ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΜΗΧΑΝΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΤΟΜΕΑΣ ΒΙΟΜΗΧΑΝΙΚΗΣ ΔΙΟΙΚΗΣΗΣ &
ΕΠΙΧΕΙΡΗΣΙΑΚΗΣ ΕΡΕΥΝΑΣ

Development of a configurable Virtual Environment for studying Human-Robot collaboration

---

Ανάπτυξη παραμετρικού εικονικού περιβάλλοντος για τη μελέτη συνεργασίας Ανθρώπου-Ρομπότ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Οικονόμου Γεώργιος

Επιβλέπων καθηγητής:
Ναθαναήλ Δημήτριος

ΑΘΗΝΑ
Φεβρουάριος 2021

Ευχαριστίες

Αρχικά, χρωστάω ένα μεγάλο ευχαριστώ στην οικογένειά μου και όλους μου τους φίλους. Σε πολλές στιγμές στη διάρκεια της φοίτησής μου και ειδικά κατά την εκπόνηση της διπλωματικής μου, συνάντησα πολλές δυσκολίες. Όλοι ήταν συνεχώς δίπλα μου να με βοηθήσουν να συνεχίσω και να με υποστηρίξουν.

Από όλους αυτούς, ένα ιδιαίτερα μεγάλο ευχαριστώ στους φίλους μου Χρήστο Πάτσιο και Δημήτρη Παπαδημητράκη. Χωρίς εσάς, υπήρξαν στιγμές που θα πλησίαζα πολύ κοντά στο να τα παρατήσω. Ένα ακόμα ευχαριστώ στη Ζωή, για τη θετική ενέργεια και την υποστήριξη που μου παρείχες όσο πλησίαζα προς το τέλος της διπλωματικής μου.

Εν συνεχεία, ένα μεγάλο ευχαριστώ στον καθηγητή μου κ. Ναθαναήλ. Υπήρχαν αρκετοί συμφοιτητές που με προέτρεψαν να συνεργαστώ μαζί του και είχαν απόλυτο δίκιο. Οι συμβουλές του και σε θέματα διπλωματικής αλλά διαχείρισης ορισμένων καταστάσεων συνέβαλαν πολύ στο να φτάσω εδώ που βρίσκομαι τώρα.

Ευχαριστώ πολύ τον Λοϊζο Ψαράκη, που ήταν συνεχώς δίπλα μου σε όλες τις δοκιμές, και σχεδίασε και παραμετροποίηση όλα τα σενάρια. Ευχαριστώ πολύ και τον Αντρέα Μουρελάτο. Οι δυο τους με βοήθησαν πολύ να σκεφτώ εφαρμόσω τις ιδέες μου στα προβλήματα που καλούμασταν να λύσουμε, χωρίς να μου δίνουν έτοιμες λύσεις.

Ευχαριστώ τον κ.Γκίκα, για τη μεγάλη προσοχή στη λεπτομέρεια και τη βοήθειά του να κατανοήσω τον τρόπο σκέψης από την πλευρά του χρήστη, και να κάνω τα σενάριο που δημιούργησα όσο πιο σωστά μπορούσα, εργονομικά.

Τέλος, ένα ευχαριστώ στον κ. Alan Zucconi. Η επικοινωνία μαζί του, πέρα από τη δουλειά του πάνω στην κινηματική των ρομπότ, συνέβαλε στο να κατανοήσω γρήγορα και αβίαστα δύσκολα σενάρια κινηματικής.

Μακάρι όλα τα πρότζεκτ που θα αναλάβω στην καριέρα μου να είναι τόσο ωραία και οι συνάδελφοί μου να είναι τόσο ανθρώπινοι, όσο σε αυτή τη διπλωματική..

Abstract

Virtual Reality (VR) is nowadays a tool established as a powerful but cheap means of using a virtual world to create and study real world scenarios and situations.

At the same time, the advanced technology of robotics is being put to a lot of use, both industrially and individually, due to the robots immense capabilities for speed, accuracy and repeatability in movements and calculations.

This thesis's intention is to create an easily modifiable industrial working environment that implements human – robot collaboration (HRC) to complete simple assembly tasks, in a Virtual Environment.

The scenarios created for this purpose are intended to be as parametrical as possible, in order to be easily modified for future tests. At the same time they have the ability to calculate metrics and provide results based on those parameters.

The basic and most important options of these scenarios, are

- The robots' velocity
- The number of objects assigned to the person or the robots
- The placement sequence of the above objects for the assembly to be completed
- An anticipation indicator for the robots' movements and current targets

All the above have been designed with the primary result in mind, being their ability to be parametrical (easily changed/modified).

Περίληψη

Η εικονική πραγματικότητα, έχει καθιερωθεί στις μέρες μας ως ένα ισχυρό μα φθηνό εργαλείο που χρησιμοποιεί έναν εικονικό κόσμο για τη δημιουργία και μελέτη πραγματικών σεναρίων και καταστάσεων.

Ταυτόχρονα, η προηγμένη τεχνολογία της ρομποτικής έχει δει εκτεταμένη χρήση, σε βιομηχανικό αλλά και ατομικό επίπεδο, χάρη στις τεράστιες δυνατότητες των ρομπότ πάνω στην ταχύτητα, ακρίβεια και επαναληψιμότητα κινήσεων και υπολογισμών.

Η διπλωματική αυτή αποσκοπεί στο να δημιουργήσει ένα εύκολα παραμετροποιήσιμο βιομηχανικό εργασιακό περιβάλλον, το οποίο εφαρμόζει τη συνεργασία Ανθρώπου – Ρομπότ για την ολοκλήρωση απλών καθηκόντων συναρμολόγησης, σε ένα Εικονικό Περιβάλλον.

Τα σενάρια που δημιουργήθηκαν για αυτό τον σκοπό προορίζονται να είναι όσο πιο παραμετρικά γίνεται, προκειμένου να είναι εύκολα ρυθμιζόμενα για μελλοντικά τεστ. Ταυτόχρονα, έχουν τη δυνατότητα να υπολογίσουν μετρικές και να παρέχουν αποτελέσματα βασισμένα σε αυτές τις παραμέτρους.

Οι πιο βασικές και σημαντικές παράμετροι αυτών των σεναρίων είναι

- Η ταχύτητα των ρομπότ
- Ο αριθμός των αντικειμένων που έχουν ανατεθεί στον άνθρωπο ή τα ρομπότ
- Η σειρά τοποθέτησης των παραπάνω για την ολοκλήρωση της συναρμολόγησης
- Ένας δείκτης προσμονής για τις κινήσεις των ρομπότ και τους στόχους τους

Όλα τα παραπάνω σχεδιάστηκαν με κύριο επιθυμητό αποτέλεσμα, τη δυνατότητά τους να είναι παραμετρικά (εύκολα ρυθμιζόμενα).

Contents

1 Introduction

Nowadays, commercial use of VR technology is something that has become easily achievable, compared to previous years. Major factor to this result was the use of VR applications for the creation of video games and other forms of entertainment, which created a need in the market for VR technology to be easily acquired by individuals.

This low-priced technology first appeared in 2015, and since then, in conjunction with many development platforms that have implemented it in their environments, it has become a common option for research purposes by a lot of scientific fields, mainly the medical field, and training simulators.

At the same time, the study of the interactions between humans and robots, and the possibilities that this produces for advancements industrially, as well as in our everyday lives, has seen a lot of benefit from the above technology. Robots are very commonly used in our everyday lives, but also in industrial operations. Robots are strong units that can operate very fast and precisely, and have proven to be a great solution to very difficult tasks to be human performed, very mundane and repeatable operations, but also in very fast production facilities. The study of these interaction in a virtual reality environment, has proven to be an easily repeated, low-cost and safe method to optimize the functionalities of robots, as well as their anticipatory danger functions, and without actually consuming the robots worktime.

This thesis however is focused on the ergonomic aspect of the cooperation possibilities between humans and robots in an industrial working environment.

The scenarios created to simulate the above concern a basic assembly task of an electrical board with different kinds of devices.

The human has to collaborate with the robots in completing the assembly in the least amount of time and with as few collisions with them, as possible.

The robots used have 5 degrees of freedom and their movement is based on the implementation of inverse and forward kinematics. As a result modifications in any target’s position can happen instantaneously.

2 Human – robot collaboration in Virtual Reality

2.1 Human-Robot collaboration

Human-Robot collaboration is a part of the research and studying of Human-Robot Interaction. We currently live in an era of huge technological advancements, and the field of A.I. (Artificial Intelligence) and Robotics are no exceptions. Robots have become parts of our everyday lives, for both commercial and industrial purposes. HRI researchers are thus aiming to perfect the interaction between robots and humans, and to primarily establish and deliver a safe, high performance and robust environment of continuous cooperation and improvement, while at the same time optimizing costs and/or time spent on tasks, and making a human's task easier to perform.

Human-Robot Interaction is thus defined as: "A scientific field of study that aims to understand, design and configure robotic systems that are operate with or by a human." [A.C. Schultz (2008)].

HRI is a multidisciplinary field. Additions-contributions to this study are made by a variety of scientific researches and studies, in the fields of robotics, artificial intelligence, design and even social sciences.

HRC is therefore a child domain of the above field of study. Collaboration specifically occurs when humans and robots strive to complete a task that involves both of them, and which takes place in a domain that they share. There, sharing rules and structures, they approach or decide on situations regarding the interactive process that takes place in that domain.

The above marks the primary aim of HRC research, which is to establish a safe and efficient environment, where human and robots work together, and not just coexist, to complete their tasks. Robots have the upper hand in operations when it comes to speed, accuracy, efficiency, power and adaptability, whereas a human is more flexible, and has high dexterity and problem solving skills. As their collaboration combines all of the above and provides a lot more efficiency, providing rules to ensure a safe environment is essential.

2.2 Regulations and Standards

Granted the fact that robots apply very advanced and complex operations, their application had to be defined by a set of regulation and standards to ensure their safe function.

The standard ISO 10218-1 firstly in 2006 (and revised in 2011), introduced the concept of Human-Robot collaboration, with its main purpose to define the required precautions. In its introduction, ISO 10218-1 created the following definitions:

- Collaborative robot: A robot that is used in a collaborative operation.

- Collaborative operation: An operation in which a robotic design and a human-user together in a common workspace.
- Collaborative workspace: A workspace (including the operating segments) inside which a robot and a human can execute tasks at the same time, during a productive operation.

Additionally, supplemental regulation ISO/TS 15066:2016 focuses more on human-robot collaborative applications in industrial environments. Human-robot collaboration in a common workspace is allowed for the first time, though not without offering rules, regarding the environmental design and common workspace, with safety in mind. According to those, HRC can implement one or more of the below safety precautions.

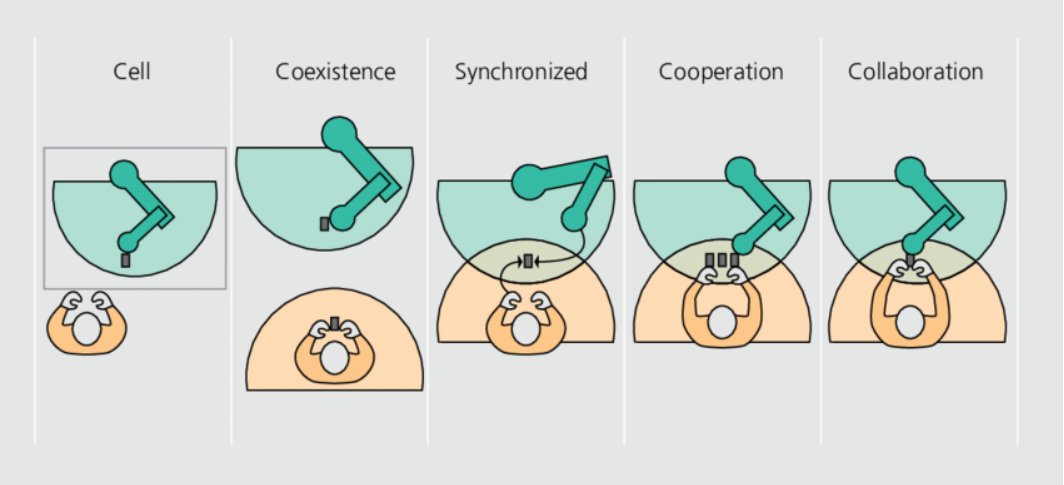
- Safety based observation and interruption: A method used to stop the movement of the robot while inside the common workspace, before the user enters it in order to interact with the robotic system and complete a task.
- Hand based guidance: In this method of operation, the user operates on a manual device to transmit movement commands to the robot.
- Speed and separation observation: Here, the robotic system and the user can work simultaneously on the common workspace. The danger of collision is avoided in this scenario by always keeping a minimum amount of distance between the robot and the human.

- Force and power restrictions: In this method, physical contact between the user and the robot is allowed to occur both intentionally and not. Any danger possibilities are controlled either by safety precautions implemented in the robot, or by an external safety control system. The latter keeps measurements that are related to the robotic system under certain values, which are obtained during the danger evaluation.

2.3 HR levels of interaction

Humans and robots can coexist with one another and work together in a variety of tasks. This is not always achieved safely by the same type of interaction between them. In the figure below the different levels of interaction (Bauer et al. (2006)) can be observed, varying from simple coexistence and individual workspaces, to not only operating in a shared workspace but also cooperating and implementing teamwork to complete specific scenarios· these levels of interaction are also defined as:

- Cell: The robot is contained in a traditional safety cage, cooperation is not so direct.
- Coexistence: The robot and the user exist in the same space, with no safety cage, but they do not share their workspace.
- Synchronized: The robot's and the user's workspaces are connected. However, only one of the two is allowed to operate on the shared part, at a time.
- Cooperation: Both the user and the robot can operate on the shared workspace simultaneously, but they cannot work on the same product/object.
- Collaboration: The user and the robot can share the workspace and also work on the same object simultaneously.



Bauer et al. (2006)

2.4 Ergonomics of HRC

One of the bigger problems that robots need to tackle and improve on, is the ergonomic aspect of HRC, whether a robot operates in a home or in an industrial or production environment. Achieving a more ergonomic functionality in any task provides a better experience for the user. Also, in the long run, and especially for an industry, a faster production would be achieved, and less physical and mental strain would affect the employees.

Ergonomic benefits are directly connected to the user experience. Areas the see benefits from a workspace that is designed better ergonomically and implements robots are briefly described here:

- Postural optimization: Robot assistants are primarily designed to help humans perform task that they previously found tiring or could not complete at all. One of the main causes for this, is that in certain everyday tasks and even more so in a productive environment, the tasks that are to be performed inflict strain on the users' posture. This issue, combined with other health issues a user may already have, or simply a user's age, factors heavily in the his/her ability. Static or unnatural positions and postures, far reaching positions, overhead or very low position of working, are primary causes of discomfort in a user/worker.
- Physically demanding operations: One of the main issues encountered that heavily affect both a worker and the overall production are physical injuries. Heavy lifting of big objects, continuous lifting or reaching for

objects in a production line, and even falls, after using a forklift or reaching for high shelves, are actions performed by people that can prove dangerous and damaging.

- Hazardous environments: Especially in an industry or research facility, working in unhealthy environments is very common. Radiation, contamination, very bright lights, heavy noises, or operations that produce particles mixed with the breathing air, or that come into contact with a person's eyes, nose and ears (for example, a welding operation), despite inflicting an immense amount of strain on a human's health, have no implications to a specially designed robot, that can be instead operated by a user in a safe environment.
- Mentally draining operations: Long hours in very repeatable operations, in a fast product line, operations that require a lot of calculations with speed and precision, or fast thinking, can inflict a lot of fatigue on a person. This can lead to mistakes, that could produce a faulty product, or maybe errors that can be seen in long term results and evaluations. At the same time, strain in a person's mental activity can add to the physical demand of operations, and increase the possibilities of physical injuries.

Tackling the above can help humans work on their strengths instead.

2.5 Virtual Reality

VR refers to Virtual Reality. This technology immerses a human into a computer generated world, and was a concept introduced by Rheingold (1991). Using this technology, a human could not only experience real world environments and locations, but could also study non – realistic microscopic or macroscopic scenarios. Even at the time though, a head mounted display (HMD) was not an unknown technology. It was developed in the 1960s and introduced as a display option in 1965 (Sutherland).

It was as late as 2015 that the technology became available to the public, with mainly entertainment purposes in mind. It was however being used extensively already by many scientific fields, and mainly the medical and engineering field. In addition, it had a lot of purpose and was found very useful in military training, and even more broadly in simulation running. It was also a major option in education and even rehabilitation and psychological evaluation and research.

Testing and evaluating Human-Robot collaboration in a Virtual Reality environment becomes relevant due to its simplicity. Especially after its major success in the entertainment market, it has become a very easy to use device. Installing and modifying it to suit any experiment is very well documented for every device by its producers, and programming it is easy as well. At the same time, setting up such an environment can be achieved in a small space, and only requires a somewhat capable computer.

Using virtual reality as a tool removes the need for actual robots. Providing, transporting and setting up robots for these tests would be an expensive and tedious task that would also require a lot of space. At the same time, an industrial robot has pre-programmed capabilities, meaning that its installments, freedom of movement, speed and power, precision, cannot be changed, even if its programming can be altered or modified in some ways. VR removes this barricade. A robot model here can be altered on the spot, adding all kinds of functionalities, and even modifying its appearance, adding different end effector tools, adding or removing joints, simulating different kinds of movements. This proves most powerful during test, where problem that have not been foreseen might appear, or if a robot model has to be used for more than one scenario, each of them serving a different purpose. It was as late as the 2000s when the first application of robotics in visual reality was introduced. Visualizing and designing a VR environment where a human could experience and train with robots without actually coming into contact with them proved to be an effective way of education. It was resulted in being a very low-cost and safe way of introducing the idea of human-robot collaboration to a human, in both physical and mental aspects of a person's health, and at the same time teaching him how to operate or cooperate with robots, that have high speed capabilities and very accurate operations.

Except from the low – cost and accident free training of a human, before he actually comes into contact with a robot, simulating HRC in VR also played a major part in a more ergonomic and optimized environment. Having workspaces assembled and operated in virtual reality simulations could greatly reduce an operations design errors, resulting in lower effort required for tasks to be completed, and better optimized actions and tasks for the occupied person, resulting in minimizing a task's completion time, by making it easier to understand and rendering the actions needed less stressful to complete.

2.6 Task Design

Creating a human-robot collaborative task in a virtual reality environment is a challenging task. These are mainly described below:

- The task has to be as close to realism as possible, being similar to real life application while being engaging at the same time, so the user can maintain focus on the task.
- The participants need to be able to understand and/or feel when a collision occurs.
- Participants have to understand and believe that there was actually collaboration with the robot.
- The task should be short, to prevent motion sickness from the head mounted display, but also long enough, for the metrics of the scenario to be able to provide sufficient amount of data.

3 Hardware-Software Setup

3.1 VR Hardware

In our lab, theses studying the concept of human – robot collaboration up to this point, were using a motion tracking and head display solution. However, as technology saw great advancements at the same time, before starting this thesis a commercial device was provided instead. The VR solution used is the Oculus Rift, developed by Oculus in 2016.

Oculus Rift with motion sensors and touch controllers – from Oculus website)

The above VR hardware proved very easy to implement and gave the ability to be used instantly in the 3D application environment. At the same time, the motion sensors give the ability to track movement in a certain space and not only shoulder to hand motion.

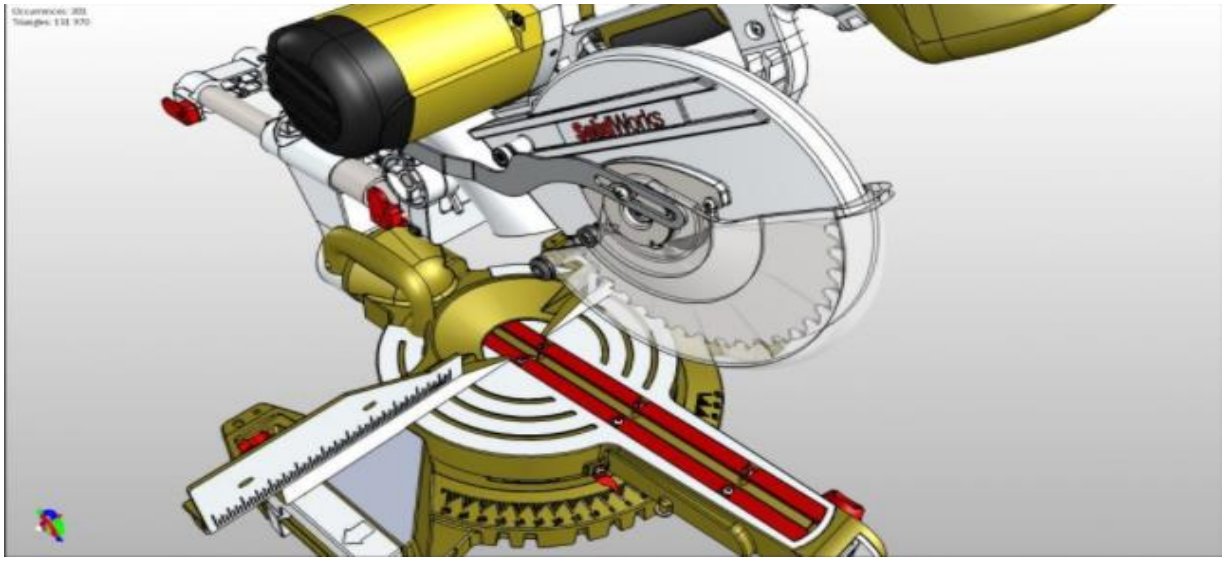
3.2 Unity for the 3D environment simulation

Unity is a cross-platform game engine, introduced in June 2005 as a game design solution. It has since then evolved and extended to support a great amount of platforms, and has seen a big variety of applications, for games, animations or film, but at the same time for architectural, engineering, construction and manufacturing purposes.

It is widely used for creating and studying real life scenarios, and its ability to implement large CAD assemblies make it an amazing tool for the engineering or manufacturing fields. The above solution was achieved by Unity's partnership with PiXYZ.

Unity was a tool used in this laboratory before the thesis took place, so additional experience along with its many great features provided an easy implementation of the Oculus Rift device into the 3D environment that was created.

For the purposes of this thesis, Unity version **2019.2.f8** was used.



CAD imported assembly (from Unity website)

3.3 Additional Software

Unity is a game engine based on the programming languages **C++** and **C#**. While Unity itself provides a great amount of tools and features, the logic and behavior of the models in any environment can be greatly upgraded through scripting.

For the purpose of this thesis, the programming language used for scripting is **C#**, and the software used to develop on is **Microsoft's Visual Studio 2017**.

In addition, while many of the objects presented in each scenario where created in Unity, since this thesis does not have 3D modeling as a primary focus, some models where imported from **Blender**, while the robot model was imported from a **SketchUp** export (Petr P.).

4 Robotic arm movement

As mentioned before, the robotic arms used in the simulation are 5 dof (degrees of freedom) models.

Up to this thesis, when running a HRC simulation, the movement of the robots was based on animations, thus making it time consuming to alter the environment, as it resulted in having to redo the animations, for the new positions the end effector of the robotic arms was supposed to target.

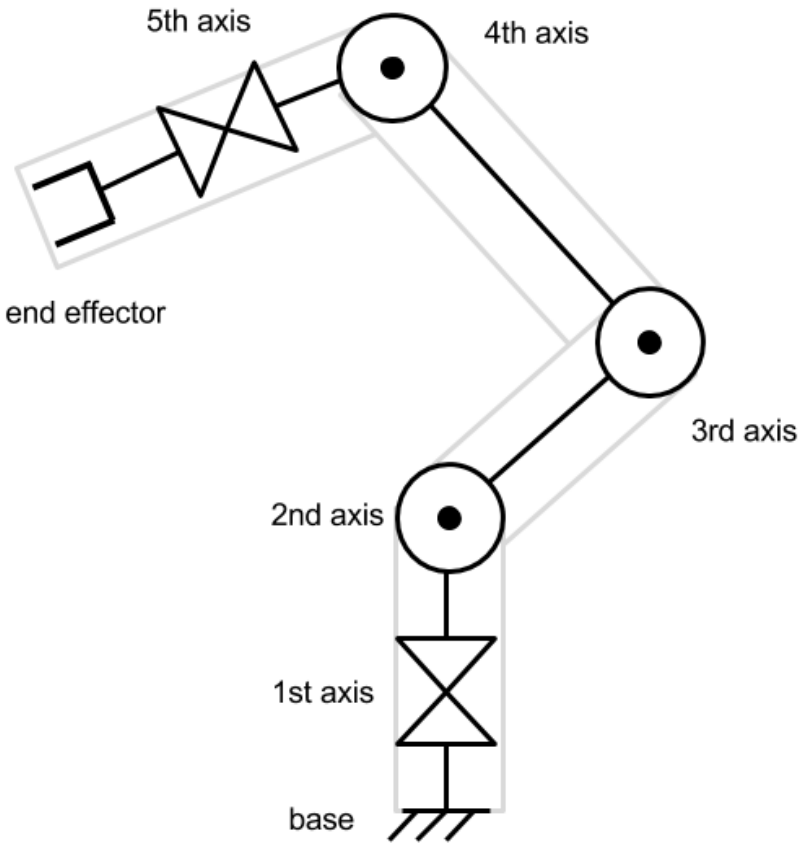
Therefore, to improve on this, the robotic arms in this thesis implement inverse kinematics (IK) and forward kinematics (FK). As a result, assigning a new target position to a robotic arm is now as simple as moving the target object in the 3D space. It is also possible for the robots to pursue a moving object within their range.

Also, with more details to follow, the use of 2 robotic arms instead of 1 that was being used so far, served as to improve the complexity and interaction that was achievable in a working scenario, while in the meantime the 2 robotic arms can be programmed to execute different kinds of operations, creating even more possibilities.

Since the kinematics focus on the reaching movement, of the end effector to the target location, the end effector does not have any movement itself and does not implement grabbing, so it is not considered a joint or the 6^{th} degree of freedom, while it could, in different context.

4.1 Forward Kinematics for robotic arms

A 2D schematic of the robotic arms would be the following.

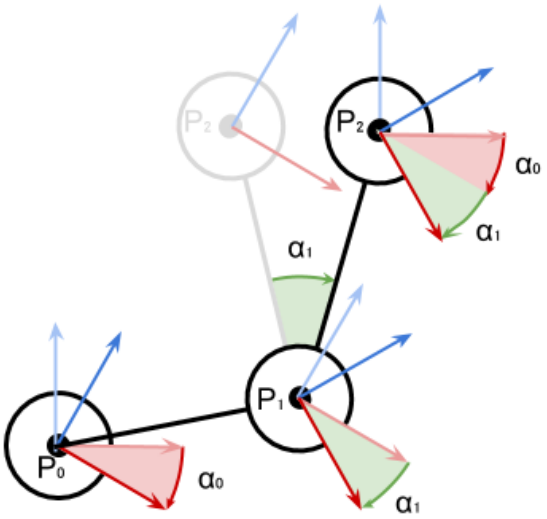


Robotic arm sketch (from alanzucconi.com)

Forward Kinematics for a robotic arm starts by knowing the position of the end effector in space and are given specific rotations for all the joints of the robotic arm.

Then, by applying these joint rotations, we are given the result, which is the new position of the end effector.

In a robotic arm, limbs and put together by the robot's joints, therefore a chain is created. As a result, when a joint is rotated, all the joints that are part of the continuation of this chain, starting with the position of the joint rotating, are all also affected by this action. Visualizing this action in a 2D sketch would look like the following



Example of rotating joints (alanzucconi.com)

As shown above, a $\alpha 1$ rotation of $P1$ joint would affect $P2$, but a $\alpha 0$ rotation of $P0$ joint would affect both $P1$ and $P2$.

Mathematically, these rotation would look like this:

$$P_1 = P_0 + rotate\,(D_1, P_0, \alpha_0)$$

But for the second joint, we would have to calculate both rotations, so it would look like this:

$$P_2 = P_1 + rotate\,(D_2, P_1, \alpha_0 + \alpha_1)$$

Generalizing the above in order to apply it to all joints the following is derived

$$P_i = P_{i-1} + rotate\left(D_i, P_{i-1}, \sum_{k=0}^{i-1} \alpha_k\right)$$

In the above examples, Di is used as the distance between the 2 joints.

4.2 Forward Kinematics in Unity

Unity has a very important feature that allows for the creation of this chain between the joints and limbs of the robotic arm. This is called “Parenting”.

Parenting allows us to assign an object as the child of another. A child object automatically inherits the position, rotation and scale of the parent object. By changing the position or rotation of the parent object, the child object is also moved. At the same time though, the child object has its own local coordinates. Changing those will affect the position/rotation of the child and all subsequent children, but will not affect the father, therefore the above mentioned kinematic chain is achieved.

(An image of the robotic arm parenting configuration in Unity

To apply the given rotation, we use Unity’s **Quaternion.AngleAxis(angle, axis)** function. Quaternions are mathematical objects that represent rotations, and are much easier to use in Unity that Euler angles. The **AngleAxis** functions rotates a single joint of the robotic arm by the specified degrees, in the specified axis X,Y or Z.

Our point starts from the base and adds the joints’ rotations to itself, while also adding the distance between the current and the previous joint. Finally, when all the joints have been rotated according to the solution, our point will actually be position of the effector.

4.3 Gradient Descent

During the scenarios, the need for the robots is to have the result position in space as input, and afterwards calculate the appropriate rotations for each joint, in order to reach that point. Thus, the need for inverse kinematics appears. Inverse kinematics in this thesis are applied using a simple implementation of the Gradient Descent algorithm, as analyzed in programming with C#, in Unity, by Alan Zucconi.

Gradient descent was preferred for this thesis, as its research provided more information and it proved easier to implement in comparison to another common inverse kinematics method, the Denavit-Hartenberg matrix.

Gradient descent is an optimization algorithm. It takes a function F() and leads to its local minimum. The idea is that we have to move in the opposite direction of the approximate gradient, because it will lead to the steepest descent, thus reduce the value of the function faster.

While the gradient is connected to the derivative of a function, it is not the same. It could be described as a directional derivative. The derivative of a function is given with the use of a limit :

$$f'(p) = \lim_{\Delta x \to 0} \frac{f(p + \Delta x) - f(p)}{\Delta x}$$

Depending on its value, the result that can be obtained is that:

- $f'(p) > 0 \Rightarrow f$ is going up, locally;
- $f'(p) < 0 \Rightarrow f$ is going down, locally;
- $f'(p) = 0 \Rightarrow f$ is flat, locally.

Because the derivate requires the usage of a limit, our gradient will be calculated as an estimation of the derivate, like so :

$$\nabla f(p) = \frac{f(p + \Delta x) - f(p)}{\Delta x}$$

The value of the derivative will point towards the steepest ascend from the point we are currently in, so we are heading towards minimizing our function as long as we move in the opposite direction.

All that remains is to update our point p_i by $-L\nabla f(p_i)$,

where "**L**" is a constant called **LearningRate**. This will be what we use in the scenarios to control the speed that the robots' joints are rotating with.

4.4 Implementing Inverse Kinematics in C# code

To implement the above described inverse kinematics logic in our code, it is also required that a forward kinematics logic is used, for evaluation purposes. A brief description of the implementation follows below.

We firstly need an array that contains all the current rotations of each of the joints of the robot, which in our case is defined as "**Solution[]**" .

After getting a target location, we cycle through each one of the robot's joints, starting from the top.

Taking the Solution[i] value for the **i-th** joint, we increase it by a constant value. Using forward kinematics, we calculate the distance of the effector with the current rotations, and afterwards we calculate the distance the effector would be in, if the above rotation of the **i-th** joint was to take place.

If the current distance is **f_x**, and with a **delta = const > 0** rotation we get a new hypothetical distance **f_x_plus_d**,

the gradient would be : **(f_x_plus_d – f_x)/delta** .

Our desired new value for **Solution[i]**, after restoring its initial value would then be

Solution[i] = Solution[i] – L * Gradient, where **L** is our **LearningRate** of choice. The code for calculating the gradient is shown below.

```
public float CalculateGradient(Vector3 target, float[] Solution, int i, float delta)
{
    // Saves the angle,
    // it will be restored later
    float solutionAngle = Solution[i];

    // Gradient : [F(x+h) - F(x)] / h
    // Update   : Solution -= LearningRate * Gradient
    float f_x = ErrorFunction(target, Solution); // ErrorFunction is DistanceFromTarget!!!!

    Solution[i] += delta;
    float f_x_plus_h = ErrorFunction(target, Solution);

    float gradient = (f_x_plus_h - f_x) / delta;

    // Restores
    Solution[i] = solutionAngle;

    return gradient;
}
```

Calculating gradient, as implemented by Alan Zucconi

The kinematic algorithm also implements a threshold for stopping. This threshold is the value of the distance that has to occur between the end effector of the robot and the target position, in order to assume that the robot has reached its destination. Otherwise, because the robot makes iterations in every frame, it would be very possible that it would overreach its target position, which would cause it to wiggle around it, never actually reaching it.

5 Environment configurations

In the Unity engine, each model that is included in the scene (a Scene represents the total of the environment) is considered an object. That includes the floor, walls etc.

Objects in Unity can be assigned components, which allow them to perform an extended number of operations and hold a variety of properties.

The **Transform** component is automatically assigned to a new object, as it contains all the info regarding its position, rotation and scale in space, both locally if it is a child object, and globally.

Some other important components are the **Collider**, which allows Unity to detect collisions between objects, the **RigidBody**, which gives a physical aspect to the object and instructs Unity to calculate kinematics and dynamics on the model, the **Material**, which gives object properties like color, the **Animator**, **Mesh** etc. Another really important and much used in this thesis component, is the **Script**, which allows the user to program more functions for a specific model or groups of models through coding.

5.1 The initial environment

The initial idea for this project was that an HRC environment would be created, where a human and a robot would cooperate in an assembly task in a shared table. All the objects would be grabbable by the human for the assembly to work.

Since the robot would be moving by implementing an inverse kinematics algorithm and not by animations, it had to be assigned the targets at startup. For this reason, the robot is made to carry a **List of GameObject** (the target objects), which is defined in the **Start**() function of Unity (Start() is a Unity function that is called once for every object, after it is initiated).

The robot removes the first GameObject of the list every time. Accessing its Transform component, it uses the inverse kinematic algorithm to reach its position. The object also has other components that declare its drop location and orientation.

The above meant that the robot has to **Grab** and **Drop** an object. In the meantime, the human has his/her own objects to place in the table and can also interact with the robots.

5.2 Increasing the complexity

Increasing the complexity of the environment, to make it accept more parameters and require more focus by the human was required.

To address this, we decided that the environment would implement 2 robots instead of one. Both robots implement the same movement algorithm, but they execute a slightly different task.

The task at this point was decided to be an assembly task of an electrical board.

To this thought, the first robot was assigned to place electrical devices on the board, like resistors and capacitors. The second robot places screws on the board, on top of objects placed by the human.

The human's task is to place his own list of objects, one at a time, while being careful of the robotic arms. When the human picks up the designated object, its location on the board is rendered.

5.2.1 Adjusting Grabbable Objects

In order to reduce the complexity of the above changes, only the human's initial targets are **Grabbable**. In addition, he can only Grab the object that he is supposed to grab, defined by the task's order of object placement. While the object is grabbed, he can see its target location on the board, and only after he places the object, is he able to grab the next one.

Programming wise, there were some adjustments made to the oculus integration, because in the version used by this thesis, there was not an option to render an object Grabbable and NOT Grabbable at will.

5.2.2 Handling Collisions

On the context that 1 human and 2 robots are now participating in the assembly, there are two kinds of collisions that can occur. The first one is a collision between the human's hands and a robot, and the second is a collision between the two robotic arms, which is not realistic in a productive environment and has to be avoided.

After a robot and a human hand collide, at first the robot would stay still until the human would remove his hand from the specified collision range. As this was not safe enough, on collision, both robots retreat to a designated retreat position. A button was firstly placed near the assembly table, and when it is pressed, the robots would continue their task. The human could still place the object he was currently holding in the designated slot, but until he pressed the button, he was not able to pick another one, and at the same time he was not able to pick the current one again, if he released it in a position other that the designated one.

The traditional start-stop method described above provides safety and at the same time gives the user more control over the pacing of the scenario. However, collaborative robots (Cobots) nowadays implement a lot of sensors in their functionality and can safely know when the workspace has been cleared.

To simulate the above functionality, the start-stop button was ultimately removed. In its place, the robots will still retreat to the designated safety position. Upon reaching it, however, they will instantly continue their interrupted task, as they will assume that the workspace has been cleared.

The robot – robot collision has to be handled as a near collision, in order to be more realistic. To this end, the robots have a volume bigger than their own (in order to not actually touch), which is specified as a collider only for the specific kind of collision. Once this occurs, the robot that is the furthest from the electrical board stops, and only the other robot continues its task, until the aforementioned volumes stop touching.

5.2.3 Achieving Cooperation

To achieve the desired level of cooperation, a degree of verticality was given to the assembly. The robots and the human can place and/or screw objects on top of each other. For that to happen, sometimes a robot has to wait for the human or for the other robot to complete an action, and also the human may have to wait for the robots to place an object, in order for his target location to render his target.

To achieve this, the objects assigned to the human have a new property that designates if they have to wait for a certain object to be placed, in order to activate the target locations rendering.

In addition, the robots both have an intermediate position (Idle Position) in the form of an empty object. After grabbing an object, they pass this new location. Before moving on the board, they check if they need to wait for an object to be placed before they continue. If they do, they are rendered idle until that object is placed.

Finally, to complement all the above, each object that is to be placed on the board, has a property called "**IsDone**" that designates when its placement is completed.

6 Calculating the metrics

For the calculation of the metrics, it was decided that the parameters studied would be based on :

- The speed of the robots' end effector
- The anticipation on the robots' movements

6.1 Creating Options for each task

An object called “OPTIONS” exists in each of the scenes, that contains the parameters that have to be selected or activated/deactivated before running the task.

The OPTIONS object contains the LearningRate parameter of each robot, which controls the speed of its joints’ rotations. By altering this, the robots operate on a different average and maximum speed of the end effector.

This object also contains a setting for the name of each participant. Finally, it contains a Boolean that designates if the task is to run with Anticipation enabled or disabled.

6.2 Anticipation

Anticipation in these scenarios refers to the following adjustment.

In the case in is enabled, the target location that each robot is moving onto at the current time, will begin flashing until the object is placed on it and the “IsDone” property of it will be set to true. Afterwards, the next target of the robots will begin flashing and so on.

The locations targeted will be flashing with a different color, to designate which robot is targeting it. In addition, the second robot, which has the screwing function assigned to it, will make the hole that it is targeting with a screw, flash.

6.3 Logging

In order to log all the calculations made during runtime, a separate logging class was created, that is called by different scripts on different objects, to write specific calculations.

These logs contain:

- The real time that the task was started on. At this point a timer starts counting the seconds the task is running.

- Time of each near-collision of the robots, in seconds since the startup.
- Time of each collision of a robot with a hand, in seconds since the startup.
- The delay each of the above collisions applied on the task, in seconds.
- The total idle time of each robot.
- The count of total near-collisions of the robots.
- The count of total robot-hand collisions.
- The time it took for each robot to complete its task, in seconds.
- The time it took the human to complete his task, in seconds.
- The average and maximum speed each of the robots achieved, in m/s .
- The time of completion since the startup, in seconds.
- The total time the human was idle, meaning that he was outside the workspace.

It is important to note here that the task is considered to have started when the human picks up the first objects. Before this occurrence, the robots are inactive and the timer has not started counting.

6.4 Additional adjustments

In order to provide better feedback to the user and get more realistic reactions, and as a result metrics, about the task or events that have occurred, some additional adjustments have been made:

I. The object that next in order for the human to grab is flashing green as an indicator.
II. A sound is implemented when the human places an object in the correct position.
III. An alert sound is played if the human's hands collide with any of the robots.
IV. If a collision has occurred, the continuation button flashes instead of the next human target object. Only one flash at a time is used as an indicator for the human to take action.
V. A clicking sound is played to designate the pressing of the button by the human.

7 Algorithmic Logic in coding

As this thesis main purpose is to have the creation of scenarios for future testing be as parametric as possible, it is a big project code wise. Therefore, some further explanation of the logic behind the main functionalities were the means to this end, should be provided.

7.1 Transporting robot

This robot (Robot 1) transfers electronic devices from its picking table to the working electrical board. It implements forward and inverse kinematics as previously described, for its movement. In addition it has some functionalities for implemented a sequence of actions and placing many objects in a specified order.

At the start of a run, it is provided with a list of game objects that consist the total of its targets. It will iterate through this list each times it places an object on the table to find its new target. After the last placement, the list will be empty, and the robot will log its metrics and retreat to a specific position. However, if the human has not picked up his/her first object, the robot will not begin its task, even if there are no pre-requirements to its first object's placing. After it starts, it continuously checks the distance its effector has from its current target, and moves accordingly. The target itself is not always the object it has to transfer to the board. The robot passes through 4 different states that change its targets accordingly, before going for the next target:

- Picking up the target: At this stage, the robots position is the position of the target it has to transport to the board.
- Picked up the target: After the target is picked up, the robot targets the intermediate position.
- Reached mid-way: The robot now targets the position that it has to release the object in. However, once it reaches this point, there is a possibility that the item it is carrying is placed after an object that is not yet placed, by the human or the other robot. Each object has a property

that designates the game object has to wait for, before placing continuing to place its own.

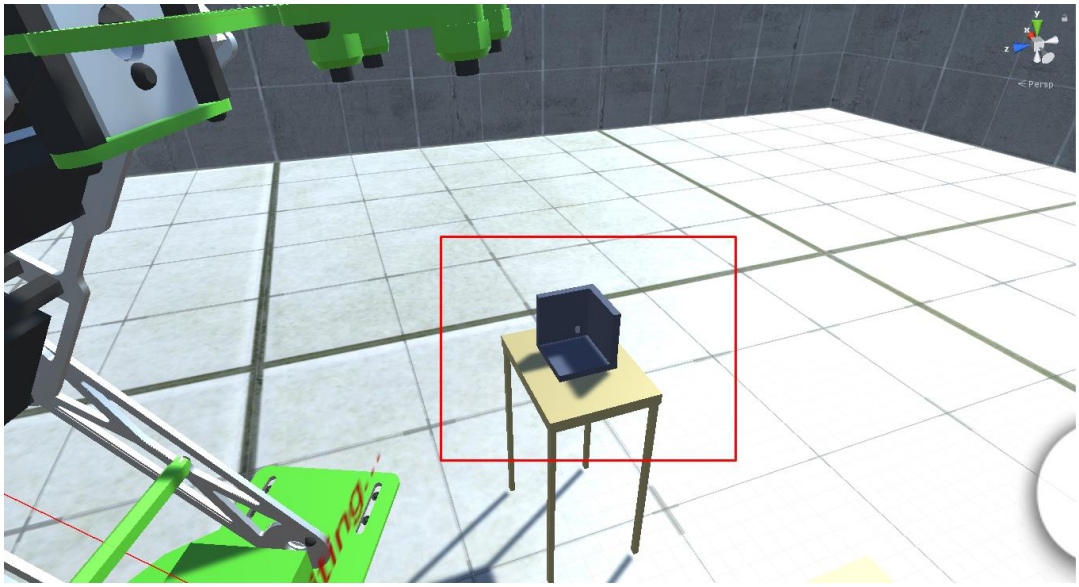
- Released object: At this point the robot targets the intermediate position once more, and after it reaches it, it will aim for the next object from the list, and the next cycle of phases.

The robot also calculates its own idle time, using the timer that times the whole project, and subtracting between the times and the start and end of its idle state.

7.2 Screwing robot

This robot (Robot 2) places screws on top of objects that were previously placed by either Robot 1 or the human user. It also implements the forward and inverse kinematics mentioned for its movement. Additionally, it makes use of the same principle for the 4 phases of moving before completing an action, and logs its metrics the same way.

A difference in its functionality is that it does not pick up screws from a table. It reaches to the same target (and therefore same position) every time and Instantiates a screw from a given model. In order for the user not to see the instantiation, a short wall is placed in front of its reaching position. At the same time, this robot does not take a list of objects, because the screws are instantiated. Therefore, Robot 2 uses a list of target locations as the targets for placing the screw. This list is defined on startup. Since the target locations belong to empty objects, the list contains only the names of these objects and not the object itself. Finally, to simulate the time it takes to actually screw, the robot will wait after releasing each screw for at least 1 second before continuing.



The screw inside the "half" cube is used as Grab position as well as model for the instantiation

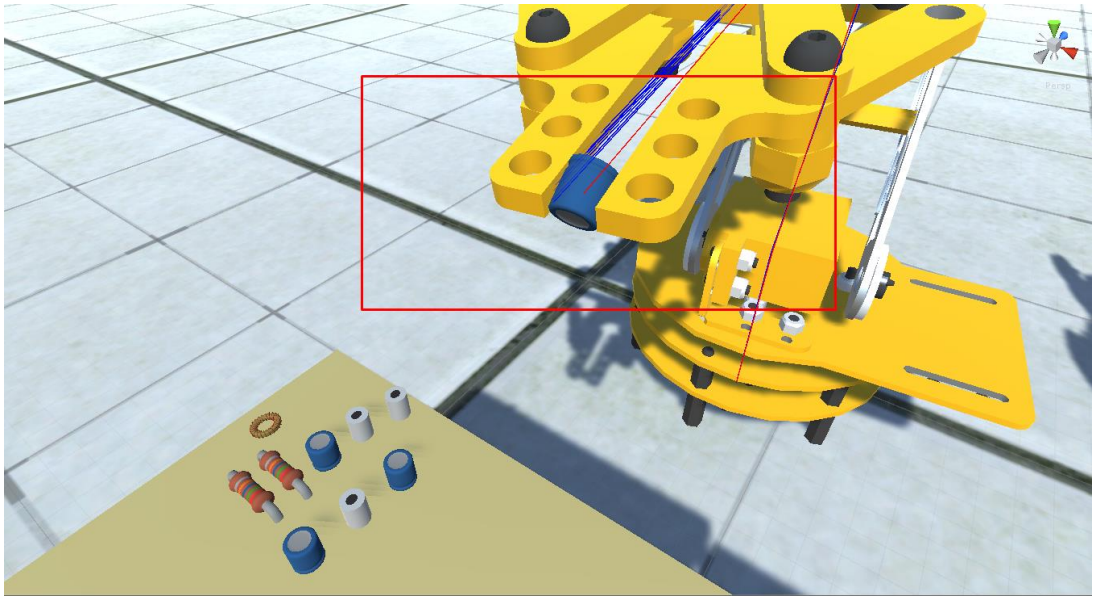
7.3 Grabbing and Dropping

Grabbing is essentially the same for both robots, but has a slightly different implementation. The Grab method is though run for both robots, once they have reached the corresponding object position.

For Robot 1, grabbing means a couple of actions. The algorithm finds the item that is to be grabbed, and assigns it as a child to the robots end effector, while making it a kinematic object, because if in a scenario it isn't already, we don't want it to be moving apart from its parent object's movements. Then, through a property set in another script, it gets the target location that the object needs to reach. This location though will be used as a target after it reaches the intermediate position. The aforementioned object's location is a component in the so called Release object. This object is the exact duplicate of the grabbed object. Though the renderer of this object is disabled in runtime, it is used in designing to give easily the rotation and position that we want to have in our grabbed object at release. Finally before returning to the main code, it render a Boolean as "true", designated that the object is now grabbed by its effector.

For Robot 2, there are very slight differences. First, it instantiates a screw according to the position and rotation of the initial object. After that, it assigns it as a child of the effector and continues its function as with Robot 1. However, as the grabbed object is just instantiated, it uses the object from the list to get

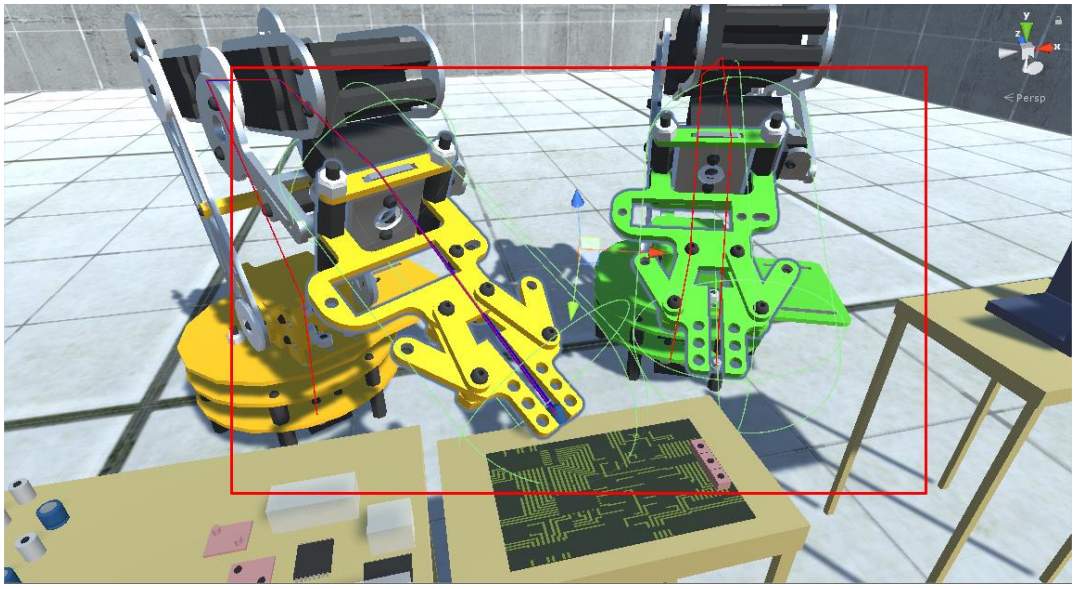
information regarding the release position's location, and also designates the object from the list as grabbed, for convenience in the calculations.



Robot 1 has grabbed a capacitor and is moving to the intermediate position

Dropping has the same functionality for both objects. Essentially, once the robot has reached its release position for the specific objects, it firstly removes it from the effectors children. Now, since the robots have a stop threshold, the current object position is not 100% accurate to the designated. For this purpose, the Drop function assigns the intended position to the object, taken from the object containing the information for the release. After that, it empties some variables, in order to have them cleared for the next object, and marks the one just placed as "Done" in some scripts to enable and disable corresponding functionalities.

7.4 Robots' near-collisions



Robots colliders (green capsules) are touching, indicating that there is a near-collision occurring.

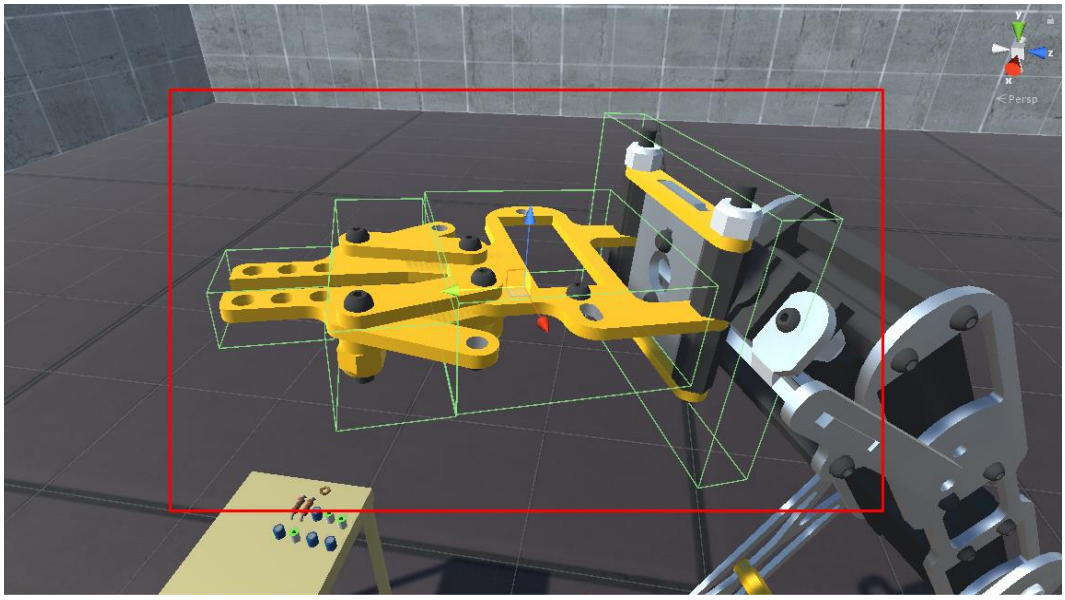
Collisions between robots, as seen above, are handled differently from human-robot ones. The main reason here, is that in a real time scenario, robots, and especially cobots (collaborative robots) would never collide with each other, considering they are programmed for their task in the specific environment, and make use of a lot of sensors. Therefore, to simulate this, the robots in our scenarios use capsule collider to ensure that they don't collide.

When the colliders touch, a near collision occurs. Through different scripts, each robot calculates for itself the distance it has from its target, as well as the other robot's distance from its own target. The robot that is the further form its target is the one that acknowledges the near collision occurrence. In the rare scenario that the two distances are exactly equal, Robot 2 is selected in a hardcoded fashion to acknowledge the collision.

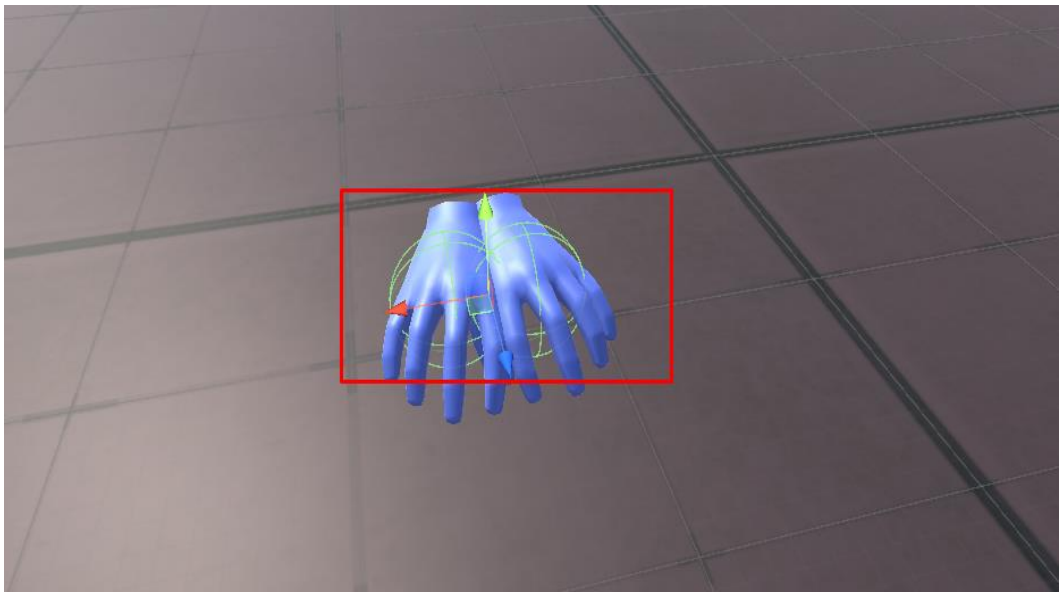
The touching of the colliders is realized in the **OnCollisionEnter()** function provided by Unity. Until the **OnCollisionExit()** is called, meaning that the colliders are no longer touching at any point, the selected from the above logic robot, will enter a near-collision state. What this basically does is enable a Boolean in the main script of the robot that declares the occurrence of a near-collision for this robot. The result is that as long as this Boolean is true, the robot

will stop executing any code after that check, in the specific script, therefore halting its movement in the scenario.

7.5 Human-Robot collisions



Robot 1's colliders used for human-robot collision (same for robot 2)



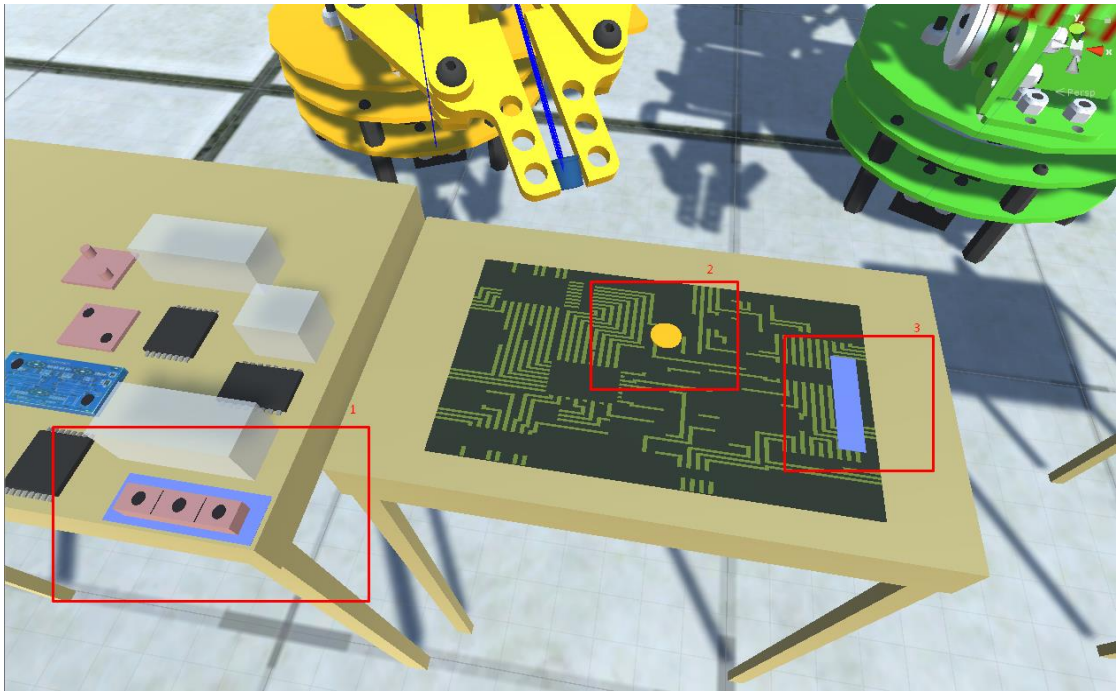
Both hands' colliders used for the human-robot collision event

As it can be seen in the above figures, the robots use different colliders for the human-robot interaction. The thought behind this is that the user must actually come in contact with the robot, to indicate a collision. Therefore the capsule colliders from the near-collision calculation are not preferable here, because they are not tangential to the robots actual volume. The cube colliders seen here are as tangential as possible. Thus the hand of the user will actually be colliding with a robot's effector in the event of a collision. The above statement has however one exception, that is the fingers of the hands. If the fingers are pointing out, they are outside the collider and will be slightly inside the effector before a collisions is triggered. But the colliders cannot be bigger because if the fingers are not pointing, the collision would occur before a hand actually touches an effector. Therefore there is a small percentage of error here that does not affect, however, the scenarios designed.

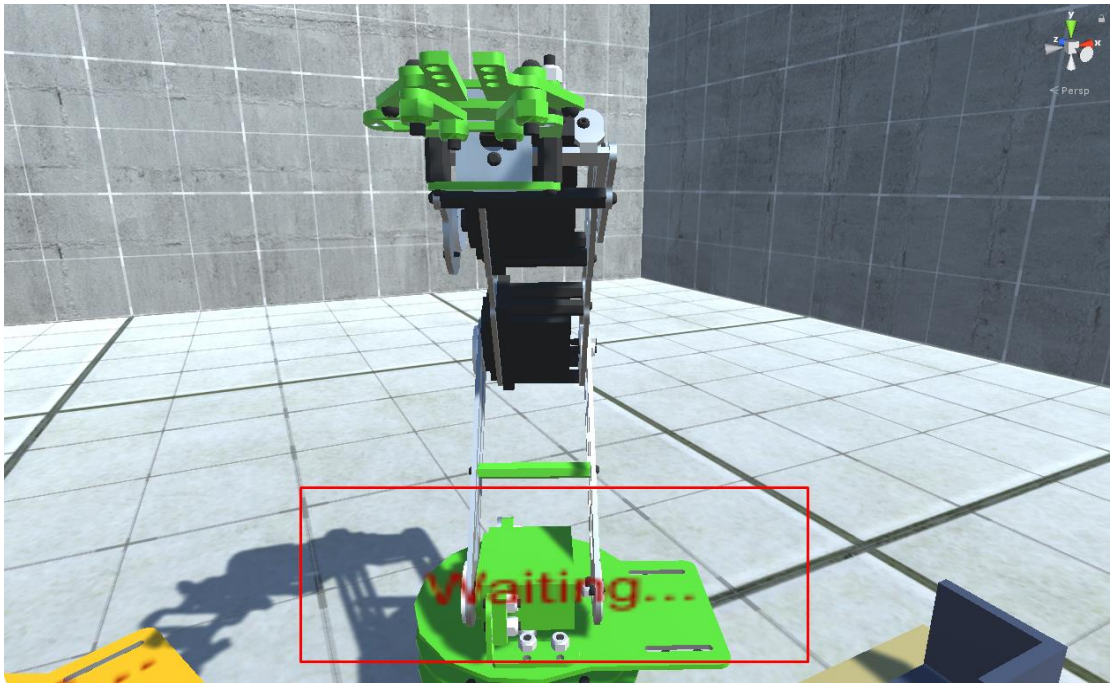
The human-robot collision is handled differently than that of a near-collision. **OnTriggerEnter()** is the method used here to acknowledge a collision. This function is different because the hands' colliders are declared as triggers. Trigger colliders are used because Unity will not calculate collisions with other colliders and apply physical restrictions, it will only inform of a collision and let the coding decide how to interpret it. Except from an integer variable, used to count how many times a collision of this kind has occurred, the corresponding script informs a Boolean variable that a collision has occurred. This variable is read by the 2 robots' main scripts. When it is true, it overrides that main code in a way, and executes another small part that only runs while a state of collision is present. The script logs the collision, writing when it occurred, with which hand and which robot, and will also give new target positions to both robots, regardless of the position they are in at the moment. The initial position they had will be kept to be restored after the collision. Both robots will therefore have specified positions to go to, designated as **"HandCollisionReset1"** and **"HandCollisionReset2"**. These are the retreat positions. This method is implemented in the Robot 2's main script. This part of code will wait until both robots have reached their retreat positions. After this happens, it will render the collision state as false, at the same time restoring the initial targets to both robots. Finally it will keep time of the delay that occurred because of the collision, as well as add to the total delay time.

7.6 Object Flashing

Object flashing is a function used to inform the subject of the sequence with which he has to pick his own objects, as well as the position and rotation he has to place them in. Additionally, if the Anticipation mode is enabled, this method is used to pinpoint where the robot is heading on the board, after it has picked up its target object, and until it places it. Finally, if Anticipation is enabled, both robots will be able to flash a message that says **"Waiting.."** above their base. The flashing of this message designates that the robot it currently in idle state, meaning it is either waiting directly for the subject to place more objects in order to continue with its own, or indirectly by waiting for the other robot to place an object, which will be delayed by the human.



1 - Object to be picked up by human is flashing
2 - Position robot 1 is moving into is flashing
3 - Position the human has to place his object (It is considered grabbed here for showing purposes.)



Waiting message appearing in robot 2 during runtime.

Object flashing is done by changing their color by accessing their material component, or by enabling and disabling their renderer component, with the use of Coroutines. Coroutines are parts of code in a script that are executed asynchronously with the rest of the code, and usually have a logical condition that enables them to complete their functionality. The condition in these scripts is:

- The human object will be flashing until it is picked up.
- The target position for the human object's placement will be flashing until the object is placed correctly by the subject. An object is considered placed correctly when its collider at least touches a collider with same shape, in the designated position, and it is not being grabbed.
- The robot's targets for placing their objects will be flashing from the moment they grab their object, up to the moment they place it and mark it as **"Done"**

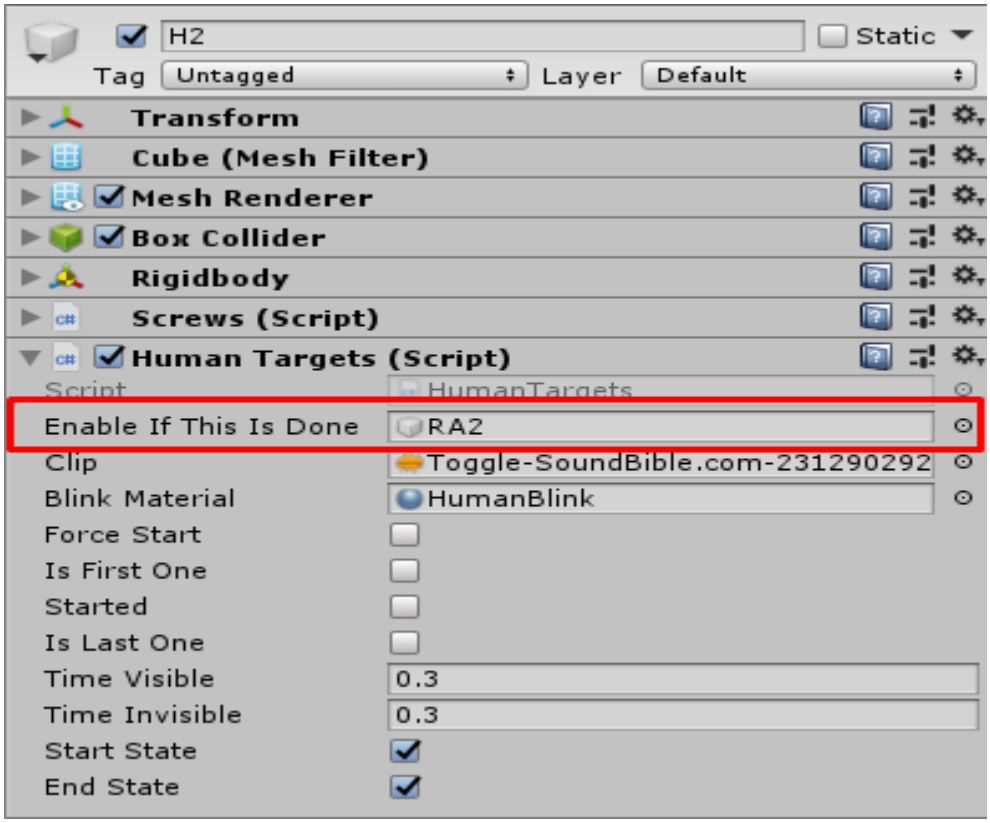
Each object in the sequence that has to be flashing, has a child object inside it that is a 2D plane with the same shape as the object, like a top-down view. This is the object that is actually flashing each time.

7.7 Human-object interactions

In the scenarios, the subject can only interact with objects that are modified as grabbable, and can only grab and release them. However, there is an algorithm that enables objects to be picked up, or disables that functionality, and also enables or disables their flashing functionality. Same as for the robot objects, there also exists a list that contain all the objects that are to be picked up by the human. For convenience in coding, the first and last objects are also designated through a Boolean property, as they have to be treated slightly differently. The first object to be picked up is also assigned the script **"Timing"**. This script marks the start of the scenario, which occurs when the first object in the subject's sequence is picked up for the first time, and it also keeps track of the scenarios total time, and logs some metrics at the end.

There is also another script (**"HTargets"**) that makes use of the list of objects mentioned above, to handle functionalities for all human objects at the same time. This script is responsible for checking if the next object in the human's sequence is able to be placed on the board. If it is, it will enable the flashing of the corresponding placement position.

It should be noted that a collision occurrence will override the above functionality. It its presence, the subject is not able to pick up anything, until the robots reach their retreat positions and the collision state is completed. This means that if he forced a collision after placing his object, he/she will briefly be unable to continue with the next object. However, if the collision occurred while he was already holding one, he/she will be able to place it, as long as he does in the correct position. If he/she places it in a wrong position or lets it go, he/she will have to wait for the collision phase to end before picking it up again, adding to his total completion time.



The Humans 2nd object has to wait for Robot 1's 2nd object to be placed, before its placement position will start flashing.

7.8 Robot-object interactions

The interaction between each robot and its corresponding objects is the same at its root for both robots, however it has small differences due to the aforementioned fact that Robot 2 instantiates objects, and keeps values of the release positions in empty objects. At the same time, the screw holes that need to be flashing, are not on the board this time, but instead exist on objects handled by the other robot, or the human. Therefore it required more scripting to complete the same functionality.

- Robot 1: This is a pretty straight forward functionality. When the robot grabs an object, it renders it as grabbed. When this situation is read in another script, another variable is enabled, that enables the object's position on the board to start flashing. The coroutine that does the flashing ends when the initial object is marked as "IsDone".

- Robot 2: Robot 2's 2D planes are placed as children in human or Robot 1's objects. At the same time, the grabbed object is instantiated. To address this, first when a screw is instantiated, it marks a variable in the target location's object that renders it as grabbed. Continuing, this empty object has a new script **("EnableHoleRendering")** that contains the object which has to flash. This object contains a script that enables its corresponding 2D plane, placed as its child, to start flashing. However, since this whole interaction is much less direct, each hole contains some additional information. This information refers to the previous screw's objects, and checks if they are done flashing. It also has a property that declares an object the screw has to wait for to be completed, before it starts flashing. Finally the last hole that has to flash is marked for convenience, and all holes except for the first one have information on the previous screw hole, to make it easier to check its current state.

It should be noted that the 2D plane is actually another object placed as child in the object that has to flash. The script that does the flashing is placed in the father object. The child object has the same name as the father plus the word **"Outline"**. For example, the first human object, named **"H1"** flashes the **"H1Outline"** object.

7.9 Naming conventions

Naming conventions that are implemented in this thesis are not obligatory and were created for coding convenience. That being said, there are hardcoded names of objects in few scripts, so the code has to be altered slightly to implement new names or modifications of the existing ones.

The naming conventions that are used in the objects participating in the sequence are presented below:

- Human objects in the sequence are called "H" plus the position that they are in, in the sequence. Therefore they are called "H1", "H2", "H3" and so on.
- The objects that represent the position the human has to place his objects in, inherit the sequence's object name plus the work "target". So "H1"

object is placed in “H1target” object’s location, “H2” in “H2target” object’s location etc.

- Robot 1’s objects are called “RA” plus their number in the sequence, meaning the names are “RA1”, “RA2” and so on.
- Robot 1’s objects that represent the placing location inherit the initial objects name plus the work “Release”. Thus, the “RA1” placement is in “RA1Release” object’s position and so on.
- The second robot’s object names are irrelevant, as the objects are instantiated. The placing location objects are named after their sequence, like so: “Screw1Loc”, “Screw2Loc”, “Screw3Loc” etc.
- The actual holes that exist on the first robot’s and the human’s objects have a similar name, indicating their sequence. These are called “ScrewHole1”, “ScrewHole2” and so on.

Finally all objects that participating in some sort of flashing have child objects that inherit their names plus the work “Outline”. The correspondence is as seen below:

H1 -> H1Outline

H1target -> H1targetOutline

RA1Release -> RA1ReleaseOutline

ScrewHole1 -> ScrewHole1Outline

The same goes for the continuation of the sequence.

It should be noted that the lists are filled at startup will up to 12 objects for each of the 3 sequences, if that many exist. If the sequence should contain more than 12 objects, they also have to be added in the script. Grouping them was avoided to make changing conventions easier to understand and implement.

8 Iterative improvement operations

After running several tests with the aforementioned settings, faults and room for improvements appeared to be present, mainly regarding the optimization of the task, with the purpose to give better, clearer signals and indications to the user, and also to differentiate the task in regards to what actions has each of the agents to execute, while enforcing the collaboration aspect of the whole scenario, wherever the individual tasks get involved with one another.

To achieve this, modifications where made to some aspects of the scenarios that were mentioned in previous parts of this thesis. Additionally, new and improved functions were implemented.

These changes are described below.

- Correction of the scale of objects for more realism: The scale of the tables, objects participating in the assembly, as well as the height and width of the Oculus avatar were modified, to give a more realistic feeling. Still in order to maintain a task that is not really hard to perform, objects such as resistors or capacitors are larger than usual, commercially, but the size is still realistic in some applications. The robot model however does not have a modifiable size. The reason behind this is the in a 1:1:1 scale (in X, Y and Z), when calculating metrics, the units per second that Unity counts as speed, can be translated to meters per second.

- Narrowing the robot's effector speed options: Since the end effector's speed is adjusted by changing the Learning Rate factor, it is made to be modified easily. However, the regulations and the scare factor of the problem have been accounted, and therefore, a range has been implemented in the Learning Rate options. This way the scenario has a more realistic speed of execution, and the subject is not scared of the robot's movements at the same time.

- Texture and color adjustments: The textures and colors of all the in game objects have been considered. Changes were made so that every in game model will have a friendly and not very bold color, so that the task is more realistic, but also the user's focus is constantly maintained. At the same

time, different objects can be easily distinguished, to avoid confusion about the sequence of tasks.

- Positions adjusted with user's field in mind: During the test, while the user would turn the table where his objects were placed, to pick them up, the assembly table was out of his field of view. This would result in the user missing some indicators of events occurring, but also missing the robots' movements, adding to his confusion. Finalizing the environment, actions were taken to move every object as close to his field of view as possible.

- Object flashing reworks: Due to the different textures and colors of objects, the flashing of objects did not appear consistent enough, and was thus removed. In its place, a 2D was placed under the human objects, and at the bottom of all the colliders on the tables, for the placement of objects. The planes geometry was in alignment with the objects top down view. The indicator of the next in sequence object for the user, as well as its placement position, and the anticipation of the robot's objects' positions on the board were now made apparent by the flashing of this planes.

- Color consistency: In order for the task to be easily identified, and for the user to be able to distinguish between his tasks and each of the robot's sequence, a certain coloring configuration was introduced. The users in game avatar's hands were colored blue, as well as the 2D planes that referred to his sequence's objects and board positions. The screwing robot's base and end effector were painted green, as well as the indicators of the screw hole it is targeting next. The other robot's base and end effector were painted yellowish, as well as the indicator planes for the positions it is targeting next.

- Anticipation rework: The anticipation option, meaning the timing at which each of the robots target on the board would flash its appointed color, slightly changed. With better correlation between the flashing and the actual corresponding movement of the robot in mind, the 2D planes were made to start flashing as soon as each robot picks up its next in sequence object, and not when it places its previous object.

- Robot's idle state notification: Each of the robot has an object that represents the word "Waiting" appointed on them. Each time the robot enters idle state, meaning that it is waiting for the human user to perform a task, or that it is waiting for the other robot, which is already in idle state waiting for the user, this object is flashing above the corresponding robot's base, as a notification that the user has been left behind on his sequence.

- Screwing delay simulation: To make the screwing robot appear to function more realistically and to improve on the complexity of the sequence, since the robot is not implementing a screwing animation, it was made to wait for at least one second, once it reaches its target screwing hole, before continuing its task.

- Human idle time: The workspace was defined by two cube colliders (game object **"Envelope"**). At any point the human's hands are not inside or touching the colliders, he/she is considered to not be participating, therefore idle.

9 Future possibilities

As the conclusion of this thesis is achieved, and results are delivered, there is apparent room for improvements and optimizations that exceed this thesis' targets and expectations. These are briefly described below:

- The implementation of inverse kinematics: Gradient descend, in the form that was implemented in this thesis, would not be accurate in a much more complicated scenario. For example, if the robots being used for this scenarios had more than one degree of freedom in some of its joints, this code would not be applicable, as the gradient descend in this form is applicable in a 2D plane, and a more complex mathematical approach would be required.

- Improved robotic arm intelligence: Improving the intelligence of the robotic arm algorithm would give a scenario a lot more possibilities to improve on its complexity and realism. One example of improvement would be object and human collision avoidance. In this thesis there times when colliders have to be disabled and so, robots can at times pass through objects. Additionally, a robot could be made to create its own sequence of actions, based on choices on the human side.

- Use of space movement: The Oculus Rift used in this thesis produces no motion sickness under sort periods of scenario executions. At the same time, the motion sensors it has implemented give a big enough space for movement. This would be well implemented in future scenarios, by creating platforms and making the user or the robots move back and forth, or by making the user perform more motions, such as reaching high for an object or ducking for one. This would give the ability to create more complex scenarios, or more implement more realistic applications.

10 Conclusion

Concluding, this thesis result is an easily modifiable and parametrical environment in Unity that can be long term used to create various experiments and scenarios that study and evaluate HRC.

Any user that picks up this project does not need to know coding to create his/her own environment and use his/her own objects. The user can also use different robot models with more or less joints, however they should be configured like the ones seen in this project, and also have one degree of freedom in every joint. The dof should be declared in the scripts.

Additionally, if the user wishes to add more than 12 objects, or change the naming conventions for any reason, it is very easy to do so, and requires minimal to no C# coding understanding.

Some functionalities that were created but not implemented in the end, are still present in scripts. Such are an alert button, used to continue after collision, a functionality where only the robot that collided with a hand would enter collision state and the other would continue normally, the flashing of objects themselves instead of their child “Outline” objects, and some smaller ones.

To enforce the above statements, all the scenarios ran during this thesis, even as tests, and even if they were not kept, originated from one initial scene, in which all the functionalities were set up, without adding any further coding, except simply expanding the list of objects accordingly.

There was an intention for multiple subjects to test all the various scenarios created and modified, as the logging system was extended and parametric itself, at the same time. However, as time restriction were big, and this thesis was developed simultaneously with the restrictions of the global Covid pandemic, it proved impossible to do so.

11 Bibliography

Bauer, W., Bender, M., Braun, M., Rally, P., & Scholtz, O. (2016). Lightweight robots in manual assembly—best to start simply. Examining companies' initial experiences with lightweight robots, Stuttgart, 1-32.

BSI Group. (2016). Robots and robotic devices—Collaborative robots (ISO/TS 15066: 2016). BSI Standards Publication.

Goodrich, M. A., & Schultz, A. C. (2008). Human-Robot Interaction: A Survey. Foundations and trends in human-computer interaction. In Now Publishers Inc..

Haddadin, S., Albu-Schäffer, A., & Hirzinger, G. (2007, June). Safety Evaluation of Physical Human-Robot Interaction via Crash-Testing. In Robotics: Science and systems (Vol. 3, pp. 217-224).

Kim, Y. M., Rhiu, I., & Yun, M. H. (2020). A systematic review of a virtual reality system from the perspective of user experience. International Journal of Human–Computer Interaction, 36(10), 893-910.

Kontrazis (2018). Creating a virtual environment and scenarios for studying human robot collaboration in virtual reality, Diploma Thesis, National Technical University of Athens.

Krüger, J., Lien, T. K., & Verl, A. (2009). Cooperation of human and machines in assembly lines. CIRP annals, 58(2), 628-646.

Matsas, E., Vosniakos, G. C., & Batras, D. (2017). Effectiveness and acceptability of a virtual environment for assessing human–robot collaboration in manufacturing. The International Journal of Advanced Manufacturing Technology, 92(9), 3903-3917.

Matsas, E., & Vosniakos, G. C. (2017). Design of a virtual reality training system for human–robot collaboration in manufacturing tasks. International Journal on Interactive Design and Manufacturing (IJIDeM), 11(2), 139-153.

Oke, G., & Istefanopulos, Y. (2001, July). Gradient-descent based trajectory planning for regulation of a two-link flexible robotic arm. In 2001 IEEE/ASME International Conference on Advanced Intelligent Mechatronics. Proceedings (Cat. No. 01TH8556) (Vol. 2, pp. 948-952). IEEE.

Ottosson, S. (2002). Virtual reality in the product development process. Journal of Engineering Design, 13(2), 159-172.

Oyekan, J. O., Hutabarat, W., Tiwari, A., Grech, R., Aung, M. H., Mariani, M. P., ... & Dupuis, C. (2019). The effectiveness of virtual environments in developing collaborative strategies between industrial robots and humans. Robotics and Computer-Integrated Manufacturing, 55, 41-54.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

Unhelkar, V. V., Siu, H. C., & Shah, J. A. (2014, March). Comparative performance of human and mobile robotic assistants in collaborative fetch-and-deliver tasks. In 2014 9th ACM/IEEE International Conference on Human-Robot Interaction (HRI) (pp. 82-89). IEEE.

Weistroffer, V., Paljic, A., Callebert, L., & Fuchs, P. (2013, October). A methodology to assess the acceptability of human-robot collaboration using virtual reality. In Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology (pp. 39-48).

Zucconi, A. (2017, April 6). Implementing Forward Kinematics. https://www.alanzucconi.com/2017/04/06/implementing-forward-kinematics/

Zucconi, A. (2017, April 10). An Introduction to Gradient Descent. https://www.alanzucconi.com/2017/04/10/gradient-descent/

Zucconi, A. (2017, April 10). Inverse Kinematics for Robotic Arms. https://www.alanzucconi.com/2017/04/10/robotic-arms/

12 Code Appendix

Below are provided most of the parts that were written for the completion of this thesis. Scripts, or parts of scripts that were results of research, aside from basic settings, are not presented here.

Inverse Kinematics

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;
using Assets.GD.IK.Scripts;

namespace GD
{
   // Error function to minimise
   public delegate float ErrorFunction(Vector3 target, float[] solution);

   public struct PositionRotation
   {
      Vector3 position;
      Quaternion rotation;
      public PositionRotation(Vector3 position, Quaternion rotation)
      {
         this.position = position;
         this.rotation = rotation;
      }
      // PositionRotation to Vector3
      public static implicit operator Vector3(PositionRotation pr)
      {
```

```
        return pr.position;
    }
    // PositionRotation to Quaternion
    public static implicit operator Quaternion(PositionRotation pr)
    {
        return pr.rotation;
    }
}
//[ExecuteInEditMode]
public class InverseKinematics : MonoBehaviour
{
    [Header("Joints")]
    public Transform BaseJoint;
    //[HideInInspector]
    [ReadOnly]
    public RobotJoint[] Joints = null;
    // The current angles
    [ReadOnly]
    public float[] Solution = null;

    [Header("Destination")]
    public Transform Effector;
    [Space]
    public Transform Destination = null;
    public float DistanceFromDestination;
    private Vector3 target;

    [Header("Inverse Kinematics")]
    [Range(0, 1f)]
    public float DeltaGradient = 0.1f; // Used to simulate gradient (degrees)
    //[Range(0, 100f)]
```

```
public float LearningRate; // How much we move depending on the gradient

[Space()]
[Range(0, 0.25f)]
public float StopThreshold = 0.1f; // If closer than this, it stops
[Range(0, 10f)]
public float SlowdownThreshold = 0.25f; // If closer than this, it linearly slows down

public ErrorFunction ErrorFunction;

[Header("Debug")]
public bool DebugDraw = true;

private Transform ActualTarget;
private GameObject effector;
public List<GameObject> Targets;

private GameObject ToWaitFor;

void Start()// Use this for initialization
{
   ToWaitFor = null;

   ActualTarget = null;
   Destination = null;

   effector = GameObject.Find("HAND");

   if (Joints == null)
      GetJoints();
```

```
ErrorFunction = DistanceFromTarget;

Targets = new List<GameObject>();
Targets.Add(GameObject.Find("RA1"));
Targets.Add(GameObject.Find("RA2"));
Targets.Add(GameObject.Find("RA3"));
Targets.Add(GameObject.Find("RA4"));

if (GameObject.Find("RA5") != null)
{
   Targets.Add(GameObject.Find("RA5"));
}

if (GameObject.Find("RA6") != null)
{
   Targets.Add(GameObject.Find("RA6"));
}

if (GameObject.Find("RA7") != null)
{
   Targets.Add(GameObject.Find("RA7"));
}
if (GameObject.Find("RA8") != null)
{
   Targets.Add(GameObject.Find("RA8"));
}

if (GameObject.Find("RA9") != null)
{
   Targets.Add(GameObject.Find("RA9"));
}
```

```
    LearningRate = GameObject.Find("OPTIONS").GetComponent<OPTIONS>().R1_LearningRate;
}


[ExposeInEditor(RuntimeOnly = false)]
public void GetJoints()
{
    Joints = BaseJoint.GetComponentsInChildren<RobotJoint>();
    Solution = new float[Joints.Length];
}

//*************************************
//Identifiers for the current task of the robot
//*************************************
private bool IsAvailable = true;
private bool IsCarryingTarget = false;
private bool IsMovingToTarget = false;
public bool MovingToInner1 = false;
public bool MovingToInner2 = false;
private bool MovingToStart = false;

public bool DontDrop = false; //dont drop object because it is avoiding collision with a hand

void Update() // Update is called once per frame
{
    if (!GameObject.Find("H1").GetComponent<Timing>().StartedTask)
    {
        return;
    }
```

```
if (GameObject.Find("H1").GetComponent<Timing>().Robot1Done)
{
   return;
}

if (GameObject.Find("Button").GetComponent<ButtonStartAlert>().RobotHandCollision)
{
   HandleIdleState();

   ActualTarget = GameObject.Find("HandCollisionReset1").transform;
   Vector3 newdirection = (ActualTarget.position - transform.position).normalized;
   target = ActualTarget.position - newdirection * DistanceFromDestination;
   //if (Vector3.Distance(Effector.position, target) > Threshold)
   if (ErrorFunction(target, Solution) > StopThreshold)
   {
      IsMovingToTarget = true;
      ApproachTarget(target);
   }
   return;
}

if (IsAvailable)
{
   SelectTarget();
   IsAvailable = false;
}

if (Destination == null)
{
   return;
}
```

```
else
{
   ActualTarget = Destination;
}

if (ActualTarget == null)
{
   return;
}



// Do we have to approach the target?
//Vector3 direction = (Destination.position - Effector.transform.position).normalized;
Vector3 direction = (ActualTarget.position - transform.position).normalized;
target = ActualTarget.position - direction * DistanceFromDestination;
//if (Vector3.Distance(Effector.position, target) > Threshold)
if (ErrorFunction(target, Solution) > StopThreshold)
{
   IsMovingToTarget = true;
   ApproachTarget(target);
}
else if (IsMovingToTarget)
{
   if (!IsCarryingTarget && !MovingToStart)
   {
      IsMovingToTarget = false;
      ActualTarget = null;

      GrabObject();
      IsCarryingTarget = true;
      MovingToInner1 = true;
```

```
}
else if (MovingToInner1)
{
   if (!ToWaitFor.GetComponent<Screws>().IsDone)
   {
      CountIdleTimeR1();
      return;
   }
   DeactivateIdle();

   Destination = ReleaseTransform;

   MovingToInner1 = false;
   MovingToInner2 = true;
}
else if (MovingToInner2)
{
   if (DontDrop)
   {
      return;
   }
   DropObject();

   IsCarryingTarget = false;
   MovingToInner2 = false;
   MovingToStart = true;
}
else if (MovingToStart)
{
   Destination = null;
   MovingToStart = false;
```

```
            IsAvailable = true;
        }
    }
    //if (DebugDraw)
    //{
    //    Debug.DrawLine(Effector.transform.position, target, Color.green);
    //    Debug.DrawLine(ActualTarget.transform.position, target, new Color(0, 0.5f, 0));
    //}
}

private string SelectedTargetsName = "";
private void SelectTarget()
{
    var target = Targets.FirstOrDefault();

    if (target == null)
    {
        Destination = GameObject.Find("InnerHandInitialPosition").transform;

        var h1 = GameObject.Find("H1").GetComponent<Timing>();
        h1.Robot1Done = true;
        var time = h1.Timer.ReturnTime();
        Logfile.Write($"{time.ToString()} sec : Robot1 - task FINISHED.");

        var J4 = GameObject.Find("JOINT 4").GetComponent<EffectorCollision>();

        float maxR1Speed = J4.maxspeed;
        float speeds = J4.AllSpeeds;
        int speedsCount = J4.SpeedsCount;
```

```
            Logfile.Write($"Robot1 (LearningRate = {LearningRate}) - max speed reached :
{maxR1Speed} m/s, average speed : {speeds/speedsCount} m/s.");


            Logfile.Write($"Robot1 - waited human for {TotalIdleTime} sec.");


            return;
        }
        //else if
(GameObject.Find("Button").GetComponent<ButtonStartAlert>().RobotHandCollision)
        //{
        //    SelectedTargetsName = target.name;
        //    GameObject.Find("Button").GetComponent<ButtonStartAlert>().R1Destination =
target.transform;


        //    ToWaitFor = Destination.GetComponent<Screws>().WaitForThis;
        //}
        else
        {
            SelectedTargetsName = target.name;
            Destination = target.transform;


            ToWaitFor = Destination.GetComponent<Screws>().WaitForThis;
        }


        Targets.RemoveAt(0);
    }


    private Vector3 ReleasePosition;
    private Quaternion ReleaseRotation;
    private Transform ReleaseTransform;
    public void DropObject()
    {
```

```
        effector.transform.GetChild(0).parent = null;

        var target = GameObject.Find(SelectedTargetsName);

        var targetRigidBody = target.GetComponent<Rigidbody>();

        targetRigidBody.transform.position = ReleasePosition;
        targetRigidBody.transform.rotation = ReleaseRotation;

        targetRigidBody.isKinematic = true;
        targetRigidBody.useGravity = false;

        SelectedTargetsName = "";
        ReleasePosition = default;

        target.GetComponent<Screws>().IsDone = true;

        GameObject.Find(target.name +
"Release").GetComponent<RenderRAReleases>().ReleaseIsDone = true;

        Destination = GameObject.Find("InnerPosition1").transform;
    }

    public void GrabObject()
    {
        if (string.IsNullOrEmpty(SelectedTargetsName))
        {
            return;
        }
        var target = GameObject.Find(SelectedTargetsName);
```

```
        target.transform.SetParent(effector.transform);

        target.transform.localRotation = effector.transform.rotation;
        target.transform.position = effector.transform.position;

        var script = target.GetComponent<ReleasePosition>();

        Destination = GameObject.Find("InnerPosition1").transform;

        ReleaseTransform = script.Position;
        ReleasePosition = script.Position.position;
        ReleaseRotation = script.Position.rotation;
    }

    public void ApproachTarget(Vector3 target)
    {
        // Starts from the end, up to the base
        // Starts from joints[end-2]
        //  so it skips the hand that doesn't move!
        for (int i = Joints.Length - 1; i >= 0; i--)
        //for (int i = 0; i < Joints.Length - 1 - 1; i++)
        {
            // FROM:    error:     [0, StopThreshold,  SlowdownThreshold]
            // TO:      slowdown:  [0, 0,              1               ]
            float error = ErrorFunction(target, Solution);
            float slowdown = Mathf.Clamp01((error - StopThreshold) / (SlowdownThreshold -
StopThreshold));

            // Gradient descent
            float gradient = CalculateGradient(target, Solution, i, DeltaGradient);
            Solution[i] -= LearningRate * gradient;/* * slowdown;*/                    //////EXW
PEIRAKSEI AUTO//////
```

```
        // Clamp
        Solution[i] = Joints[i].ClampAngle(Solution[i]);

        // Early termination
        if (ErrorFunction(target, Solution) <= StopThreshold)
            break;
    }


    for (int i = 0; i < Joints.Length - 1; i++)
    {
        Joints[i].MoveArm(Solution[i]);
    }
}

/* Calculates the gradient for the inverse kinematic.
 * It simulates the forward kinematics the i-th joint,
 * by moving it +delta and -delta.
 * It then sees which one gets closer to the target.
 * It returns the gradient (suggested changes for the i-th joint)
 * to approach the target. In range (-1,+1)
 */
public float CalculateGradient(Vector3 target, float[] Solution, int i, float delta)
{
    // Saves the angle,
    // it will be restored later
    float solutionAngle = Solution[i];

    // Gradient : [F(x+h) - F(x)] / h
    // Update   : Solution -= LearningRate * Gradient
    float f_x = ErrorFunction(target, Solution); // ErrorFunction is DistanceFromTarget!!!!
```

```
    Solution[i] += delta;
    float f_x_plus_h = ErrorFunction(target, Solution);

    float gradient = (f_x_plus_h - f_x) / delta;

    // Restores
    Solution[i] = solutionAngle;

    return gradient;
}

// Returns the distance from the target, given a solution
public float DistanceFromTarget(Vector3 target, float[] Solution)
{
    Vector3 point = ForwardKinematics(Solution);
    return Vector3.Distance(point, target);
}



/* Simulates the forward kinematics,
 * given a solution. */
public PositionRotation ForwardKinematics(float[] Solution)
{
    Vector3 prevPoint = Joints[0].transform.position;
    //Quaternion rotation = Quaternion.identity;

    // Takes object initial rotation into account
    Quaternion rotation = transform.rotation;
    for (int i = 1; i < Joints.Length; i++)
    {
```

```
        // Rotates around a new axis
        rotation *= Quaternion.AngleAxis(Solution[i - 1], Joints[i - 1].Axis);
        Vector3 nextPoint = prevPoint + rotation * Joints[i].StartOffset;

        if (DebugDraw)
            Debug.DrawLine(prevPoint, nextPoint, Color.blue);

        prevPoint = nextPoint;
    }

    // The end of the effector
    return new PositionRotation(prevPoint, rotation);
}



private float TotalIdleTime = 0;
private bool IdleTimeActivated = false;
private float IdleTimerStartingValue = 0;
public void CountIdleTimeR1()
{
    if (!IdleTimeActivated)
    {
        IdleTimeActivated = true;

        var h1 = GameObject.Find("H1").GetComponent<Timing>();
        IdleTimerStartingValue = h1.Timer.ReturnTime();
    }
}
public void DeactivateIdle()
{
    if (IdleTimeActivated)
```

```
        {
            var h1 = GameObject.Find("H1").GetComponent<Timing>();

            TotalIdleTime += h1.Timer.ReturnTime() - IdleTimerStartingValue;

            IdleTimerStartingValue = 0;

            IdleTimeActivated = false;
        }
    }
    public void HandleIdleState()
    {
        if (IdleTimeActivated)
        {
            DeactivateIdle();
        }
    }
  }
}
```

IK2

```
using GD;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;
using Assets.GD.IK.Scripts;

namespace GD
{
```

```
public class IK2 : MonoBehaviour
{
    [Header("Joints")]
    public Transform BaseJoint;
    //[HideInInspector]
    [ReadOnly]
    public RobotJoint[] Joints = null;
    // The current angles
    [ReadOnly]
    public float[] Solution = null;

    [Header("Destination")]
    public Transform Effector;
    [Space]
    public Transform Destination = null;
    public float DistanceFromDestination;
    private Vector3 target;

    [Header("Inverse Kinematics")]
    [Range(0, 1f)]
    public float DeltaGradient = 0.1f; // Used to simulate gradient (degrees)

    //[Range(0, 100f)]
    public float LearningRate; // How much we move depending on the gradient

    [Space()]
    [Range(0, 0.25f)]
    public float StopThreshold = 0.1f; // If closer than this, it stops
    [Range(0, 10f)]
    public float SlowdownThreshold = 0.25f; // If closer than this, it linearly slows down
```

```
public ErrorFunction ErrorFunction;

[Header("Debug")]
public bool DebugDraw = true;

private Transform ActualTarget;
private GameObject effector;
public List<GameObject> Targets;

private int ScrewsCreated = 0;
private GameObject ToWaitFor;
public string CurrentScrewLocationName = "";

GameObject Screw;
void Start()// Use this for initialization
{
   Screw = GameObject.Find("Screw");

   ToWaitFor = null;
   ActualTarget = null;
   Destination = null;

   effector = GameObject.Find("HAND2");

   if (Joints == null)
      GetJoints();

   ErrorFunction = DistanceFromTarget;
```

```
Targets = new List<GameObject>();
Targets.Add(GameObject.Find("Screw1Loc"));
Targets.Add(GameObject.Find("Screw2Loc"));
Targets.Add(GameObject.Find("Screw3Loc"));
Targets.Add(GameObject.Find("Screw4Loc"));

var S5 = GameObject.Find("Screw5Loc");
if (S5 != null)
{
   Targets.Add(GameObject.Find("Screw5Loc"));
}

var S6 = GameObject.Find("Screw6Loc");
if (S6 != null)
{
   Targets.Add(GameObject.Find("Screw6Loc"));
}

var S7 = GameObject.Find("Screw7Loc");
if (S7 != null)
{
   Targets.Add(GameObject.Find("Screw7Loc"));
}

var S8 = GameObject.Find("Screw8Loc");
if (S8 != null)
{
   Targets.Add(GameObject.Find("Screw8Loc"));
}

var S9 = GameObject.Find("Screw9Loc");
```

```
    if (S9 != null)
    {
        Targets.Add(GameObject.Find("Screw9Loc"));
    }

    LearningRate = GameObject.Find("OPTIONS").GetComponent<OPTIONS>().R2_LearningRate;
}

[ExposeInEditor(RuntimeOnly = false)]
public void GetJoints()
{
    Joints = BaseJoint.GetComponentsInChildren<RobotJoint>();
    Solution = new float[Joints.Length];
}

//*************************************
//Identifiers for the current task of the robot
//*************************************
private bool IsAvailable = true;
private bool IsCarryingTarget = false;
private bool IsMovingToTarget = false;
public bool MovingToInner1 = false;
public bool MovingToInner2 = false;
private bool MovingToStart = false;

public bool DontDrop = false; //dont drop object because it is avoiding collision with a hand
public bool LastScrew = false;

void Update() // Update is called once per frame
{
    if (!GameObject.Find("H1").GetComponent<Timing>().StartedTask)
```

```
{
    return;
}

if (GameObject.Find("H1").GetComponent<Timing>().Robot2Done)
{
    return;
}

if (GameObject.Find("Button").GetComponent<ButtonStartAlert>().RobotHandCollision)
{
    HandleIdleState();

    ActualTarget = GameObject.Find("HandCollisionReset2").transform;
    Vector3 newdirection = (ActualTarget.position - transform.position).normalized;
    target = ActualTarget.position - newdirection * DistanceFromDestination;
    //if (Vector3.Distance(Effector.position, target) > Threshold)
    if (ErrorFunction(target, Solution) > StopThreshold)
    {
        IsMovingToTarget = true;
        ApproachTarget(target);
    }
    return;
}

if (IsAvailable)
{
    GoForScrew();
    IsAvailable = false;
}
```

```
if (Destination == null)
{
   return;
}
else
{
   ActualTarget = Destination;
}

if (ActualTarget == null)
{
   return;
}



// Do we have to approach the target?
//Vector3 direction = (Destination.position - Effector.transform.position).normalized;
Vector3 direction = (ActualTarget.position - transform.position).normalized;
target = ActualTarget.position - direction * DistanceFromDestination;
//if (Vector3.Distance(Effector.position, target) > Threshold)
if (ErrorFunction(target, Solution) > StopThreshold)
{
   IsMovingToTarget = true;
   ApproachTarget(target);
}
else if (IsMovingToTarget)
{
   if (!IsCarryingTarget && !MovingToStart)
   {
      IsMovingToTarget = false;
      ActualTarget = null;
```

```
    GrabScrew();
    IsCarryingTarget = true;
    MovingToInner1 = true;
}
else if (MovingToInner1)
{
    if (!ToWaitFor.GetComponent<Screws>().IsDone)
    {
        CountIdleTimeR2();
        return;
    }
    DeactivateIdle();

    GetScrewPlacement();
    Destination = ReleaseTransform;

    MovingToInner1 = false;
    MovingToInner2 = true;
}
else if (MovingToInner2)
{
    if (DontDrop)
    {
        return;
    }
    DropObject();

    IsCarryingTarget = false;
    MovingToInner2 = false;
    MovingToStart = true;
```

```
        }
        else if (MovingToStart)
        {
            Destination = null;
            MovingToStart = false;
            IsAvailable = true;
        }
    }
    //if (DebugDraw)
    //{
    //    Debug.DrawLine(Effector.transform.position, target, Color.green);
    //    Debug.DrawLine(ActualTarget.transform.position, target, new Color(0, 0.5f, 0));
    //}
}

private string SelectedTargetsName = "";
private void GoForScrew()
{
    if (LastScrew)
    {
        Destination = GameObject.Find("Hand2InitialPosition").transform;

        var h1 = GameObject.Find("H1").GetComponent<Timing>();
        var time = h1.Timer.ReturnTime();
        h1.Robot2Done = true;
        Logfile.Write($"Robot2 - task FINISHED. Time : {time.ToString()} sec.");

        var J24 = GameObject.Find("JOINT 24").GetComponent<Ik2EffectorCollision>();

        float maxR2Speed = J24.maxspeed;
        float speeds = J24.AllSpeeds;
```

```
            int speedsCount = J24.SpeedsCount;
            Logfile.Write($"Robot2 (LearningRate = {LearningRate}) - max speed reached :
{maxR2Speed} m/s, average speed : {speeds / speedsCount} m/s.");

            Logfile.Write($"Robot2 - waited human for {TotalIdleTime} sec.");

            return;
        }

        Destination = GameObject.Find("Screw").transform;
        return;
    }

    private void GetScrewPlacement()
    {
        var target = Targets.FirstOrDefault();

        if (target == null)
        {
            Destination = GameObject.Find("Hand2InitialPosition").transform;
            return;
        }
        else
        {
            Destination = target.transform;
        }
        ReleaseTransform = target.transform;
        ReleasePosition = target.transform.position;
        ReleaseRotation = target.transform.rotation;

        Targets.RemoveAt(0);
```

```
    if (Targets.FirstOrDefault() == null)
    {
        LastScrew = true;
    }
}

private Vector3 ReleasePosition;
private Quaternion ReleaseRotation;
private Transform ReleaseTransform;

private void DropObject()
{
    effector.transform.GetChild(0).parent = null;

    var target = GameObject.Find(SelectedTargetsName);

    //var targetRigidBody = target.GetComponent<Rigidbody>();

    target.transform.position = ReleasePosition;
    target.transform.rotation = ReleaseRotation;

    //targetRigidBody.isKinematic = true;
    //targetRigidBody.useGravity = false;

    SelectedTargetsName = "";
    ReleasePosition = default;

    GameObject.Find(CurrentScrewLocationName).GetComponent<Screws>().IsDone = true;
    CurrentScrewLocationName = "";
```

```
    Destination = GameObject.Find("InnerPosition3").transform;
}

private void GrabScrew()
{
    if (LastScrew)
    {
        return;
    }
    var newScrew = Instantiate(Screw, Screw.transform.position, Screw.transform.rotation);
    newScrew.tag = "Screw";
    newScrew.name = $"Screw{ScrewsCreated}";
    ScrewsCreated++;

    SelectedTargetsName = newScrew.name;

    newScrew.transform.SetParent(effector.transform);

    newScrew.transform.localRotation = effector.transform.rotation;
    newScrew.transform.position = effector.transform.position;

    Destination = GameObject.Find("InnerPosition3").transform;

    var target = Targets.FirstOrDefault();
    CurrentScrewLocationName = target.name;
    var tScript = target.GetComponent<Screws>();
    ToWaitFor = tScript.WaitForThis;
}

public void ApproachTarget(Vector3 target)
{
```

```
        // Starts from the end, up to the base
        // Starts from joints[end-2]
        //  so it skips the hand that doesn't move!
        for (int i = Joints.Length - 1; i >= 0; i--)
        //for (int i = 0; i < Joints.Length - 1 - 1; i++)
        {
            // FROM:    error:     [0, StopThreshold,  SlowdownThreshold]
            // TO:      slowdown:  [0, 0,              1              ]
            float error = ErrorFunction(target, Solution);
            float slowdown = Mathf.Clamp01((error - StopThreshold) / (SlowdownThreshold - StopThreshold));

            // Gradient descent
            float gradient = CalculateGradient(target, Solution, i, DeltaGradient);
            Solution[i] -= LearningRate * gradient;/* * slowdown;*/                    //////EXW PEIRAKSEI AUTO//////
            // Clamp
            Solution[i] = Joints[i].ClampAngle(Solution[i]);

            // Early termination
            if (ErrorFunction(target, Solution) <= StopThreshold)
                break;
        }

        for (int i = 0; i < Joints.Length - 1; i++)
        {
            Joints[i].MoveArm(Solution[i]);
        }
    }

    /* Calculates the gradient for the inverse kinematic.
```

```
 * It simulates the forward kinematics the i-th joint,
 * by moving it +delta and -delta.
 * It then sees which one gets closer to the target.
 * It returns the gradient (suggested changes for the i-th joint)
 * to approach the target. In range (-1,+1)
 */
public float CalculateGradient(Vector3 target, float[] Solution, int i, float delta)
{
   // Saves the angle,
   // it will be restored later
   float solutionAngle = Solution[i];

   // Gradient : [F(x+h) - F(x)] / h
   // Update   : Solution -= LearningRate * Gradient
   float f_x = ErrorFunction(target, Solution);

   Solution[i] += delta;
   float f_x_plus_h = ErrorFunction(target, Solution);

   float gradient = (f_x_plus_h - f_x) / delta;

   // Restores
   Solution[i] = solutionAngle;

   return gradient;
}

// Returns the distance from the target, given a solution
public float DistanceFromTarget(Vector3 target, float[] Solution)
{
   Vector3 point = ForwardKinematics(Solution);
```

```
    return Vector3.Distance(point, target);
}


/* Simulates the forward kinematics,
 * given a solution. */
public PositionRotation ForwardKinematics(float[] Solution)
{
    Vector3 prevPoint = Joints[0].transform.position;
    //Quaternion rotation = Quaternion.identity;

    // Takes object initial rotation into account
    Quaternion rotation = transform.rotation;
    for (int i = 1; i < Joints.Length; i++)
    {
        // Rotates around a new axis
        rotation *= Quaternion.AngleAxis(Solution[i - 1], Joints[i - 1].Axis);
        Vector3 nextPoint = prevPoint + rotation * Joints[i].StartOffset;

        if (DebugDraw)
            Debug.DrawLine(prevPoint, nextPoint, Color.blue);

        prevPoint = nextPoint;
    }

    // The end of the effector
    return new PositionRotation(prevPoint, rotation);
}

private float TotalIdleTime = 0;
private bool IdleTimeActivated = false;
```

```
private float IdleTimerStartingValue = 0;
public void CountIdleTimeR2()
{
   if (!IdleTimeActivated)
   {
      IdleTimeActivated = true;

      var h1 = GameObject.Find("H1").GetComponent<Timing>();
      IdleTimerStartingValue = h1.Timer.ReturnTime();
   }
}
public void DeactivateIdle()
{
   if (IdleTimeActivated)
   {
      var h1 = GameObject.Find("H1").GetComponent<Timing>();

      TotalIdleTime += h1.Timer.ReturnTime() - IdleTimerStartingValue;

      IdleTimerStartingValue = 0;

      IdleTimeActivated = false;
   }
}
public void HandleIdleState()
{
   if (IdleTimeActivated)
   {
      DeactivateIdle();
   }
}
```

```
    }
}
```

TriggerRobotHandCollision

```
using Assets.GD.IK.Scripts;
using GD;
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TriggerRobotHandCollision : MonoBehaviour
{
    public AudioClip Clip;
    // Start is called before the first frame update

    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    private int HumanRobotCollisions = 0;
    private void OnTriggerEnter(Collider other)
    {
        var btn = (GD.ButtonStartAlert)FindObjectOfType(typeof(GD.ButtonStartAlert));
        //THELW NA STAMATANE KAI TA 2 ROBOT STO COLLISION
        if ((other.gameObject.name == "HAND" || other.gameObject.name == "HAND2" ||
other.gameObject.name == "Obstacle") && (!btn.RobotHandCollision))
        {
            HumanRobotCollisions++;

            AudioSource.PlayClipAtPoint(Clip, other.gameObject.transform.position);

            //var btn = (GD.ButtonStartAlert)FindObjectOfType(typeof(GD.ButtonStartAlert));
            var iK = (GD.InverseKinematics)FindObjectOfType(typeof(GD.InverseKinematics));
            var iK2 = (GD.IK2)FindObjectOfType(typeof(GD.IK2));

            btn.RobotHandCollision = true;
            btn.HaveToBlink = true;
            btn.CurrentRobotHandCollisionNumber = HumanRobotCollisions;
            btn.TotalCollisions++;
```

```
        btn.R1Destination = iK.Destination;
        btn.R2Destination = iK2.Destination;

        iK.DontDrop = true;
        iK2.DontDrop = true;

        iK.Destination = GameObject.Find("HandCollisionReset1").transform;
        iK2.Destination = GameObject.Find("HandCollisionReset2").transform;

        if (other.gameObject.name == "HAND" || other.gameObject.name == "Obstacle")
        {
           var h1 = GameObject.Find("H1").GetComponent<Timing>();
           var time = h1.Timer.ReturnTime();

           Logfile.Write($"{time.ToString()} sec : Collision {HumanRobotCollisions} occured between
Robot1 and hand.");
        }
        else if (other.gameObject.name == "HAND2")
        {
           var h1 = GameObject.Find("H1").GetComponent<Timing>();
           var time = h1.Timer.ReturnTime();

           Logfile.Write($"{time.ToString()} sec : Collision {HumanRobotCollisions} occured between
Robot2 and hand.");
        }
      }
      else
      {
        return;
      }
   }
}
```

Screws

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

public class Screws : MonoBehaviour
{
   [Header("Joints")]
   public GameObject WaitForThis;
   public bool IsDone = false;
}
```

Timing

```
using Assets.GD.IK.Scripts;
using GD;
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Timing : MonoBehaviour
{
   public bool StartedTask = false;

   public bool StartedTimer = false;
   public Timer Timer;

   public bool Robot1Done = false;
   public bool Robot2Done = false;
   public bool HumanDone = false;

   public bool TaskFinished = false;
   // Start is called before the first frame update
   void Start()
   {

   }

   // Update is called once per frame
   void Update()
   {
      if (!StartedTimer)
      {
         if (StartedTask)
         {
            StartedTimer = true;

            Timer.Start();
            Logfile.Write(Environment.NewLine + DateTime.Now.ToString() + " - TASK STARTED.");
         }
      }

      if (Robot1Done && Robot2Done && HumanDone && !TaskFinished)
      {
         var totalCollisions =
GameObject.Find("Button").GetComponent<ButtonStartAlert>().TotalCollisions;

         Logfile.Write($"{Timer.ReturnTime()} sec : TASK FINISHED. {totalCollisions} total collisions.");
         TaskFinished = true;
      }
   }
```

```
}
```

RenderScrewHoles

```
using GD;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class RenderScrewHoles : MonoBehaviour
{
   private bool AnticipationEnabled;

   public GameObject PreviousLoc;

   public GameObject CorrespondingScrew;

   public bool LocIsDone;

   private bool IsFirstLoc;

   public bool IsLastLoc;

   public GameObject WaitForDone;
   // Start is called before the first frame update
   void Start()
   {
      AnticipationEnabled =
GameObject.Find("OPTIONS").GetComponent<OPTIONS>().RobotAnticipation;

      if (!IsLastLoc && PreviousLoc == null)
      {
         IsFirstLoc = true;
      }
   }

   // Update is called once per frame
   void Update()
   {
      if (!GameObject.Find("H1").GetComponent<Timing>().StartedTask)
      {
         return;
      }

      if (!AnticipationEnabled)
      {
         return;
      }
```

```
if (!IsFirstLoc)
{
   if (CorrespondingScrew.GetComponent<Screws>().IsDone)
   {
      PreviousLoc.GetComponent<RenderScrewHoles>().LocIsDone = true;
   }
}

bool canEnable;
if (WaitForDone == null)
{
   canEnable = true;
}
else
{
   canEnable = WaitForDone.GetComponent<Screws>().IsDone;
}

Renderer renderer = GetComponent<Renderer>();

if (IsFirstLoc && canEnable)
{
   var button = GameObject.Find("Button");
   if (!button.GetComponent<ButtonStartAlert>().RobotHandCollision)
   {
      if (!IsFlashing)
      {
         StartCoroutine(ColorFlash());
      }
      //renderer.material.color = Color.blue;
      return;
   }
   else
   {
      renderer.material.color = Color.black;
   }
}

if (IsLastLoc)
{
   if (LocIsDone)
   {
      renderer.enabled = false;
      return;
   }
}

if (PreviousLoc != null)
{
   if (PreviousLoc.GetComponent<RenderScrewHoles>().LocIsDone && canEnable)
```

```
        {
            var button = GameObject.Find("Button");
            if (!button.GetComponent<ButtonStartAlert>().RobotHandCollision)
            {
                if (!IsFlashing)
                {
                    StartCoroutine(ColorFlash());
                }
                return;
            }
            else
            {
                renderer.material.color = Color.black;
            }
        }
    }

    private bool IsFlashing = false;
    public float timeVisible = 0.3f;
    public float timeInvisible = 0.3f;
    public bool startState = true;
    public bool endState = true;
    public IEnumerator ColorFlash()
    {
        IsFlashing = true;

        Renderer renderer = GetComponent<Renderer>();
        renderer.enabled = startState;

        yield return new WaitForSeconds(0.1f);

        while (!GameObject.Find("Button").GetComponent<ButtonStartAlert>().RobotHandCollision &&
!LocIsDone)
        {
            if (startState)
            {
                renderer.material.color = Color.blue;
                yield return new WaitForSeconds(timeInvisible);
                renderer.material.color = Color.black;
                yield return new WaitForSeconds(timeVisible);
            }
            else
            {
                renderer.material.color = Color.black;
                yield return new WaitForSeconds(timeVisible);
                renderer.material.color = Color.blue;
                yield return new WaitForSeconds(timeInvisible);
            }
        }
        renderer.material.color = Color.black;
```

```
        IsFlashing = false;

    }
}
```

RenderRAReleases

```
using GD;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class RenderRAReleases : MonoBehaviour
{
    private bool AnticipationEnabled;

    public GameObject PreviousRARelease;

    public bool ReleaseIsDone;

    private bool IsFirstRARelease;

    public bool IsLastRARelease;

    public GameObject WaitForDone;

    private Color OriginalColor;
    // Start is called before the first frame update
    void Start()
    {
        OriginalColor = this.GetComponent<Renderer>().material.color;

        AnticipationEnabled =
GameObject.Find("OPTIONS").GetComponent<OPTIONS>().RobotAnticipation;

        GetComponent<Renderer>().enabled = false;

        if (!IsLastRARelease && PreviousRARelease == null)
        {
            IsFirstRARelease = true;
        }
    }

    // Update is called once per frame
    void Update()
    {
        if (!GameObject.Find("H1").GetComponent<Timing>().StartedTask)
        {
            return;
        }
```

```
if (!AnticipationEnabled)
{
   return;
}

if (ReleaseIsDone)
{
   GetComponent<Renderer>().enabled = false;
   return;
}

bool canEnable;
if (WaitForDone == null)
{
   canEnable = true;
}
else
{
   canEnable = WaitForDone.GetComponent<Screws>().IsDone;
}

Renderer renderer = GetComponent<Renderer>();
if (IsFirstRARelease)
{
   var button = GameObject.Find("Button");
   if (!button.GetComponent<ButtonStartAlert>().RobotHandCollision)
   {
      if (!IsFlashing)
      {
         StartCoroutine(ColorFlashRA());
      }
      return;
   }
   else
   {
      renderer.material.color = OriginalColor;
   }
}

if (IsLastRARelease)
{
   if (ReleaseIsDone)
   {
      renderer.enabled = false;
      return;
   }
}

if (PreviousRARelease != null)
{
```

```
        if (PreviousRARelease.GetComponent<RenderRAReleases>().ReleaseIsDone && canEnable)
        {
            var button = GameObject.Find("Button");
            if (!button.GetComponent<ButtonStartAlert>().RobotHandCollision)
            {
                if (!IsFlashing)
                {
                    StartCoroutine(ColorFlashRA());
                }
                return;
            }
            else
            {
                renderer.material.color = OriginalColor;
            }
        }
    }

    private bool IsFlashing = false;
    public float timeVisible = 0.3f;
    public float timeInvisible = 0.3f;
    public bool startState = true;
    //public bool endState = true;
    private IEnumerator ColorFlashRA()
    {
        IsFlashing = true;

        Renderer renderer = GetComponent<Renderer>();
        renderer.enabled = startState;

        yield return new WaitForSeconds(0.1f);

        while (!GameObject.Find("Button").GetComponent<ButtonStartAlert>().RobotHandCollision &&
!ReleaseIsDone)
        {
            if (startState)
            {
                renderer.material.color = Color.cyan;
                yield return new WaitForSeconds(timeInvisible);
                renderer.material.color = OriginalColor;
                yield return new WaitForSeconds(timeVisible);
            }
            else
            {
                renderer.material.color = OriginalColor;
                yield return new WaitForSeconds(timeVisible);
                renderer.material.color = Color.cyan;
                yield return new WaitForSeconds(timeInvisible);
            }
        }
```

```
        if (ReleaseIsDone)
        {
            renderer.enabled = false;
        }
        else
        {
            renderer.material.color = OriginalColor;
        }

        IsFlashing = false;
    }
}
```

IK2EffectorCollision

```
using Assets.GD.IK.Scripts;
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Ik2EffectorCollision : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        prev = transform.position;
        maxspeed = 0;
        prevTime = Time.fixedTime;

        AllSpeeds = 0;
        SpeedsCount = 0;
    }

    public float AllSpeeds;
    public int SpeedsCount;

    private Vector3 prev;
    private float prevTime;

    private Vector3 curr;
    private float currTime;

    private float speed;
```

```
public float maxspeed;
private void FixedUpdate()
{
   if (transform.position.y > 2)
   {
      return;
   }

   curr = transform.position;
   currTime = Time.fixedTime;

   if (curr == prev)
   {
      return;
   }

   speed = (curr - prev).magnitude / (currTime - prevTime);

   if (speed > maxspeed)
   {
      maxspeed = speed;
   }

   if (speed > 0)
   {
      AllSpeeds += speed;
      SpeedsCount++;
   }

   prev = curr;
   prevTime = currTime;
}

// Update is called once per frame
void Update()
{

}

private int NearCollisions = 0;
private int Collisions = 0;
private Transform EffectorsTarget = null;

private void OnCollisionEnter(Collision collision)
{

   if (collision.gameObject.name == "JOINT 4" && EffectorsTarget == null)
   {
      var h1 = GameObject.Find("H1").GetComponent<Timing>();
      var time = h1.Timer.ReturnTime();
```

```
            NearCollisions++;
            Logfile.Write($"{time.ToString()} sec : Near-collision {NearCollisions} between the robots.");

            var oppositeRobotHand = GameObject.Find("JOINT 4");

            if (this.transform.position.y >= oppositeRobotHand.transform.position.y)
            {
               Collisions++;

               var iK2 = (GD.IK2)FindObjectOfType(typeof(GD.IK2));

               EffectorsTarget = iK2.Destination;

               iK2.Destination = null;
            }
         }
      }

      private void OnCollisionExit(Collision collision)
      {
         if ((collision.gameObject.name == "JOINT 4") && (Collisions > 0) )
         {
            Collisions--;
         }

         if (Collisions == 0 && EffectorsTarget != null)
         {
            var iK2 = (GD.IK2)FindObjectOfType(typeof(GD.IK2));

            iK2.Destination = EffectorsTarget;

            EffectorsTarget = null;
         }
      }
   }
```

EffectorCollision

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

public class EffectorCollision : MonoBehaviour
{
   // Start is called before the first frame update
```

```
void Start()
{
   prev = transform.position;
   maxspeed = 0;
   prevTime = Time.fixedTime;

   AllSpeeds = 0;
   SpeedsCount = 0;
}

public float AllSpeeds;
public int SpeedsCount;

private Vector3 prev;
private float prevTime;

private Vector3 curr;
private float currTime;

private float speed;
public float maxspeed;
private void FixedUpdate()
{
   if (transform.position.y > 2)
   {
      return;
   }

   curr = transform.position;
   currTime = Time.fixedTime;

   if (curr == prev)
   {
      return;
   }

   speed = (curr - prev).magnitude / (currTime - prevTime);

   if (speed > maxspeed)
   {
      maxspeed = speed;
   }

   if (speed > 0)
   {
      AllSpeeds += speed;
      SpeedsCount++;
   }

   prev = curr;
   prevTime = currTime;
```

```
    }
    // Update is called once per frame
    void Update()
    {

    }

    private int Collisions = 0;
    public Transform EffectorsTarget = null;

    private void OnCollisionEnter(Collision collision)
    {

        if (collision.gameObject.name == "JOINT 24" && EffectorsTarget == null)
        {
            var oppositeRobotHand = GameObject.Find("JOINT 24");

            if (this.transform.position.y > oppositeRobotHand.transform.position.y)
            {
                Collisions++;

                var iK = (GD.InverseKinematics)FindObjectOfType(typeof(GD.InverseKinematics));

                EffectorsTarget = iK.Destination;

                iK.Destination = null;
            }
        }
    }

    private void OnCollisionExit(Collision collision)
    {
        if ((collision.gameObject.name == "JOINT 24") && (Collisions >0))
        {
            Collisions--;
        }

        if (Collisions == 0 && EffectorsTarget != null)
        {
            var iK = (GD.InverseKinematics)FindObjectOfType(typeof(GD.InverseKinematics));

            iK.Destination = EffectorsTarget;

            EffectorsTarget = null;
        }
    }
}
```

Timer

```
using System;
using UnityEngine;

[Serializable]
public struct Timer {

   public float startTime;
   public float duration;

   public Func<float> GetTime;

   public void Start (bool restart = true)
   {
      GetTime = () => { return Time.time; };

      // If it is done
      // Or if it is not done, but it is set to restart anyway...
      if (
         IsDone()
         ||
         ( ! IsDone() && restart)
         )
         startTime = GetTime();
   }

   public bool IsDone ()
   {
      return GetTime() >= startTime + duration;
   }

   // From zero to one, how complete is it
   public float GetNormalised ()
   {
      if (duration == 0)
         return 1f;
      return Mathf.Clamp01(   (GetTime() - startTime) / (duration)    );
   }

   public float ReturnTime()
   {
      return GetTime() - startTime;
   }
}
```

ButtonStartAlert

```
using Assets.GD.IK.Scripts;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace GD
{
   public class ButtonStartAlert : MonoBehaviour
   {
      [Header("Ready")]
      public bool Continue = false;

      public bool RobotHandCollision;
      public int CurrentRobotHandCollisionNumber;
      public int TotalCollisions;

      public AudioClip Clip;

      public Transform R1Destination;
      public Transform R2Destination;

      private float TimingCollisionsStart;
      // Start is called before the first frame update
      void Start()
      {
         RobotHandCollision = false;

         R1Destination = null;
         R2Destination = null;

         TotalCollisions = 0;

         OriginalColor = this.GetComponent<Renderer>().material.color;
      }

      // Update is called once per frame
      public bool HaveToBlink = false;
      private bool StopBlinking = false;
      void Update()
      {
         if (HaveToBlink)
         {
            HaveToBlink = false;

            if (IsRunning)
            {
               try { StopCoroutine(buttonblink()); }
               catch { }
            }
            var H1 = GameObject.Find("H1").GetComponent<Timing>();
```

```
            TimingCollisionsStart = H1.Timer.ReturnTime();
            StartCoroutine(buttonblink());
        }
    }

    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.name == "ButtonBase"
            || collision.gameObject.tag.Contains("RobotHandCollision"))
        {
            return;
        }
    }

    private void OnTriggerEnter(Collider collision)
    {
        if (R1Destination == null && R2Destination == null)
        {
            return;
        }

        if ((collision.gameObject.name.Contains("GrabVolumeBig")) || (Continue))
        {
            AudioSource.PlayClipAtPoint(Clip, gameObject.transform.position);

            var h1 = GameObject.Find("H1").GetComponent<Timing>();
            var time = h1.Timer.ReturnTime();
            Logfile.Write($"{time} sec : Continuing after robot-hand collision
{CurrentRobotHandCollisionNumber}. Delayed {(time - TimingCollisionsStart).ToString()} sec.");

            Continue = false;
            StopBlinking = true;

            if (RobotHandCollision)
            {
                if (R1Destination != null)
                {
                    var iK = (GD.InverseKinematics)FindObjectOfType(typeof(GD.InverseKinematics));
                    iK.Destination = R1Destination;

                    iK.DontDrop = false;
                    R1Destination = null;
                }

                if (R2Destination != null)
                {
                    var iK = (GD.IK2)FindObjectOfType(typeof(GD.IK2));
                    iK.Destination = R2Destination;

                    iK.DontDrop = false;
                    R2Destination = null;
```

```
            }

            RobotHandCollision = false;
        }
    }
}


private Color OriginalColor;
private int Countdown = 0;
private float timeVisible = 0.3f;
private float timeInvisible = 0.3f;
private bool startState = true;
public bool IsRunning = false;
public IEnumerator buttonblink()
{
    IsRunning = true;

    Renderer renderer = GetComponent<Renderer>();
    renderer.enabled = startState;

    yield return new WaitForSeconds(Countdown);

    while (!StopBlinking)
    {
        if (startState)
        {
            renderer.material.color = Color.yellow;
            yield return new WaitForSeconds(timeInvisible);
            renderer.material.color = OriginalColor;
            yield return new WaitForSeconds(timeVisible);
        }
        else
        {
            renderer.material.color = OriginalColor;
            yield return new WaitForSeconds(timeVisible);
            renderer.material.color = Color.yellow;
            yield return new WaitForSeconds(timeInvisible);
        }
    }
    StopBlinking = false;
    renderer.material.color = OriginalColor;

    IsRunning = false;
}
}
}
```

HTargets

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;
using Assets.GD.IK.Scripts;
using System;

public class HTargets : MonoBehaviour
{
   public List<string> HObjects;

   public bool ManuallyEnableCollider;

   public bool IsConsideredGrabbed;
   // Start is called before the first frame update
   void Start()
   {
      HObjects = new List<string>();

      HObjects.Add("H1");
      HObjects.Add("H2");
      HObjects.Add("H3");
      HObjects.Add("H4");

      if (GameObject.Find("H5") != null)
      {
         HObjects.Add("H5");
      }
      if (GameObject.Find("H6") != null)
      {
         HObjects.Add("H6");
      }
      if (GameObject.Find("H7") != null)
      {
         HObjects.Add("H7");
      }
      if (GameObject.Find("H8") != null)
      {
         HObjects.Add("H8");
      }
      if (GameObject.Find("H9") != null)
      {
         HObjects.Add("H9");
      }

      foreach (var item in HObjects)
      {
         GameObject.Find(item + "target").GetComponent<Renderer>().enabled = false;
```

```
            GameObject.Find(item + "target").GetComponent<BoxCollider>().enabled = false;
        }

        //StartCoroutine(H1Started());
    }

    // Update is called once per frame
    void Update()
    {
        var target = GameObject.Find(HObjects.FirstOrDefault());
        if (target == null)
        {
            return;
        }

        GameObject waitingForDone = target.GetComponent<HumanTargets>().EnableIfThisIsDone;

        bool canEnable;
        if (waitingForDone == null)
        {
            canEnable = true;
        }
        else
        {
            canEnable = waitingForDone.GetComponent<Screws>().IsDone;
        }

        if ((target.transform.GetComponent<OVRGrabbable>().isGrabbed && canEnable) ||
(IsConsideredGrabbed && canEnable))
        {
            var targetname = target.name + "target";
            var targetplace = GameObject.Find(targetname);

            if (targetplace == null) return;

            var rend = targetplace.GetComponent<Renderer>();
            var coll = targetplace.GetComponent<BoxCollider>();

            rend.enabled = true;

            coll.enabled = true;
        }

        if (!target.transform.GetComponent<OVRGrabbable>().isGrabbed &&
!IsConsideredGrabbed)//&& Input.GetKeyDown(KeyCode.Alpha2))
        {
            var targetname = target.name + "target";
            var targetplace = GameObject.Find(targetname);

            if (targetplace == null) return;
```

```
            var rend = targetplace.GetComponent<Renderer>();
            var coll = targetplace.GetComponent<BoxCollider>();

            if (rend.enabled)
            {
                rend.enabled = false;
            }

            if (coll.enabled)
            {
                coll.enabled = false;
            }
        }
    }

    private void LateUpdate()
    {
        var target = GameObject.Find(HObjects.FirstOrDefault());

        if (target == null)
        {
            return;
        }

        if (target.GetComponent<Screws>().IsDone)
        {
            HObjects.RemoveAt(0);
        }
    }
}
```

HumanTargets

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;
using Assets.GD.IK.Scripts;
using System;
using GD;

public class HumanTargets : MonoBehaviour
```

```
{
    public GameObject EnableIfThisIsDone;

    public AudioClip Clip;

    public bool ForceStart = false;

    private Color OriginalColor;
    public bool IsFirstOne;
    public bool Started = false;
    public bool IsLastOne;
    // Start is called before the first frame update
    void Start()
    {
        OriginalColor = this.GetComponent<Renderer>().material.color;

        this.GetComponent<OVRGrabbable>().allowOffhandGrab = false;
    }

    private bool Blinked = false;
    // Update is called once per frame
    void Update()
    {
        var currentTargetName =
GameObject.Find("HumanTask").GetComponent<HTargets>().HObjects.FirstOrDefault();

        if (GameObject.Find("Button").GetComponent<ButtonStartAlert>().RobotHandCollision)
        {
            if (!this.GetComponent<OVRGrabbable>().isGrabbed)
            {
                if (this.name == currentTargetName && Blinked)
                {
                    this.GetComponent<OVRGrabbable>().allowOffhandGrab = false;

                    try { StopCoroutine(blink()); }
                    catch { }

                    Blinked = false;
                }
            }
        }

        if (this.name == currentTargetName && !Blinked &&
!GameObject.Find("Button").GetComponent<ButtonStartAlert>().IsRunning)
        {
            Blinked = true;
            this.GetComponent<OVRGrabbable>().allowOffhandGrab = true;
            StartCoroutine(blink());
        }

        if (!Started)
```

```
    {
        if ((IsFirstOne && this.GetComponent<OVRGrabbable>().isGrabbed) || ForceStart)
        {
            Started = true;

            try { this.GetComponent<Timing>().StartedTask = true; }
            catch { }
        }
    }
}

//public int Countdown = 0;
public float timeVisible = 0.3f;
public float timeInvisible = 0.3f;
public bool startState = true;
public bool endState = true;
public IEnumerator blink()
{

    Renderer renderer = GetComponent<Renderer>();
    renderer.enabled = startState;

    yield return new WaitForSeconds(0.1f);

    while (!this.transform.GetComponent<OVRGrabbable>().isGrabbed &&
(!this.GetComponent<Screws>().IsDone) &&
!GameObject.Find("Button").GetComponent<ButtonStartAlert>().RobotHandCollision)
    {
        if (startState)
        {
            renderer.material.color = Color.green;
            yield return new WaitForSeconds(timeInvisible);
            renderer.material.color = OriginalColor;
            yield return new WaitForSeconds(timeVisible);
        }
        else
        {
            renderer.material.color = OriginalColor;
            yield return new WaitForSeconds(timeVisible);
            renderer.material.color = Color.green;
            yield return new WaitForSeconds(timeInvisible);
        }
    }
    renderer.material.color = OriginalColor;


}

public IEnumerator blink2()
{
```

```
    Renderer renderer = GetComponent<Renderer>();
    renderer.enabled = startState;

    yield return new WaitForSeconds(0.1f);

    int blinks = 0;

    while (blinks < 2)
    {
        if (startState)
        {
            renderer.material.color = Color.green;
            yield return new WaitForSeconds(timeInvisible);
            renderer.material.color = OriginalColor;
            yield return new WaitForSeconds(timeVisible);
        }
        else
        {
            renderer.material.color = OriginalColor;
            yield return new WaitForSeconds(timeVisible);
            renderer.material.color = Color.green;
            yield return new WaitForSeconds(timeInvisible);
        }
        blinks++;
    }
    renderer.material.color = OriginalColor;
}
private void OnCollisionStay(Collision collision)
{
    if (!this.name.Contains("H") || collision.gameObject.name == "HumanTask")
    {
        return;
    }
    else if ((this.name + "target") == collision.gameObject.name)
    {
        if (!this.transform.GetComponent<OVRGrabbable>().isGrabbed)
        {
            StopCoroutine(blink());

            this.transform.position = collision.gameObject.transform.position;
            this.transform.rotation = collision.gameObject.transform.rotation;

            collision.gameObject.SetActive(false);

            this.GetComponent<Rigidbody>().useGravity = false;
            this.GetComponent<Rigidbody>().isKinematic = true;

            AudioSource.PlayClipAtPoint(Clip, this.transform.position);

            this.GetComponent<Screws>().IsDone = true;
            this.GetComponent<OVRGrabbable>().allowOffhandGrab = false;
```

```
            if (IsLastOne)
            {
                var h1 = GameObject.Find("H1").GetComponent<Timing>();
                var time = h1.Timer.ReturnTime();

                h1.HumanDone = true;
                Logfile.Write($"{time.ToString()} sec : Human - task FINISHED.");
            }
        }
    }
}
```

LogFile

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;
using UnityEngine.SceneManagement;

namespace Assets.GD.IK.Scripts
{
    static class Logfile
    {
        private static string SceneName;
        private static string Name;
        private static bool Anticipation;
        private static string Task;
        public static void Write(string log)
        {
            SceneName = SceneManager.GetActiveScene().name;

            Name = GameObject.Find("OPTIONS").GetComponent<OPTIONS>().Participator;
            Anticipation = GameObject.Find("OPTIONS").GetComponent<OPTIONS>().RobotAnticipation;

            if (string.IsNullOrWhiteSpace(Name))
            {
                Name = "EmptyName";
            }

            if (Anticipation)
            {
```

```
            Task = "Anticipation";
        }
        else
        {
            Task = "Normal";
        }

        log += Environment.NewLine;

        string path = Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory);

        path += $"\\Vr Data\\{DateTime.Today.ToString("MM_dd_yyyy")}\\{Name}";

        Directory.CreateDirectory(path);

        try
        {
            System.IO.File.AppendAllText(path + $"\\{SceneName}_{Task}.txt", log);
        }
        catch
        { }
    }
  }
}
```