Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Automation and signal processing	Spring semester, 2018 Open
Author: Rolf Jacob Dramdal	...Rolf J. Dramdal..... (signature of author)

Programme coordinator: Morten Mossige

Supervisor(s): Morten Mossige

Title of master's thesis:

Visualization of process values as a function of a robots path in 3D Virtual Reality

Norwegian title:
Presentasjon av prosessverdier som funksjon av en robotbane i 3D virtuell virkelighet

Credits:
ECTS: 30

Keywords: Virtual reality, HTC Vive, 3D graphics, User interface, Industrial robotics,	Number of pages: 28 + attached files: Source code, application files, demonstration video Stavanger, 15.06.2018 date/year

Abstract

Modern industrial robotics can display a lot of information, especially when something goes wrong or unexpected behaviour occurs. Often developers are presented with too much information, rather than too little. This makes it time consuming to locate the problem, and debugging the robotic system becomes difficult. This thesis will explore solutions for visualising the process values of a robotic system inside a virtual world.

Virtual reality is an enabling technology, that has a wide area of applications in industries. It has been proven useful in a variety of applications, stretching from product design to educational training. The software running on industrial robots are often created offline in a virtual environment. This opens the possibility of exploring virtual reality functionality for development of industrial robot tasks.

This thesis is done in collaboration with ABB, with the purpose of visualizing process values as a function of a robotic path, using the HTC Vive virtual reality system. ABB has implemented virtual reality for their offline programming software (Robotstudio). The implementation presented in this thesis graphs a robots process values as a function of the path run by the robot. Graphics are designed based on the data recorded and loaded into the virtual world. User interface is added to enable the user to move the robot along its recorded path. The graphics implemented offers the user an intuitive way of analyzing process values while immersed in a virtual world.

The resulting solution of the thesis has been tested on generated test data as well as actual data taken from runs of a paint robot at ABB. Being immersed in the scene enables developers to validate robotic systems for issues that otherwise would not be detectable until physical prototyping.

List of Abbrevations

VR - Virtual Reality

IVR - Immersive Virtual Reality

UI - User Interface

HUD - Heads Up Display

HMD - Head Mounted Display

IPD - Interpupillary distance

Contents

Chapter 1

Introduction

1.1 Motivation

Industrial robotics are becoming more complex every day, as the demand for automation in manufacturing grows larger. Modern industrial robots can display a great amount of information in the form of process values, error messages, or the general state of the robot. During development of such systems, getting an overview of all the processes involved can be overwhelming. Often the problem is that developers are presented with too much information, rather than too little.

Offline programming is often utilized in order to create, analyze and make changes to the software that runs on industrial robots. This involves going from the manufacturing environment, to viewing a copy of the robot inside a virtual environment, on a computer screen. The main advantages of this approach is that it reduces the down time that is required for online robot programming, and the ease with which one can test different approaches to develop, update or debug the robot.

Virtual environments open the possibility of creating Virtual Reality (VR) applications. Virtual Reality received a lot of attention in the early 2010s, as the equipment became widely accessible to the general public for development purposes. VR systems like the Oculus Rift and HTC Vive gained a lot of publicity and paved the way for developers to extend virtual world functionality.

This project is conducted in collaboration with ABB. Its main purpose is to utilize Virtual Reality to explore methods for visualization inside a virtual environment, namely Robotstudio (described in section 2.3).

Viewing a robots position for a given process value can make the system easier to analyze, especially if an unexpected spike in the signal occurs during a run. Being able to move the robot along its path and view its process values for that exact position can provide valuable insight.

1.2 Virtual Reality

History

VR is a technology which allows the user to interact with a computer-simulated environment. The first development of a VR system was in 1963, when Dr. Ivan Sutherland created the first Head Mounted Display (HMD) [1]. Since then there has been a continuous development of VR systems, resulting in great enhancement of the technology. Up until the 1990s, limitations in

computer graphics and processing power had limited the technological advancements. Today VR has spread to a wide variety of fields, with the most prominent being education and medicine [2].

Even though processing power and computer graphics had seen advancements, development of VR applications in the 1990s and 2000s was difficult. This was because of the cost of implementation and development being time consuming. After VR equipment and resources became accessible to developers at a significantly lower cost in the 2010s, through gear like HTC Vive and Oculus Rift, VR development has seen a great surge in both industrial and commercial applications[3].

VR applications have already added value to several fields, e.g. educational applications have been made for the purpose of training medical personnel [4].

Immersion

Immersion refers to three dimensional (3D) VR applications, which can be divided in to immersive and non-immersive.

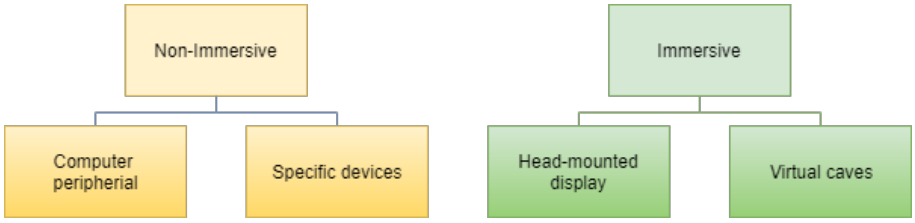


Figure 1.1: Different types of VR applications

Non-immersive systems are either computer peripheral devices, like keyboard and mouse, or specially designed devices used to simulate real control, like vehicle control cockpits.

Immersive Virtual Reality (IVR) surrounds the user in the computer generated environment. The most common way of achieving this is by using a head mounted display (HMD). HMDs use a 3D stereo display in combination with head tracking to create a human-centered, rather than a computer-determined point of view (POV). This human-centric view strengthens the sense of presence for the user, or the feeling of "being there", and enhances the users sense of reality in the virtual environment. For this project, a HMD is central in the solution of the problem statement(detailed in section 2.1).

Head Mounted Display

A VR headset is a heads-up display (HUD) which allows users to interact with the simulated world in a first person view. Modern VR headsets can be placed into two categories; mobile or tethered. Mobile headsets are the most accessible, relatively inexpensive, and user friendly option. However, they are limited as mobile phones are not specifically made for VR, and the resolution and functionality is low. Tethered headsets are physically connected to a PC. They are more expensive and requires a more comprehensive set up, but offer a more immersive user experience and functionality [5].

Most HMDs are binocular, meaning they display a screen to each eye, offering the user a stereoscopic view. Interpupillary distance (IPD) is the distance between the center of the pupils of the eyes. Most VR headsets offer adjustments for IPD to match the binocular screens.

Properly spaced lenses can significantly improve image quality. This is often overlooked and results in blurry imagery [6].

1.3 Problem statement

This projects main purpose is to explore methods for visualizing a robots process values in an intuitive and effective way, while the user is immersed in a virtual world. This involves creating graphical components, placing them in a 3D scene, and creating UI to control and interact with the graphics.

1.4 The structure of this thesis

Chapter 1 - Introduction
An introduction and motivation of the thesis topic, an introduction to virtual reality, and the problem statement is defined.

Chapter 2 - Prerequisites
The relevant theory, technology used, and previous work which this thesis builds upon.

Chapter 3 - Implementation
Descriptions of the methods used to create the application.

Chapter 4 - Results
A display of the results of the application and a test case to review the performance.

Chapter 5 - Discussion
The fifth and final chapter will detail the results, discuss the limitations of the solution and make suggestions for improvements and future work.

Chapter 2

Prerequisites

2.1 User Interface design

2.1.1 Placement of user interface

In non-VR applications, user interface (UI) and visual information is often displayed on the screen itself. This kind of display is called non-diegetic, as it is not a part of the scene. Non-diegetic UI does not work well in a VR-application as it is uncomfortable and straining for the human eye to focus on something that close. A logical solution to this is to explore diegetic methods for displaying information in a HUD.

2.1.2 The gestalt principles

It is important to keep in mind conventional UI guidelines when designing UI for VR applications. "Gestalt" (from German: "shape" or "form") is a cognitive theory, developed in the late 1880s. Today, some of the Gestalt principles are used in UI design. They explain how the mind categorize and create meaningful assumptions based on what is observed. [7]

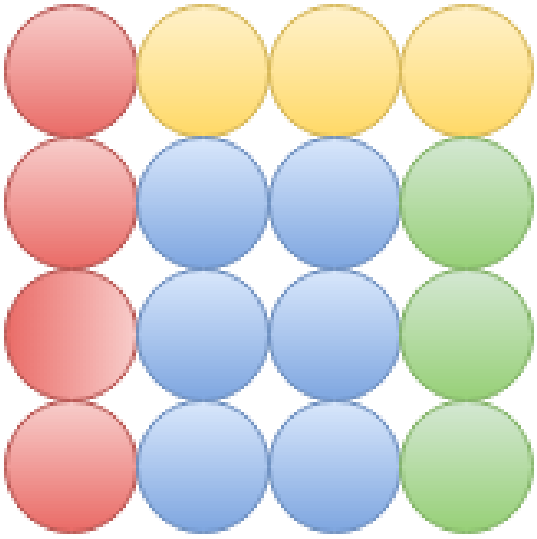


Figure 2.1: The Gestalt principle of similarity demonstrated with color

The most relevant principle for this project is the principle of similarity. The principle states that elements which share visual characteristics, such as color and shape, will be seen as belong-

ing together. [7] This principle is central in the design of the graphics and their functionality in this project.

2.1.3 Manipulating the virtual world

Less effort is required to move one hand relative to the other versus moving a single hand relative to an abstract 3D location [8]. This type of bimanual (two handed) interface offers an intuitive way of exploring the UI. Attaching UI within the scene itself can lead to the interface being too complicated and difficult to manage, as movement inside of virtual worlds are limited for most VR systems. The most straight forward solution is to attach the UI to the tracked equipment.

2.2 HTC Vive

For this project, the HTC Vive system is used. The HTC Vive system consists of a headset, two motion controllers and two base stations. The two base stations (also called lighthouses) should be set up in the opposite sides of the room. This provide the user up to $20m^2$ area of motion tracking, depending on the size of the room. The motion controllers can be used to interact with the virtual world through motion tracked by the base stations and button inputs. Below is information that describes the equipments specifications and functionality.

Figure 2.2: The HTC Vive equipment

	HTC Vive
Screen	OLED 2.160 x 1.2000 pixels
Refresh Frequency (Hz)	90
Platform Web	Steam VR
Field of View (°)	110
Integrated Audio	Yes
Integrated Micophone	Yes
Controls	Two wireless controllers
Sensors	Accelerometer, gyroscope, laser position sensor, front camera, base stations
Connections	HDMI, USB 2.0,USB 3.0
Minimum Requirements	NVIDIA GTX 970/AMD Radeon R9 290. Intel i5-4590, 4 GB RAN, HDMI 1.3
price(euro)	899

Table 2.1: HTC Vive specifications

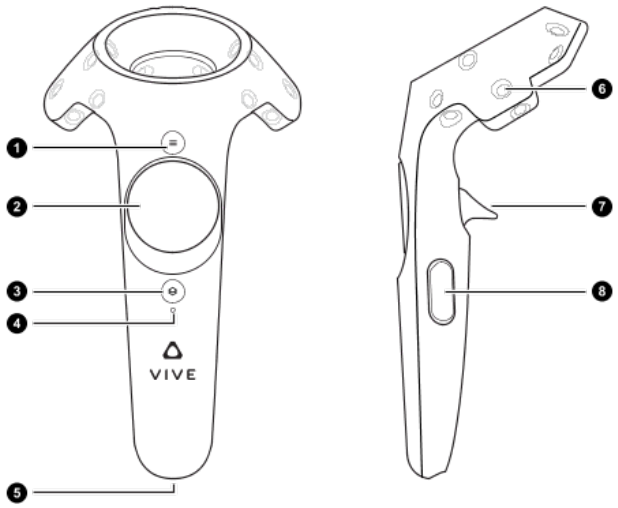


Figure 2.3: The HTC Vive motion controller functionality

1: Menu button

2: Touchpad

3: System button

4: Status light

5: Micro-USB port

6: Tracking sensor

7: Trigger

8: Grip button

The HTC Vive has considerable computer requirements[9]:

Processor: Intel™ Core™ i5-4590 or AMD FX™ 8350, equivalent or better.

Grahics: NVIDIA GeForce™ GTX 1060 or AMD Radeon™ RX 480, equivalent or better.

Memory: 4 GB RAM or more

Video output: 1x HDMI 1.4 port, or DisplayPort 1.2 or newer.

USB: 1x USB 2.0 port or newer

Operating system: Windows™ 7 SP1, Windows™ 8.1 or later or Windows™ 10

All the units of the system are being tracked by an accelerometer and a gyroscope. This tracking only measures the position relative to itself, which can result in a drift of the position. If the positions were to drift, this would be corrected by the lighthouses. Each device is installed with optical sensors, which are triggered by infrared light sent from the lighthouses. When these sensors are triggered the absolute position and orientation of the devices are measured.

2.3 Robotsudio

ABBs Robotstudio is a simulation and offline programming software which allows robot programming to be done offline on a PC without shutting down production. Robotstudio is built on the ABB VirtualController, an exact copy of the software that runs the robots in online production. This allows for very realistic simulations, using real robot programs and configuration files identical to those used on the shop floor [10].

Robotstudio functionality can be extended and custom behaviour can be implemented through the Robotsudios software development kit (SDK). By using the SDK, extensions called add-ins can be created. A Robotstudio Add-in is a .Net assembly that extend the functionality within Robotstudio, by calling the Robotstudio application programming interface (API).

The robot paths used in development/testing have been created/loaded in RAPID, which is the programming language of Robotstudio.

2.4 The graphics engine

Robotstudio graphics engine is based on DirectX 11. An important difference between Robotstudio graphics engine and DirectX is its conventions for how to apply transformation-matrices to the vectors that make up the 3D graphics. Robotstudio is using the mathematical convention of pre-multiplication (contrary to DirectX), as shown in the formula below.

$$\begin{bmatrix} m11 & m12 & m13 & m13 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Formula 2.1: Robotstudio using pre-multiplication to transform vectors in 3D-space

$$V' = V * Rx * Ry * Rz$$

Formula 2.2: Example of rotating Vector V, using rotation matrices.

The convention for multiplication is important when it comes to the order in which transformations are being applied. In the formula below, V will be rotated around the z-axis first, then y, and lastly x. Robotstudio is using a right handed coordinate system, in contrast to traditional DirectX engines.

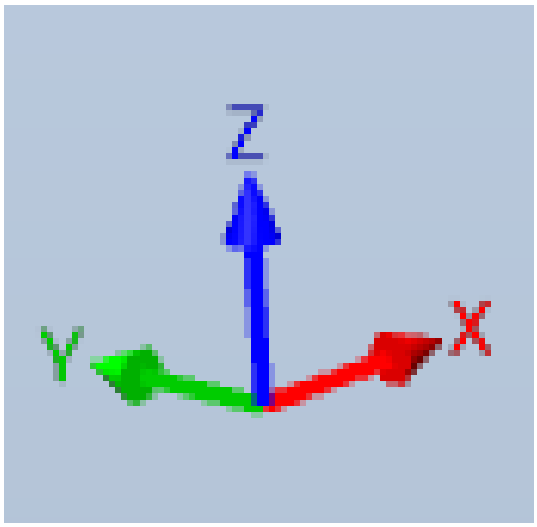
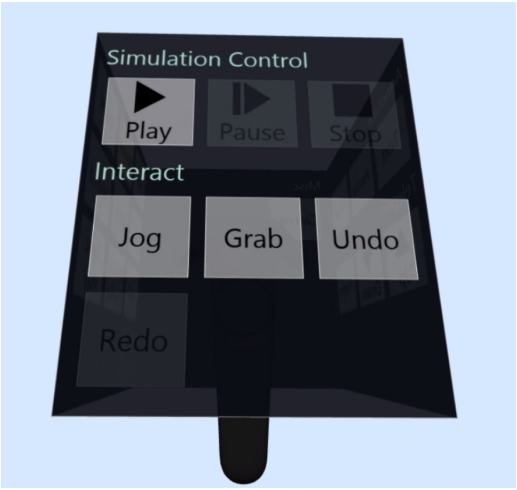


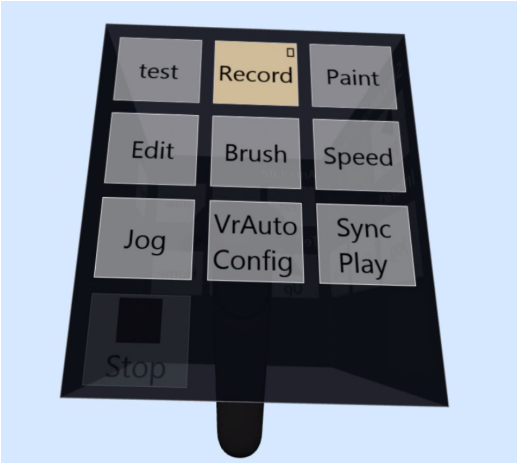
Figure 2.4: The right handed coordinate system of Robotstudio

2.5 VR inside Robotstudio

2.5.1 Virtual controls in Robotstudio



(a) Simulation Control



(b) Applications

Figure 2.6: The menu cube attached to the left controller, used to access applications inside the VR of Robotstudio

A menu cube is attached to the left motion controller and is used as interface inside Robotstudio once the VR addin is activated. The cube has four sides, but only the two shown in figure 2.6 is used in this project. By using the scroll on the left motion controller, the menu cube will rotate and reveal more buttons for functionality. These buttons are activated by aiming the right motion controller at them and clicking the trigger.

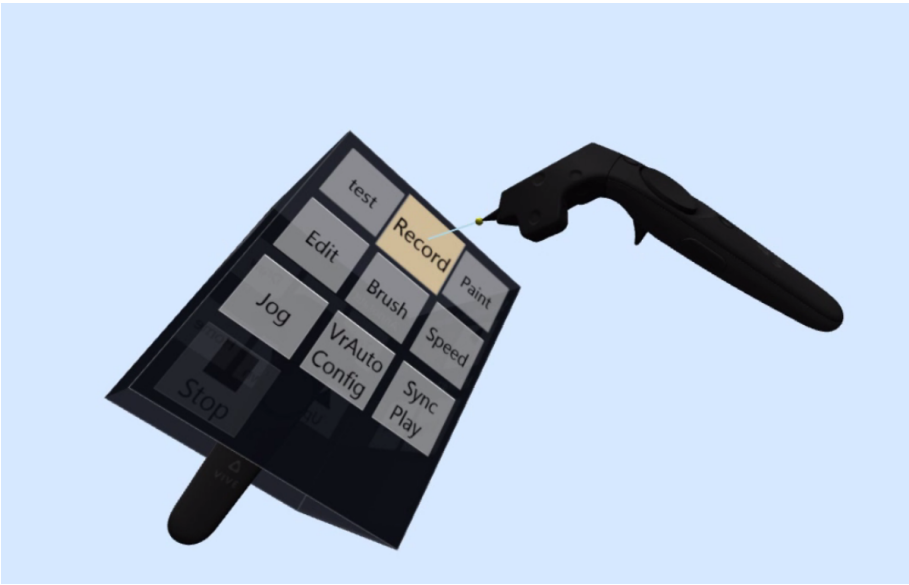


Figure 2.7: Buttons on the menu cube are enabled by aiming and clicking the right motion controller

2.5.2 Virtual movement in Robotstudio

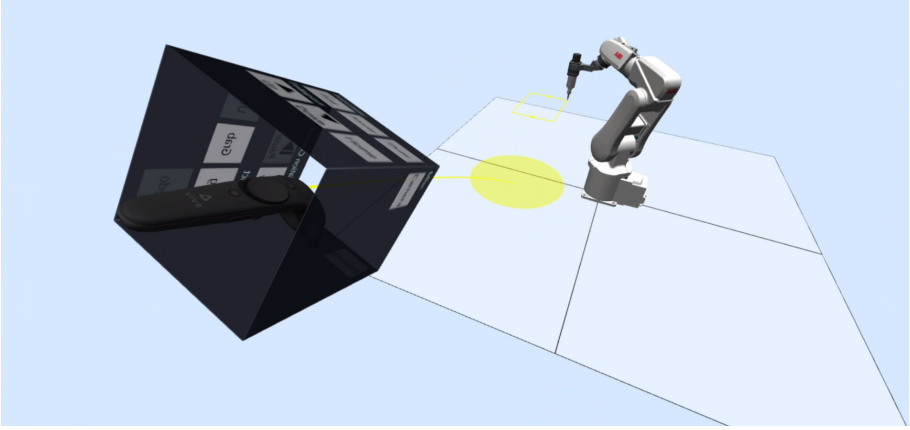


Figure 2.8: Movement inside the virtual environment

Figure 2.8 shows the touchpad of the the left motion controller being pressed down. While the touchpad is held down, a yellow line pointing to a circle field will be displayed on the floor of the virtual environment. Letting go of the touchpad will move the user to this position.

2.6 Development Environment

The development environment used for this solution is the .Net (C#) environment in Microsoft Visual Studios. The following sections explains what assemblies the solution imports and the abstract classes it extends.

The Robotstudio SDK
The Robotstudio SDK (software development kit) and it's .NET assemblies are used to create the Add-in for this solution.

VrPaintAddin
The VrPaintAddin.sln is a .Net package solution provided by ABB that contains helpful functions for adding new functionality to the VR environment. The input functionality implemented in this thesis builds upon the motion controller objects from this package. Most importantly, the package includes VrInputMode.cs, which is an abstract class that requires 3 abstract methods to be inherited:

Activate(): This method controls what the application runs once initialized

Update(): This method continuously listens for inputs while the application is running.

Deactivate(): This method controls what runs once the application is terminated.

Chapter 3

Implementation

3.1 System Overview

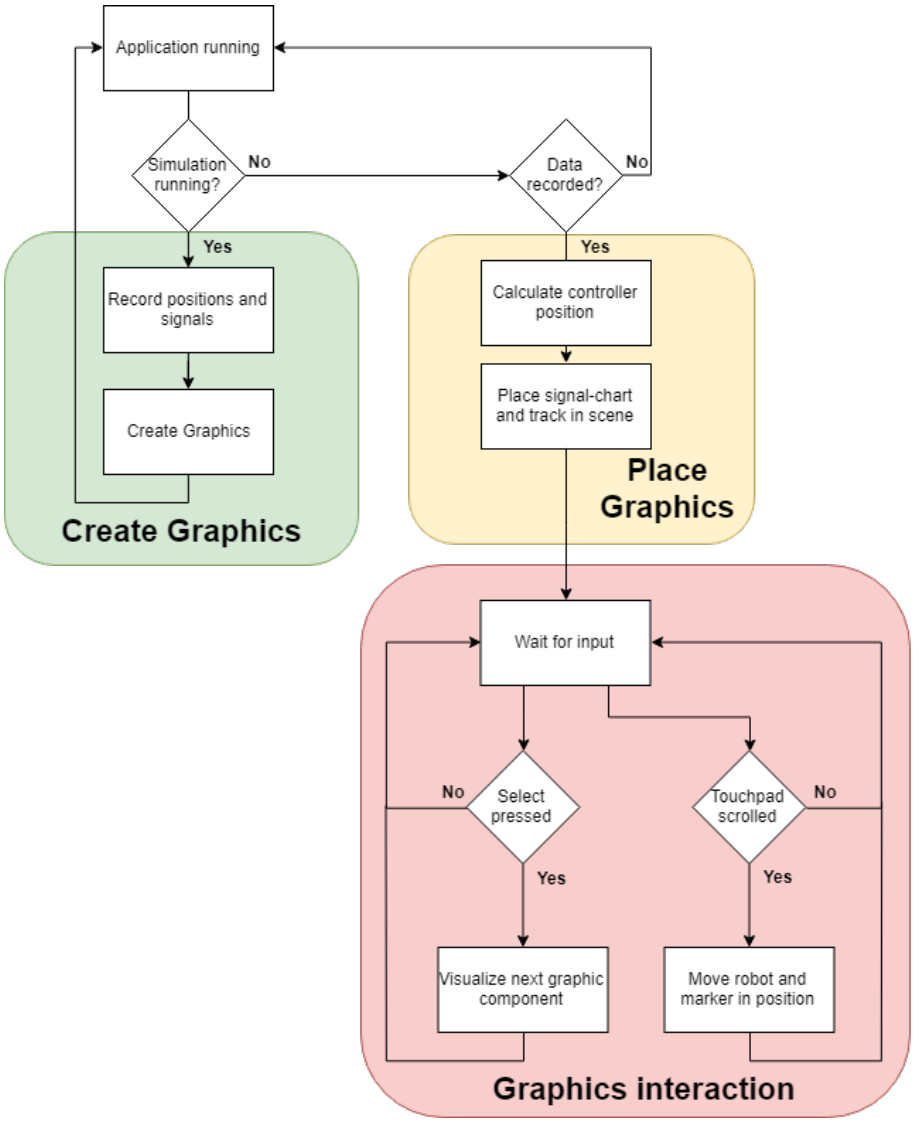


Figure 3.1: System overview of how the three main parts of the solution is connected.

The implementation of this solution will be described in 3 parts:

1. The recording of the data and creation of the graphics (described in section 3.2 and 3.3)

2. The placement of the graphics (described in section 3.4)

3. The control and functionality of the graphics (described in section 3.5)

3.2 Recording data

The recording of the robots positional values and process values are done in the RecData.cs class. If the robots joint values change as a result of a simulation being run, the recording will start. An event handler triggers each time the robots joint values change. The components that are tracked are the robots joint values, the active tools coordinates, and the active tools rotation. These values are then stored in three separate static lists.
If there are any process values available, their values are recorded each time the event handler triggers. Reading and storing a process value for every positional value ensures that the signal samples correspond to the correct position. This way there are no chances of a mismatch between a signal value and positional value.
The following public lists are created in this class to keep track of all the data used in implementing the application:

List name	**Data type**	**Containing**
jointValuesList	$< double : vector >$	The robots jointvalues
toolValuesList	$< Vector4 >$	The active tools position
TransPosList	$< Vector4 >$	The active tools position
signalList	$< double : vector >$	All process values
signalWidthList	$< int >$	Width of line
colorList	$< Color >$	Color of line

Table 3.1: Lists of data used in application

Below are the most significant methods for recording the data:
MyJointListener() is a listener that is activated when the robot moves (changes its positional values). When a simulation is running and the robots position has changed, its positional values are added to the lists.
RunPos() initializes the EventHandler.
RecSignal reads all the process values.
ResetList() empties all the recorded values form the lists. This method is used when a new simulation starts, or the application is terminated.
The full source code for RecData.cs is attached to this thesis.

3.3 Creating the graphics

This solution designs and creates 3 graphical components:

A graph window: A bitmap image created for the purpose of displaying a graph.

A 3D path: To show the robot path.

An arrow: To point at positions in the graph window.

These elements are to be designed to visualize the process data as a function of the robot path.

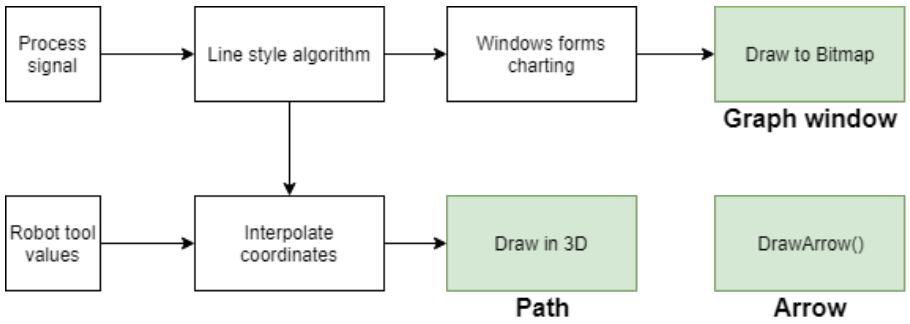


Figure 3.2: An overview of the graphics design. The input for the design is the robotic tool values and the selected process signal. The arrow is created separately.

Figure 3.2 shows how the application creates the graphics that are to be placed in the scene. The graphics only exist for the currently selected signal and is only temporarily saved to the local disk. Once the user selects the next process signal available, the current graphics are deleted, recreated with the new signal, and put back into the scene.
The arrow graphic is not dependent on any data for its creation, it is only dependent on the data recordings for its position and movement. It is created with the DrawArrow() method from the Robotstudio SDK. The following sections describe how each method in figure 3.2 is implemented.

3.3.1 The linestyle algorithm

The linestyle algorithm is responsible for designing the colorization and line-width of the recorded path, which is also the one used to design the graph. The method is based on the RGB color model. The color is determined by the current signal value in relation to the maximum signal value in the signal. As the components in the RGB method ranges from [0,255], the desired color for positive process values are calculated in the following algorithm:

```
double yMax = ySignal.Max();
foreach(double y in ySignal) {
  if (y == Math.Abs(y)) {
     double r = yProp > 50 ? 255 : ((2 * yProp / 100) * 255);
     double g = yProp < 50 ? 255 : ((2 * (100-yProp) / 100) * 255);
        }
}
int rA = (int)Math.Floor(r);
int gA = (int)Math.Floor(g);
int bA = 0;

colorList.Add(Color.FromArgb(rA,gA,bA))
```

Listing 3.1: Calculation of the linestyle. The proportion of the current signal value, in comparison to the maximal value in the signal, is used to set the RGB

The algorithm is designed to give a colorization that goes from green (low process values) to yellow (medium process values) to red (high process values). A similar algorithm is run for the negative values, ranging from green (low negative values) to blue (high negative values). The line width is slightly increased as the signal value is above 90% of its maximal value.

3.3.2 The graphic window

Since the signal to be graphed is of unknown value, a scaling problem is introduced. The graph window is created using System.Windows.Forms.DataVisualization.Charting in the .Net framework. The main advantages of using this namespace is that it solves the automatic scaling of the graph window axes.
The CreateChart() method takes lineWidthList,colorList,signalList and draws the graph using DataVisualization.Charting in Windows.Forms.
The DeleteChart() method deletes the temporary BMP image. The method is invoked on app-unload.

```
int signr = SigPos.GetCurrentSignal();
int l = RecData.signalList[signr].Count();
for (int i = 0; i < l; i++)
   {
   chart1.Series.Points.Add(RecData.signalList[0][i]);
   chart1.Series.Points[i].Color = RecData.colorList[i];
   chart1.Series.Points[i].BorderWidth = RecData.lineWidthList[i];
   }
```

Listing 3.2: Part of the CreateChart() method that draws the graph

Listing 3.2 shows how the class gets the recorded data and the currently selected signal, draws the graph onto the graph window, and saves the bitmap image to the local disk.

```
chart1.Size = new Size(1100 + l*2, 520);
this.chart1.SaveImage(@"mychart.bmp", ChartImageFormat.Bmp);
```

Listing 3.3: The window width is adjusted by the length of the current signal displayed, and then saved

In order for the graphic window to be able to display longer signal sequences, the window width is determined by the signal length. The signal length is determined by the length of the path ran by the robot. The image is temporarily saved locally for the purpose of loading it into Robotstudio by the SigPos.cs class.

3.4 Placing the graphics

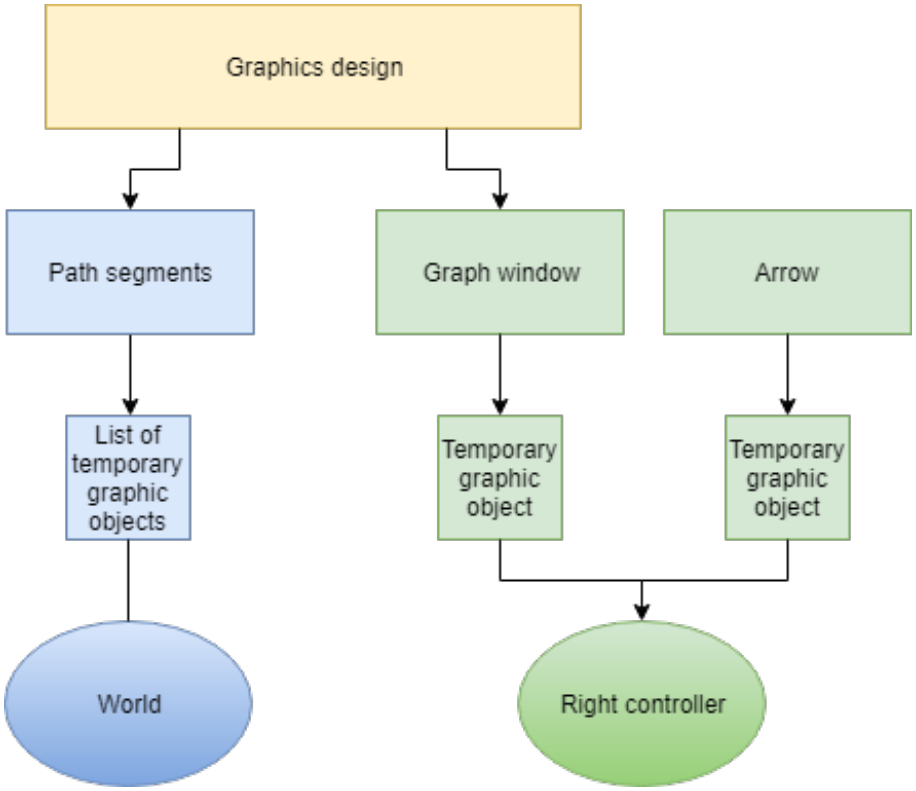


Figure 3.3: The placement of the graphics in the 3D virtual world.

The three graphic components created are placed in the 3D world in real time. The first component (the path) is placed in world coordinates and represents the robots path. The second component (the graph window) is placed in right controller coordinates, at the tip of the controller. Both of these graphics are adapted based on the process value selected by the user. All the graphics used in this solution are TemporaryGraphic objects inherited from the Robotstudio SDK. The path consists of several TemporaryGraphic objects and are placed in a list.

3.4.1 The path graphic

The path graphic is drawn in the 3D world using the recorded coordinates of the active tool of the robot. It has the exact same colorization as the graph in the graph window.The recorded values are interpolated using the **DrawLine()** function inherited from the Robotstudio SDK library.

```
double l = RecData.signalList[GetCurrentSignal()].Count();
for (int i = 0; i < l-1; i++)
        {
        _PathAtTrack =  station.TemporaryGraphics.DrawLine(RecData...
        transposList[i], RecData.transposList[i + 1], RecData...
        lineWidthList[i], RecData.colorList[i]);
        }
```

Listing 3.4: The placement of the path in the 3D world, and the design of the path

The paths color and width is determined by the same color and width of the graph in the graphics window.

3.4.2 The graph window

The chart window is placed at the tip of the right motion controller. It is set to follow the controller in the 3D virtual world.
In order for the graphics window to match the view, rotations are needed. First a 180 degree rotation around the y-axis, followed by a 90 degree rotation around the z-axis. The following matrices are calculated:

$$Roty = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix}$$

Formula 3.1: Rotation around the y-axis

$$Rotz = \begin{bmatrix} cos(\phi) & -sin(\phi) & 0 \\ sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Formula 3.2: Rotation around the z-axis

Since the window is attached to the controller, no translation is needed and we get the following Transformation matrix (Inserted ($\theta = 180$) and ($\phi = 90$):

$$Tzy = \begin{bmatrix} 0.2682 & -0.8940 & 0.3590 & 0 \\ -0.5350 & -0.4481 & -0.7162 & 0 \\ 0.8012 & 0 & -0.5985 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Formula 3.3: Transformation with rotation around the y-axis and z-axis, with no translation

3.4.3 The arrow pointer

The purpose of the arrow is to show the process value on the graph window compared to the robots position. The arrow is placed and attached to the right motion controller, pointing at the graph window. Some additional transformations are needed to make it fit in place on the bitmap image.

```
Vector3 vorigin = rcontrollerpos.Translation;
double x = RecData.GetCurrentValue()*...
RecData.ToolValuesList.Count()*0.00000025+.0415
double y = 0.015;
double z = -0.02;
Vector3 voriginmove = new Vector3(x, y, z);
```

Listing 3.5: The function moving the arrow in 3D space, when the robot changes position

In listing 3.5 the translation vector for placing and moving the arrow along the x-axis of the

graph window is listed. When it reaches the end of the graph it will return to the start (as the robot will have finished the path at this point). The current position of the robot, multiplied with the total length of the signal, and then scaled, makes the pointer fit the graphic window for any signal.

3.5 Controlling the graphics

3.5.1 Control of graphics

The control of the graphics are based on the current position of the robot and the process value selected. This is controlled with two variables:

currentValue The currentValue is a static variable that keeps track of the robots current position in the path recorded. It is implemented with getter and setter methods. The setter method ensures that the variable returns to zero if scrolled further than the end of the list (When scrolling the robot further than the end of its path).

currentSignal The currentSignal keeps track of what process value dataset is selected. It cycles trough the count of signal value datasets that was present during the simulation.

These two variables are used for all the graphic components, which ensures the arrow pointing at the graph window matching with the robots current position, while the user scrolls the robot back and forward through the simulated track.
The class uses the position lists to move the robot back/forward based on user input. It is also responsible for keeping track of the robots current position.
The setter and getter methods described above, along with the methods below, make up the most critical functions of controlling the graphics in real time.

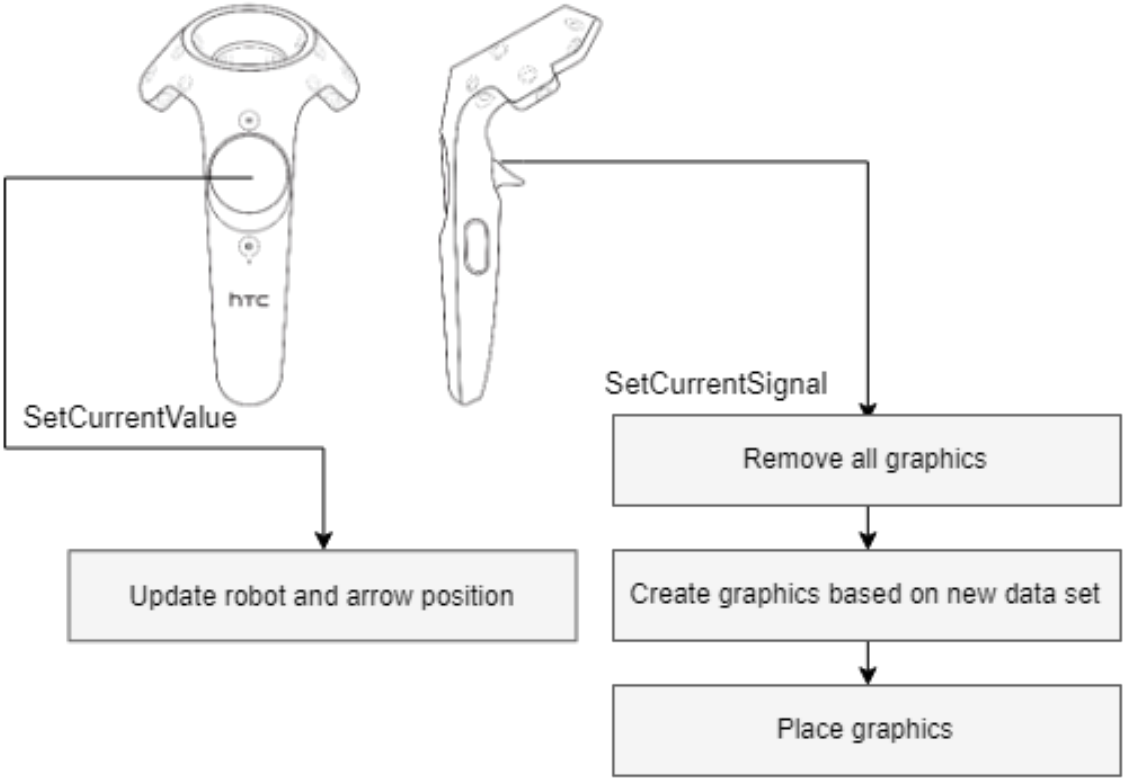


Figure 3.4: The adaptation of the graphics based on user input

NextPosition() sets the currentValue to the next in the position list.
PrevPosition() sets the currentValue to the previous in the position list.
SetRandomVal() sets the currentValue to a random position in the position list.
setRobPos() sets the robot position according to currentValue.
ResetPos() resets the robots position and deletes the position list.

3.5.2 Controlling user input

The functions discussed in the previous secion (3.5.1) are triggered by the trigger button and touchpad on the right motion controller. The inputs are controlled in the SigPos.cs class, which inherits the abstract methods from VrInputMode.cs (ref. section 2.6).

Activate() This method is run when the application is started. The active station is loaded, along with the motion controller objects.

Update() This method is listening as long as the application is running. If a simulation is running, the EventHandler for recording data is invoked. If not, the method listens for user input.

Deactivate() The method is invoked when the application is terminated. The temporary graphic objects are removed from the controller and scene, and the recorded lists are deleted.

The sensitivity of the scroll on the touchpad is dependent on the length of the signal: $0.1 * signalList.Count()$. This makes it easier to scroll through the robots position in the case of long paths.

Chapter 4

Resulting system

The resulting system extends the VR functionality within Robotstudio. This chapter will present the solution in the following order:

4.1: The added functionality of the motion controllers

4.2-4.3: Each resulting graphical component of the solution

4.4: Presentation of the components working together for a predetermined path and signal.

4.5: Test case, using real data from a paint robot at ABB.

4.1 The graph window

The advantage of the graph window used in this solution is its high level of customization and adjustable interface. This is necessary as the signals to be displayed are arbitrary.

Figure 4.1: The graph window to be loaded into Robotstudio and attached to the controller.

Figure 4.1 is showing the resulting graph with an example sinusoid. The y-axis of the window is representing the process values, while The x-axis corresponds to the number of positions and process values recorded. The color of the graph goes from green to red as the signal value goes high and to blue as the signal value goes low. The x-axis and y-axis of the graph is automatically adjusted to match the value of the signal.

4.2 The arrow graphic

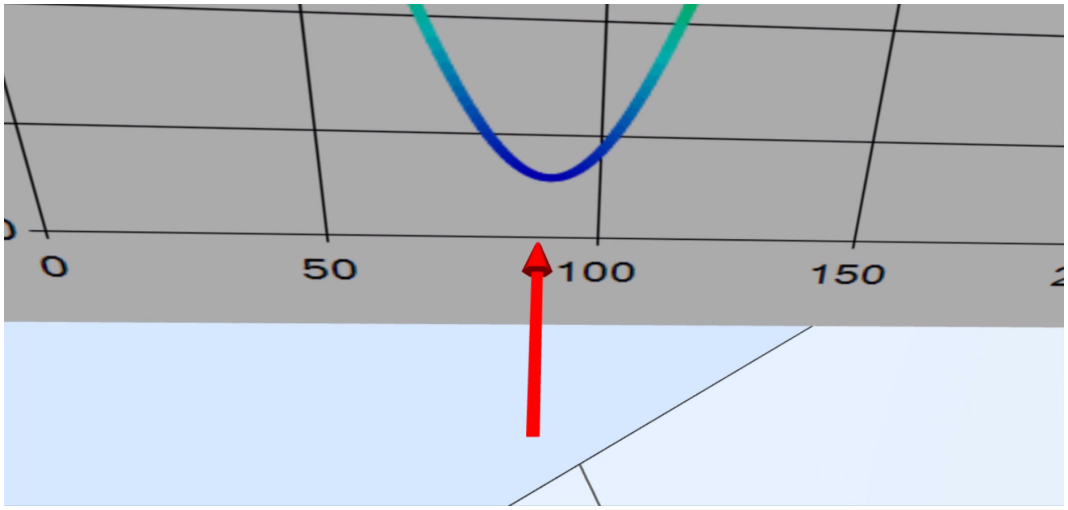


Figure 4.2: The arrow (red) pointing at the graph window.

The arrow is shown in figure 4.2 in red color. As the user scrolls the robot back/forward in position, it will move along the graph window. This way the user is able to keep track of what section of the signal he/she is in.

4.3 The motion controllers

The application is controlled by the inputs on the right motion controller.

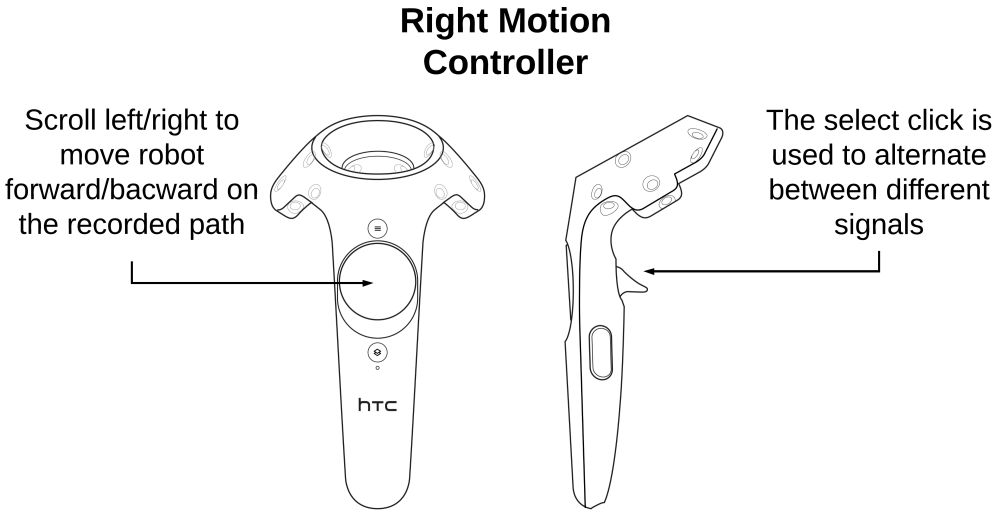


Figure 4.3: How to control the application with the right motion controller

On the left side of figure 4.3 the touchpad functions of the right motion controller is shown. By scrolling left/right the robot moves back/forward on it's recorded path. This will simultaneously move the pointer on the graph window accordingly.
On the right side of figure 4.3 the select-click for the right motion controller is shown. By pressing the button the graphics will adapt to the next process value dataset in the recording.

4.4 The resulting system

Here the visual results of the solution will be presented by running a simple path in Robotstudio along with a sinusoid signal.

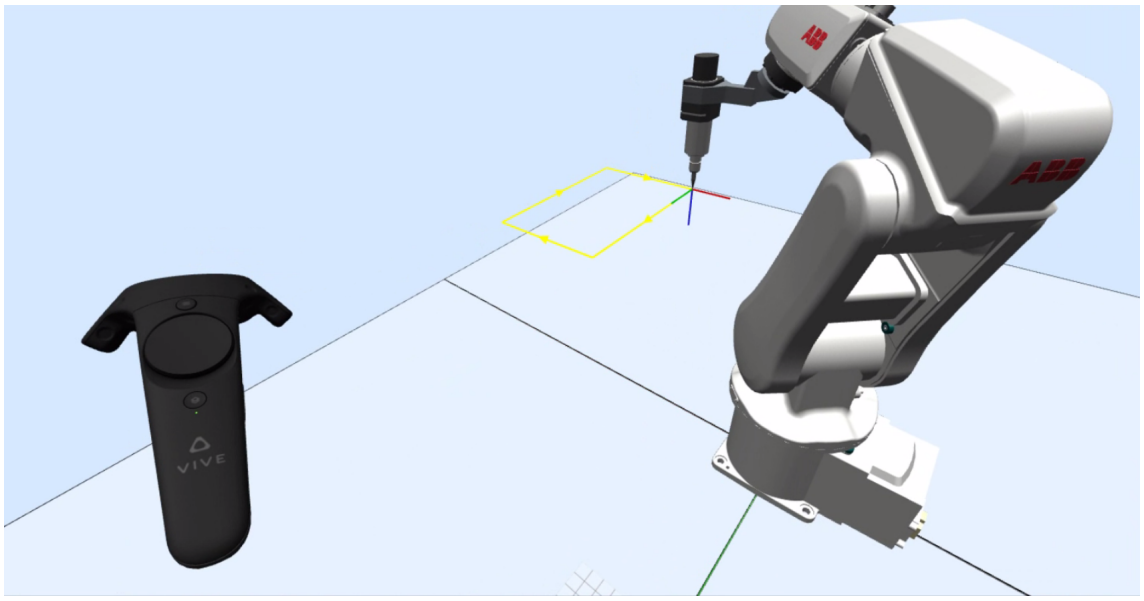


Figure 4.4: Image from inside the VR in Robotstudio showing the path to be run for testing the solution

The robot path used is created in RAPID for the purpose of performing a simple test of example signals to review the solution (shown in figure 4.4). The application is activated on the menu cube and the path is simulated along with the example signals.

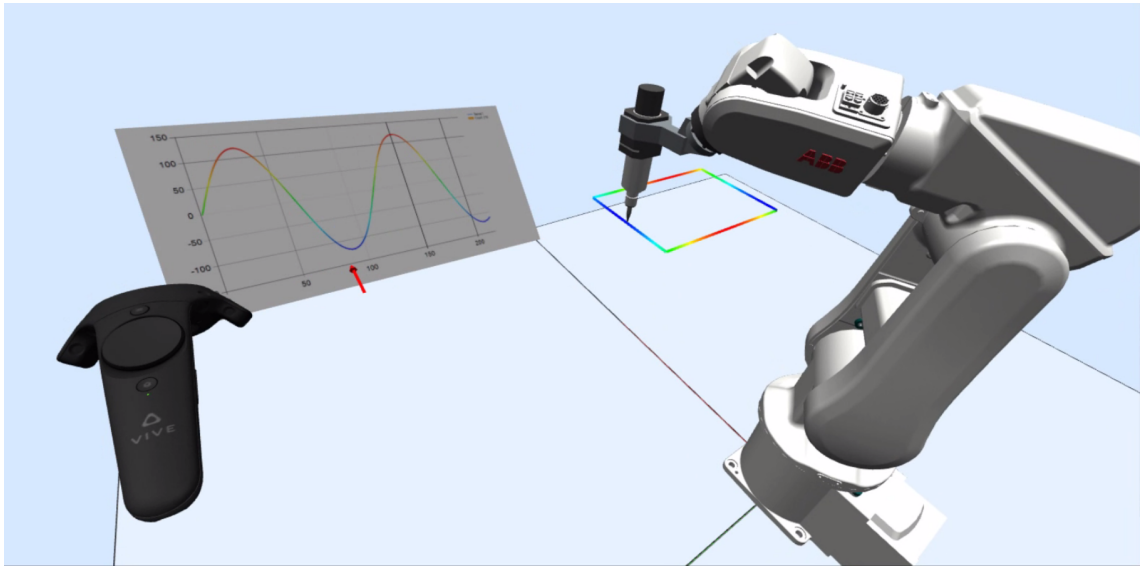


Figure 4.5: Post-run view inside Robotstudio showing the post-run resulting graphics

Figure 4.4 shows the new colorized path along with the graph attached to the motion controller after the simulation is run. The color of the sinusoid corresponds to the painted path of the robot. The user has scrolled the robot forward in position, as seen by the red pointer on the

graph. The axis and values written on the plot window is scaling according to the data value of the current signal it is visualizing.

4.5 Test case

The robot path and process data used in this test case is gathered from physical runs of a paint robot at ABB Bryne. The data is loaded in to Robotstudio and analyzed using the Virtual Reality add-in.
The path and signal is first loaded into Robview, which is a software used to view a robots path and process valeus from a PC.

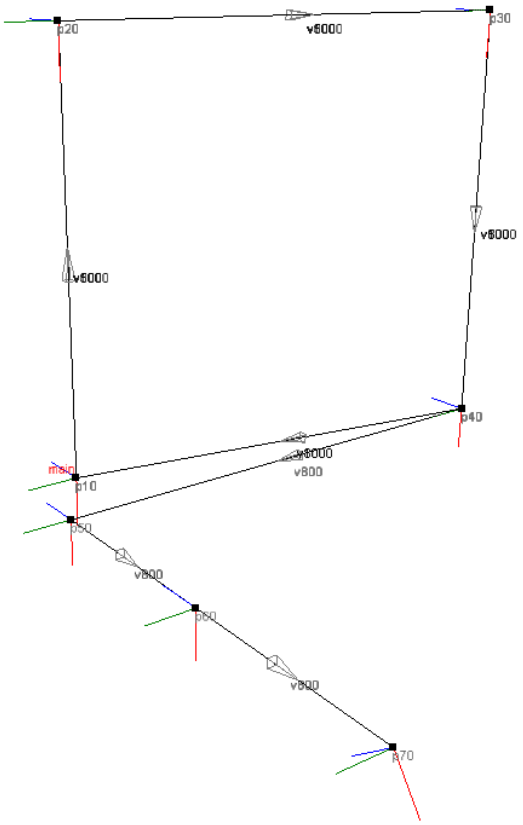


Figure 4.6: The path to be loaded into Robotstudio to perform tests on the application, seen from Robview

Figure 4.6 shows the recorded path of a paint robot at ABB. The tool of the paint robot moves to a frame and completes three rounds in a square pattern.

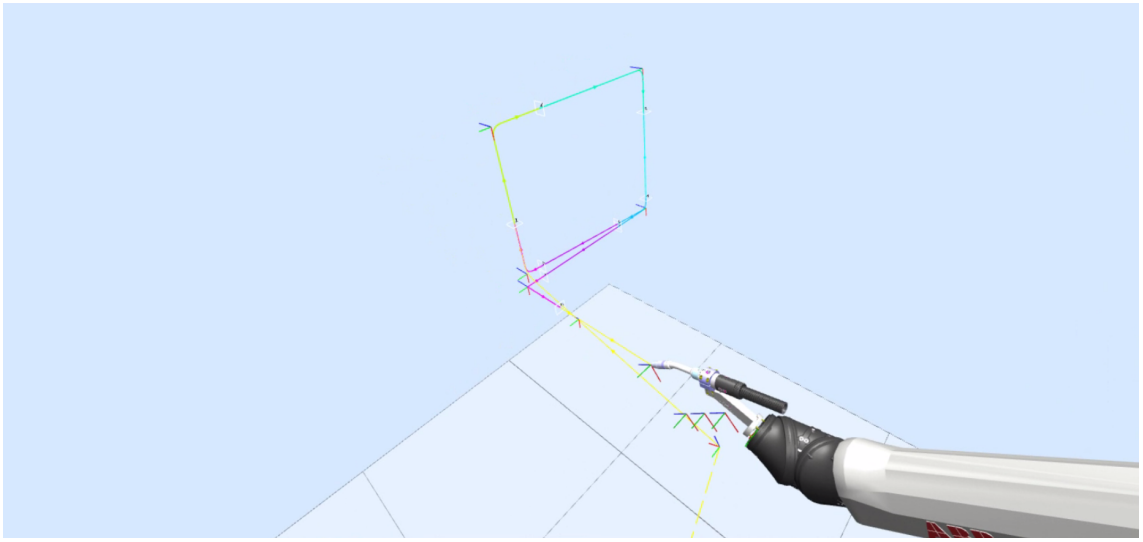


Figure 4.7: The path from the previous figure loaded in to Robotstudio

Figure 4.7 show the same path loaded into the 3D environment of Robotstudio, pre-simulation.

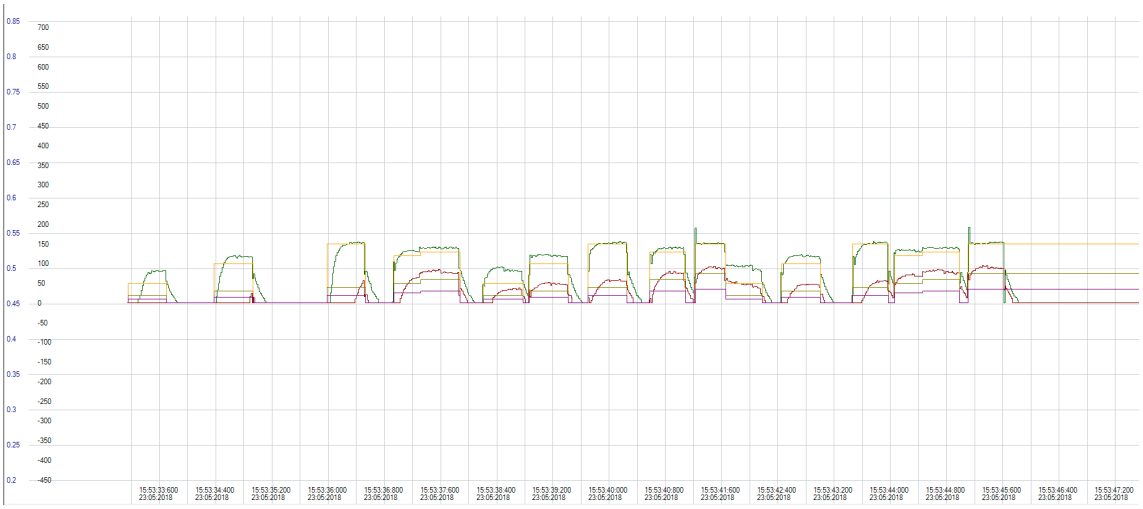


Figure 4.8: The process signals viewed from RobView

Figure 4.8 shows the process values recorded during the run of the paint robot. These are to be loaded into Robotstudio and simulated along with the path. The signals consists of inputs, in the form of step responses (yellow, purple), and the corresponding outputs, nano liter pr. minute (nl/min).

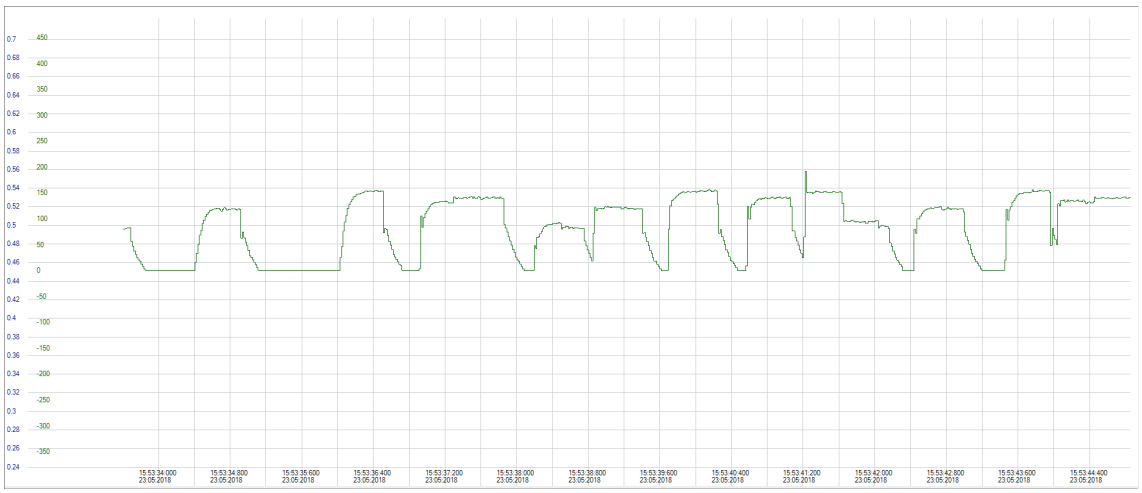


Figure 4.9: A single signal picked from the process signals to be compared with the image from Robotstudio

Figure 4.9 shows the signal selected to be displayed visualized inside Robotstudio. The signal

4.5.1 Rigid graph window

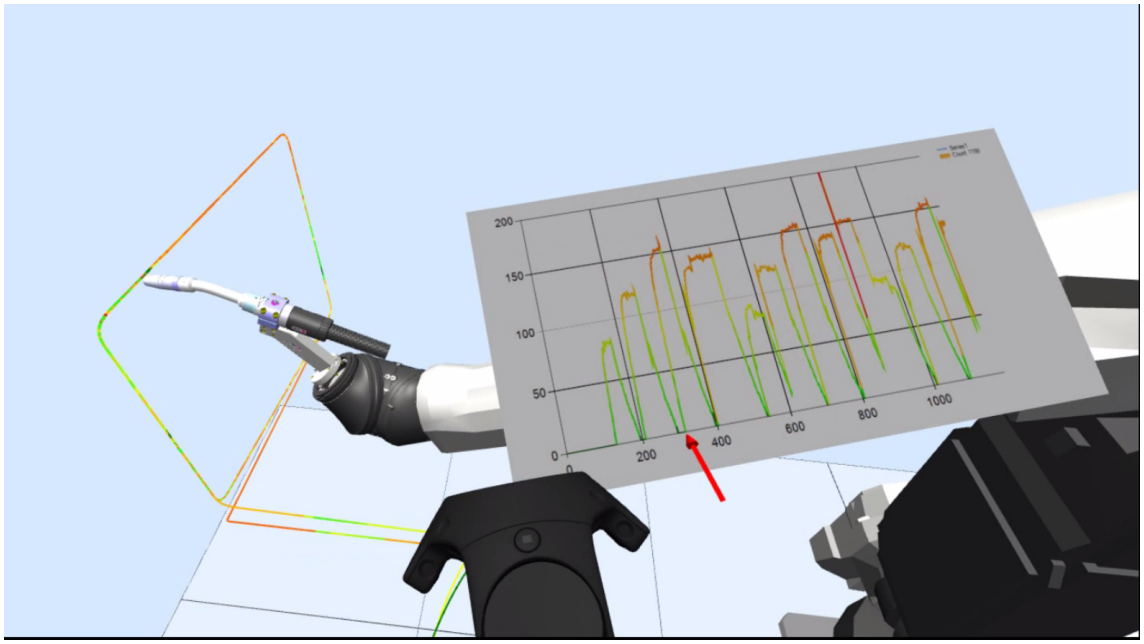


Figure 4.10: Post-run path and signal display

Figure 4.10 is showing the resulting colorized path and graph. The robot has been scrolled almost halfway into its first round, and the signal value corresponding to this position is shown by the red arrow.

4.6 Scaling graph window

The graphics window is extended for long signals, which results in a better reconstruction of the actual signal. The graph window becomes a larger object. Zooming in on specific sections of the graph is done by moving the motion controller closer to the HMD.

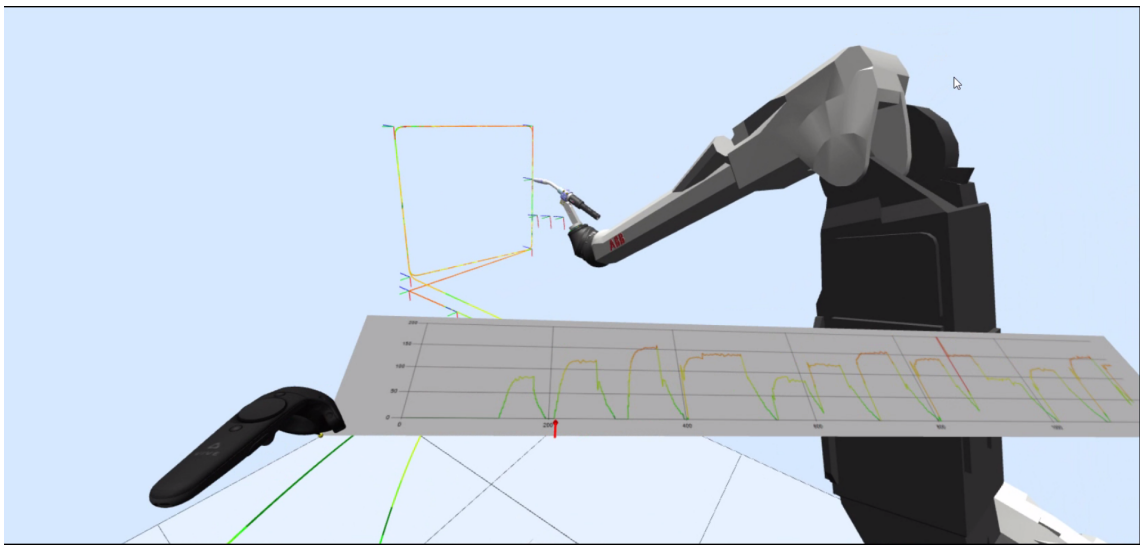


Figure 4.11: Post-run path and signal display for a longer path

In figure 4.11 the motion controller is held at a distance and the entire graph window is visible.

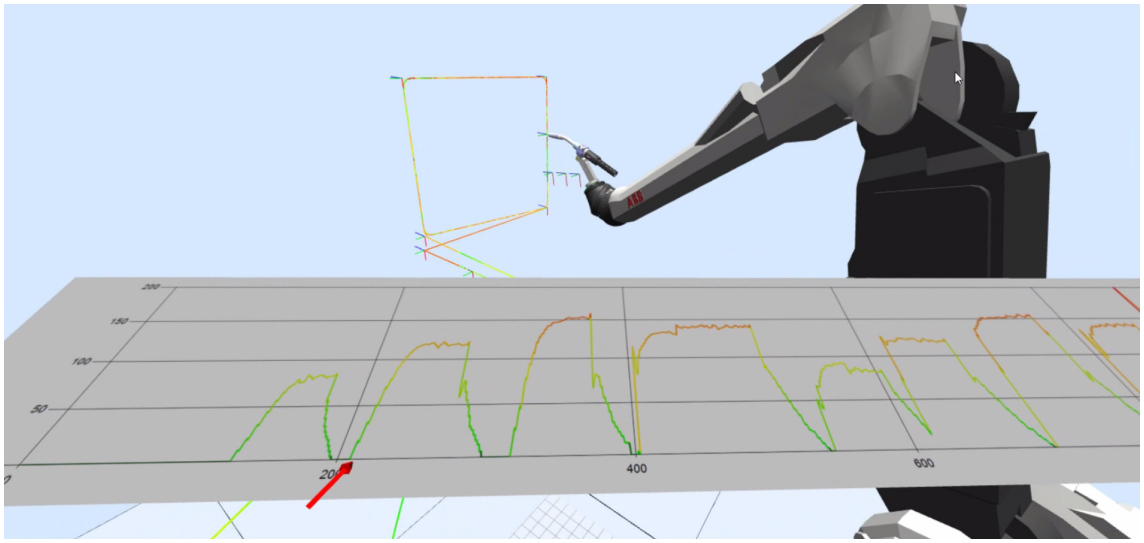


Figure 4.12: A closer look at the graph window from the previous figure

In figure 4.12 the motion controller has been moved closer for a better look at a specific section of the signal. The robot has been moved along the path by the scroll on the touchpad. The red arrow points at the process value for the robots position.

Chapter 5

Discussion

5.1 Performance

It is not easy to represent the visuals of solution trough pictures or video. In order to experience the solution fully, it has to be done using the HTC Vive. However, the results shows the solution is successful in visualizing the process values as a function of the robots path. The exact match between the robot path and the graph window is best shown in figure 4.5, where the signal displayed is a simple sinusoid. The position of the robot matches the pointer on the graph window.
The test case is ran both with a rigid graph window (section 4.5) and with auto scaling of the graph window (section 4.6). By extending the width of the graph window the recreation of the signal is improved. Zooming to view specific sections of the graph provides accurate information of the signal

5.1.1 Long runs

When presented with very long paths, the visualization is limited. The solution reaches a limit, where it is not possible to fit the processing data on the graph window. The graph window will either be too wide for the view , or the graph itself will become truncated and thus the recreation of the signal is misrepresented. The graph window itself is a rigid object once loaded into Robotstudio and cannot be manipulated. The solution does not support functionality for selecting and displaying specific sections of the current signal or path.

5.1.2 Overlapping paths

When visualizing a path that is looped several times, the color does not provide any visual information as the colors of the path are blending. A solution could be to have an option to only display the critical high values in the signal and leave the rest of the path transparent.

5.2 Further work

5.2.1 Warnings and errors

Scaling 3D objects to represent a signal increasing / decreasing or ”pop-up”-objects for specific errors/warnings can be added. The positional value of the warning/error could be recorded by invoking a log-message method inside the joint values listener.

5.2.2 Overlapping robotic path

If the robot runs the same path over and over, the color-scheme of the path will overlap and the visuals of the path itself will be limited. A solution to this could be to remove the portions of the path not selected in the graph-window.

5.2.3 Multiple graphs

When two or more signals are displayed in the same window, a scaling issue is introduced. The current solution scales the x- and y-axis based on the first signal loaded and adds the following signals on top of the graph window. If the values of the signals to be displayed differ significantly in value, the signals will not be fitting the window. This solution will not be adequate.
It is possible to display multiple process values in the graph window, however the solution presented in this thesis does not support functionality for selecting multiple signals.

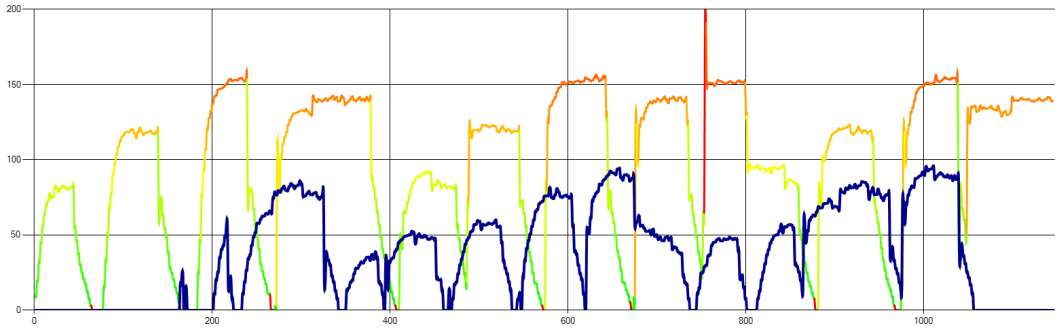


Figure 5.1: Chart graphic displaying two sets of data

Figure 5.1 is showing two of the signals taken from data sets of actual runs at ABB. Since the signals are of relatively similar value, the solution handles this well.
The resulting solution in this thesis does not support selecting specific signals for displaying multiple data series in the same window. Solutions for combining signals inside VR can be explored.

5.3 Conclusion

This thesis has presented a solution for displaying and analyzing signals in VR using the HTC Vive system. Being able to load a recording and look at the robots position for a given signal value provides a good overview of the system. Being immersed in the scene enables developers to validate robotic systems for issues that otherwise would not be detectable until physical prototyping.
To make Virtual Reality systems more relevant when it comes to development of industrial robotic systems, more work has to be done in extending VR UI and visualisation, like the solution discussed in this thesis.

Bibliography

[1] Sutherland, I.E. The ultimate display. In Information Processing 1965, Proceedings of the IFIP Congress, London, UK, 1965; Macmillan and Co.: London, UK, 1965; pp. 506–508

[2] Erin Carson. 9 Industries using virtual reality. https://www.techrepublic.com/article/9-industries-using-virtual-reality/

[3] Will GreenWald. The Best VR (Virtual Reality) Headsets of 2018. https://www.pcmag.com/article/342537/the-best-virtual-reality-vr-headsets

[4] Virtual Reality Simulation for the operating room: Proficiency-Based training as a paradigm shift in surgical skills training. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1356924/

[5] Science daily. https://www.sciencedaily.com/terms/virtual_reality.htm

[6] Binocular vision in a virtual world: visual deficits following the wearing of a head-mounted display. - Mark Mon-Williams John P. Warm Simon Rushton

[7] The Glossary of Human Computer Interaction, 22. Gestalt principles of form perception - Mads Soegaard. https://www.interaction-design.org/literature/book/the-glossary-of-human-computer-interaction/gestalt-principles-of-form-perception

[8] Understanding Virtual Reality: Interface, Application, and Design. Page 300 - William R. Sherman, Alan B. Craig https://books.google.no/books?hl=no&lr=&id=b3OJpAMQikAC&oi=fnd&pg=PP1&dq=virtual+reality+developer&ots=3EGUkluNUo&sig=66-futjH18ucM82t3jZ5HeP4YVE&redir_esc=y#v=onepage&q=virtual%20reality%20developer&f=false

[9] HTC Vive system requirements. https://www.vive.com/us/ready/

[10] Robotstudio. https://new.abb.com/products/robotics/robotstudio

Appendix

The appendix contains the 7z files and are attached below:

(a). Application Robotstudio Add-in with ReadMe

(b). Source code 7z file with complete source code

(c). Video Short demonstration video of the VR-view.