support = (radius * 0.0007) / 0.525 # same as rotary circle
101     scaling_x_rotary_support = scaling_y_rotary_support
102     scaling_z_rotary_support = 1 / 0.525
103
104     support = findNodePath("Your_Line/Your_EkoSort/RotarySupport", True)
105     setTransformNodeScale(support, scaling_x_rotary_support,
106                           scaling_y_rotary_support, scaling_z_rotary_support)
107
108 def load_Stations(nMovers, radius):
109     scaling_x_stat = radius * 0.0007 # same as RotaryCircle
110     scaling_y_stat = scaling_x_stat
111     scaling_z_stat = 1
112
113     global n_stat, limit_stat, starting_value_stat
114     n_stat = nMovers - 1 # default value (total number of stations present)
115     limit_stat = nMovers # used to limit the for cycle
116     starting_value_stat = 1 # default value
117
118     if nMovers <= 6: # in this case skip the positioning of 1 station (just
119         delete Station0)
120         starting_value_stat = 1
121         limit_stat = nMovers
122         n_stat = nMovers - 1
123     else: # in this case skip the positioning of 3 stations
124         starting_value_stat = 2
125         limit_stat = nMovers - 1
126         n_stat = nMovers - 3
127
128     starting_station = findNodePath("Your_Line/Your_EkoSort/AllStations/Station0",
129                                     True) # select the station by using path in Scenegraph
130     starting_station.setWorldTranslation(0, radius, 0) # translate the mover to
131     initial position
132
133     if scaling_x_stat <= 1:
134         setTransformNodeScale(starting_station, scaling_x_stat, scaling_y_stat,
135                               scaling_z_stat) # scale along X, Y, Z
136     else:
137         setTransformNodeScale(starting_station, 1, 1, 1) # scale along X, Y, Z
138
139     for i in range(starting_value_stat, limit_stat):
140         # this is the syntax to DUPLICATE. The duplicate is named starting from
141         Mover1
142         stat_clone_temp = cloneNode(starting_station, True)
143
144         # set the duplicate's rotation pivot to world center
145         setTransformNodeRotatePivot(stat_clone_temp, 0, 0, 0, True)
146         # rotate the duplicate
147         setTransformNodeRotation(stat_clone_temp, 0, 0, i * 360 / nMovers)
148
149         if nMovers > 6:
150             # if i have to skip more than 1 station, i have to adjust names to
151             make the code work
152             stat_clone_temp.setName("Station" + str(i))
153
154     # hide Station0 (without deleting it), since it is clipping in the conveyor

```

```
149     hideNode(starting_station)
150
151     cloned_stations_list_temp = []
152
153     for i in range(starting_value.stat, limit_stat): # fill the list
154         a = findNodePath("Your_Line/Your_EkoSort/AllStations/Station"
155                         + str(i), True)
156         cloned_stations_list_temp.append(a)
157
158     selectNodes(cloned_stations_list_temp)
159     groupSelection() # group all cloned stations
160
161     rename_temp = findNodePath("Your_Line/Your_EkoSort/AllStations/"
162                               "Grouped_Nodes1", True)
163     rename_temp.setName("ClonedStations")
164
165
166 def load_Windows(radius):
167     scaling_y_wind = radius * 0.0007
168     scaling_x_wind = scaling_y_wind
169     scaling_z_wind = 1
170
171     wind = findNodePath("Your_Line/Your_EkoSort/Windows", True)
172     setTransformNodeScale(wind, scaling_x_wind, scaling_y_wind, scaling_z_wind)
173
174 # ----- R
175
176 def reset_EkoSort():
177     cloned_movers = findNodePath("Your_Line/Your_EkoSort/AllMovers/"
178                                 "ClonedMovers", True)
179     cloned_stations = findNodePath("Your_Line/Your_EkoSort/AllStations/"
180                                   "ClonedStations", True)
181
182     trash_clone = [cloned_movers, cloned_stations]
183     deleteNodes(trash_clone, False)
184
185     removeAllMeasurements()
186
187     # select the station by using path in Scenegraph
188     starting_station = findNodePath("Your_Line/Your_EkoSort/AllStations/"
189                                    "Station0", True)
190     showNode(starting_station) # show again Station0
191
192
193 # ----- MAIN -----
194
195 build_EkoSort()
```

---

# Appendix B

## ExtraPack Building Code

---

```
1 # ----- EDITOR FUNCTIONS -----
2
3 # ----- B
4
5 def build_ExtraPack(palletDistance, three_per_arc):
6     reset_ExtraPack()
7
8     r = d / 2
9     link_after_wrapping = findNodePath("Your_Line/ExtraPack_After_Wrapping/"
10                                         "Link_AfterWrapping", True)
11     coord_pack = findLinkCoordinates(link_after_wrapping)
12
13     global x_pack, y_pack
14     x_pack = coord_pack[0]
15     y_pack = coord_pack[1]
16
17     load_Transport(x_pack, y_pack, r)
18
19     build_Slider_Support(x_pack, y_pack, p, three_per_arc)
20
21     load_Slider(x_pack, y_pack)
22
23     if not three_per_arc: # 2 pallet stations per arc
24         load_PalletStations_2perArc(y_pack, p, palletDistance)
25     else: # 3 pallet stations per arc
26         load_PalletStations_3perArc(y_pack, p, palletDistance)
27
28
29 def build_Slider_Support(x_pack, y_pack, nPallets, three_per_arc):
30     global nArcs
31
32     # N.B. In nArcs I must insert an additional + 1, in order to store pallet
33     # buffers!
34     if not three_per_arc: # 2 pallet stations per arc
35         if nPallets % 2 == 0: # even number of pallet stations
36             nArcs = int(nPallets / 2) + 1
37         else: # odd number of pallet stations
38             nArcs = 1 + int(nPallets / 2) + 1
39             # in this way, nArcs is the ceiling of nPallets/2 (n of arcs needed in
40             # the support) + 1 ADDITIONAL
41     else: # 3 pallet stations per arc
42         if nPallets % 3 == 0: # number of pallet stations is a multiple of 3
```

```

41         nArcs = int(nPallets / 3) + 1
42     else:
43         nArcs = 1 + int(nPallets / 3) + 1
44
45     global limit_arc_cent, coord_last_arc
46     limit_arc_cent = nArcs - 2
47     coord_last_arc = [0, 0]
48
49     load_FirstArc(x_pack, y_pack) # load first arc
50
51     if limit_arc_cent != 0: # if there is need for central arc(s)
52         link_first_arc = findNodePath("Your_Line/Your_ExtraPack/SliderSupport/"
53                                     "CampataIniziale/Link.PrimaCampata", True)
54         # save coordinates given by link sphere in first arc
55         coord_second_arc = findLinkCoordinates(link_first_arc)
56         x_second_arc = coord_second_arc[0]
57         y_second_arc = coord_second_arc[1]
58
59         load_MiddleArc(x_second_arc, y_second_arc)
60
61         # the first center arc has already been positioned. If only that one is
62         # required (limit_arc_cent == 1), the code won't enter in the next "for" cycle
63         for i in range(1, limit_arc_cent):
64             first_middle_arc = findNodePath("Your_Line/Your_ExtraPack/Slider"
65                                             "Support/CampataCentrale_"
66                                             "NoUltimiPiloni0", True)
67             # clone the first middle arc, named starting from "
68             CampataCentrale_NoUltimiPiloni1"
69             mid_arc_clone_temp = cloneNode(first_middle_arc, True)
70
71             link_previous_arc = findNodePath("Your_Line/Your_ExtraPack/"
72                                             "SliderSupport/Campata"
73                                             "Centrale_NoUltimiPiloni" + str(i - 1) +
74                                             "/Link_CampataCentrale_NoUltimiPiloni",
75                                             True)
76             # get coordinates of previous arc's link sphere
77             coord_arc_temp = findLinkCoordinates(link_previous_arc)
78             x_arc_temp = coord_arc_temp[0]
79             y_arc_temp = coord_arc_temp[1]
80
81             # translate current arc in link sphere's position
82             mid_arc_clone_temp.setWorldTranslation(x_arc_temp, y_arc_temp, 0)
83
84             link_second_last_arc = findNodePath("Your_Line/Your_ExtraPack/Slider"
85                                                 "Support/CampataCentrale_NoUltimi"
86                                                 "Piloni" + str(limit_arc_cent - 1) +
87                                                 "/Link_CampataCentrale_NoUltimiPiloni")
88             # get coordinates of last middle arc's link sphere
89             coord_last_arc = findLinkCoordinates(link_second_last_arc)
90
91     else: # if there is NO need for central arc(s)
92         link_second_last_arc = findNodePath("Your_Line/Your_ExtraPack/Slider"
93                                             "Support/CampataIniziale/Link.Prima"
94                                             "Campata", True)
95         # obtain coordinates of previous arc's link sphere
96         coord_last_arc = findLinkCoordinates(link_second_last_arc)
97
98     node_camp_cent = findNodePath("Your_Line/Your_ExtraPack/SliderSupport/"

```



```

97         "CampataCentrale_NoUltimiPiloni0", True)
98     hideNode(node_camp_cent) # hide middle arc
99
100     x_last_arc = coord_last_arc[0]
101     y_last_arc = coord_last_arc[1]
102
103     loadLastArc(x_last_arc, y_last_arc)
104
105     if nArcs > 3: # in case I had to clone middle arc
106         all_slider_supp_cloned_nodes = []
107         for i in range(1, limit_arc_cent):
108             a = findNodePath("Your_Line/Your_ExtraPack/SliderSupport/"
109                             "CampataCentrale_NoUltimiPiloni" + str(i), True)
110             all_slider_supp_cloned_nodes.append(a)
111
112         selectNodes(all_slider_supp_cloned_nodes)
113         groupSelection() # group all cloned arcs
114         rename_temp_movers = findNodePath("Your_Line/Your_ExtraPack/SliderSupport"
115                                           "/Grouped_Nodes1", True)
116         rename_temp_movers.setName("Clones_CampataCentrale")
117
118         temp_list_to_hide = []
119         for i in range(1, limit_arc_cent):
120             temp_linknode_to_hide = findNodePath("Your_Line/Your_ExtraPack/"
121                                                  "SliderSupport/Clones_Campata"
122                                                  "Centrale/CampataCentrale_No"
123                                                  "UltimiPiloni" + str(i) +
124                                                  "/Link_CampataCentrale_"
125                                                  "NoUltimiPiloni", True)
126             temp_list_to_hide.append(temp_linknode_to_hide)
127
128         hideNodes(temp_list_to_hide)
129
130
131 # ----- F
132
133 def findLinkCoordinates(linkNode):
134     bb = linkNode.getBoundingBox()
135
136     # doing X_max - X_min of the bounding box gives the X direction's length
137     bb_x_length = bb[3] - bb[0]
138     # doing Y_max - Y_min of the bounding box gives the Y direction's length
139     bb_y_length = bb[4] - bb[1]
140     # doing Z_max - Z_min of the bounding box gives the Z direction's length
141     bb_z_length = bb[5] - bb[2]
142
143     x_link = round(bb[0] + (bb_x_length / 2), 3)
144     y_link = round(bb[1] + (bb_y_length / 2), 3)
145     z_link = round(bb[2] + (bb_z_length / 2), 3)
146
147     coord_link = [x_link, y_link, z_link]
148     return coord_link
149
150
151 # ----- L
152 def loadFirstArc(x_pack, y_pack):
153     slider_support = findNodePath("Your_Line/Your_ExtraPack/SliderSupport", True)
154     slider_support.setWorldTranslation(x_pack, y_pack, 0)

```

```

155
156     firstarc = findNodePath("Your_Line/Your_ExtraPack/SliderSupport/"
157                             "CampataIniziale", True)
158     firstarc.setLocalTranslation(0, 0, 0)
159     setTransformNodeScale(firstarc, 1, 1, 1)
160
161
162 def load_LastArc(x_last_arc, y_last_arc):
163     last_arc = findNodePath("Your_Line/Your_ExtraPack/SliderSupport/"
164                             "CampataFinale", True)
165     last_arc.setWorldTranslation(x_last_arc, y_last_arc, 0)
166     setTransformNodeScale(last_arc, 1, 1, 1)
167
168
169 def load_MiddleArc(x_mid_arc, y_mid_arc):
170     mid_arc = findNodePath("Your_Line/Your_ExtraPack/SliderSupport/"
171                             "CampataCentrale_NoUltimiPiloni0", True)
172     mid_arc.setWorldTranslation(x_mid_arc, y_mid_arc, 0)
173     setTransformNodeScale(mid_arc, 1, 1, 1)
174
175
176 def load_PalletStations_2perArc(y_pack, nPallets, palletDistance):
177     all_pallets_nodes = [] # list of all clones, to be grouped after
178     hw = 600 # half width of the pallet
179     y_offset = 1750
180
181     first_slider = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
182                                 "PalletStation0", True)
183
184     setTransformNodeScale(first_slider, 1, 1, 1) # scaling is kept constant
185
186     prima_campata = findNodePath("Your_Line/Your_ExtraPack/SliderSupport"
187                                 "/CampataIniziale", True)
188     coord_prima_campata = findLinkCoordinates(prima_campata)
189
190     campata_finale = findNodePath("Your_Line/Your_ExtraPack/SliderSupport"
191                                 "/CampataFinale", True)
192     coord_campata_finale = findLinkCoordinates(campata_finale)
193
194     for i in range(0, nArcs):
195         if i == 0: # FIRST ARC
196             first_slider.setWorldTranslation(coord_prima_campata[0] - (
197                 palletDistance / 2) - hw, y_pack + y_offset, 0)
198             pallet_clone_temp = cloneNode(first_slider, True) # clone pallet0
199             pallet_clone_temp.setWorldTranslation(coord_prima_campata[0] + (
200                 palletDistance / 2) + hw, y_pack + y_offset, 0)
201             all_pallets_nodes.append(pallet_clone_temp)
202         elif i == nArcs - 1: # FINAL ARC
203             pallet_buffer = findNodePath("Your_Line/Your_ExtraPack/"
204                                         "AllPallets/Pallets_Buffer",
205                                         True)
206             pallet_buffer.setWorldTranslation(coord_campata_finale[0], y_pack +
207                 y_offset, 0)
208         elif nArcs > 2 and i == 1: # FIRST MIDDLE ARC
209             prima_campata_centrale = findNodePath("Your_Line/Your_ExtraPack/S"
210                                                     "liderSupport/CampataCentrale"
211                                                     "_NoUltimiPiloni0", True)
212             coord_prima_camp_cent = findLinkCoordinates(prima_campata_centrale)

```

```

210
211     pallet_clone_temp = cloneNode(first_slider, True) # clone pallet0
212     pallet_clone_temp.setWorldTranslation(coord_prima_camp_cent[0] - (
palletDistance / 2) - hw, y_pack + y_offset, 0)
213
214     pallet_clone_temp1 = cloneNode(first_slider, True) # clone pallet0
215     pallet_clone_temp1.setWorldTranslation(coord_prima_camp_cent[0] + (
palletDistance / 2) + hw, y_pack + y_offset, 0)
216
217     all_pallets_nodes.append(pallet_clone_temp)
218     all_pallets_nodes.append(pallet_clone_temp1)
219     else: # CLONES OF MIDDLE ARC
220         clone_campata_cent = findNodePath("Your_Line/Your_ExtraPack/"
221                                           "SliderSupport/Clones_Campata"
222                                           "Centrale/CampataCentrale-"
223                                           "NoUltimiPiloni" + str(i - 1),
224                                           True)
225         # (i - 1) must be used, since the first case with a clone of central
arc occurs at i == 2
226         coord_clone_camp_cent = findLinkCoordinates(clone_campata_cent)
227
228         pallet_clone_temp = cloneNode(first_slider, True) # clone pallet0
229         pallet_clone_temp.setWorldTranslation(coord_clone_camp_cent[0] - (
palletDistance / 2) - hw, y_pack + y_offset, 0)
230
231         pallet_clone_temp1 = cloneNode(first_slider, True) # clone pallet0
232         pallet_clone_temp1.setWorldTranslation(coord_clone_camp_cent[0] + (
palletDistance / 2) + hw, y_pack + y_offset, 0)
233
234         all_pallets_nodes.append(pallet_clone_temp)
235         all_pallets_nodes.append(pallet_clone_temp1)
236
237     selectNodes(all_pallets_nodes)
238     groupSelection() # group all stations
239
240     rename_temp_movers = findNodePath("Your_Line/Your_ExtraPack/AllPallets"
241                                       "/Grouped_Nodes1", True)
242     rename_temp_movers.setName("Clones_Pallet")
243
244     if nPallets % 2 == 0: # even number of pallet stations
245         pass
246     else: # odd number of pallet stations
247         pallet_to_delete = findNodePath("Your_Line/Your_ExtraPack/AllPallets"
248                                         "/Clones_Pallet/PalletStation" +
249                                         str(nPallets), True)
250         deleteNode(pallet_to_delete)
251
252
253 def load_PalletStations_3perArc(y_pack, nPallets, palletDistance):
254     all_pallets_nodes = [] # list of all clones, to be grouped after
255     w = 1200 # width of the pallet
256     y_offset = 1750
257
258     first_slider = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
259                                 "PalletStation0", True)
260
261     setTransformNodeScale(first_slider, 1, 1, 1) # scale on Y dimension
262

```

```

263     prima_campata = findNodePath("Your_Line/Your_ExtraPack/SliderSupport/"
264                                   "CampataIniziale", True)
265     coord_prima_campata = findLinkCoordinates(prima_campata)
266
267     campata_finale = findNodePath("Your_Line/Your_ExtraPack/SliderSupport/"
268                                   "CampataFinale", True)
269     coord_campata_finale = findLinkCoordinates(campata_finale)
270
271     for i in range(0, nArcs):
272         if i == 0: # FIRST ARC
273             first_slider.setWorldTranslation(coord_prima_campata[0] -
274 palletDistance - w, y_pack + y_offset, 0)
275
276             pallet_clone1_temp = cloneNode(first_slider, True) # clone pallet0
277             pallet_clone1_temp.setWorldTranslation(coord_prima_campata[0], y_pack +
278 y_offset, 0)
279
280             pallet_clone2_temp = cloneNode(first_slider, True) # clone pallet0
281             pallet_clone2_temp.setWorldTranslation(coord_prima_campata[0] +
282 palletDistance + w, y_pack + y_offset, 0)
283
284             all_pallets_nodes.append(pallet_clone1_temp)
285             all_pallets_nodes.append(pallet_clone2_temp)
286         elif i == nArcs - 1: # FINAL ARC
287             pallet_buffer = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
288                                           "Pallets_Buffer", True)
289             pallet_buffer.setWorldTranslation(coord_campata_finale[0], y_pack +
290 y_offset, 0)
291         elif nArcs > 2 and i == 1: # FIRST MIDDLE ARC
292             prima_campata_centrale = findNodePath("Your_Line/Your_ExtraPack/"
293                                                     "SliderSupport/CampataCentrale"
294                                                     "_NoUltimiPiloni0", True)
295             coord_prima_camp_cent = findLinkCoordinates(prima_campata_centrale)
296
297             pallet_clone0_temp = cloneNode(first_slider, True) # clone pallet0
298             pallet_clone0_temp.setWorldTranslation(coord_prima_camp_cent[0] -
299 palletDistance - w, y_pack + y_offset, 0)
300
301             pallet_clone1_temp = cloneNode(first_slider, True) # clone pallet0
302             pallet_clone1_temp.setWorldTranslation(coord_prima_camp_cent[0], y_pack
303 + y_offset, 0)
304
305             pallet_clone2_temp = cloneNode(first_slider, True) # clone pallet0
306             pallet_clone2_temp.setWorldTranslation(coord_prima_camp_cent[0] +
307 palletDistance + w, y_pack + y_offset, 0)
308
309             all_pallets_nodes.append(pallet_clone0_temp)
310             all_pallets_nodes.append(pallet_clone1_temp)
311             all_pallets_nodes.append(pallet_clone2_temp)
312         else: # CLONES OF MIDDLE ARC
313             clone_campata_cent = findNodePath("Your_Line/Your_ExtraPack/Slider"
314                                                 "Support/Clones_CampataCentrale/"
315                                                 "CampataCentrale_NoUltimiPiloni"
316                                                 + str(i - 1), True)
317
318             # (i - 1) must be used, since the first case with a clone of central
319 arc occurs at i == 2
320             coord_clone_camp_cent = findLinkCoordinates(clone_campata_cent)

```

```

313     pallet_clone0_temp = cloneNode(first_slider, True) # clone pallet0
314     pallet_clone0_temp.setWorldTranslation(coord_clone_camp_cent[0] -
palletDistance - w, y_pack + y_offset, 0)
315
316     pallet_clone1_temp = cloneNode(first_slider, True) # clone pallet0
317     pallet_clone1_temp.setWorldTranslation(coord_clone_camp_cent[0], y_pack
+ y_offset, 0)
318
319     pallet_clone2_temp = cloneNode(first_slider, True) # clone pallet0
320     pallet_clone2_temp.setWorldTranslation(coord_clone_camp_cent[0] +
palletDistance + w, y_pack + y_offset, 0)
321
322     all_pallets_nodes.append(pallet_clone0_temp)
323     all_pallets_nodes.append(pallet_clone1_temp)
324     all_pallets_nodes.append(pallet_clone2_temp)
325
326     selectNodes(all_pallets_nodes)
327     groupSelection() # group all stations
328
329     rename_temp_movers = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
330                                     "Grouped_Nodes1", True)
331     rename_temp_movers.setName("Clones_Pallet")
332
333     if nPallets % 3 == 0: # number of pallet stations is a multiple of 3
334         pass
335     elif nPallets % 3 == 2: # need to delete 1 excess stations
336         pallet_to_delete = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
337                                     "Clones_Pallet/PalletStation" +
338                                     str(nPallets), True)
339         deleteNode(pallet_to_delete)
340     else: # need to delete 2 excess stations
341         pallet_to_delete = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
342                                     "Clones_Pallet/PalletStation" +
343                                     str(nPallets), True)
344         pallet_to_delete_bis = findNodePath("Your_Line/Your_ExtraPack/"
345                                     "AllPallets/Clones_Pallet/"
346                                     "PalletStation" +
347                                     str(nPallets + 1), True)
348         deleteNode(pallet_to_delete)
349         deleteNode(pallet_to_delete_bis)
350
351
352     def load_Slider(x_pack, y_pack):
353         slider = findNodePath("Your_Line/Your_ExtraPack/SliderRobot", True)
354         setTransformNodeScale(slider, 1, 1, 1) # scale on Y direction
355         slider.setWorldTranslation(10000 + x_pack, y_pack, 0)
356
357
358     def load_Transport(x_pack, y_pack, radius):
359         scaling_y_transport = radius * 0.0007
360
361         transport = findNodePath("Your_Line/Your_ExtraPack/Trasporto", True)
362         transport.setWorldTranslation(x_pack, y_pack, 0)
363         setTransformNodeScale(transport, 1, scaling_y_transport, 1) # scale on Y
direction
364
365
366     # ----- R

```

```
367
368 def reset_ExtraPack():
369     old_camp_cent = findNodePath("Your_Line/Your_ExtraPack/SliderSupport/"
370                                   "Clones_CampataCentrale", True)
371     old_pallets = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
372                                 "Clones_Pallet", True)
373
374     delete = [old_camp_cent, old_pallets]
375     deleteNodes(delete, False)
376
377     node_camp_cent = findNodePath("Your_Line/Your_ExtraPack/SliderSupport/"
378                                   "CampataCentrale_NoUltimiPiloni0", True)
379     showNode(node_camp_cent) # show middle arc (in case it was hidden)
380
381     removeAllMeasurements()
382
383
384 # ----- MAIN -----
385
386 build_ExtraPack(palletDistance, three.per.arc)
```

---

# Appendix C

## Animation Code

---

```
1 # ----- ANIMATION FUNCTIONS -----
2
3 # ----- B
4
5 def build_AllMovers_Animation(x_grasp, y_grasp, z_grasp, z_tile_on_station, nMovers,
    nTiles, max_handling_height, frame_nearest_tile_arrived, frame_gap_btw_tiles):
6     # starting FRAME of mover movement (1 frame before grasping)
7     z_descent_frame0 = frame_nearest_tile_arrived - 1
8     gap_anim_mover = frame_gap_btw_tiles * nMovers
9     counterTiles = nTiles
10    counterMovers = nMovers
11    j = 0
12
13    if nTiles < 10:
14        k = 1
15    else:
16        k = 0
17
18    while counterTiles > 0 and counterMovers > 0:
19        frame_on_station1 = all_frames_mov_on_station[nTiles - 1 - j]
20
21        # from mover to mover, the animation is slipped of frame_gap_btw_tiles
22        z_tr_init_frame = z_descent_frame0 + frame_gap_btw_tiles * j
23        # starting TIME of the descent trajectory of mover i
24        z_tr_init_time = frameToSec(z_tr_init_frame)
25        frame_put_down1 = all_frames_tile_put_down[nTiles - 1 - j]
26
27        tr0 = Vec3f()
28        tr0.__init__(x_grasp, y_grasp, z_grasp)
29        # grasp tile on conveyor belt
30        tr1 = Vec3f()
31        tr1.__init__(x_grasp, y_grasp, z_grasp - max_handling_height)
32        # mover stands still for a frame after arriving on top of the station
33        tr2 = Vec3f()
34        tr2.__init__(x_grasp, y_grasp, z_grasp)
35        # release tile on station (first)
36        tr4 = Vec3f()
37        tr4.__init__(x_grasp, y_grasp, z_tile_on_station + 10 * k)
38
39        if j == 0:
40            mover_temp = findNodePath("Your_Line/Your_EkoSort/AllMovers/Mover0",
41                                     True)
```

```

42     else:
43         mover_temp = findNodePath("Your_Line/Your_EkoSort/AllMovers/"
44                                   "ClonedMovers/Mover" + str(j), True)
45
46         # create vectors of translation transform vectors (x, y, z), and
corresponding times to impose them
47         tr_vectors = [tr0, tr1, tr0, tr2, tr4, tr0]
48         tr_times = [z_tr_init_time, frameToSec(z_tr_init_frame + 1),
49                   frameToSec(z_tr_init_frame + 2), frameToSec(frame_on_station1),
50                   frameToSec(frame_put_down1 - 5), frameToSec(frame_put_down1),
51                   frameToSec(frame_put_down1 + 4)]
52
53         # create trajectory of the mover
54         addTranslationControlPoints(mover_temp, tr_times, tr_vectors, True)
55
56         counterTiles = counterTiles - 1
57         counterMovers = counterMovers - 1
58         j = j + 1
59         k = k + 1
60
61     # In case (nTiles = nMovers) or (nTiles < nMovers), the first while will
suffice: counterTiles will be 0 and no other animations will be needed.
However, if nTiles > nMovers, another loop will be needed in order to animate
the (nTiles - counterTiles) tiles "left behind" from the first cycle
62     if counterTiles == 0:
63         pass
64     else:
65         for i in range(0, counterTiles):
66             # the last j positions of the vector all_frames_mov_on_station have
already been used inside the first loop
67             frame_on_station2 = all_frames_mov_on_station[nTiles - 1 - i - j]
68
69             # the start of the second batch of trajectories is given by
z_descent_frame0 (start of first batch) plus the amount of frames used to
animate all of the first batch
70             z_tr_init_frame_second = z_descent_frame0 + gap_anim_mover +
frame_gap_btw_tiles * i
71             frame_put_down2 = all_frames_tile_put_down[nTiles - 1 - i - j]
72
73             tr0 = Vec3f()
74             tr0.__init__(x_grasp, y_grasp, z_grasp)
75             # grasp tile on conveyor belt
76             tr1 = Vec3f()
77             tr1.__init__(x_grasp, y_grasp, z_grasp - max_handling_height)
78             # mover stands still for a frame after arriving on top of the station
79             tr2 = Vec3f()
80             tr2.__init__(x_grasp, y_grasp, z_grasp)
81             # release tile on station (second)
82             tr5 = Vec3f()
83             tr5.__init__(x_grasp, y_grasp, z_tile_on_station + 10 * i + 10 * nMovers)
84
85             if i == 0:
86                 mover_temp = findNodePath("Your_Line/Your_EkoSort/AllMovers/"
87                                           "Mover0", True)
88             else:
89                 mover_temp = findNodePath("Your_Line/Your_EkoSort/AllMovers/"
90                                           "ClonedMovers/Mover" + str(i), True)
91

```



```

92     # create vectors of translation transform vectors (x, y, z), and
corresponding times to impose them
93     tr_vectors = [tr0, tr1, tr0, tr2, tr5, tr0]
94     tr_times = [frameToSec(z.tr_init_frame_second),
95                 frameToSec(z.tr_init_frame_second + 1),
96                 frameToSec(z.tr_init_frame_second + 2),
97                 frameToSec(frame_on_station2),
98                 frameToSec(frame_put_down2 - 5),
99                 frameToSec(frame_put_down2),
100                 frameToSec(frame_put_down2 + 4)]
101
102     # create trajectory of the mover
103     addTranslationControlPoints(mover_temp, tr_times, tr_vectors, True)
104
105     # Create animation block for every mover
106     for i in range(0, nMovers):
107         if i == 0:
108             mover_temp = findNodePath("Your_Line/Your_EkoSort/AllMovers/"
109                                       "Mover0", True)
110         else:
111             mover_temp = findNodePath("Your_Line/Your_EkoSort/AllMovers/"
112                                       "ClonedMovers/Mover" + str(i), True)
113         createAnimationBlockForNode(mover_temp, True)
114
115
116 def build_Machinery_Animation(z.rot.angle.package, frame_nearest_tile_arrived):
117     setCurrentFrame(0)
118
119     # ----- MACHINERY MOV
120
121     mach_mov = findNodePath("Your_Line/Your_EkoSort/Machinery/"
122                             "Machinery_mov", True)
123
124     # to have a good behavior, the pack must be taken from the station AT MOST at
42 frames
125     if frame_nearest_tile_arrived < 42:
126         key_frame_anim_mach = frame_nearest_tile_arrived
127     else:
128         key_frame_anim_mach = 42
129
130     if key_frame_anim_mach - 42 > 0:
131         frame_start_anim_mach = key_frame_anim_mach - 42
132     else:
133         frame_start_anim_mach = 0
134
135     # ROTATIONS
136     z.rot.angle_mach = 180 + z.rot.angle.package
137
138     rot0 = Vec3f()
139     rot0.__init__(0, 0, 0)
140     rot26 = Vec3f()
141     rot26.__init__(0, 0, z.rot.angle_mach)
142     rot72 = Vec3f()
143     rot72.__init__(0, 0, z.rot.angle_mach)
144     rot98 = Vec3f()
145     rot98.__init__(0, 0, 0)
146

```

```

147 # create vectors of rotation transform vectors (x, y, z), and corresponding
148 times to impose them
149 rot_vectors = [rot0, rot26, rot72, rot98]
150 rot_times = [frameToSec(frame_start_anim_mach),
151             frameToSec(key_frame_anim_mach - 16),
152             frameToSec(key_frame_anim_mach + 30),
153             frameToSec(key_frame_anim_mach + 56)]
154
155 # create final trajectories
156 addRotationControlPoints(mach_mov, rot_times, rot_vectors)
157 createAnimationBlockForNode(mach_mov, True)
158
159 # ----- CROSS
160
161 cross = findNodePath("Your_Line/Your_EkoSort/Machinery/Machinery_mov/Croci",
162                     True)
163 coord_cross = findLinkCoordinates(cross)
164 conv_bottom = findNodePath("Your_Line/Your_EkoSort/ConveyorComplete/"
165                             "Conveyor Bottom", True)
166 coord_conveyor_bottom = findLinkCoordinates(conv_bottom)
167
168 # TRANSLATIONS
169 # N.B: to properly use translations, i have to counterbalance the effect of
170 the SCALING of each part in the editor! To do so, i have to divide
171 translations by scaling_y_machinery
172
173 tr0 = Vec3f()
174 tr0.__init__(0, 0, 0)
175 tr26 = Vec3f()
176 tr26.__init__(0, 0, 0)
177 tr38 = Vec3f()
178 tr38.__init__(0, 0, 158 / 2)
179 tr42 = Vec3f()
180 tr42.__init__(0, 0, 208 / 2)
181 tr72 = Vec3f()
182 tr72.__init__(0, 1662.03, 208 / 2)
183 tr98 = Vec3f()
184 tr98.__init__(0, 1662.03, 40 / 2)
185 tr144 = Vec3f()
186 tr144.__init__(8.469, (coord_conveyor_bottom[1] - coord_cross[1]) /
187 scaling_y_machinery, 40 / 2)
188 tr156 = Vec3f()
189 tr156.__init__(8.469, (coord_conveyor_bottom[1] - coord_cross[1]) /
190 scaling_y_machinery, -65 / 2)
191 tr168 = Vec3f()
192 tr168.__init__(13.170, 1662.03, -50.877 / 2)
193 tr194 = Vec3f()
194 tr194.__init__(17.338, 0, 0)
195
196 # create vectors of translation transform vectors (x, y, z), and corresponding
197 times to impose them
198 tr_cross_vectors = [tr0, tr26, tr38, tr42, tr72, tr98, tr144, tr156, tr168,
199 tr194]
200 tr_cross_times = [frameToSec(frame_start_anim_mach),
201                 frameToSec(key_frame_anim_mach - 16),
202                 frameToSec(key_frame_anim_mach - 4),
203                 frameToSec(key_frame_anim_mach),
204                 frameToSec(key_frame_anim_mach + 30),
205                 frameToSec(key_frame_anim_mach + 56),

```

```

198         frameToSec(key_frame.anim_mach + 102),
199         frameToSec(key_frame.anim_mach + 114),
200         frameToSec(key_frame.anim_mach + 126),
201         frameToSec(key_frame.anim_mach + 152)]
202
203 # create final trajectories
204 addTranslationControlPoints(cross, tr_cross_times, tr_cross_vectors, True)
205 createAnimationBlockForNode(cross, True)
206
207 # ----- CROSS INTERNAL
208
209 cross_internal = findNodePath("Your_Line/Your_EkoSort/Machinery/Machinery_mov"
210                               "/Croci/Interne", True)
211
212 # ROTATIONS
213 rot_int0 = Vec3f()
214 rot_int0.__init__(0, 0, 0)
215 rot_int26 = Vec3f()
216 rot_int26.__init__(0, 0, 0)
217 rot_int38 = Vec3f()
218 rot_int38.__init__(-20, 0, 0)
219 rot_int42 = Vec3f()
220 rot_int42.__init__(-25, 0, 0)
221 rot_int72 = Vec3f()
222 rot_int72.__init__(-30, 0, 0)
223 rot_int98 = Vec3f()
224 rot_int98.__init__(-10, 0, 0)
225 rot_int144 = Vec3f()
226 rot_int144.__init__(-10, 0, 0)
227 rot_int156 = Vec3f()
228 rot_int156.__init__(5, 0, 0)
229 rot_int168 = Vec3f()
230 rot_int168.__init__(9, 0, 0)
231 rot_int194 = Vec3f()
232 rot_int194.__init__(0, 0, 0)
233
234 # create vectors of translation transform vectors (x, y, z), and corresponding
    times to impose them
235 rot_int_vectors = [rot_int0, rot_int26, rot_int38, rot_int42, rot_int72,
236                   rot_int98, rot_int144, rot_int156, rot_int168, rot_int194]
237 rot_int_times = [frameToSec(frame_start.anim_mach),
238                 frameToSec(key_frame.anim_mach - 16),
239                 frameToSec(key_frame.anim_mach - 4),
240                 frameToSec(key_frame.anim_mach),
241                 frameToSec(key_frame.anim_mach + 30),
242                 frameToSec(key_frame.anim_mach + 56),
243                 frameToSec(key_frame.anim_mach + 102),
244                 frameToSec(key_frame.anim_mach + 114),
245                 frameToSec(key_frame.anim_mach + 126),
246                 frameToSec(key_frame.anim_mach + 152)]
247
248 # create final trajectories
249 addRotationControlPoints(cross_internal, rot_int_times, rot_int_vectors)
250 createAnimationBlockForNode(cross_internal, True)
251
252 # ----- CROSS EXTERNAL
253

```

```

254     cross_external = findNodePath("Your_Line/Your_EkoSort/Machinery/Machinery_mov/
    Croci/Esterne", True)
255
256     # ROTATIONS
257     rot_ext0 = Vec3f()
258     rot_ext0.__init__(0, 0, 0)
259     rot_ext26 = Vec3f()
260     rot_ext26.__init__(0, 0, 0)
261     rot_ext38 = Vec3f()
262     rot_ext38.__init__(20, 0, 0)
263     rot_ext42 = Vec3f()
264     rot_ext42.__init__(25, 0, 0)
265     rot_ext72 = Vec3f()
266     rot_ext72.__init__(30, 0, 0)
267     rot_ext98 = Vec3f()
268     rot_ext98.__init__(10, 0, 0)
269     rot_ext144 = Vec3f()
270     rot_ext144.__init__(10, 0, 0)
271     rot_ext156 = Vec3f()
272     rot_ext156.__init__(-5, 0, 0)
273     rot_ext168 = Vec3f()
274     rot_ext168.__init__(-9, 0, 0)
275     rot_ext194 = Vec3f()
276     rot_ext194.__init__(0, 0, 0)
277
278     # create vectors of translation transform vectors (x, y, z), and corresponding
    times to impose them
279     rot_ext_vectors = [rot_ext0, rot_ext26, rot_ext38, rot_ext42, rot_ext72,
280                       rot_ext98, rot_ext144, rot_ext156, rot_ext168, rot_ext194]
281     rot_ext_times = [frameToSec(frame_start_anim_mach),
282                     frameToSec(key_frame_anim_mach - 16),
283                     frameToSec(key_frame_anim_mach - 4),
284                     frameToSec(key_frame_anim_mach),
285                     frameToSec(key_frame_anim_mach + 30),
286                     frameToSec(key_frame_anim_mach + 56),
287                     frameToSec(key_frame_anim_mach + 102),
288                     frameToSec(key_frame_anim_mach + 114),
289                     frameToSec(key_frame_anim_mach + 126),
290                     frameToSec(key_frame_anim_mach + 152)]
291
292     # create final trajectories
293     addRotationControlPoints(cross_external, rot_ext_times, rot_ext_vectors)
294     createAnimationBlockForNode(cross_external, True)
295
296     # ----- UP/DOWN MACH
297
298     updown_mach = findNodePath("Your_Line/Your_EkoSort/Machinery/Machinery_mov/"
299                               "up-down_mach", True)
300     coord_updown_mach = findLinkCoordinates(updown_mach)
301
302     # TRANSLATIONS
303     tr_ud0 = Vec3f()
304     tr_ud0.__init__(0, 0, 0)
305     tr_ud26 = Vec3f()
306     tr_ud26.__init__(0, 0, 0)
307     tr_ud38 = Vec3f()
308     tr_ud38.__init__(0, 0, 158)
309     tr_ud42 = Vec3f()

```

```

310 tr_ud42.__init__(0, 0, 208)
311 tr_ud72 = Vec3f()
312 tr_ud72.__init__(0, 1662.03, 208)
313 tr_ud98 = Vec3f()
314 tr_ud98.__init__(0, 1662.03, 40)
315 tr_ud144 = Vec3f()
316 tr_ud144.__init__(0, (coord.conveyor_bottom[1] - coord.updown_mach[1]) /
scaling.y_machinery, 40)
317 tr_ud156 = Vec3f()
318 tr_ud156.__init__(0, (coord.conveyor_bottom[1] - coord.updown_mach[1]) /
scaling.y_machinery, -65)
319 tr_ud168 = Vec3f()
320 tr_ud168.__init__(0, 1662.03, -50.877)
321 tr_ud194 = Vec3f()
322 tr_ud194.__init__(0, 0, 0)
323
324 # create vectors of translation transform vectors (x, y, z), and corresponding
times to impose them
325 tr_ud_vectors = [tr_ud0, tr_ud26, tr_ud38, tr_ud42, tr_ud72, tr_ud98,
tr_ud144, tr_ud156, tr_ud168, tr_ud194]
326
327 tr_ud_times = [frameToSec(frame_start_anim_mach),
frameToSec(key_frame_anim_mach - 16),
frameToSec(key_frame_anim_mach - 4),
frameToSec(key_frame_anim_mach),
frameToSec(key_frame_anim_mach + 30),
frameToSec(key_frame_anim_mach + 56),
frameToSec(key_frame_anim_mach + 102),
frameToSec(key_frame_anim_mach + 114),
frameToSec(key_frame_anim_mach + 126),
frameToSec(key_frame_anim_mach + 152)]
337
338 # create final trajectories
339 addTranslationControlPoints(updown_mach, tr_ud_times, tr_ud_vectors, True)
340 createAnimationBlockForNode(updown_mach, True)
341
342
343 def build_Package_Animation(nTiles, x_packaging_position,
conveyor_velocity_mmPerFrame, z_rot_angle, frame_nearest_tile_arrived):
344 # ----- MAIN PACKAGE
345
346 old_package_to_delete = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
"Package_First_Station", True)
347
348 deleteNode(old_package_to_delete)
349
350 # to have a good behavior, the pack must be taken from the station AT MOST at
42 frames
351 if frame_nearest_tile_arrived < 42:
352     key_frame_anim_mach = frame_nearest_tile_arrived
353 else:
354     key_frame_anim_mach = 42
355
356 if key_frame_anim_mach - 42 > 0:
357     frame_start_anim_mach = key_frame_anim_mach - 42
358 else:
359     frame_start_anim_mach = 0
360
361 frame_pack_ready = all_frames_tile_put_down[0]
362 # go to frame in which the pack is completed

```

```

363     setCurrentFrame(frame_pack_ready)
364     cloned_tiles_for_package_list = []
365
366     for i in range(0, nTiles):
367         tile_temp = findNodePath("Switch.Demo.Tiles/10.DemoTiles/"
368                                 "Piastrrella.Prova" + str(i), True)
369         clone_tile_temp = cloneNode(tile_temp, True) # clone each tile
370         # delete animations inherited from original tiles
371         trashAnimation(clone_tile_temp)
372         cloned_tiles_for_package_list.append(clone_tile_temp)
373
374     selectNodes(cloned_tiles_for_package_list)
375     groupSelection() # group all cloned tiles
376     rename_temp_package = findNodePath("Switch.Demo.Tiles/10.DemoTiles/"
377                                         "Grouped.Nodes1", True)
378     rename_temp_package.setName("Package.First.Station")
379     coord_package = findLinkCoordinates(rename_temp_package)
380     main_package = rename_temp_package
381
382     conv_bottom = findNodePath("Your.Line/Your.EkoSort/Conveyor.Complete/"
383                               "Conveyor.Bottom", True)
384     coord_conveyor_bottom = findLinkCoordinates(conv_bottom)
385
386     # set rotation pivot to package center
387     setTransformNodeRotatePivot(main_package, coord_package[0], coord_package[1],
388     coord_package[2], True)
389     setCurrentFrame(0) # go back to starting frame
390
391     # TRANSLATIONS
392     tr0 = Vec3f()
393     tr0.__init__(0, 0, 0)
394     tr38 = Vec3f()
395     tr38.__init__(0, 0, 7)
396     tr42 = Vec3f()
397     tr42.__init__(0, 0, 57)
398     tr72 = Vec3f()
399     tr72.__init__(-coord_package[0], -coord_package[1], 57)
400     tr98 = Vec3f()
401     tr98.__init__(-coord_package[0], -coord_package[1], -98)
402     tr144 = Vec3f()
403     tr144.__init__(-coord_package[0], coord_conveyor_bottom[1] - coord_package[1],
404     -98)
405     tr152 = Vec3f()
406     tr152.__init__(-coord_package[0], coord_conveyor_bottom[1] - coord_package[1],
407     -152)
408     tr_pack = Vec3f()
409     tr_pack.__init__(x_packaging_position, coord_conveyor_bottom[1] - coord_package
410     [1], -152)
411
412     # frame used to arrive at packaging point
413     frame_to_reach_packing = (x_packaging_position) / conveyor_velocity_mmPerFrame
414     frame_packing = round(key_frame_anim_mach + 110 + frame_to_reach_packing, 0)
415
416     # create vectors of translation transform vectors (x, y, z), and corresponding
417     times to impose them
418     tr_vectors = [tr0, tr0, tr38, tr42, tr72, tr98, tr144, tr152, tr_pack]
419     tr_times = [0, frameToSec(frame_start_anim_mach),
420                 frameToSec(key_frame_anim_mach - 4),

```

```

416         frameToSec(key_frame.anim_mach),
417         frameToSec(key_frame.anim_mach + 30),
418         frameToSec(key_frame.anim_mach + 56),
419         frameToSec(key_frame.anim_mach + 102),
420         frameToSec(key_frame.anim_mach + 110),
421         frameToSec(frame_packing)]
422
423     # ROTATIONS
424     rot0 = Vec3f()
425     rot0.__init__(0, 0, 0)
426     rot72 = Vec3f()
427     rot72.__init__(0, 0, 0)
428     rot98 = Vec3f()
429     rot98.__init__(0, 0, -180 - z.rot_angle)
430
431     # create vectors of rotation transform vectors (x, y, z), and corresponding
    times to impose them
432     rot_vectors = [rot0, rot72, rot98]
433     rot_times = [0, frameToSec(key_frame.anim_mach + 30),
434                 frameToSec(key_frame.anim_mach + 56)]
435
436     # VISIBILITY
437     vis_values = [1, 0]
438     vis_times = [0, frameToSec(frame_packing)]
439
440     # create final trajectories
441     addTranslationControlPoints(main_package, tr_times, tr_vectors, True)
442     addRotationControlPoints(main_package, rot_times, rot_vectors)
443     addVisibleControlPoints(main_package, vis_times, vis_values)
444     createAnimationBlockForNode(main_package, True)
445
446     # ----- PACKAGE PACKED
447
448     packed_package = findNodePath("Switch.Demo.Tiles/10_DemoTiles/"
449                                   "Package.Packed", True)
450
451     # TRANSLATIONS
452     tr_packed0 = Vec3f()
453     tr_packed0.__init__(x.packaging_position + coord.package[0],
454                        coord.conveyor_bottom[1], 616)
455
456     # Since i have a time horizon (given by time_horizon), i need to know how much
    space the pack has to travel from the packing frame to the end of animation.
    It obviously depends from the conveyor speed
456     x_end_of_run = x.packaging_position + 1472 + conveyor_velocity_mmPerFrame * (
    time_horizon - frame_packing)
457
458     tr_packed_final = Vec3f()
459     tr_packed_final.__init__(x_end_of_run, coord.conveyor_bottom[1], 616)
460
461     # create vectors of translation transform vectors (x, y, z), and corresponding
    times to impose them
462     tr_packed_vectors = [tr_packed0, tr_packed_final]
463     tr_packed_times = [frameToSec(frame_packing), frameToSec(time_horizon)]
464
465     # VISIBILITY
466     vis_packed_values = [0, 1]

```

```

467 vis_packed_times = [0, frameToSec(frame_packing)] # in frame_packing + 1, the
packed tiles become visible
468
469 # create final trajectories
470 addTranslationControlPoints(packed_package, tr_packed_times, tr_packed_vectors,
True)
471 addVisibleControlPoints(packed_package, vis_packed_times, vis_packed_values)
472 createAnimationBlockForNode(packed_package, True)
473
474 # ----- PACKAGE PACKED BIS
475
476 packed_package_bis = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
"Package_Packed_bis", True)
477
478
479 # TRANSLATIONS
480 tr_packed_bis0 = Vec3f()
481 tr_packed_bis0._init_(x_end_of_run, coord_conveyor_bottom[1], 616)
482
483 if conveyor_velocity <= 1.7: # conveyor velocity in m/s is a global var
484     pallet_package = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
"PalletStation0", True)
485 else:
486     x = int(round(1 + p / 2, 0))
487     pallet_package = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
"Clones_Pallet/PalletStation1", True)
488
489
490 coord_pallet = findLinkCoordinates(pallet_package)
491 # find the X and Y value of the center of first pallet station
492 x_pallet_station = coord_pallet[0]
493 y_pallet_station = coord_pallet[1]
494
495
496 tr_packed_bis_pallet = Vec3f()
497 tr_packed_bis_pallet._init_(x_pallet_station, coord_conveyor_bottom[1], 616)
498 tr_packed_bis_rest = Vec3f()
499 tr_packed_bis_rest._init_(x_pallet_station, coord_conveyor_bottom[1], 666)
500 tr_packed_bis276 = Vec3f()
501 tr_packed_bis276._init_(x_pallet_station, y_pallet_station, 666)
502 tr_packed_bis300 = Vec3f()
503 tr_packed_bis300._init_(x_pallet_station, y_pallet_station, 194)
504
505 # create vectors of translation transform vectors (x, y, z), and corresponding
times to impose them
506 tr_packed_bis_vectors = [tr_packed_bis0, tr_packed_bis_pallet,
tr_packed_bis_rest, tr_packed_bis276, tr_packed_bis300]
507
508
509 # frame in which pack reaches pallet
510 global frame_pallet
511 frame_pallet = round((x_pallet_station - x_end_of_run) /
conveyor_velocity_mmPerFrame, 0)
512 tr_packed_bis_times = [0, frameToSec(frame_pallet),
frameToSec(frame_pallet + 15),
frameToSec(frame_pallet + 27),
frameToSec(frame_pallet + 51)]
513
514
515
516
517 # ROTATIONS
518 rot_packed_bis276 = Vec3f()
519 rot_packed_bis276._init_(0, 0, 0)
520

```



```

521     rot_packed.bis290 = Vec3f()
522     rot_packed.bis290.__init__(0, 0, -90)
523
524     # create vectors of rotation transform vectors (x, y, z), and corresponding
525     # times to impose them
526     rot_packed.bis_vectors = [rot_packed.bis276, rot_packed.bis290]
527     rot_packed.bis_times = [frameToSec(frame_pallet + 27),
528                             frameToSec(frame_pallet + 41)]
529
530     # create final trajectories
531     addTranslationControlPoints(packed_package_bis, tr_packed.bis.times,
532                                tr_packed.bis_vectors, True)
533     addRotationControlPoints(packed_package_bis, rot_packed.bis.times,
534                              rot_packed.bis_vectors)
535     createAnimationBlockForNode(packed_package_bis, True)
536
537 def buildRotaryCircleAnimation(nMovers, nTiles, frame_gap_btw_tiles,
538                               step_rotation):
539     # need a float to calculate a rotation that is NOT an integer
540     fl_nMovers = float(nMovers)
541     z_rot_angle_rotary = round(float(-360 / fl_nMovers), 1)
542     rotation_init_frame = all_frames_grasp_position[nTiles - 1] + 4
543     rotary_circle = findNodePath("Your_Line/Your_EkoSort/RotaryCircle", True)
544     rot_vectors = []
545     rot_times = []
546
547     rot0 = Vec3f()
548     rot0.__init__(0, 0, 0)
549     rot_vectors.append(rot0)
550     rot_times.append(0)
551     rot_bis = Vec3f()
552     rot_bis.__init__(0, 0, 0)
553
554     rot_vectors.append(rot_bis)
555     rot_times.append(frameToSec(rotation_init_frame))
556
557     for i in range(1, nTiles + 1):
558         if i == nTiles:
559             # In this case, i have finished the list all_frames_grasp_position, so
560             # i would take a wrong value. I have to take the value of the previous cycle and
561             # then add the frame gap between tiles animation
562             frame_grasp_position = all_frames_grasp_position[nTiles - 1 - (i - 1)]
563             \
564                 + 4 + frame_gap_btw_tiles
565         else:
566             frame_grasp_position = all_frames_grasp_position[nTiles - 1 - i] + 4
567             # start from the second-to-last one
568
569         if step_rotation == 1:
570             frame_on_station = all_frames_mov_on_station[nTiles - i] # -1
571             frame_next_movement = frame_on_station
572             frame_last_movement = frame_next_movement
573         elif step_rotation == 2:
574             frame_on_station = all_frames_tile_on_miss_stat_1[nTiles - i]
575             frame_next_movement = all_frames_mov_on_station[nTiles - i]
576             frame_last_movement = frame_next_movement
577         else: # step_rotation == 3

```

```

571         frame.on_station = all_frames.tile_on_miss_stat.1[nTiles - i]
572         frame.next_movement = all_frames.tile_on_miss_stat.2[nTiles - i]
573         frame.last_movement = all_frames.mov_on_station[nTiles - i]
574
575         rot1 = Vec3f()
576         rot1.__init__(0, 0, z_rot.angle_rotary * i)
577         rot_vectors.append(rot1)
578         rot_times.append(frameToSec(frame.on_station))
579         rot2 = Vec3f()
580         rot2.__init__(0, 0, z_rot.angle_rotary * i)
581         rot_vectors.append(rot2)
582         rot_times.append(frameToSec(frame.grasp_position))
583
584         if i == nTiles and step_rotation != 1:
585             rot3 = Vec3f()
586             rot3.__init__(0, 0, z_rot.angle_rotary * (i + 1))
587             rot_vectors.append(rot3)
588             rot_times.append(frameToSec(frame.next_movement))
589             rot4 = Vec3f()
590             rot4.__init__(0, 0, z_rot.angle_rotary * (i + 1))
591             rot_vectors.append(rot4)
592             rot_times.append(frameToSec(frame.grasp_position +
593                                     frame_gap btw.tiles))
594
595             if step_rotation == 3:
596                 rot5 = Vec3f()
597                 rot5.__init__(0, 0, z_rot.angle_rotary * (i + 2))
598                 rot_vectors.append(rot5)
599                 rot_times.append(frameToSec(frame.last_movement))
600                 rot6 = Vec3f()
601                 rot6.__init__(0, 0, z_rot.angle_rotary * (i + 2))
602                 rot_vectors.append(rot6)
603                 rot_times.append(frameToSec(frame.grasp_position + (
604                     frame_gap btw.tiles * 2)))
605
606         # create trajectory of RotaryCircle
607         addRotationControlPoints(rotary_circle, rot_times, rot_vectors)
608         createAnimationBlockForNode(rotary_circle, True)
609
610         # same rotation trajectory must be followed by the group AllMovers
611         all_mov = findNodePath("Your_Line/Your_EkoSort/AllMovers", True)
612         addRotationControlPoints(all_mov, rot_times, rot_vectors)
613         createAnimationBlockForNode(all_mov, True)
614
615     def build_Slider_Animation(x_pack, y_pack):
616         x.start_slider = x_pack + 10000
617
618         if conveyor_velocity <= 1.7: # conveyor velocity in m/s is a global var
619             pallet_package = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
620                                     "PalletStation0", True)
621         else:
622             pallet_package = findNodePath("Your_Line/Your_ExtraPack/AllPallets/"
623                                     "Clones_Pallet/PalletStation1", True)
624
625         coord_pallet = findLinkCoordinates(pallet_package)
626         x.pallet_station = coord_pallet[0]
627         y.pallet_station = coord_pallet[1]

```

```

628
629 # frame_pallet is a global variable, already calculated when animating
    packages. Every frame of slider movements MUST be referred to frame_pallet!
630
631 # ----- SLIDER MAIN
632
633 slider = findNodePath("Your_Line/Your_ExtraPack/SliderRobot", True)
634
635 # TRANSLATIONS
636 tr0 = Vec3f()
637 tr0.__init__(x.start_slider, y_pack, 0)
638 tr170 = Vec3f()
639 tr170.__init__(x.start_slider, y_pack, 0)
640 tr220 = Vec3f()
641 tr220.__init__(x.pallet_station - 842, y_pack, 0) # 842 is the displacement of
    the center of grasp w.r.t. the rotation pivot of the slider
642 tr312 = Vec3f()
643 tr312.__init__(x.pallet_station - 842, y_pack, 0)
644 tr336 = Vec3f()
645 tr336.__init__(x.start_slider, y_pack, 0)
646
647 # create vectors of translation transform vectors (x, y, z), and corresponding
    times to impose them
648 tr_vectors = [tr0, tr170, tr220, tr312, tr336]
649 tr_times = [0, frameToSec(frame_pallet - 79), frameToSec(frame_pallet - 29),
    frameToSec(frame_pallet + 63), frameToSec(frame_pallet + 87)]
650
651
652 # create final trajectories
653 addTranslationControlPoints(slider, tr_times, tr_vectors, True)
654 createAnimationBlockForNode(slider, True)
655
656 # ----- UP/DOWN
657
658 setCurrentFrame(0) # go to starting animation frame
659 up_down = findNodePath("Your_Line/Your_ExtraPack/SliderRobot/up-down", True)
660 coord_up_down = findLinkCoordinates(up_down)
661
662 # TRANSLATIONS
663 tr_ud0 = Vec3f()
664 tr_ud0.__init__(0, 0, 0)
665 tr_ud170 = Vec3f()
666 tr_ud170.__init__(0, 0, 0)
667 tr_ud220 = Vec3f()
668 tr_ud220.__init__(0, -460, 0)
669 tr_ud264 = Vec3f()
670 tr_ud264.__init__(0, -460, 0)
671 tr_ud276 = Vec3f()
672 tr_ud276.__init__(0, y_pallet_station - coord_up_down[1], 0)
673 tr_ud312 = Vec3f()
674 tr_ud312.__init__(0, y_pallet_station - coord_up_down[1], 0)
675 tr_ud336 = Vec3f()
676 tr_ud336.__init__(0, 0, 0)
677
678 # create vectors of translation transform vectors (x, y, z), and corresponding
    times to impose them
679 tr_ud_vectors = [tr_ud0, tr_ud170, tr_ud220, tr_ud264, tr_ud276, tr_ud312,
    tr_ud336]
680
681 tr_ud_times = [0, frameToSec(frame_pallet - 79),

```

```

682         frameToSec(frame_pallet - 29),
683         frameToSec(frame_pallet + 15), frameToSec(frame_pallet + 27),
684         frameToSec(frame_pallet + 63), frameToSec(frame_pallet + 87)]
685
686     # create final trajectories
687     addTranslationControlPoints(up_down, tr_ud_times, tr_ud_vectors, True)
688     createAnimationBlockForNode(up_down, True)
689
690     # ----- GRASP
691
692     grasp = findNodePath("Your_Line/Your_ExtraPack/SliderRobot/up-down/"
693                          "COLONNA/Grasp", True)
694
695     # TRANSLATIONS
696     tr_gr0 = Vec3f()
697     tr_gr0.__init__(0, 0, 0)
698     tr_gr170 = Vec3f()
699     tr_gr170.__init__(0, 0, 0)
700     tr_gr220 = Vec3f()
701     tr_gr220.__init__(0, 0, -370)
702     tr_gr234 = Vec3f()
703     tr_gr234.__init__(0, 0, -958)
704     tr_gr249 = Vec3f()
705     tr_gr249.__init__(0, 0, -1010)
706     tr_gr276 = Vec3f()
707     tr_gr276.__init__(0, 0, -959)
708     tr_gr300 = Vec3f()
709     tr_gr300.__init__(0, 0, -1432)
710     tr_gr312 = Vec3f()
711     tr_gr312.__init__(0, 0, -500)
712     tr_gr336 = Vec3f()
713     tr_gr336.__init__(0, 0, 0)
714
715     # create vectors of translation transform vectors (x, y, z), and corresponding
716     # times to impose them
717     tr_gr_vectors = [tr_gr0, tr_gr170, tr_gr220, tr_gr234, tr_gr249, tr_gr276,
718                     tr_gr300, tr_gr312, tr_gr336]
719     tr_gr_times = [0, frameToSec(frame_pallet - 79),
720                   frameToSec(frame_pallet - 29),
721                   frameToSec(frame_pallet - 15), frameToSec(frame_pallet),
722                   frameToSec(frame_pallet + 27), frameToSec(frame_pallet + 51),
723                   frameToSec(frame_pallet + 63), frameToSec(frame_pallet + 87)]
724
725     # create final trajectories
726     addTranslationControlPoints(grasp, tr_gr_times, tr_gr_vectors, True)
727     createAnimationBlockForNode(grasp, True)
728
729     # ----- ROTATOR
730
731     rotator = findNodePath("Your_Line/Your_ExtraPack/SliderRobot/up-down/"
732                            "COLONNA/Grasp/Rotator", True)
733
734     # ROTATIONS
735     rot0 = Vec3f()
736     rot0.__init__(0, 0, 0)
737     rot170 = Vec3f()
738     rot170.__init__(0, 0, 0)
739     rot220 = Vec3f()

```

```

739     rot220.__init__(0, 0, 0)
740     rot276 = Vec3f()
741     rot276.__init__(0, 0, 0)
742     rot290 = Vec3f()
743     rot290.__init__(0, 0, -90)
744     rot312 = Vec3f()
745     rot312.__init__(0, 0, -90)
746     rot336 = Vec3f()
747     rot336.__init__(0, 0, 0)
748
749     # create vectors of rotation transform vectors (x, y, z), and corresponding
750     # times to impose them
751     rot_vectors = [rot0, rot170, rot220, rot276, rot290, rot312, rot336]
752     rot_times = [0, frameToSec(frame_pallet - 79), frameToSec(frame_pallet - 29),
753                 frameToSec(frame_pallet + 27), frameToSec(frame_pallet + 41),
754                 frameToSec(frame_pallet + 63), frameToSec(frame_pallet + 87)]
755
756     # create final trajectories
757     addRotationControlPoints(rotator, rot_times, rot_vectors)
758     createAnimationBlockForNode(rotator, True)
759
760     # ----- CARRELLO SX
761
762     carr_sx = findNodePath("Your_Line/Your_ExtraPack/SliderRobot/up-down/COLONNA"
763                           "/Grasp/Rotator/P42000.0051.1/CARRELLO_SX", True)
764     carr_sx_2 = findNodePath("Your_Line/Your_ExtraPack/SliderRobot/up-down/"
765                              "COLONNA/Grasp/Rotator/P42000.0051.1/"
766                              "CARRELLO_SX.CILINDRO", True)
767
768     # TRANSLATIONS
769     tr_sx0 = Vec3f()
770     tr_sx0.__init__(0, 0, 0)
771     tr_sx234 = Vec3f()
772     tr_sx234.__init__(0, 0, 0)
773     tr_sx249 = Vec3f()
774     tr_sx249.__init__(0, 145, 0)
775     tr_sx300 = Vec3f()
776     tr_sx300.__init__(0, 145, 0)
777     tr_sx312 = Vec3f()
778     tr_sx312.__init__(0, 0, 0)
779
780     # create vectors of translation transform vectors (x, y, z), and corresponding
781     # times to impose them
782     tr_sx_vectors = [tr_sx0, tr_sx234, tr_sx249, tr_sx300, tr_sx312]
783     tr_sx_times = [0, frameToSec(frame_pallet - 15), frameToSec(frame_pallet),
784                   frameToSec(frame_pallet + 51), frameToSec(frame_pallet + 63)]
785
786     # create final trajectories
787     addTranslationControlPoints(carr_sx, tr_sx_times, tr_sx_vectors, True)
788     addTranslationControlPoints(carr_sx_2, tr_sx_times, tr_sx_vectors, True)
789     createAnimationBlockForNode(carr_sx, True)
790     createAnimationBlockForNode(carr_sx_2, True)
791
792     # ----- CARRELLO DX
793
794     carr_dx = findNodePath("Your_Line/Your_ExtraPack/SliderRobot/up-down/COLONNA"
795                           "/Grasp/Rotator/P42000.0051.1/CARRELLO_DX", True)
796     carr_dx_2 = findNodePath("Your_Line/Your_ExtraPack/SliderRobot/up-down/"

```

```

795         "COLONNA/Grasp/Rotator/P42000.0051.1/"
796         "CARRELLO_DX_CILINDRO", True)
797
798     # TRANSLATIONS
799     tr_dx0 = Vec3f()
800     tr_dx0.__init__(0, 0, 0)
801     tr_dx234 = Vec3f()
802     tr_dx234.__init__(0, 0, 0)
803     tr_dx249 = Vec3f()
804     tr_dx249.__init__(0, -152, 0)
805     tr_dx300 = Vec3f()
806     tr_dx300.__init__(0, -152, 0)
807     tr_dx312 = Vec3f()
808     tr_dx312.__init__(0, 0, 0)
809
810     # create vectors of translation transform vectors (x, y, z), and corresponding
      times to impose them
811     tr_dx_vectors = [tr_dx0, tr_dx234, tr_dx249, tr_dx300, tr_dx312]
812     tr_dx_times = [0, frameToSec(frame_pallet - 15), frameToSec(frame_pallet),
813                   frameToSec(frame_pallet + 51), frameToSec(frame_pallet + 63)]
814
815     # create final trajectories
816     addTranslationControlPoints(carr_dx, tr_dx_times, tr_dx_vectors, True)
817     addTranslationControlPoints(carr_dx_2, tr_dx_times, tr_dx_vectors, True)
818     createAnimationBlockForNode(carr_dx, True)
819     createAnimationBlockForNode(carr_dx_2, True)
820
821     # ----- CINGHIA LEFT-RIGHT
822
823     cinghia = findNodePath("Your_Line/Your_ExtraPack/SliderRobot/left-right/"
824                           "Cinghia Left-Right", True)
825
826     # TRANSLATIONS
827     tr_cin0 = Vec3f()
828     tr_cin0.__init__(0, 0, 0)
829     tr_cin_m79 = Vec3f()
830     tr_cin_m79.__init__(0, 0, 0)
831     tr_cin_m29 = Vec3f()
832     tr_cin_m29.__init__(0, -565, 0)
833     tr_cin15 = Vec3f()
834     tr_cin15.__init__(0, -565, 0)
835     tr_cin24 = Vec3f()
836     tr_cin24.__init__(0, 806, 0)
837     tr_cin66 = Vec3f()
838     tr_cin66.__init__(0, 806, 0)
839     tr_cin87 = Vec3f()
840     tr_cin87.__init__(0, 0, 0)
841
842     # create vectors of translation transform vectors (x, y, z), and corresponding
      times to impose them
843     tr_cin_vectors = [tr_cin0, tr_cin_m79, tr_cin_m29, tr_cin15, tr_cin24,
844                      tr_cin66, tr_cin87]
845     tr_cin_times = [0, frameToSec(frame_pallet - 79),
846                    frameToSec(frame_pallet - 29),
847                    frameToSec(frame_pallet + 15), frameToSec(frame_pallet + 24),
848                    frameToSec(frame_pallet + 66), frameToSec(frame_pallet + 87)]
849
850     # SCALING

```

```

851     sc_cin0 = Vec3f()
852     sc_cin0.__init__(1, 1, 1)
853     sc_cin24 = Vec3f()
854     sc_cin24.__init__(1, 1, 1)
855     sc_cin27 = Vec3f()
856     sc_cin27.__init__(1, 0.7, 1)
857     sc_cin63 = Vec3f()
858     sc_cin63.__init__(1, 0.7, 1)
859     sc_cin66 = Vec3f()
860     sc_cin66.__init__(1, 1, 1)
861
862     # create vectors of scaling transform vectors (x, y, z), and corresponding
863     # times to impose them
864     sc_cin_vectors = [sc_cin0, sc_cin24, sc_cin27, sc_cin63, sc_cin66]
865     sc_cin_times = [0, frameToSec(frame_pallet + 24),
866                    frameToSec(frame_pallet + 27), frameToSec(frame_pallet + 63),
867                    frameToSec(frame_pallet + 66)]
868
869     # create final trajectories
870     addTranslationControlPoints(cinghia, tr_cin_times, tr_cin_vectors, True)
871     addScaleControlPoints(cinghia, sc_cin_times, sc_cin_vectors)
872     createAnimationBlockForNode(cinghia, True)
873
874 def buildTilesAnimation(x_spawn, y_spawn, z_spawn, step, nMovers, nTiles,
875                        z_tile_on_station, max_handling_height, conveyor_vel_mmToFrame, frame_gap_anim,
876                        step_rotation):
877     fl_nMovers = float(nMovers)
878     z_rot_angle = round(float(-360 / fl_nMovers), 1)
879     # Z coordinate of tile 0 when put on the station above the others. It is set
880     # in this way in order to parametrize the next "for" cycle
881     z_last_tile_on_station = z_tile_on_station + 90
882     # frame in which Tile0 is taken by its mover (frame in which he reaches x = 0)
883     # . N.B: the minus is there because x_spawn is NEGATIVE
884     grasp_frame0 = round(-x_spawn / conveyor_vel_mmToFrame, 0)
885
886     global all_frames_mov_on_station, all_frames_tile_on_miss_stat1, \
887            all_frames_tile_on_miss_stat2, all_frames_grasp_position, \
888            all_frames_tile_put_down
889     # vector of frames in which movers arrive on designed station, in descending
890     # order. N.B: goes from 0 to (nTiles - 1)
891     all_frames_mov_on_station = []
892     # vector of frames with tiles on station 1 (missing), descending order.
893     all_frames_tile_on_miss_stat1 = []
894     # vector of frames with tiles on station 2 (missing), descending order.
895     all_frames_tile_on_miss_stat2 = []
896     # vector of frames in which tiles are being grasped (mover down), descending
897     # order.
898     all_frames_grasp_position = []
899     # vector of frames in which tiles are put down on station, descending order.
900     all_frames_tile_put_down = []
901
902     for i in range(0, nTiles):
903         # ----- MAIN TILES
904         tile_temp = findNodePath("Switch.Demo.Tiles/10_DemoTiles/"
905                                "Piastrrella_Prova" + str(i), True)
906         # set right rotation pivot (X axis = world's X axis)
907         setTransformNodeRotatePivot(tile_temp, 0, -y_spawn, 0, False)

```

```

902
903     # TRANSLATIONS
904     tr0 = Vec3f()
905     tr0.__init__(x_spawn + step * i, y_spawn, z_spawn)
906
907     global x_closest_tile
908     if i == 9:
909         x_closest_tile = x_spawn + step * i
910
911     tr424 = Vec3f()
912     tr424.__init__(0, y_spawn, z_spawn)
913     tr425 = Vec3f()
914     tr425.__init__(0, y_spawn, z_spawn + max_handling_height)
915     tr442 = Vec3f()
916     tr442.__init__(0, y_spawn, z_spawn + max_handling_height)
917     tr453 = Vec3f()
918     tr453.__init__(0, y_spawn, z_last_tile_on_station - 10 * i)
919
920     # create vectors of translation transform vectors (x, y, z)
921     tr_vectors = [tr0, tr424, tr425, tr442, tr453]
922
923     # N.B: tile descent on station is 5 frames long
924     if step_rotation == 1:
925         frame_tile_put_down = grasp_frame0 + 20 - frame_gap_anim * i + 5
926     elif step_rotation == 2:
927         frame_tile_put_down = grasp_frame0 - frame_gap_anim * (i - 1) + 25
928     else: # step_rotation == 3
929         frame_tile_put_down = grasp_frame0 - frame_gap_anim * (i - 2) + 25
930
931     tr_times = [0, frameToSec(grasp_frame0 - frame_gap_anim * i),
932                frameToSec(grasp_frame0 + 1 - frame_gap_anim * i),
933                frameToSec(frame_tile_put_down - 5),
934                frameToSec(frame_tile_put_down)]
935
936     # create final translation trajectories of the tile
937     addTranslationControlPoints(tile_temp, tr_times, tr_vectors, True)
938
939     if step_rotation == 1:
940         # ROTATIONS
941         rot0 = Vec3f()
942         rot0.__init__(0, 0, 0)
943         rot428 = Vec3f()
944         rot428.__init__(0, 0, 0)
945         rot442 = Vec3f()
946         rot442.__init__(0, 0, z_rot.angle * step_rotation)
947
948         # create vectors of rotation transform vectors (x, y, z)
949         rot_vectors = [rot0, rot428, rot442]
950
951         frame_mover_rot_on_station = grasp_frame0 + 19 - frame_gap_anim * i
952         all_frames_mov_on_station.append(frame_mover_rot_on_station)
953         all_frames_grasp_position.append(grasp_frame0 - frame_gap_anim * i)
954         all_frames_tile_put_down.append(frame_tile_put_down)
955         rot_times = [0, frameToSec(frame_mover_rot_on_station - 15),
956                    frameToSec(frame_mover_rot_on_station)]
957
958     elif step_rotation == 2:
959         # ROTATIONS

```



```

960     rot0 = Vec3f()
961     rot0.__init__(0, 0, 0)
962     rot1 = Vec3f()
963     rot1.__init__(0, 0, 0)
964     rot2 = Vec3f()
965     rot2.__init__(0, 0, z_rot.angle)
966     rot3 = Vec3f()
967     rot3.__init__(0, 0, z_rot.angle)
968     rot4 = Vec3f()
969     rot4.__init__(0, 0, z_rot.angle * step.rotation)
970
971     # create vectors of rotation transform vectors (x, y, z)
972     rot_vectors = [rot0, rot1, rot2, rot3, rot4]
973     frame_grasp_position = grasp_frame0 - frame_gap_anim * i
974     all_frames_grasp_position.append(frame_grasp_position)
975     frame_tile_rot_on_missing_station_1 = frame_grasp_position + 19
976     all_frames_tile_on_miss_stat_1.append(frame_tile_rot_on_missing_station_1
)
977     frame_mover_rot_on_station = grasp_frame0 - frame_gap_anim * (i - 1) +
19
978     # obtained counting 19 frames (4 grasping, 15 rotation) from grasping
of NEXT TILE
979     all_frames_mov_on_station.append(frame_mover_rot_on_station)
980     all_frames_tile_put_down.append(frame_tile_put_down)
981     rot_times = [0, frameToSec(frame_grasp_position + 4),
982                 frameToSec(frame_tile_rot_on_missing_station_1),
983                 frameToSec(grasp_frame0 - frame_gap_anim * (i - 1) + 4),
984                 frameToSec(frame_mover_rot_on_station)]
985
986     else: # step_rotation == 3
987         # ROTATIONS
988         rot0 = Vec3f()
989         rot0.__init__(0, 0, 0)
990         rot1 = Vec3f()
991         rot1.__init__(0, 0, 0)
992         rot2 = Vec3f()
993         rot2.__init__(0, 0, z_rot.angle)
994         rot3 = Vec3f()
995         rot3.__init__(0, 0, z_rot.angle)
996         rot4 = Vec3f()
997         rot4.__init__(0, 0, z_rot.angle * 2)
998         rot5 = Vec3f()
999         rot5.__init__(0, 0, z_rot.angle * 2)
1000        rot6 = Vec3f()
1001        rot6.__init__(0, 0, z_rot.angle * step.rotation)
1002
1003        # create vectors of rotation transform vectors (x, y, z)
1004        rot_vectors = [rot0, rot1, rot2, rot3, rot4, rot5, rot6]
1005        frame_grasp_position = grasp_frame0 - frame_gap_anim * i
1006        all_frames_grasp_position.append(frame_grasp_position)
1007        frame_tile_rot_on_missing_station_1 = frame_grasp_position + 19
1008        all_frames_tile_on_miss_stat_1.append(frame_tile_rot_on_missing_station_1
)
1009        frame_tile_rot_on_missing_station_2 = grasp_frame0 - frame_gap_anim * (i
- 1) + 19
1010        all_frames_tile_on_miss_stat_2.append(frame_tile_rot_on_missing_station_2
)

```

```

1011         frame_mover_rot_on_station = grasp_frame0 - frame_gap_anim * (i - 2) +
19
1012     all_frames_mov_on_station.append(frame_mover_rot_on_station)
1013     all_frames_tile_put_down.append(frame_tile_put_down)
1014     rot_times = [0, frameToSec(frame_grasp_position + 4),
1015                 frameToSec(frame_tile_rot_on_missing_station1),
1016                 frameToSec(grasp_frame0 - frame_gap_anim * (i - 1) + 4),
1017                 frameToSec(frame_tile_rot_on_missing_station2),
1018                 frameToSec(grasp_frame0 - frame_gap_anim * (i - 2) + 4),
1019                 frameToSec(frame_mover_rot_on_station)]
1020
1021     # create final rotation trajectories of the tile
1022     addRotationControlPoints(tile_temp, rot_times, rot_vectors)
1023     createAnimationBlockForNode(tile_temp, True)
1024
1025
1026     global time_horizon
1027     time_horizon = all_frames_tile_put_down[0] + 24
1028
1029     for i in range(0, nTiles):
1030         # ----- "BIS" TILES - used to give a continuity illusion
1031         if i == 0:
1032             pass # Tile0 doesn't need a "bis" one
1033         else:
1034             tile_bis_temp = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
1035                                         "Piastrrella_Prova" + str(i) + "_bis",
1036                                         True)
1037             # total frames of animation for tile.i.bis
1038             frame_anim_i_bis = step * i / conveyor_vel_mmToFrame
1039             # set right rotation pivot (Y axis = world's Y axis)
1040             setTransformNodeRotatePivot(tile_bis_temp, 0, -y_spawn, 0, False)
1041
1042             # TRANSLATIONS
1043             tr_bis0 = Vec3f()
1044             tr_bis0.__init__(x_spawn, y_spawn, z_spawn)
1045             tr_bis431 = Vec3f()
1046             tr_bis431.__init__(x_spawn, y_spawn, z_spawn)
1047             # at the end of the animation loop, the "bis" tile must be in the same
1048             place where the original one is at time 0
1049             tr_bis457 = Vec3f()
1050             tr_bis457.__init__(x_spawn + step * i, y_spawn, z_spawn)
1051
1052             # create vectors of translation transform vectors (x, y, z)
1053             tr_vectors_bis = [tr_bis0, tr_bis431, tr_bis457]
1054             tr_times_bis = [0, frameToSec(time_horizon - frame_anim_i_bis),
1055                             frameToSec(time_horizon)]
1056
1057             # create trajectory of tile bis
1058             addTranslationControlPoints(tile_bis_temp, tr_times_bis,
1059                                       tr_vectors_bis, True)
1060             createAnimationBlockForNode(tile_bis_temp, True)
1061
1062     # ----- C
1063
1064     def calc_FrameGapBetweenTiles(step_tiles, conveyor_velocity_mmPerFrame):
1065         frame_gap = round(step_tiles / conveyor_velocity_mmPerFrame, 0)
1066         # [mm] / [mm / frame] = frames

```

```

1067     # round the quantity to 0 decimal values
1068     return frame_gap
1069
1070
1071 def calc_StepTiles(nMovers, radius):
1072     beta = 360 / nMovers
1073     beta_rad = beta * math.pi / 180
1074     # distance in mm between subsequent tiles (rounded to 2 decimal values)
1075     arc = round(beta_rad * radius, 2)
1076     return arc
1077
1078
1079 # ----- F
1080
1081 def frameToSec(frame):
1082     sec_per_frame = 0.041666666 # 1/24
1083     time = frame * sec_per_frame
1084     return time
1085
1086
1087 # ----- H
1088
1089 def hide_1_tile():
1090     hide_tile_1 = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
1091                                "Piastrella_Prova9", True)
1092     hide_tile_1.bis = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
1093                                    "Piastrella_Prova9_bis", True)
1094
1095     to_hide_tiles = [hide_tile_1, hide_tile_1.bis]
1096     hideNodes(to_hide_tiles)
1097
1098
1099 def hide_2_tiles():
1100     hide_tile_1 = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
1101                                "Piastrella_Prova9", True)
1102     hide_tile_2 = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
1103                                "Piastrella_Prova8", True)
1104     hide_tile_1.bis = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
1105                                    "Piastrella_Prova9_bis", True)
1106     hide_tile_2.bis = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
1107                                    "Piastrella_Prova8_bis", True)
1108
1109     to_hide_tiles = [hide_tile_1, hide_tile_2, hide_tile_1.bis, hide_tile_2.bis]
1110     hideNodes(to_hide_tiles)
1111
1112
1113 # ----- M
1114
1115 def mPerSec_in_mmPerFrames(velocity_mPerSec):
1116     # since 1 m = 1000 mm and 1 sec = 24 frames:
1117     velocity_mmToFrames = velocity_mPerSec * 1000 / 24
1118     return velocity_mmToFrames
1119
1120
1121 # ----- S
1122
1123 def show_hiddenTiles():
1124     show_tile_1 = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"

```

```

1125         "Piastrella_Prova9", True)
1126     show_tile_2 = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
1127         "Piastrella_Prova8", True)
1128     show_tile_1_bis = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
1129         "Piastrella_Prova9_bis", True)
1130     show_tile_2_bis = findNodePath("Switch_Demo_Tiles/10_DemoTiles/"
1131         "Piastrella_Prova8_bis", True)
1132
1133     to_show_tiles = [show_tile_1, show_tile_2, show_tile_1_bis, show_tile_2_bis]
1134     showNodes(to_show_tiles)
1135
1136
1137 # ----- ANIMATION CREATOR -----
1138
1139
1140 def create_Animation(nMovers, diameter):
1141     setCurrentFrame(0) # go to starting animation frame
1142     show_hiddenTiles()
1143     radius = diameter / 2
1144
1145     # the number of tiles of the demo is selected based on the limitations of
1146     # certain cases
1147     if n == 5 and d > 3000 and d <= 3650:
1148         nTiles = 9
1149         hide_1_tile()
1150     elif n == 5 and d > 3650:
1151         nTiles = 8
1152         hide_2_tiles()
1153     else:
1154         nTiles = 10
1155
1156     # [mm/frame]
1157     conveyor_velocity_mmPerFrame = mPerSec.in_mmPerFrames(conveyor_velocity)
1158     link_start_line = findNodePath("Your_Line/ExtraPack.Before.Sorting/"
1159         "Link_StartInspecting", True)
1160     coord_start_line = findLinkCoordinates(link_start_line)
1161     x_start_line = coord_start_line[0] # x coordinate of the start of line
1162     y_start_line = coord_start_line[1] # y coordinate of the start of line
1163     x_spawn_tiles = x_start_line + 1000
1164     y_spawn_tiles = y_start_line
1165
1166     if diameter <= 2500:
1167         z_spawn_tiles = 1105
1168     else:
1169         z_spawn_tiles = 1108.5
1170
1171     # X, Y, Z coordinates of movers when in grasp position
1172     x.start = 0
1173     y.start = radius
1174     z.start = 1150
1175     # height of tile when released on station
1176     z.tile_release = 730
1177     # max height of the mover w.r.t. conveyor
1178     max_handling_height = 40
1179     # X value where the pack is wrapped
1180     x.packaging_position = 9845
1181
1182     if nMovers <= 6 and not bool_stations_cancelled:

```

```

1182     step_for = 1
1183 elif nMovers > 6 and bool_stations_cancelled:
1184     step_for = 3
1185 else: # if nMovers <= 6 with stations cancelled or in the default case of
1186     nMovers > 6
1187     step_for = 2
1188
1189 fl_nMovers = float(nMovers) # need a float to calculate a rotation that is
1190 NOT an integer
1191 z_rot.angle = round(float((-360 / fl_nMovers) * step_for), 1)
1192
1193 # ----- Delete Previous Animations
1194 all_anim = getAnimBlockNodes(True)
1195 trashAnimations(all_anim)
1196
1197 # ----- Animate TILES
1198 step_tiles.normal = calc.StepTiles(nMovers, radius) # distance to be kept
1199 between tiles (to be synchronized with movers)
1200
1201 # if (step_tiles.normal / conveyor.velocity) < 785.4, the speed of the rotary
1202 circle cannot keep up with the tiles arrival rate. So in this case step_tiles
1203 must be doubled!
1204 condition = step_tiles.normal / conveyor.velocity
1205 if condition < 785.4: # used to be if step_for == 3
1206     step_tiles = step_tiles.normal * 2
1207 else:
1208     step_tiles = step_tiles.normal
1209
1210 frame_gap_btw_tiles = calc.FrameGapBetweenTiles(step_tiles,
1211 conveyor.velocity_mmPerFrame) # frame difference between animations of
1212 subsequent tiles
1213 frame_nearest_tile_arrived = round((-x_spawn_tiles /
1214 conveyor.velocity_mmPerFrame) - (frame_gap_btw_tiles * (nTiles - 1)), 0)
1215
1216 buildTilesAnimation(x_spawn_tiles, y_spawn_tiles, z_spawn_tiles, step_tiles, n
1217 , nTiles, z_tile_release, max_handling_height, conveyor.velocity_mmPerFrame,
1218 frame_gap_btw_tiles, step_for)
1219
1220 # ----- Animate MOVERS
1221 # after positioning tiles, the animation should start 4 frames after the
1222 nearest tile reaches x = 0
1223 buildAllMoversAnimation(x_start, y_start, z_start, z_tile_release, nMovers,
1224 nTiles, max_handling_height, frame_nearest_tile_arrived, frame_gap_btw_tiles)
1225
1226 # ----- Animate PACKAGES
1227 buildPackageAnimation(nTiles, x_packaging_position,
1228 conveyor.velocity_mmPerFrame, z_rot.angle, frame_nearest_tile_arrived)
1229
1230 # ----- Animate ROTARY CIRCLE
1231 buildRotaryCircleAnimation(nMovers, nTiles, frame_gap_btw_tiles, step_for)
1232
1233 # ----- Animate MACHINERY
1234 buildMachineryAnimation(z_rot.angle, frame_nearest_tile_arrived)
1235
1236 # ----- Animate SLIDER
1237 buildSliderAnimation(x_pack, y_pack)
1238
1239 setCurrentFrame(0) # go to starting animation frame

```



# Appendix D

## Final Editor Code: Main Section

---

```
1 # ----- EDITOR MAIN SECTION -----
2
3 # ----- Clean VR tools and menus
4 clean_vr_tools()
5 delete_existing_extra_vr_menus()
6
7 # ----- Rebuild VR custom tools and menus (kept hidden)
8 create_hidden_vr_menus()
9
10 # ----- Delete Previous Animations
11 delete_previous_animations()
12
13 # ----- Hide existing annotations
14 executePython('selectVariantSet("Hide All Annotations")')
15
16 # ----- Start Editor
17 line_0 = findNodePath("Line0", True)
18 line_1 = findNodePath("Line1", True)
19 line_2 = findNodePath("Line2", True)
20
21 global nLines
22 if nLines == 1:
23     showNode(line_0)
24     hideNode(line_1)
25     hideNode(line_2)
26 elif nLines == 2:
27     showNode(line_0)
28     showNode(line_1)
29     hideNode(line_2)
30 else: # nLines == 2
31     showNode(line_0)
32     showNode(line_1)
33     showNode(line_2)
34
35 line_distance = 15000 # distance between lines [mm]
36
37 # ----- Global lists used
38 global list_d, list_p, list_n, list_palletDistance, list_three_per_arc,
    list_conveyor_velocity, list_limit_stat, list_limit_movers,
    list_starting_value_stat, list_scaling_y_machinery, list_nArcs, list_x_pack,
    list_y_pack, list_limit_arc_cent, list_length_m, list_bool_stations_cancelled,
    all_time_horizons
```

```

39
40 for j in range(0, 3):
41     list_limit_stat[j] = list_n[j] - 1
42
43 # ----- Main "for" cycle
44 for i in range(0, nLines):
45     setCurrentFrame(0) # go to starting animation frame
46
47     diameter = list_d[i]
48     r = list_d[i] / 2
49     num_pallets = list_p[i]
50     num_movers = list_n[i]
51
52     show_All_LinkSpheres(i)
53
54     # ----- EkoSort
55     build_EkoSort(i, line.distance)
56
57     # ----- EkoWrap
58     link_wrapping = findNodePath("Line" + str(i) + "/Your_Line/Your_EkoSort/
59     ConveyorComplete/Link_EkoSort", True)
60     coord_ekowrap = findLinkCoordinates(link_wrapping)
61     x_wrapping = coord_ekowrap[0]
62     y_wrapping = coord_ekowrap[1]
63
64     load_EkoWrap(x_wrapping, y_wrapping, r, i)
65
66     # ----- ExtraPack After Wrapping
67     link_extrapack_after_wrapping = findNodePath("Line" + str(i) + "/Your_Line/
68     EkoWrap/Link_EkoWrap", True)
69     coord_extrapack_after_wrapping = findLinkCoordinates(
70     link_extrapack_after_wrapping)
71     x_extrapack_after_wrapping = coord_extrapack_after_wrapping[0]
72     y_extrapack_after_wrapping = coord_extrapack_after_wrapping[1]
73
74     load_ExtraPack_AfterWrapping(x_extrapack_after_wrapping,
75     y_extrapack_after_wrapping, r, i)
76
77     # ----- ExtraPack Before Sorting
78     link_extrapack_before_sorting = findNodePath("Line" + str(i) + "/Your_Line/
79     Your_EkoSort/ConveyorComplete/Link_Before_EkoSort", True)
80     coord_extrapack_before_sorting = findLinkCoordinates(
81     link_extrapack_before_sorting)
82     x_extrapack_before_sorting = coord_extrapack_before_sorting[0]
83     y_extrapack_before_sorting = coord_extrapack_before_sorting[1]
84
85     load_ExtraPack_BeforeSorting(x_extrapack_before_sorting,
86     y_extrapack_before_sorting, r, i)
87
88     # ----- ExtraPack Palletizer
89     build_ExtraPack(num_pallets, r, list_palletDistance[i], list_three_per_arc[i],
90     i)
91
92     length_tot = measure_LineLength(i)
93     # convert to meters and round to 1 decimal
94     list_length_m[i] = round(length_tot / 1000, 1)
95
96     # ----- Lights and Shadows

```



```
89     compute_line_shadows(i)
90
91     # ----- Animation
92     create_animation(num_movers, diameter, i, line_distance)
93
94 longest_time_horizon = max(all_time_horizons)
95
96 for i in range(0, nLines):
97     diameter = list_d[i]
98     num_movers = list_n[i]
99
100    create_bis_animation(num_movers, diameter, i, line_distance,
101                          longest_time_horizon)
102
103    hide_all_link_spheres(i)
104    deselect_all()
105
106 # ----- ShadowPlane Shadows
107 # go to starting animation frame (so that there are no "strange" shadows when
108 #   everything is still)
109 set_current_frame(0)
110 compute_shadowplane_shadows()
111 deselect_all()
112
113 # ----- Update Annotations
114 update_annotations_position()
115 deselect_all()
116
117 # ----- Update dummy VR tools
118 update_vr_tools()
119
120 # ----- Close loading screen and open final reset message
121 widget_complete_editor.editor_end_widget.show()
122 widget_complete_editor.loading_widget.hide()
123
124 # ----- Go To VR
125 execute_python('selectVariantSet("VR - Open Editor in VR")')
126
127 # ----- Starting View
128 execute_python('selectVariantSet("Control Room View")')
```

---



# List Of References

- [1] Autodesk. *About Variant Sets*. URL: <https://knowledge.autodesk.com/support/vred-products/learn-explore/caas/CloudHelp/cloudhelp/2019/ENU/VRED/files/Variants/VRED-Variants-About-Variant-Sets-html-html.html>.
- [2] Autodesk. *Collaboration*. URL: <https://knowledge.autodesk.com/support/vred-products/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/VRED/files/Collaboration/VRED-Collaboration-Collaboration-1-html-html.html>.
- [3] Autodesk. *To Create Variants*. URL: <https://knowledge.autodesk.com/support/vred-products/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/VRED/files/GUID-F913D186-40A3-4F5B-87DD-0B22D1D44611-hm.html>.
- [4] Autodesk. *VRED commandline options*. URL: <https://knowledge.autodesk.com/support/vred-products/troubleshooting/caas/sfdcarticles/sfdcarticles/VRED-commandline-options.html>.
- [5] Qt Company. *About Qt*. URL: [https://wiki.qt.io/About\\_Qt](https://wiki.qt.io/About_Qt).
- [6] Qt Company. *PySide2.QtWidgets*. URL: <https://doc.qt.io/qtforpython/PySide2/QtWidgets/index.html#module-PySide2.QtWidgets>.
- [7] Qt Company. *Qt for Python Documentation - Qt Modules*. URL: <https://doc.qt.io/qtforpython/modules.html>.
- [8] HTTP for Humans. *Requests: HTTP for Humans*. URL: <https://requests.readthedocs.io/en/master/>.
- [9] Python.org. *threading - Thread-based parallelism*. URL: <https://docs.python.org/3/library/threading.html>.
- [10] SACMI S.C. *Automatic tiles sorting, packaging and palletizing lines*. URL: <https://www.sacmi.it/en-us/Ceramics/Tiles/automatic-tiles-sorting,-packaging-and-palletizing-lines>.
- [11] Nuova Sima S.p.A. *EkoSort - Sorting line catalogue*. URL: [http://www.nuovasima.com/System/00/02/37/23717/635467419552422037\\_1.pdf](http://www.nuovasima.com/System/00/02/37/23717/635467419552422037_1.pdf).
- [12] Nuova Sima S.p.A. *Ekowrap, Ekoroll Catalogue*. URL: [http://www.nuovasima.com/System/00/02/18/21899/635156446316392104\\_1.pdf](http://www.nuovasima.com/System/00/02/18/21899/635156446316392104_1.pdf).
- [13] Nuova Sima S.p.A. *Palletizer by Nuova Fima*. URL: [http://www.nuovasima.com/System/00/01/74/17448/634206860217197500\\_1.pdf](http://www.nuovasima.com/System/00/01/74/17448/634206860217197500_1.pdf).

- [14] Nuova Sima S.p.A. *Tiles inspection systems by Surface Inspection*. URL: [http://www.nuovasima.com/System/00/02/34/23496/635421528748930645\\_1.pdf](http://www.nuovasima.com/System/00/02/34/23496/635421528748930645_1.pdf).
- [15] Gabriele Verducci. *Corso di formazione: Modellazione 3D, Rendering, Animazione*. URL: [http://www.hardcad.it/rhino/corso\\_01.pdf](http://www.hardcad.it/rhino/corso_01.pdf).
- [16] Vive. *VIVE Cosmos Overview*. URL: <https://www.vive.com/us/product/vive-cosmos/overview/>.
- [17] Vive. *VIVE Pro Overview*. URL: <https://enterprise.vive.com/us/product/vive-pro/>.
- [18] VRgineers. *XTAL Overview*. URL: <https://vrgineers.com/xtal/>.
- [19] XlsxWriter. *Creating Excel files with Python and XlsxWriter*. URL: <https://xlsxwriter.readthedocs.io/index.html>.