# Towards Digital Twins for soccer robots
# a use case in reusing artifacts

Walravens, Gijs

*Award date:*
2022


Department of Mathematics and Computer Science
Software Engineering and Technology Cluster

# Towards Digital Twins for soccer robots: a use case in reusing artifacts

*Master Thesis*

G. Walravens
Student ID: 0904152

**Tutor:** Hossain Muhammad Muctadir
**Supervisor:** Loek Cleophas

Eindhoven, Monday 7th March, 2022

# Abstract

Digital twins (DT) and related concepts have recently received significant attention both in industry and academia. A DT can be seen as a pair of two entities, one physical and one virtual, between which historical or real time data exchange takes place. The virtual entity is then able to mimic its physical counterpart very closely, both visually and functionally. Using DTs it is possible to gain a greater understanding of the entity in question, develop tools to better analyze the process, and ease testing, to name a few.

Various sources propose that the ideal time to start developing a DT is during the prototyping phase of a product. With DTs being a relatively new concept, quite often the physical counterpart already exists. Given this context, a different approach may thus be necessary. In particular, the reuse of existing artifacts that have been created over the years is of interest. These could potentially reduce the time to realize the DT, and make the DT easier to maintain and use due to the reuse of already applied technologies. Present research on DTs does not cover this specific aspect of development, making it an interesting candidate for an engineering case study which investigates the possible benefits that can be reaped from the reuse of artifacts.

The goal of this project is to take steps towards realizing a DT for Eindhoven University of Technology's robot soccer team Tech United's soccer robots, most notably for the virtual entity. Tech United's soccer robot team is a well-supported project that has existed for over ten years. This makes it an excellent candidate for exploring the advantages and disadvantages of reusing artifacts from an existing project in creating a DT. The currently existing artifacts, such as 3D CAD models, Matlab models and simulator, were utilized to aid in creating this DT. We have underlined which techniques we applied in our creation process, how the artifacts have saved us a significant amount of time, but also that they were not without issues. Lastly we have also explored possible future applications and expansions in the context of robot soccer as well.

# Preface

This thesis project marks the conclusion of my Master in Computer Science Engineering at the Eindhoven University of Technology. The skills and knowledge I acquired in this study and the preceding bachelor study at the same university culminated together in this project.

This graduation project has taken just over six months to complete. During this time I have worked together with members of the Software Engineering and Technology Cluster, and with Tech United themselves. They have assisted me thoroughly throughout the course of the thesis from start to finish, and without their kind assistance I would not have been able to realize this project.

I would like to express my thanks to Loek Cleophas and Hossain Muhammad Muctadir for allowing me to choose this thesis subject that they proposed. A special mention goes out to David Manrique Negrin as well for his input in our meeting and coming up with great ideas. Furthermore many thanks to them for guiding me, reviewing my report many times over and inspiring me in the choices made along the way. A big thanks also goes out to Tech United, especially to Ferry Schoenmakers, for assisting me in this project. Without their robots and their guidance this thesis would not have been possible. I would also like to thank my parents for working so hard to support me in these times such that I can actually do this project in the first place. Finally, thanks to my great friends for being there for me to take my mind of things from time to time, with special thanks to Dario Babić , Bas Joosten, Hugo Dik, Stijn Akkermans, Melissa van Hoorn and Bram van den Berg.

Gijs Walravens

Son en Breugel, February 2022

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction

This chapter will introduce the topic at hand and is structured as follows. First Section 1.1 will describe the relevant context of the project. Then Section 1.2 will describe a conceptual definition of a digital twin. Lastly in Section 1.3 the goal of this thesis project shall be described.

## 1.1 Project context

Tech United [2] is a multidisciplinary organization consisting of (former) students, PhD's and TU/e employees. One of the main parts of this group is a robot soccer team that they build, maintain, and compete with. The so called Turtle robots [3] on this team observe the status of the field, communicate with each other and execute strategies and tactics versus other soccer robot teams in order to gain victory in a similar way to human soccer. As one can imagine, this is a quite complex endeavour, with many people involved specialized in helping bring many different systems and components together; the robot has to be built, the software has to be written, and, after all, it also has to be maintained. In order to aid in completing this venture, the team has created several tools. Most notably, besides the physical robot itself there are two more entities which the software can run on, namely a 2D simulator and a rudimentary 3D simulator.

The simulators use the same Matlab models that drive the actual robots. They behave, work together, and communicate just like the real robots. Furthermore the simulators also use the same communication channels as the actual robots do. This allows the simulators to be a quite auspicious tool for developing and maintaining the robots, that can work as a stand in when the actual robot is not available.

While these simulators are a great help in testing the robot's behaviour (both individually and working together as a team), the user might still miss out on many of the intricacies of the system; how are the robots and all their subsystems behaving in real time? How are they reacting to their environment, and are there any possible issues with any of the components? Since the robot is a quite complex machine, having just a simple sprite representing it means there is no visual representation of feedback from the hardware subsystems' current state. Furthermore if one wants to take a look at certain components, the only way would be to take a look at (and possibly take apart) the physical robot. Other more complex development work, testing, or analyzing the robot's behaviour would also require access to the physical robots, and go through a complete setup to create for instance a mock-up game. More complex undertakings such as conducting predictive maintenance, troubleshooting from afar, or testing the physical behaviour of new component candidates would require new, separate simulations or models to be built. Ideally some solution would be realized that can aid in the many aspects of developing, using, and improving the Turtle robot, while making use of the artifacts that are already available. One form that this solution can possibly take would be a *Digital Twin* (DT).

## 1.2 Definition of a Digital Twin

The concept of digital twins was first introduced by Grieves in 2002 [4]. In earlier stages this concept was also known as the "Mirrored Spaces Model" and the "Information Mirroring Model", before settling on the name digital twin. The model of the digital twin according to Grieves' definition is that every system is not an entity by itself, but actually a pairing between two systems; a physical version and a virtual version. The physical version, e.g. an assembly machine, would operate in the real space. Its digital counterpart operates on the same principles in virtual space [5].

Tao et al. [1] expands on this and defines a digital twin as a 5 dimensional system consisting of a physical entity (the robot in this case), a virtual entity (which mimics the robot), services for both the physical and virtual entities (such as a monitoring service, energy consumption service, calibration service), data that gets exchanged between the digital twin parts, and connections between all of these entities.

One of the first notable applications of digital twins is that by NASA and the US Air Force [6]. The usage of the digital twin in that use case is described as the integration of a high fidelity simulation with subsystems such as health management and the (historic) data for their vehicles. The digital twin was capable of going through all events that its physical counterpart was capable of as well. This would allow for more extensive and effective development, while saving resources that are normally spent on i.e. physical testing. Once the physical side of the system is deployed, the digital twin would continue to provide benefit by allowing continuous monitoring, supervision, and optimizing.

This provides a beneficial approach to how the data flow works in a DT. In a DT the computations on the virtual side of the DT can lead to an adjustment of the physical side of the DT. This can happen automatically, such as through the adjustment of operating parameters, or manually, by e.g. adding an extra sensor based on the found results.

Some sources also provide a concept similar to the DT that is less strict in it's requirements, namely the *Digital Shadow* (DS). The main difference between a digital twin and a digital shadow is that in a DT there are connections back and forth between the DT and the physical real life system [7]. A change in state in either the physical object or the digital object would lead to a change in the other object as well. For a digital shadow this is not the case. In a DS data is fed in a one-way flow, where a change in the physical object will lead to a change in the digital object, but not vice versa.

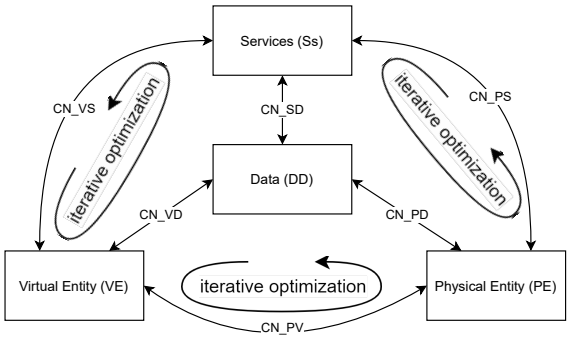



Figure 1.1: Illustration of the 5D DT taken from [1]

In the high tech systems sector, use cases can be found in the fields of automation [8] and manufacturing applications [9]. For example, Vijayakumar et al. [10] presents an industry application of digital twins. This paper proposes a digital twin approach for a manufacturing facility. The ease of simulation that a digital twin offers compared to the more traditional approaches allows for more effective and more time efficient ways of simulating the facility's operations. This can be used to find out e.g. ideal facility layout and forklift management. Furthermore a digital twin is more resilient to the many changes that the system often has to go through.

However, the benefits experienced here are of possible relevance to a soccer robot as well: since a digital twin would allow the users to simulate and virtually take apart the robot from their own computer without needing access to the actual robot, it could help greatly in developing and maintaining the robot [11], especially when working with multiple people at the same time. Fur-

thermore, the robots can be used to simulate full games in a life-like environment [12], supported by visual feedback, replays that allow for free camera movement, and automated data gathering and analysis. Even from the comfort of their own home one would then be able to set up experiments, gauge the team's performance, fine tune and analyze the tactics and strategy without the need for a single physical Turtle robot.

However, this might pose the next question: how does this notion of digital twins differ from simulations and models. Wright et al. [13] describes the threat to the potential of this technology stemming from confusion in its definition. With different interpretations among the wide range of applications, the concept may just be seen as the latest hype or buzzword. Wright defines that "a digital twin without a physical twin is a model". To give an analogy for this requirement; biological twins are created at the same time, look the same at that moment and (to trivialize it here), grow up and change simultaneously as well. Digital twins should adhere to this principle too. As a result of this it would be implied that the concept only starts to play a part from the prototyping stage: use the test data from the prototype to update the parameters in the model of the prototype, use the updated model to e.g. predict performance, and then update the design based on the outcome. While the model used in a digital twin would ideally be as accurate as needed for the intended purpose(s), this may not always be feasible. Processes like physics calculations may prove to be too computationally intense. As a summary, Wright notes that a model used in a digital twin is:

- sufficiently physics-based that updating parameters within the model based on measurement data is a meaningful thing to do,
- sufficiently accurate that the updated parameter values will be useful for the application of interest, and,
- sufficiently quick to run that decisions about the application can be made within the required timescale.

These three criteria allude to three aspects that are of importance for a DT. Notice by the abundant presence of the term 'sufficiently' here that these aspects tend to differ quite a bit on a case by case basis. The existence of a model and its properties that influence these three criteria heavily impact the possible applications of a digital twin for that system. What the digital twin will eventually be used for may affect the significance of each of these point as well. For example, with a highly physics-based model use case that tests a system's safety, accuracy may be of greater importance than performance. Krasikov et al. [14] provides a comparison between traditional simulations and digital twins as well. Here it is mentioned that simulations are mostly used in the early stage of the lifecycle, try to emulate what may happen, and only analyze the processes inside the model. digital twins on the other hand follow the entity through its lifecycle, know what is going on in the exact moment, and know of all processes inside the model and interactions between the physical and digital counterparts.

Compared to a simulation, a DT can best be seen as a more encompassing digital recreation of the system as a whole. All parts of the system and the environment in which it operates are included. A simulation is a more specific digital representation of (a part of) the system, to investigate a predefined goal (for example, the amount of water pressure a fluid system setup can take). A DT on the other hand is more of a platform, upon which many things can tested, changed and new applications like remote troubleshooting and predictive maintenance applied. DTs are thus also capable of being used for simulations, but have functionalities reaching beyond this as well.

For the remainder of this report, we shall adhere to Tao et al.'s 5D definition of a DT, and refer back to it multiple times when we describe the creation process of our DT.

## 1.3 Project objectives & Research questions

The objective of this master thesis project is to investigate the reuse of existing artifacts in the context of DT development, especially in regards to the DT's virtual entity component. We will do this through a use case where we apply the concept of DT to the earlier described Tech United turtle soccer robots. During this process we will document our experiences in creating a DT for a well established project. Of interest will be what artifacts we can reuse, what work is necessary to make them usable, and what benefit the reuse brings us compared to creating assets from scratch. In the context of this specific use case, we will also explore some of the possible (future) applications that our DT can be used for, such that our work will also be of benefit to the turtle soccer robot project.

While we shall focus this project on the development of a DT, the great amount of time that this actually requires, and the pandemic limitations that we are faced with during the entirety of this thesis, might not make this attainable. In this case, we shall at least aim to achieve the digital shadow that we described in Section 1.2. Since the definitions of a DT and DS lie so closely together, their development can follow the same process, and any findings we would thus find along the way are relevant for either definition. To this end we will refer to just a DT from here on out. We will come back to the specifics of what we have created later when reflect back on our results.
To now concretize the goals of this project, the following research questions have been formulated:

- **RQ1**: What are the requirements for the virtual entity of a digital twin for turtle soccer robots?
- **RQ2**: What currently existing artifacts can be used in creating the virtual entity of the digital twin?
- **RQ3**: What tools and methods can we apply to create the virtual entity of a digital twin for the turtle soccer robot?
- **RQ4**: What discrepancies between the virtual and physical entities are of significance in a digital twin for the turtle soccer robots, and how can they be negated if necessary?
- **RQ5**: Given the available artifacts, what findings and functionalities can be realized using the digital twin?

We address RQ1 by defining the definition of DTs in Chapter 1 and by looking at some other applications of DTs in Chapter 2. Next we answer RQ2 in Chapter 3 by examining the current state of the turtle soccer robots and what artifacts are available to us. RQ3 we answer in Chapter 4, where we go over the approaches and methods that are possible to create our DT. After this we touch on RQ4 in. Lastly RQ5 will be answered by Chapter 5, where we will showcase the new findings and functionalities that we realized. We will return to these research questions in Chapter 6 and formulate concrete answers for them.

# Chapter 2

# Related work

In this chapter the literature and previous work regarding digital twins and their history will be reviewed. In Section 2.1 we will take a look at some of the previous use cases of DTs. Then in Section 2.2 we describe several of the advantages that a digital twin offers.

## 2.1 Use cases of Digital Twins

Having observed some of the digital twin concept in Chapter 1, we shall delve into some further work in this field here. Kuts et al. [15] presents work in which a digital twin of a robot arm was created. The CAD models created during the original development of the arm were used as a baseline to create a rigged model. This makes movement in a Digital Environment (in this case that of Unity3D) possible. Scripts for controlling the robot arm were added, as well as a virtual reality toolkit to allow users to observe and interact with the digital twin in VR. Using the digital twin the system can be more easily demonstrated or used for educational purposes. The robot arm contains a collision detection system used to prevent the robot from colliding in operation with unintended objects. This system is essential for the safe operation of the robot arm. Since the digital twin provides a high fidelity simulation of the physical device in a virtual space, this system can be tested without any material or personal risk. As long as the digital environment and 3D model are made to closely resemble the physical counterpart, a verification of correctness of the collision prevention system in the digital environment would translate to it working in the real world as well. Another advantage of the digital twin here is the capability of overriding the control of the arm from the real world. This means that it is possible to adjust the arm manually or change its behaviour without having any downtime on the production line. The real world counterpart of the arm can then just keep operating as usual.

Savolainen et al. [16] gives another example. Here a digital twin of a mine environment is created. The big advantage of this digital twin is the ability to simulate aspects of the environment in digital space that would be hard and costly to do in real time. Using simulation, different system configurations for the mine can be tested to find the most cost efficient setup with the greatest relative throughput, and long term implications can be analysed to figure out the ideal maintenance intervals.

A more elaborate account of the technical side is given by Ait-Alla et al. [17]. In this publication the connection between the physical and digital parts are explored. If one wishes to feed information from the real world to the virtual, there needs to be some connection. This connection can be fed with information using sensors. There is a balance in applying this however: too many sensors and there will be too much communication traffic; too few sensors and the data is not accurate enough. The experiment showcases that there is often not a clear-cut answer. Instead this should be tested on a case by case basis. This could be a prelude to the fact that while digital twins may save work in some areas, it can also create more overhead in new ones.

The usage of digital twins could also allow for the usage or integration of new, state of the art

technologies as well that previously might not have been envisioned: Matulis et al. [8] provides a use case in which another digital twin was made for a robotic arm. In this case however, the digital twin was used for the training of the operation of the robot arm by means of reinforcement learning. Since the digital twin offers a close to 1 to 1 representation of an actual physical robot arm, it would be possible to train a model to operate the robot arm, and then to apply this model to an actual physical robot arm. Since reinforcement learning requires many iterations to go through, digital twins are especially suited: instead of using a physical robot for this, with all the strain and operational costs that come with that, the digital counterpart can be used instead. Frequently 3D (CAD) models are already created during the design phase, which can be used to create the digital twin.

As observed from these previous examples, the integration of digital twins in the so called industry of the future, also known as Industry 4.0 [18], seems like a common context in which the digital twin sees use. Industry 4.0, also known as the Fourth Industrial Revolution, pertains to the increased automation in many areas of manufacturing and industry. With this comes the integration of smart technologies and IoT in traditional practices. Negri et al. [19] provides a review of the role of the digital twin in relation to Cyber Physical Systems, specifically in that of Industry 4.0. Indeed here many of the papers support the importance of the digital twin concept for this field. Yang et al. [20] argues how in Industry 4.0 one would need to be able to deal with the customization that products are nowadays expected, requiring a great deal of flexibility and adaptability. With technologies such as increased automation, IoT and cyber-physical systems, digital twins offer a way to deal with these aspects in manufacturing.

Aside from specific applications that can be developed using a DT, the high fidelity visualization that comes with a DT is a benefit in itself. A DT of the city of Zurich for example is not only used to model noise and air pollution, but also visualize architectural designs [21].

## 2.2 Advantages of a Digital Twin

We have seen some of the use cases of DTs now, and observed several of the benefits that were achieved. We argue that there the concept of DT provides several advantages that make it a worthwhile endevour to explore further and apply in the field. The work by Tao et al. [1] that was mentioned previously already speaks of several advantages that DTs can bring. These are:

- **Increased understanding**. Since it is a requirement of digital twins to have a high-fidelity model that is kept up to date with the physical counterpart, benefits can be reaped from this alone. A good 3D model will provide more immersion, overview and visibility of the object.
- **Reduced time to market**. A DT can provide developers of a product hands on insights about the product before it is fully completed. This can already root out possible failures that might have otherwise only been discovered post launch. Tying in with the previous point, the DT also gives a highly accurate visual representation of the product from which insights can be gained as well. This all can help in reducing the amount of iterations the design has to go through before the final release takes place.
- **Optimal operation**. As a DT offers a possibility for the system to be analyzed in different conditions in real-time, optimizations can be made in time to ensure everything works as intended.
- **Reducing energy consumption**. A DT allows the discovery of degraded components to be discovered earlier and replaced. Tests can also point out processes that can be started or stopped at set times to save energy.
- **Reducing maintenance cost**. Data generated by a DT can be used to predict lifetimes of components and possible future failures. Maintenance events can be planned at optimized times to mitigate possible issues from happening up. Predictive maintenance can be done to prevent critical failure or increased costs due to wear on more components.

- **Increasing user engagement**. Since users can interact with a high fidelity virtual model of the product before it is finalized, hands on experience and feedback can be acquired early. This allows adjustments to the design be made before they are fully realized in real life, leading to a better product at launch.
- **Fusing information technologies**. Since a DT provides a fully digitized counterpart to a physical system, digital information technologies can be more easily applied. Machine learning, (physics) simulations, modeling, cloud computing, big data integration, and IoT are just some of the possibilities that can be applied to improve the product, analyze it, or even add new functionality altogether.

Qiao et al. [22] provides a supporting sentiment in a setting of machining tool condition prediction. They state that the advantages of a DT include the ability to process and incorporate data from different sources and monitor the real working conditions, aid in design by providing a feedback mechanism, more precise localization, improved continuity and naturally a beneficial visualization. Beyond these sources we argue that other possible benefits of DTs could include a simplification of interaction with the physical object, the ability to work with the system when the real world counterpart is not (easily) available, and the possibility of extending the DT's function more easily compared to other methods.

# Chapter 3

# Background

Here we will describe the current situation of Tech United and their Turtle soccer robot team. First in Section 3.1 we will give a brief overview of the setup of the robt soccer competition. Next in Section 3.2 we describe the main components of the Turtle robots and their modus operandi. After this in Section 3.3 we describe the current main way of analysing and testing the robots digitally using the simulator. Lastly in Section 3.4 we show and describe the existing 3D models of the Turtle robots. The information in this chapter comes from information published by Tech United [23] and from contact with them.

## 3.1 Robot soccer

In robot soccer, teams compete against each other using their own team of robotic players. Like in standard human soccer, the way to win is quite straightforward; the team that gets the most goals by the end of the match wins. The tournaments in which teams similar to Tech United compete with each other are organized by the RoboCup Federation [24]. This organization, established in 1997, had the original goal of creating a robot soccer team capable of competing with the human soccer World Cup champions by 2050. Since then it has evolved into an overarching institution for several robot competitions around the world. This makes the tournaments not only a spectator sport and competitive international endeavor, but also a field in which new robotic technologies may be invented and tried.

Within the field of robot soccer several divisions exist, just like in human soccer. In there RoboCup, there is the Simulation League, which is a league in which only virtual robots participate. Next are the Small Size League and the Middle Size League, which allow teams to create their own robots as long as they hold themselves to certain requirements, most specifically with regards to weight and size. The Standard Platform League forces all teams to play with the same robot hardware. The emphasis in this league is thus fully on the software side. Finally in the Humanoid League team compete with robots that resemble actual human autonomy. Tech Turtle robots currently participate in the Middle Size League.

Similarly to human soccer organization FIFA, RoboCup also sets rules that the robots have to follow. These are inspired by human soccer rules as well. For the Middle Size League, these are some of the most important rules: players are not allowed to exert severe physical trauma to each other, the goalie is the only player with different rules, and in case of violations free kicks and penalties may be applied. These are then naturally adjusted for them to make sense in the context of robots instead of humans. For example, the robots may not be allowed to have some central server do all the processing for them, or have a top down camera provide them visual information. Instead, each robot should have their own processor and vision apparatuses, similar to real human players.

## 3.2 The Turtle robots

The Turtle robots themselves consist of several subsystems that work together to create a coherent team. Each robot has a body containing an arrangement of modules that work together to provide the robot with sight, movement, and decision making. Cameras provide vision the robot. Special wheels provide mobility and agility. A CPU and GPU provide the computing power to the robot to handle this information. The robots talk to each other as well to combine their perceived environment. By sharing their data, strategies and tactics are chosen and executed. A portrait of one of the robots can be seen in Figure 3.1. Some of these systems of note will be detailed here.

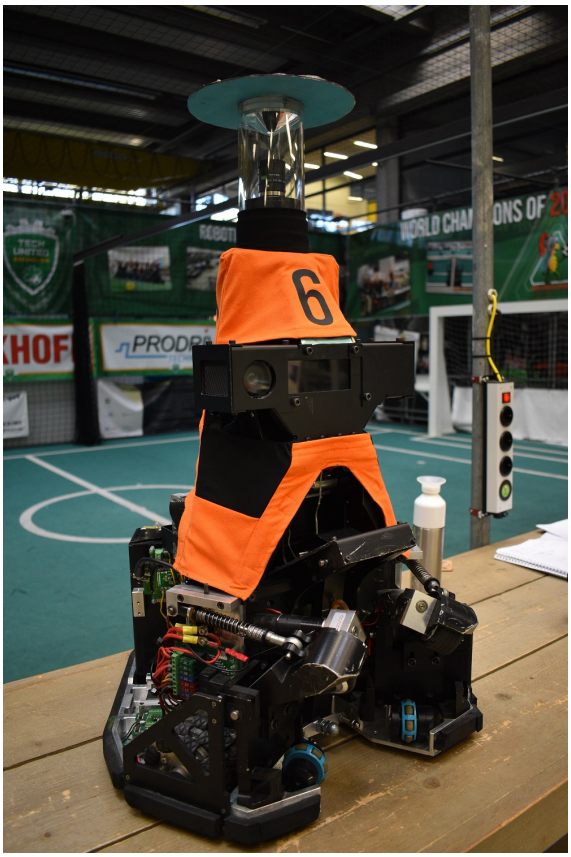



Figure 3.1: Portrait of the current Turtle robot

### 3.2.1 Motion

For the robots to perform at a competitive level, there are several factors of importance with regards to mobility; the robots should be able to accelerate quickly, move fast, and rotate rapidly. A standard wheeled system (like on a car) is the simplest approach, but has shortcomings. The robot would have to slow down significantly during turns. Turns would also require extra space to maneuver due to the turning circle. Furthermore turning on the spot would not be possible, while it can aid greatly in positioning to receive or shoot the ball.

To this end an alternative mobility system was applied. This system comes in the form of the *omni-wheels*. Omni-wheels are wheels which house within them smaller wheels that move perpendicular to the direction of travel. The advantage of this movement method is that the robot is able to move in a certain direction at speed while still being capable of rotating itself 360 degrees without interruption, something that is not possible with an e.g. tracked system. For the robots this offers a significant advantage, as they can advance to a certain position, while at the

same time orienting themselves to be able to receive a ball that is passed to them. A close up of the omni-wheels can be seen in Figure 3.2.

Figure 3.2: Close up of the omni-wheels used on the Turtle robots

### 3.2.2 Vision

In order to play the game, it is vital that each robot is able to clearly see the field, its teammates, the opponents, and the ball. To achieve this, each robot has access to two pieces of hardware; a so called '*OmniVision*'-camera up top, and a *Kinect* on the front. The OmiVision camera functions by aiming a camera orthogonally to a convex mirror housed above it. This provides the camera with a curved perspective (fish eye) view of the area around it, reaching up to about 6 meters away. The robot is still capable of knowing what lies beyond this range however, as will be described in Section 3.2.3. While the OmniVision might provide vision of the robot's surroundings, in the end it is still just a single camera. As a result of this the depth perception of the robot is rather limited using only this visual input, as is evident in nature too due to the disadvantages of monocular vision [25]. It also faces problems in detecting action that happens above 80 centimeters from the ground, due to the mirror placement. For this reason the robots are also equipped with a Kinect camera on the front. The Microsoft Kinect 2 was originally developed for the Xbox console, capable of detecting people and movement for usage in e.g. dancing games. This makes the Kinect a suitable and affordable piece of hardware to compensate for the shortcomings of the OmniVision camera [26].

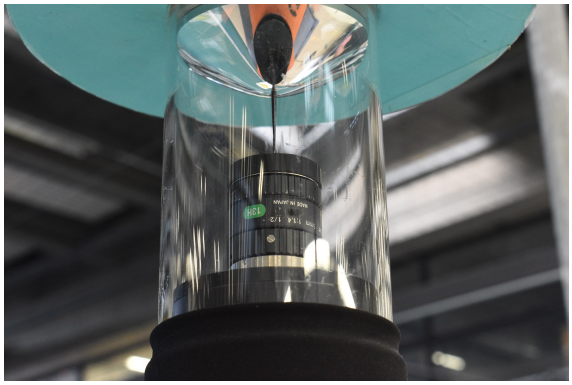


Figure 3.3: Close up of the Omnivision camera and its parabolic mirror

### 3.2.3 World model

Each of the robots combines the visual information gathered by the aforementioned systems to create their view of the field. However, even with these two cameras per robot this is still a rather limited amount of information. Unlike a human soccer player, the robots naturally do not possess a sense of intuition of where entities outside of its field of view may be, or what they are doing. As mentioned earlier, the parabolic mirror sees up to about 6 meters away, and the Kinect camera only sees things in front of the robot. In order to remedy this, each of the robots share their visual information with each other. All this information together each robot then uses to create itself a World Model. The world model is a map of the whole soccer pitch, containing the robot itself, positions of the robots, the number of other robots in the game, the ball and the goals. The robot is equipped with a compass that it can then use to determine which side of the pitch belongs to which team. A computer that is connected to all of the robots keeps track of each of the world models to monitor their reliability.


Figure 3.4: Close up of the front facing Kinect camera

### 3.2.4 Strategy

With the robots being able to create their view of the world and share it with each other, they are now capable of acting upon it. This is done using the *Strategy* system. The strategy system contains the *Defcon* module, which takes all observed data into account and interprets them. These observations can include conditions such as who has possession of the ball, which side of the field the robots are on, and how teammates are positioned relative to opponents. These are all combined to create a plan of attack (or defense). The execution of strategies starts by assigning roles to the robots. They can be attackers, defenders, or goal keepers. Unlike in real life football where each player has a set role, in robot soccer this can be dynamic. The team then executes strategies according to the *Skills, Tactics and Plays* (STP) architecture [27]. Each robot is capable of a set of skills, which consist out of basic actions such as moving, shooting and intercepting. These skills are then used to execute tactics. Tactics include things such as defending a certain opponent, attacking with the ball, or passing to a teammate. Plays are the strategies that robots will collectively employ in order to score a goal. A preset playbook with strategies is known to the robots. The robots all vote on a strategy based on their perception of the game state. Once a strategy is chosen, roles are assigned and everyone tries to execute their set of tactics with the right timing, with hopefully a goal as a result.

### 3.2.5 Communication

Communication exists between several different entities of the turtle robot project. Firstly there is a communication channel between the robots themselves. This is used mainly to create the earlier described world model, where the robots exchange visual information with each other. Furthermore there is also communication with a controlling computer. To do this Tech United has a service called the *Turtle Remote Control*, or TRC in short. The TRC is capable of basic commands such as starting and stopping the robots, ordering a substitution, penalties and free kicks, and assigning a team to each robot. The rules forbid the computer to do any work or direct control of the robots during play, with the exception of earlier mentioned interactions and the referee commands as described in Section 3.2.6.

The main framework that lies beneath all of this communication is a Real-Time Database

(RTDB) system designed by another robot soccer team, team CAMBADA [28]. This system, based on the concept of shared memory, was made publicly available by CAMBADA, and has since been used by many other teams, Tech United included. The advantage of this approach is that many of the subsystems can exchange data with each other, without knowing exactly what is on the other end. Effectively many of the subsystems are black boxes for each other, that just plainly receive and send data back and forth. This makes adding new subsystems quite easy, and adjustments can be made to a subsystem with less likelihood of breaking the system as a whole.

### 3.2.6 Refbox

Just like in real life soccer, the players in robot football may commit fouls or other actions that require the intervention of a referee to solve. To do this, the robots can be instructed by a referee application, the so called Refbox. The Refbox is capable of ordering the robots to robots to start or stop a match, or penalize a certain team by giving their opponents a free kick. The Refbox application is an official program used by Robocup. All robots that participate in their tournaments should be able to interpret the commands given by the Refbox, such that they can all be arbitrated during actual matches. Tech United has implemented support for the Refbox in their TRC application, such that they can issue referee commands during practice runs as well. An excerpt of this can be seen in Figure 3.5.

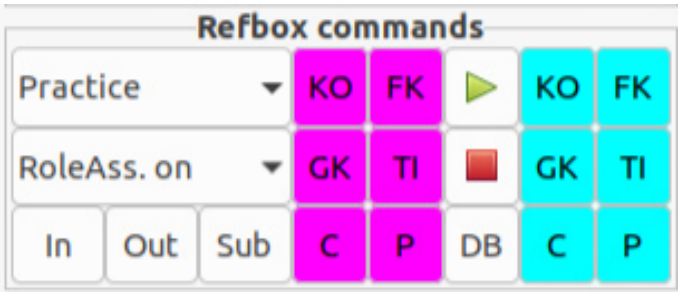



Figure 3.5: The Refbox integrated in the TRC

During actual tournaments, both teams in play are connected to an implementation of the Refbox as shown in Figure 3.6. Both teams can then receive commands from the referee behind a single computer.


Figure 3.6: The Refbox in use in a tournament match

## 3.3 Simulator

To aid in testing the robot software and any changes in it, Tech United has created a simulator. This simulator is capable of simulating matches using the team's current software models, allowing the robots to play against each other as well. Two types of visualizations exists for this simulator; a 2D top down simulator, and a rudimentary 3D simulator. First the 2D simulator shall be described, after which the same will be done for the 3D simulator.

### 3.3.1 2D simulator

The 2D simulator is the main tool that the team uses to virtually test their robots. Up to six robots may be enabled in this simulator. These robots can be set to be part of one of two teams; team cyan or team magenta. Several team composition templates exist in the simulator as well to ease setup. If the command to start the robots is then given, they will start to simulate a match. It is possible to influence the game during play as well, such as by moving the ball or by placing obstacles in the field. The simulator uses a quite simplified representation of the real world, as there is naturally no Z axis involved, nor do the robots have to detect objects around themselves in a 3D space. Nevertheless, the simulator provides the team with a way of testing their changes using a quite fast to set up tool. The 2D simulator can be seen in Figure 3.7.

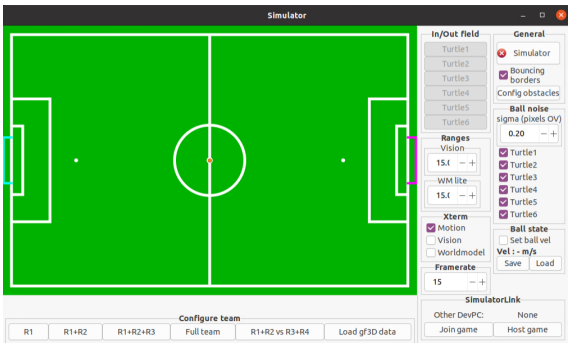



Figure 3.7: Tech United's 2D simulator, using a top down view and simple sprites

### 3.3.2 3D simulator

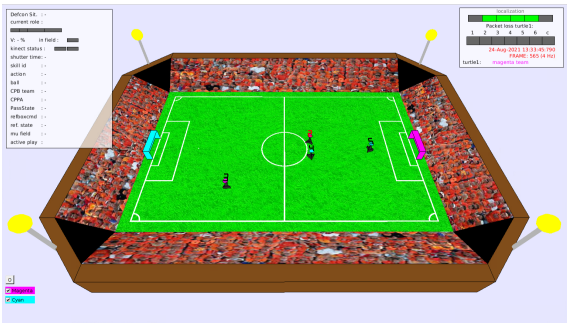


Figure 3.8: Tech United's 3D simulator, using a flying camera and rudimentary shapes

Functionality wise the 3D simulator is very similar to the 2D one. The most obvious difference is the extra dimension. A user is now able to look around using the camera as the simulations runs as well. The robots, obstacles, field, pitch and goals are now represented with simple 3D sprites. It is also possible to toggle some extra visual indicators, such as where a selected robot is aiming its shot at. From a visual perspective, the 3D simulator is a fair bit more informative than its 2D counterpart. In practice however, the 3D simulator is not used nearly as much as the 2D simulator by Tech United when testing their turtle software. This is due to the 3D simulator not adding much in relevant functionality compared to the 2D simulator. When using the simulators Tech United is often only looking to see behavioural differences that their software changes caused. For this the faster to set up 2D simulator usually suffices. Furthermore the performance of the 3D simulator is noticeably worse than the 2D simulator, making it even less suitable for quick usage. The 3D simulator is visible in Figure 3.8.

## 3.4 3D model

Lastly an important component in the development and upkeep of the Turtle soccer robots is their usage of 3D models. These have been specifically created for the construction of the robot, as many of the parts have to be individually created. These models are rather accurate, as some parts have been directly machined from these 3D models. Having a to scale 3D model of the subject in question already available is quite advantageous, as a high fidelity 3D model is necessary for realizing a proper DT. However, there are a few issues with the provided 3D model. Firstly, it is not completely up to date anymore. Several of the components that appear in the 3D model are differing from how they actually are in reality. The protective case is missing, some electronic components are different, and the movement system has been changed as well, which will be further detailed in Section 5.6.1.2 will show an example of this. Secondly the CAD model was originally not made with usage in a real-time lighting environment in mind. Due to this there are a rather large number of faces used in the model. Using it as is would lead to significant performance degradation solely based on the high rendering impact. Lastly there were are a few minor points in importing the model into Blender, which shall be mentioned in Section 5.6.1.

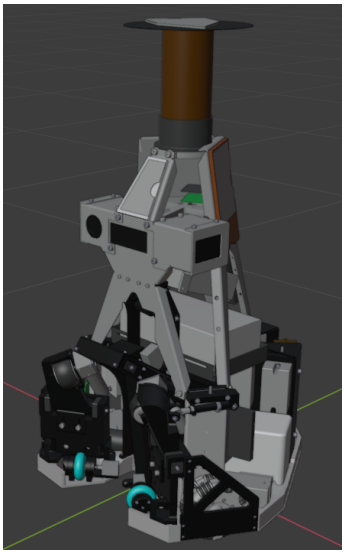


Figure 3.9: Provided CAD model of the Turtle soccer robot in Blender

# Chapter 4

# Elements of a Turtle Digital Twin

In this chapter we set about determining the elements that should make up our DT. In Section 4.1 we consider Tao et al.'s 5D DT again, to establish what is required to complete our DT. In the sections following this we will look at each of the 5D DT components in detail, going over the aspects of each of those components that are important to keep in mind as we create our DT. In particular in Section 4.3, which covers the virtual entity, we will go in depth. Here we discuss the models that are of interest for our use case, and also explore the possible methods that we can apply to go about creating the VE.

## 4.1 Requirements

Before we can define what techniques and methods are needed to create our DT, we first have to establish what is required, what is already available, and what our final result should adhere to. In Section 1.2 we have described the possible definitions for DTs, focusing on Tao et al.'s 5D DT (as depicted in Figure 1.1), and in Section 2.1 we have seen some of the use cases. Tao et al.'s 5D DT definition provides us with a framework that we can use not only as a starting point for determining what work we have to do, but also to verify that when we have completed this work that we have created a working DT.

We will now dissect these requirements following Tao et al.'s 5D DT. Notice that we will put the biggest emphasis on the virtual entity, as this is where the majority of the work will need to be done, something that we will observe in more detail in Chapter 5:

- Physical entity: the part of the system as a whole that exists in the physical space, consisting of all the subsystems, their actuators and sensors. These subsystems collectively perform a set of tasks, of which the sensors then record data.
- Virtual entity: the various models that virtually replicate the physical counterpart of the system. Tao et al. distinguishes models as being geometrical, physics, behavioral, or rule models [1]. The required degree of fidelity of the virtual entity and how accurately it mimics its physical counterpart will depend on the purposes of the DT. We wish to either reuse the existing models from Tech United, re-purpose them for our own goals, or create them from scratch, if necessary.
- Services: interfaces and abstractions that can interact with and monitor the physical and virtual entities. Either already existing services need to be reused or new ones created to use in the DT.
- Data: information that gets generated and exchanged between the two entities and the services. The DT needs access to this data to act upon and generate results. If Tech United already has historic data or means of collecting real-time data for the robots available, then we may be able to use this data in our DT as well. Otherwise, we need to make changes

such that this is possible, which may include changes to the hardware (by adding sensors to the robot) and software (making software changes to record the data from these sensors, or in case of the simulator, expose data endpoints to give access to this data).

- Connections: the connections that connect all components of the 5D DT and move data between them. Most notably, we need connections to communicate data between our physical entity and the virtual entity for our DT to function, and for the services such that we can monitor our DT and handle input.

## 4.2 Physical entity

The physical entity for a DT needs to be mature enough that it can at minimum be taken into basic operation. We observed this in the literature in Section 1.2 as well, where it was argued that at least a working prototype of the physical entity needs to exist. This ensures that it is capable of generating data that is needed for our DT. The DT would not be of much interest, nor would it be possible to reliably determine its correctness, without realistic data from the physical entity. If a physical entity is not available, then a simulator or virtual prototype can substitute for the physical entity, given that it is an accurate enough representation of the real physical entity for the desired end goals of the envisioned DT.

In order for data to be available to us, it may be required to make some modifications to the physical entity. For example, sensors might need to be attached to the physical entity to provide telemetry data of the system in operation. We also have to determine what data we want to gather, what accuracy would be required, and what the update rate of the data would need to be.

## 4.3 Virtual entity

To model a virtual entity that forms a counterpart to our physical entity, it is first essential to explore some of the properties that it could posses. At the core of the virtual entity lies its ability to mimic the physical entity. But to what extent and in what form varies greatly on a case by case basis. A combination of the various different models mentioned earlier together make the virtual entity of a specific DT. The desired purpose of the DT and the context of the system in question should be leading in the design of these models. In our use case for example, geometrical models can aid in visualizing the robot remotely and in designing the robot, physics models can help in testing new components, and behaviour models allow the robots strategies to be analyzed.

In order to realize these models, most notably the geometrical and physics ones, we can create a 3D model. A high fidelity 3D model can be used to help visualize the robot, express behaviour, and test the physics of parts of the robots virtually. Using a 3D model we may not only reuse any already existing 3D models of the Turtle soccer robots, but also provide a good visualization for the DT as well. To this end we shall cover the aspects important to a 3D model in Section 4.3.1, where we describe the possible methods, tools and techniques that can be used.

With a 3D model, we also need to be able to render it, interact with it, and for it to be able to interact with other objects in its surrounding as well. Integrated physics tools would also be of great benefit, as this would save us the effort of programming this from scratch. An answer to these needs that can round out our virtual entity is a *digital environment*. A digital environment is a piece of software that handle these requirements: it is capable of rendering 3D models placed in a scene, provide tools and interfaces to attach scripts to them, and offers support for physics. Using digital environments that are popular also means that a large amount of reference material, extra software packages are available to us, and support can be sought out from others that have done similar projects using that digital environment. In Section 4.3.2 we will be taking a look at a selection of programs available to us.

### 4.3.1 3D model

This subsection will describe the steps required to create a 3D model for the digital twin. In order to create a high fidelity digital twin in a 3D environment, naturally 3D models are required as well.

#### 4.3.1.1 High poly model

In order to produce a high-fidelity model it is desirable to capture as many details as possible of the physical entity in its digital counterpart. However, there is also the computational impact to take into account. The decision making and computations will all take place on a single machine when using the simulator. In our use case multiple robots have to be visualized at the same time as well. Therefore the rendering impact can start to be quite significant if care is not taken during the modeling phase. Given this polygonal budget, a high poly to low poly workflow can offer a solution [29]. The idea behind the high poly to low poly workflow is as follows:

1. A high resolution (high poly) model is created that is as detailed as possible, generally without much regard to the number of polygons.

2. Next a lightweight (low poly) model is created of the same entity, that in essence encapsulates the high poly model.

3. The high poly model can then be baked into a normal map that is then applied to the low poly model. In texture baking rays are cast through the low poly model inwards towards the high poly model [30]. These rays are cast along the normals of the encapsulating low poly model, the normals being lines perpendicular to the surface of the faces. The collision that follows from this ray with the surface of the high poly model gets recorded in a normal map. A normal map is an implementation of a bump map: a 2D image texture that adds surface details to objects. This way the illusion of details from the high poly model can be created on the low poly model while inducing little to no extra computational strain. Figure 4.1 provides an illustration of how this process works.

This workflow will be further explained and exemplified in Section 5.6.1.2, where we apply it our own use case.

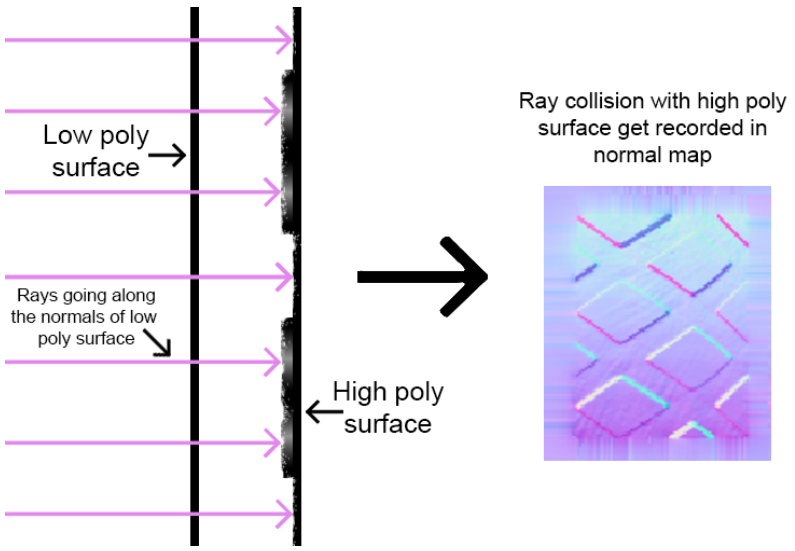



Figure 4.1: Representation of high to low poly baking

In the case of the Turtle robot, CAD files were provided by Tech United that functioned as a baseline high poly model. The CAD model was not fully up to date however, so some adjustments were necessary solely to bring it up to date already.

While the visual details of our high poly model might not be that important to realize our DT, it is still reasonable to take this approach. For one the CAD models tend to be high poly anyway, so a low poly has to be made regardless. Taking this free detail from the CAD model over to the final lower poly mesh is only beneficial to the visualization. Baking certain aspects in instead of modeling them will also save on the polygonal budget, even if one were to create a low poly model. Figure 4.2 shows an example of this process being applied to a part of the Turtle robot.

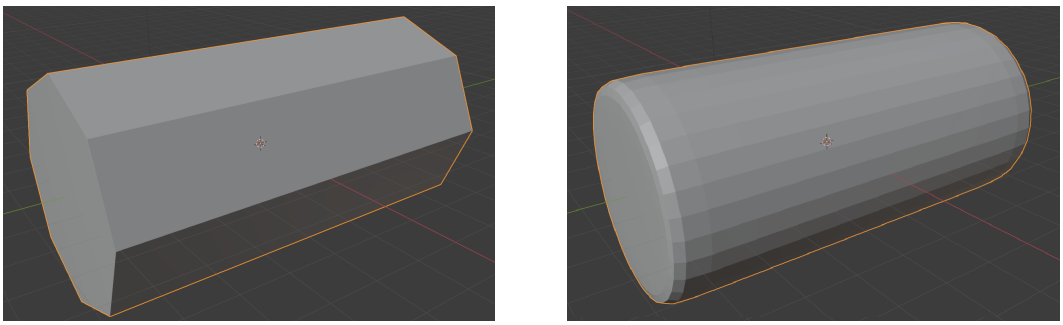
(a) High poly (b) Low poly (c) Low poly with high baked in

Figure 4.2: Showcasing the application of the high to low poly workflow on a component of the Turtle robot

#### 4.3.1.2 Low poly model

To create a low poly model for use in the high poly to low poly workflow, several approaches exist. The first method is available when non-destructive methods were used, such as modifiers, to create more detail on the high poly model. Modifiers are built in operations that are often found in 3D modeling software. These allow for the mirroring, deformation, and increasing the resolution of an object by using them. These modifiers sit on a stack that can be moved around to change the order in which they are used, and can be disabled and enabled at will as well. The most important modifiers in 3D modeling software related to high and low poly modeling are the subdivision surface modifier, which divides every face into several subfaces, and the mirror modifier, which mirrors the faces along one or more axis or mirror points.

Since modifiers can be added and removed as desired, a low poly model can be made from a high poly model using this method: Copy the high poly model and remove or disable the modifiers that increase the resolution, and clean up any non essential edges if necessary. In Blender, the most common modifier to create a higher resolution object is the subdivision surface modifier, which divides all the faces of the model into a number of subfaces.



(a) Object without a subdivision surface modifier (b) Object with a subdivision surface modifier

Figure 4.3: The result of using a subdivision surface modifier on an octogonal prism object

The second method is largely the same but applies if a non-destructive method is not applicable. This happens if no modifiers were used, or they were applied already (either manually or automatically, as is the case when exporting). With applying, we refer to the actual destructive action here of permanently adding the modifier's changes to the model, after which the modifier is removed from the stack. In this case a copy of the high poly mesh can be made, after which edges and vertices are manually dissolved or deleted to simplify the object.

A third method is a quite common one as well; rebuilding the model but with fewer polygons and simple shapes. This allows for a great degree of accuracy and control over the number of polygons. A challenge with this approach is replicating complex or smooth objects.

The last way is the fastest but generally least successful. Similarly to modifiers that increase the resolution of an object, modifiers that decrease it exist as well. Such "decimate" methods reduce the number of polygons on an object while trying to maintain the original shape. This often leads to suboptimal results and bad topology though. In reality a combination of the aforementioned methods is often used. The choice of method depends on what is more apt for the subobject that is being worked on.

#### 4.3.1.3 UV mapping

In order for a model to be textured, it first need to be UV mapped. This process consists of dissecting a 3D model into a 2D shape. The "U" and "V" in the term come from the axes of the 2D image texture ("X", "Y" and "Z" are already used to denote the axis of the 3D object in 3D space). See Figure 4.4 for a visual example. Here several approaches are available as well [31].

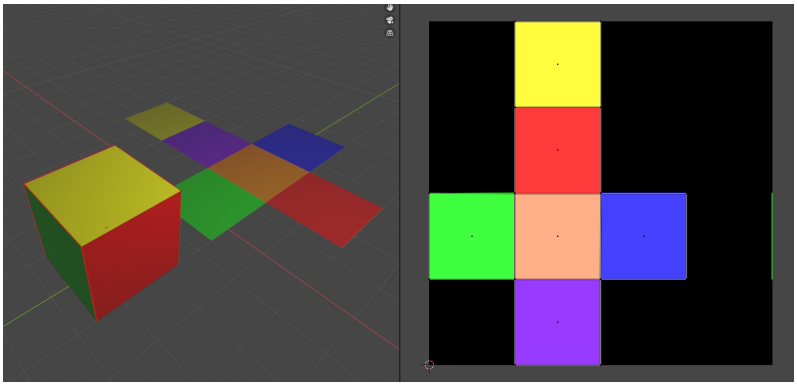


Figure 4.4: Dissecting a 3D object into a 2D shape.

A common method is to use seams. Here edges are marked as seams, by which the modeling program will then split the edge apart. By adding enough seams to the model, the 3D object can be flattened out.

Another popular method is to project from view. Faces can be selected, looked at from certain (orthogonal) angles, and then projected onto the UV map. This is repeated for all faces from different directions, such that all are visible on the map. The downside of this method is that it does not work well for shapes that do not line up with perspective views (front, side, top etc.) properly, such as curved surfaces.

The last common method is by doing smart UV projection. In this case the modeling software is instructed to compute a UV unwrapping, based on some passed parameters. This is by far the fastest method, but does create a lot of seams in many places of the model, and makes the UV map hard to oversee. The first point is problematic because the texturing will leave visible edges around the seams. This is especially true for textures where alignment is clear to see, such as with stripes or lettering. The second point is mostly of concern when texturing in 2D as well. With 3D texturing software this is not as much of an issue.

Due to the complexity of the turtle robot and the many components that make up the 3D model, choices have to be made with which methods are applied. Doing everything using seams is not realistic in the given timeframe, even if it would lead to the best result. The approach for this use case will thus be to first apply smart UV projection, and then manually add seams and project from view the largest, most important components such that good visual are achieved without it being too much of a time sink.

#### 4.3.1.4 Materials and texturing

In order to achieve a consistent visualization of the 3D model in different lighting and rendering settings, the computer graphics approach known as *Physically Based Rendering*, or PBR for short, will be used. This idea was largely introduced by Pharr et al.'s book "Physically Based Rendering" [32]. The concept of PBR rests in creating graphics that follow the real world properties of the material. It may incorporate properties of materials such as how metallic or reflective it is, or take the material's refractive index in account.

In Section 5.6.2.1 this approach will be expanded more with regards to applying it in the digital environment.

#### 4.3.1.5 Rigging

With all but one of the visual aspects of the 3D model covered, there is just one left to address that is functional as well: movement. When the robot is in operation, there are several components that can be in motion, such as the wheels and the shooting mechanism. When these move on the real robot, it would be desirable to move them in our DT as well to fully cover the visualization in our virtual setting. The primary way of moving parts on a 3D model is using animations. These can be achieved in a procedural or non procedural way.

The non procedural, 'classic' way of creating animations is to first add a rig to model. A rig, also known as a skeleton, consists of bones that are added to the 3D model. Parts relevant to a bone are then mapping to this bone, causing any movement of that bone to result in movement of those associated parts of the model as well. Animations are then created by inserting a keyframe, placing the rig in a certain position, placing another keyframe some time later, and then placing the rig in another position. This will then create an animation from the first pose to the second pose [33]. In order to make creating these poses easier, it is also possible to make use of *inverse kinematics*. Inverse kinematics, also abbreviated as IK, is a process where control bones are used to calculated the variable joint parameters needed to move a collection of bones to a position and orientation relative to the control bone. This controller bone forms a chain consisting of a family line of bones. The advantage of using IK is that complex movements can be achieved using simple manipulations of the controller bones, as opposed to moving all bones individually. IK has been a popular topic in computer graphics for several years now, with many different solutions and applications [34].

The procedural approach starts off similar to the non-procedural way, where a rig needs to be created first. Instead of then creating every pose manually however, the desired movement is instead created in a programmatic manner. With a walking animation for example, every step is individually calculated by determining a spot to move one of the feet to once the point of weight of the character has shifted by a minimum amount. The rest of the leg can then follow suit using IK as well. An advantage of this approach is that it is very adaptable to different situations, as the animation will automatically adapt properly to e.g. differing levels of terrain [35].

### 4.3.2 Digital environment

When we are satisfied with our 3D model, we can turn our attention to the digital environment. As described earlier, the digital environment is a piece of software that provides the necessary infrastructure for our DT, most notably real-time computer graphics and physics. Options include (specialized) proprietary software, subscription based software like CryEngine [36], and free licensed software bound to profit limits such as Unreal Engine [37] and Unity [38]. As each program has its own perks, the choice will largely depend on the context of the project at hand, and what the most vital aspects of the DT. Unreal Engine for example offers great visuals and an easy to use visual scripting tool called Blueprints, while Unity is more suited for lower end systems and has seen many use cases of DTs where it has been applied. We shall give our choice of digital environment software and arguments for using it in Chapter 5.

## 4.4 Services

Services provide a degree of abstraction to our DT. They are essential to allow us to interact with our DT, but also to e.g. evaluate, validate and optimize it. They provide an encapsulation to the specialized purposes that our DT would have, hiding any complex actions from the user. A service could be expressed through a simple terminal or a fully fledged user interface. They may be provided with input and demands, to which the service will then return the results, such as optimized parameters, predicted hardware status, or an evaluation report. The usage of services thus make the DT simpler to use, especially for user groups that have less affinity with the technical side of the DT. While services can take a plethora of forms and serve varying functions, there is still a common ground to be found in them. We can distinguish two possible categorizations that the services can fall into:

1. General services: the services that fall into this category are not specific to a certain DT or system. Instead, they may be used to They may even use a popular framework or software suite that is used in many other projects as well. An example of such a service is a user interface that can be used to control a system's optimization suite. These services may not directly help realize a specific functionality of a DT, but they are more likely to be available already, either from other parts of the project at hand or publicly available from other sources.

2. Use case specific services: these services are a lot more specific to the case at hand, and are likely to have been created for a specific purpose. To give an example in the context of the turtle soccer robots, imagine a tool that can be used to design strategies using visual programming, and seeing the feedback of the major hardware components, all remotely without access to the robots physically. It is highly unlikely that one will find e.g. open source software for such a specific combination of hardware and software, and if it exists it is likely only available in the hands of the robot developers. These services are thus hard to borrow from other projects or sources, and if not available already would have to be created from scratch if they are to be used in a DT.

Since we are working with a long existing use case in this project, we would ideally reuse some of the services that Tech United already has currently. The TRC for example, that Tech United uses to control their robots, could be re-purposed to control the DT as well. This would save us time that we would have to spend on creating this service ourselves, while also making the DT easier to operate by Tech United due to familiarity with the reused services.

## 4.5 Data

The next component in our 5D DT is data. Data is of great importance for a DT to work; data input is necessary for the virtual entity to properly mimic the physical entity, and data output is what gives us much of the interesting findings that the DT can provide. This data can be either:

- Historical: data that was recorded and saved earlier. This may for example be match data that can be used to replay a previously played game.
- Real-time: data that is created when the system is in operation. For example, a position data update from each of the robots may be sent out at a set frequency.

Whatever of the two kinds the data may be, it is of first importance to determine which data is actually of interest. The highest priority is data that is responsible for the core functionality of the robots. Most notably this includes positional data of the robots, indicating where on the field each robot is, and information on the ball location. After this data of each of the robot's components is of interest, such as the state of the battery, motor temperatures and shooting system settings. These may all be of interest when creating new functionalities in the DT. It is also of importance to know in which units the data will be provided. Taking the positional data for example, it is of importance whether these are in millimeters, centimeters or meters, and we need to know what this data is relative to; the center of the pitch, the friendly side's goal, or perhaps one of the corners of the field. We will also need to find out in what space the generated data is oriented. All these points are of importance when we introduce the data into the digital environment of our DT, as it may cause incorrect results if we do not take care of any necessary conversions.

## 4.6 Connections

Lastly we need connections to bring all of the 5D DT components together. Most notably we will require connections between our physical entity and the virtual entity. For this many approach are possible. In the case that the DT is mostly stationary, even simple wired connections can work. Our case obviously does not support this, so this would be out of the question. Looking at wireless solutions, we could have the robots be connected to a local Wi-Fi network from which they send their data and communication to a computer or server. The advantage with an approach like this is that this connection can also be extended later to work over greater distances, by sending this data in the local network over the web to a remote computer.

In case that the data is not coming from a robot but rather locally from a simulator, a slightly different approach would be necessary. Instead of connecting to a local Wi-Fi network, the data can just remain on the same device that is actually emulating the data. In this case a simple locally hosted TCP connection may suffice. The simulator could be compiled into an executable that is then loaded into our DT to provide the data directly. When assessing the current state of the turtle soccer robots in Chapter 3, we also went over Tech United's use of the RTDB to handle much of their communications. If possible this would also be an interesting approach to handle our connection, where we can send data either from the real robots or the simulator to our DT using shared memory.

# Chapter 5

# Resulting DT

This chapter describes how the techniques mentioned in Chapter 4 were applied in combination with the existing artifacts to realize the Turtle DT. In Section 5.1 we first go over the approach we have taken to create our DT. We then step through each of the components in the 5D DT and describe what work we have done related to it in order to create our DT.

- We start with the physical entity in Section 5.2 where we quickly go over the state of the actual robots and how we apply the simulator in our development
- Next in Section 5.3 we describe how we reuse the existing services, specifically the TRC, to manage and control our DT
- Then in Section 5.4 we cover how we have labeled the data that we use in our DT
- After this in Section 5.5 we go over the work we have done to reuse Tech United's current communication channels to create the connection necessary for our DT
- We then cover the focus of our work, the virtual entity, in Section 5.6. Here we describe the creation of our 3D model, how we created the data connection, and all the new features that our DT provides. Notice that this order is different than the previous chapter. This was done because work done in the virtual entity section relies on the things we have achieved in the other sections

Then lastly in Section 5.7 we summarize the work that we have done and describes our final baseline DT.

## 5.1 Workflow

In the previous chapter we have taken a look at the 5D DT and used it as a guide to dissect what we require of each of the components of our DT, and what methods we can employ to realize them. To follow the same decomposition here, we shall go through each of the components again, now looking at how we can reuse each of their respective artifacts:

- Physical entity: the actual turtle soccer robots. These are already quite mature and require no work from our side.
- Virtual entity: the virtual entity that mimics the turtle soccer robots. We have some base 3D models of the robot available that we can use as a starting point.
- Services: interfaces and tools that we can use to control our DT. Here we have tools available that Tech United already uses, such as the TRC which they use to control the Turtle robots and simulators. This can be extended to then also control our to be created VE and any VE-related services.

- Data: information that is generated by the entities in the system. We wish to introduce relevant data from the Turtle soccer robots into our DT to create new behaviour and functionality.
- Connections: the connections between our DT and other components of the system. We can reuse Tech United's current communication channels that they already use for other parts of the robots here.

To tackle this work, we will approach each of the DT components one by one. For each of them we will investigate the state of the relevant artifacts, and then based on that apply the methods described in the previous chapter to them. This then culminates in our creation of a baseline DT for the Turtle soccer robots. For a graphical representation of our workflow, refer to Figure 5.1.

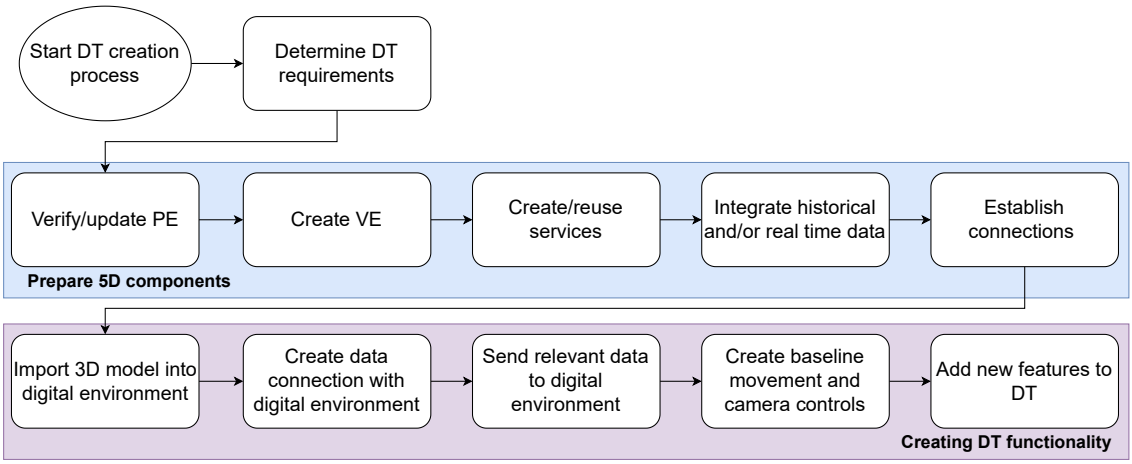



Figure 5.1: Workflow diagram of the approach taken to create the Turtle DT

## 5.2 Physical entity

The physical entity, in this use case the turtle soccer robot, is quite mature as we have observed earlier. They are already equipped with a plethora of sensors to measure many of their components in operation. This is of great benefit to us, as this means that for our initial baseline DT, we do not need to do any hardware modifications to the physical entity in order to gather our data. There is however the challenge of actually getting access to the robots. Since these are not always available to us and take some time to set up as well, it was not feasible for us to be at the Tech United headquarters and have them set up the robots for us such that we could develop our DT.

This is where the simulator that was described in Chapter 3 comes into play. Since the team has a simulator that is capable of emulating the data and behaviour of the actual robots virtually on a computer using the simulator to test out any of their code changed, we can also use it help create our DT. Instead of using data from the actual robots to build and test our DT, we shall thus be using the simulator instead. Since the simulator is capable of running the same code and communication channels as the real robots, doing this exchange should be relatively safe without causing too many discrepancies compared to building the DT based off data from the real robots.

## 5.3 Services

As we discussed in Chapter 4, we would like to have services capable of interacting with our DT. Luckily, Tech United also has an artifact available here that can be of use: the TRC. The TRC, which is used to control robots and manage the team, is also used when controlling them in the simulator. In fact, the simulator itself is built into the TRC, from where it can be booted up.

This comes as a great benefit to us, but to the team as well. We can use the TRC to manage the robots in our DT without any adjustments to the TRC itself. This means that we do not have to create a service for this of our own, and Tech United can use tools they are acquainted with to control the DT later as well.

For functionalities that the TRC does not offer, such as the camera control and data monitoring in our DT that we will see later, we need to create these ourselves. When we create these new functions, we will not do so by changing the TRC, but will instead do it in the digital environment. This way we do not have to potentially break their TRC by making our changes, we keep the DT decoupled from their service that they use for other endpoints as well, and we make the DT easier to use too; the TRC will remain reserved for functionality that Tech United is already acquainted with, but all other new functionality will be constrained to the

## 5.4 Data

For the data the biggest challenge in working with such a long standing project is not in being able to obtain data, but more about figuring out which data is of importance. Tech United has over the lifespan of the turtle robot project added a large number of data sources and parameters that they can use. Now that we wish to use it in our DT project, we have to determine which information is of interest. As mentioned in the previous chapter, we will focus on data that is necessary for the baseline behaviour of the DT, and then expand from there to create new features. This was done by a combination of reading through Tech United documentation, investigating the software repository, trying (parts of) sample scripts provided by the team, and having meeting with team members to guide us. We determined which data elements were relevant to our project, and then created a shared library file that exposed these data elements, as will be described in Section 5.5. An example of this can be seen in Figure 5.2.

```
short current_xyo[3];      Current robot position and rotation
short current_xy_dot[3];   Current robot velocity
short shot_target_xy[2];
short my_Target_xy[3];
roleInfo_t my_CurrentRole_info;Current role of the robot (attacker, defender)
short subtarget_xy[2];
char robotInField;
skillID_shotType_t skill_and_shottype;
char teamColor;Current robot's team
char CPPArobot;
char CPBrobot;
char CPBteam;Which team has the ball
char batteryVoltage;
char emergencyStatus;
MotionStatus_t motionStatus;
passState_t my_PassState_and_comment;
char my_PassStateSync;
char my_PassStateSyncReply;
char my_RefboxStatus;
unsigned char my_RefboxRegionBitmask;
char  my_PassReceiverTurtleID;
short my_PassTarget_xy[2];Target to pass to
unsigned char my_PassTargetQuality;
unsigned char motor_temperature[3];
softVersion_t softVersion;
kinectStatus_t kinectStatus;
char poseLRF_found;
PacketLoss_t packetLoss;
unsigned char actionID;
char debugActionState;
char muFieldNr;
```

Figure 5.2: Determining some of the most important data for the DT

## 5.5 Connections

To feed data from the Turtle robots into our digital environment a connection between the Matlab models that drive the Turtle robots and our digital environment software will need to be

established. To pull this data, we first created a C/C++ file. In this file we create methods for the relevant data, such as getting the Turtle positions and the Turtle status. Each of these methods has a set of parameters which are data elements that are accessible using this method. This method can be called, with the parameters passing the retrieved data via reference. The inclusion of the RTDB library from Tech United is what allows us to pull this data from either the real robots or one of the simulators interchangeably. For example, calling the function on line 64 in Listing A.3 as `get_turtle_position(1, x, y, z, rot)` would put the X, Y, Z and rotation values of the Turtle robot with ID 1 in the respective passed parameters.

The challenge now is getting this all into our digital environment. For starters, the software that we are going to use for our digital environment (which we will discuss in Section 5.6.2) uses C#, compared to the C/C++ that the Turtle robots use. To solve this incompatibility, the applied solution is to compile the plugin to a library file. Specifically, the code was compiled into an Executable and Linkable Format called *Shared Object* (.so). Such .so files can be seen as the Unix analogue to a .dll file in Windows. Our digital environment is then capable of importing such files and accessing the functions that are available in them. A Makefile file was then created to aid in compiling this plugin to a shared object file, as well as placing it within our digital environment project files. The Makefile that compiles the earlier mentioned script it into a shared library can be seen in Listing A.4.

## 5.6 Virtual entity

In this section we describe how the above described components come together with the work we applied to adjust the artifacts we wish to use in our DT, and create the virtual entity. First in Section 5.6.1 we describe the how the 3D model was created, textured, and rigged. In Section 5.6.2 we cover how the digital environment was setup and the baseline functionality was realized. The digital environment is of importance, as it holds the 3D model and any associated physics and scripting support necessary to make our DT work. Then in Section 5.6.3 we go over all the newly added features and enhancements that have been created in our DT.

### 5.6.1 3D model

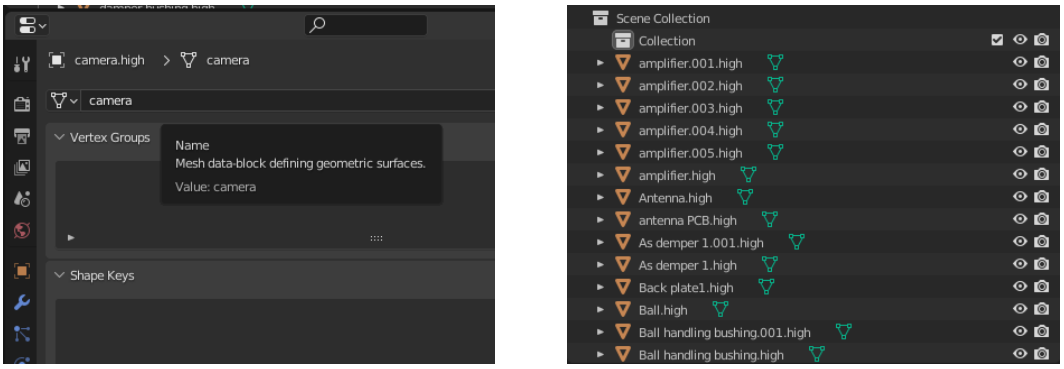
The first step in creating the virtual entity is to create a 3D model of the Turtle robot such that it can be used in the DT. To do this the provided CAD model of the robot was used. The following sections describe the steps that were taken to reuse this artifact. In Section 5.6.1.1 we describe the preparation work done on the provided 3D model artifact. Then in Section 5.6.1.2 we go over the actual modeling of our new 3D model. We proceed with Section 5.6.1.3, where we explain the UV mapping process of the 3D model. Section 5.6.1.4 covers the texturing of the model. We then describe the rigging of the model for animation purposes in Section 5.6.1.5. Lastly in Section 5.6.1.6 we recap the resulting 3D model.

#### 5.6.1.1 Preparing for modeling

To do the 3D modeling several options are available to us, such as Autodesk 3DS Max, SolidWorks and Blender. The latter was chosen for this project. Blender is a free, open source 3D modelling program, capable of full 3D modeling workflows [39]. The choice to use Blender was made because it is free (most other 3D modeling programs require significant fees), well suited to the workflow mentioned earlier, and author experience is the greatest with this program. This does lead to another issue however; the CAD format used by Tech United's models is not supported by Blender. To remedy this a program like CAD Assistant [40] can act as a middle man to convert it to a format that Blender does accept.

First some preparatory work has to be done. We encountered our first issue after importing the converted CAD model into Blender: all of the objects that the model consists of had rather ambiguous names. This makes it rather challenging to identify which component is which from the

list of objects, which would of importance later on when applying the high to low poly workflow as well. However, the original names were luckily still available in the mesh data. In order to extract these we had to manually look into the metadata of each object and copy paste the information to the object name in Blender. Since there are over a thousand objects in the provided model, we have written a small Python script (which is the scripting language that Blender uses) to do this. The script in Listing A.1 goes over all the currently selected objects and renames them to the name variable found in the mesh data. After this, a small modification to naming had to be done. Namely, postfixing the names of all the objects by ".high". This serves two functions; for one it is more easy to recognise these objects as the dense, high polygon objects from the CAD model, and some texturing programs also specifically look for this tag when baking. To not have to do all this by hand a very similar script was created for this, which can be seen in Appendix A.2. Similarly here, plainly selecting all objects in the scene and running the script will append all of them with ".high". Figure 5.3a show the metadata of the objects in Blender, and Figure 5.3b shows the names of the objects after running the scripts.




(a) Name metadata stored in the objects

(b) Object names after running the scripts

Figure 5.3: Object names in Blender before and after running the scripts

#### 5.6.1.2 3D Modeling

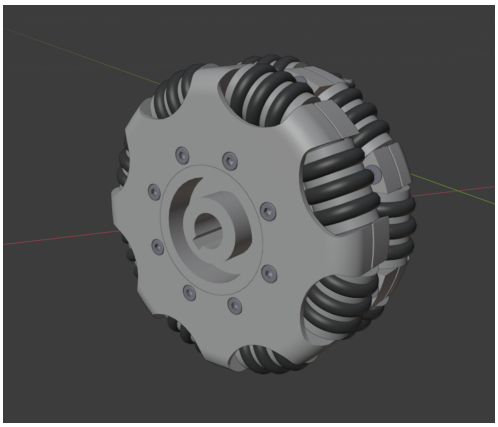
With the provided 3D model now in our 3D modeling software of choice, we can go about doing the work necessary to get a 3D model capable of being used in our DT. We opted to apply a high to low poly workflow here for a number of reasons:

- The provided CAD model is, as described earlier, out of date, so some components had to be remodeled and updated.

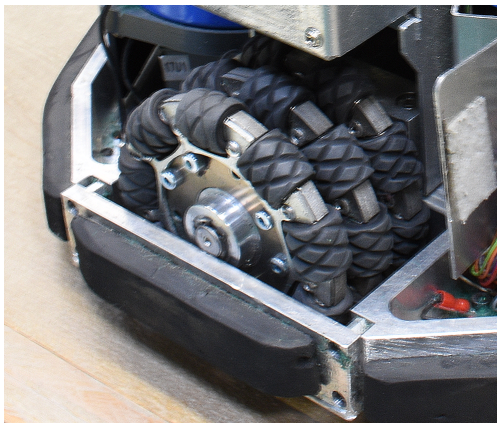
- The CAD model is very dense: about 2.5 million triangulated faces. Given that in a standard match one would have ten robots, that would lead to over 25 million faces in total in frame. That is a significant amount of computational power to spend, especially considering that there are more things to render, such as the environment the robot is in.

- Many of the details on the robot, such as the threading on the wheels or nut and bolts holding the components together, are of visual interest yet increase the amount of polygons by a lot

To give an example of some of the discrepancies between the real Turtle robots and the provided 3D models, take a look at the following images:



(a) Wheels on the CAD model



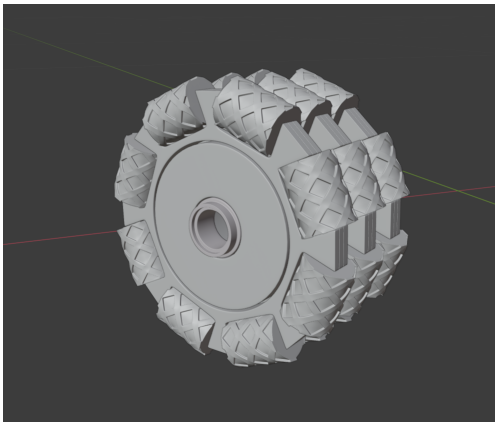
(b) Wheels on the physical robot

Figure 5.4: Comparing the wheels in the 3D CAD model to the actual wheels on the robot

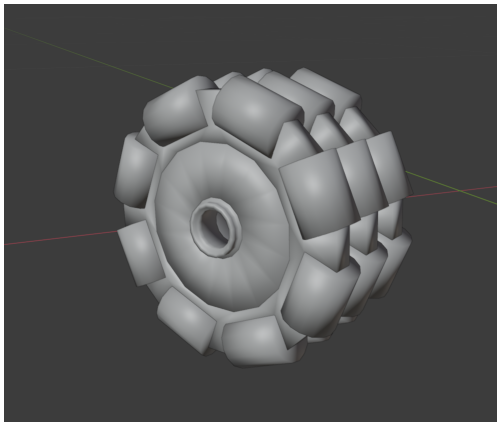
In Figure 5.4a we can observe the wheels as they are in the CAD model, and as they are in reality. The wheels in the provided CAD model have two OmniWheels on each side, where each of the sub wheels seemingly contain rollers embedded within as well. However in reality there are three OmniWheels on each side of the robot, and the subwheels are now of a solid, treaded type.

Furthermore, the wheels on the CAD model also have a great deals of internal parts, down to every bolt and ring, included in it. While nice to have visually (for those parts that are actually visible), they do not add much to the model, while they do take up significant graphical computation time.

These points make it prudent that we create new models for the wheels, one that is high poly and contains details such as the threading of the wheels, and one that is a lightweight, low poly version.



(a) New high poly wheels



(b) New low poly wheels

Figure 5.5: Newly created 3D wheels of the Turtle's OmniWheels

In Chapter 4 we went over several approaches that are possible for creating the low poly model in the high to low poly workflow. For our use case the method where we remove modifiers that add extra resolution, like a subdivide surface modifier, is not possible: since the model was originally made in CAD, no such modifiers are available when it is imported in Blender. We can however make use of modifiers for any new parts that we have had to model, which makes it easier to later make modifications and keep the model up to date due to the non-destructive nature. For the parts from the CAD model a combination of the second and third methods were used. Fitting

with the high to low poly workflow, a high poly, detailed model (Figure 5.5a), and a low poly, lightweight, model (Figure 5.5b) was created.

#### 5.6.1.3 UV mapping

Next the techniques described in Section 4.3.1.3 were applied to the new low poly model. Initially, it was attempted to fully unwrap the model using the smart UV projection method. While this was really fast, it led to some results, where many of the textures were either looking stretched or in the wrong place altogether. This is mostly due to the rather large number of objects that the robot consists out of. The smart UV projection method cuts every object up into several pieces, and then scales the pieces in the UV mapping according to their size relative to each other. Lastly they are packed together using a packing algorithm. The downside of this is that many UV islands (the faces of the object cut out into 2D shapes) may be packed together very closely. This causes the seams of the islands to become rather noticeable, and sometimes even bleed colors from other parts into it, leading to the bad visuals.

In order to remedy this, the manual method of adding seams was used for some of the biggest components. This improved the situation significantly already, as the problems are not that visible on smaller components. Furthermore some of the components that can be found in multiple places on the robot were set up such that they share UV islands, further reducing the total number needed. Lastly a margin was added between all the islands to prevent bleeding. The final UV map can be seen in Figure 5.6.

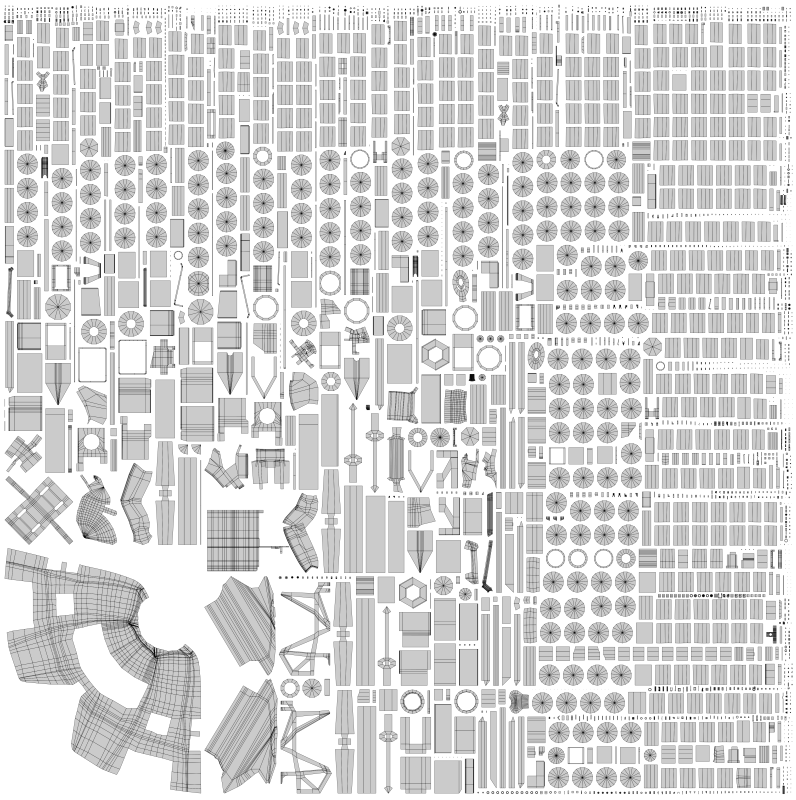
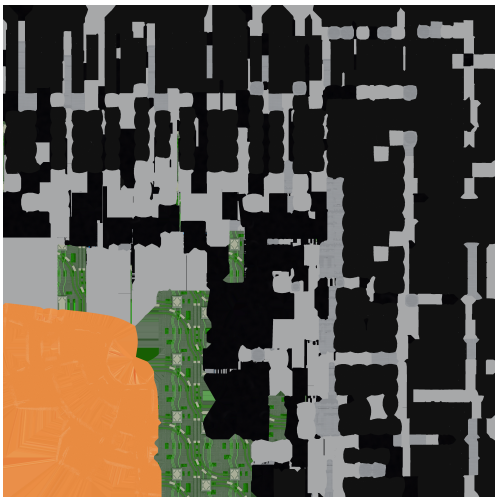


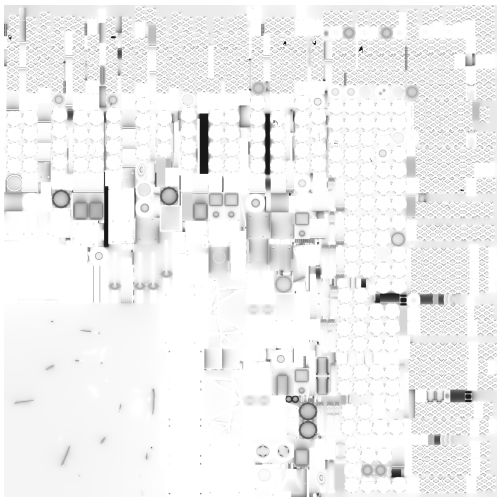
Figure 5.6: UV map of the whole robot

#### 5.6.1.4 Materials and texturing

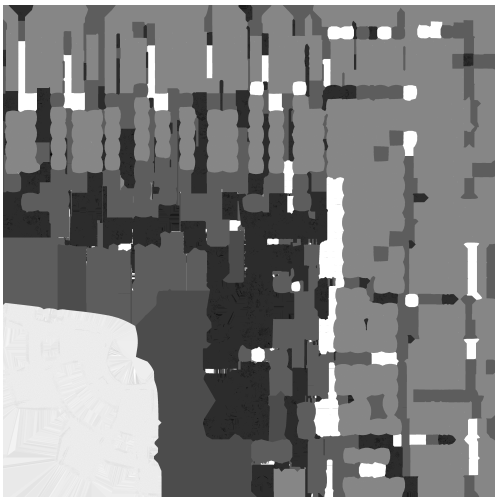
After the robot's shape was modeled in detail, the next step was to texture it. This is done to aid in the fidelity and recognizability of the components of the robot. Photos of the robot were made that were used as reference for the texturing. In order to save on drawcalls, a single material was used for the whole robot. This material would use 4K textures to still provide enough visual clarity. A color ID map was created in Blender to distinguish which parts of the UV map belongs to each part of the robot. The texturing was done in Substance Painter 2 [41], a 3D texturing program. This program then produced the following four image files, which were subsequently added to our digital project project files:



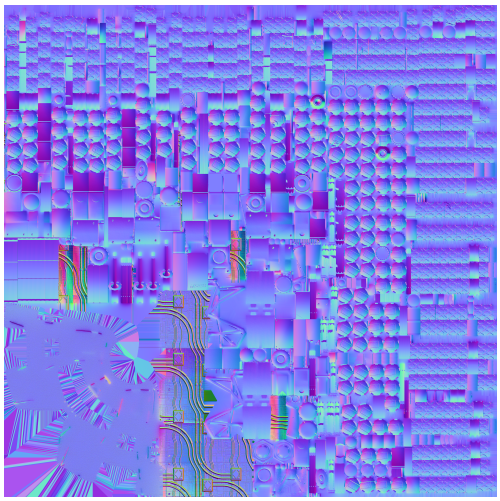
(a) Albedo Transparency texture



(b) Ambient Occlusion texture



(c) Metallic Smoothness texture



(d) Normal texture

Figure 5.7: Created textures necessary to make a PBR material in our digital environment for the 3D model

The reason for these four images is that this is how the shaders of the software that we will use (and the shaders of many other similar software as well) are set up to render their materials. In Figure 5.8 we can see the material of the robot in that software, with the above textures applied to it.

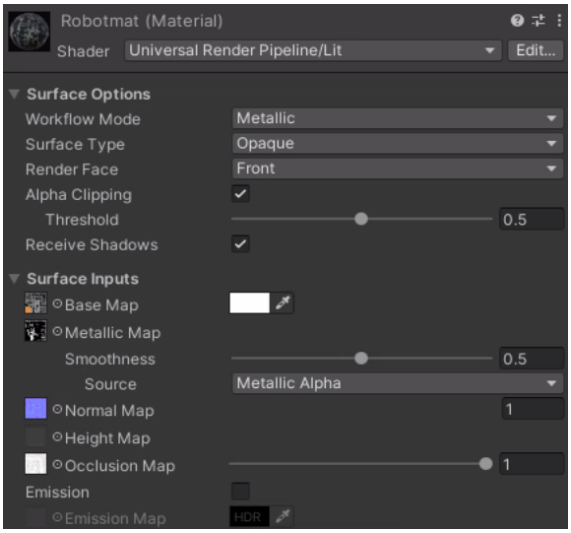



Figure 5.8: Material of the robot model in the digital environment

The basemap contains the Albedo Transparency texture, which gives colors to the robot and includes an alpha layer for e.g. glass.

Then the metallic map has the Metallic Smoothness texture, which contains information for the renderer to know which parts are reflective and which are not.

Next is the normal map, which contains our Normal texture. This is important for all the detail from our high-poly model, the provided CAD file. This allows us to create the illusion of detail without actually spending rendering capacity on it.

Lastly there is the occlusion map, which is map to the Ambient Occlusion texture. Ambient Occlusion, also known as AO, is the ambient lighting that can be found around edges and crevices of the model. The shadow this creates has been recorded into a texture and applied to the model, increasing the visual quality without incurring a rendering impact.

#### 5.6.1.5 Rigging

Given that the original model was not rigged while it would still be desirable to have animations for e.g. the robot shooting with the leg, these had to be created from scratch as well. In order to make animating easier, this rig was equipped with the inverse kinematics described in Chapter 4. IK is a commonly used process in robotics, making it even more apt to apply here. Currently we have a rig that is connected to the shooting leg of robot and its associated plunger.

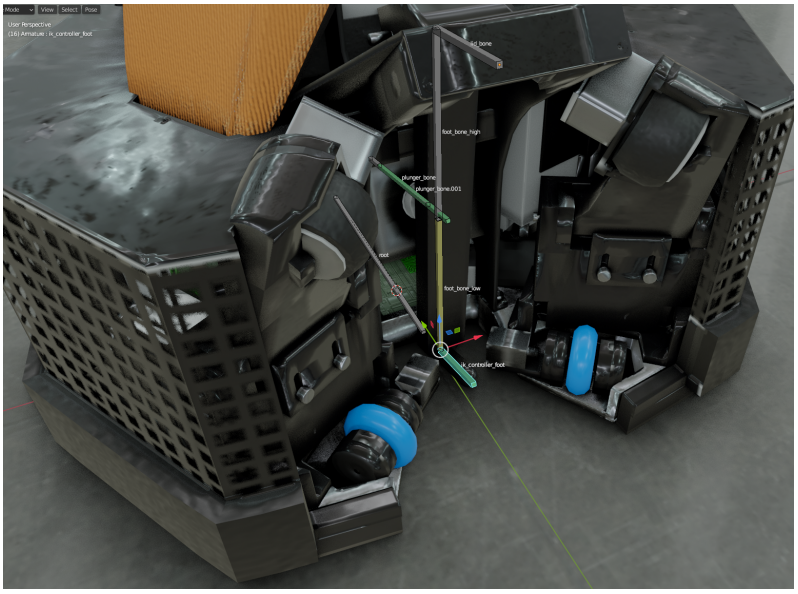
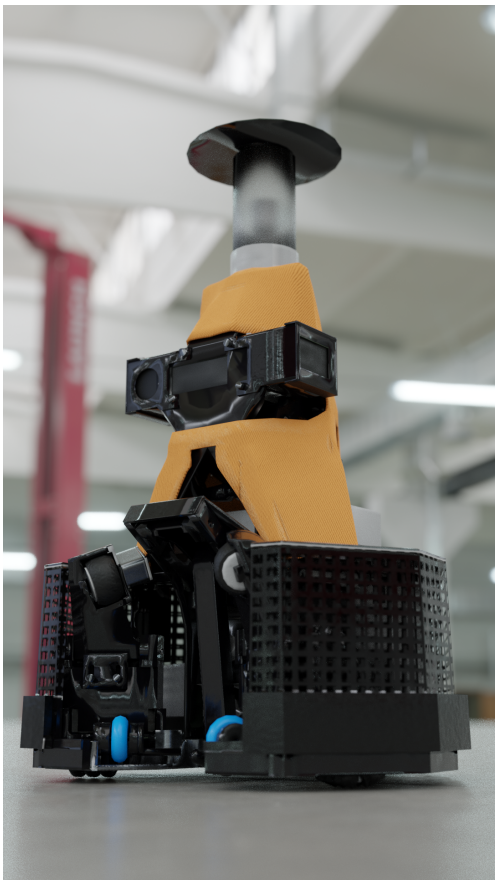


Figure 5.9: The rig on the 3D model

This whole setup can be operated using a single control bone. Moving this bone makes it possible to move the leg forward and backward to imitate the robot taking a shot, as well as up and down to adjust whether the ball will be shot through the air or across the field. The plunger will move automatically as required of the leg it is attached to as well. Constraints have been put into place to make sure that the virtual leg can only be moved according to the freedom of movement it has in reality. This makes it quite simple to create animations for the shooting mechanism. In the future, this can be expanded to other moving parts as well in a similar fashion, for example for the wheels. The correctness of the rig is currently based upon the workings of the mechanics of the robots as demonstrated in person to us by Tech United. While the exact movements of the real robot might not be fully correct (since they are just done from memory and image references), the movement is quite easy to adapt by simply changing a set of constraining parameters. Later when the DT might also be used to test the physics behaviour of certain components of the robot, the rig could be adapted to more precisely mimic the range of movement of the real robot.

#### 5.6.1.6 Resulting 3D model

By applying all the aforementioned techniques we have created the following 3D model. It is up to date, textured, has a rig, and, most importantly, is a lot less computationally intensive as well. It contains about 50 thousand faces when triangulated, a significant improvement upon what we had with the provided 3D model. Since we have applied a non destructive workflow with parts automated with scripts as well, adjusting the model later is easier to do as well.



(a) With the protective case

(b) Without the protective case

Figure 5.10: Renders of the final Turtle robot model

### 5.6.2 Setting up digital environment

In the Chapter 4 we have discussed what the purpose of a digital environment is, and some of the software choices available to us here. We now need to choose a digital environment to use, and do some preparation work for it to be used in our DT. For our project we have opted to use Unity.

Unity, developed by Unity Technologies, was initially released in 2005 as a Mac OS X-exclusive engine that focused on mobile and web products. It has since been extended to a multitude of platforms and is a popular engine to use for creating games [42]. This makes Unity better suited for our purposes, as other options such as Unreal engine focus more on high end graphics, while our target user group's hardware is not likely to support that.

Besides games, other applications in Unity see use too, such as in engineering and architecture [43]. DTs have been created in it before [44], and even whole companies have designed their DT software packages in it, such as Prespective's DT toolset [45]. It also offers support for several subsystems that could be of interest when creating applications for the DT, such as a physics system, VR [46] and machine learning support [47]. Lastly due to author experience with Unity being the greatest amongst the choices, it was decided to use Unity for this project.

To use our 3D model and associated textures into Unity, is very straightforward. This is done by simply dragging the exported 3D model into our scene. The real challenge comes from making it respond to information from the physical entity. Note that in our case the PE is actually the simulator, as we described in Section 5.2. To do this we will first cover how the shared library file that was compiled from our plugin script (that exposes the data from the simulator) will be integrated into Unity in Section 5.6.2.2. Then in Section 5.6.2.3 we use this data to implement the baseline behaviour for the robots to play a match.

#### 5.6.2.1 Render pipeline

As of this writing, Unity is currently in the long term process of consolidating and modernizing their workflow. There are currently two main workflows available: the *High Definition Render Pipeline* (HDRP) [48] and the *Universal Render Pipeline* (URP) [49]. The former is aimed at higher end systems, while the latter at lower end systems.

Since both render pipelines have quite differing feature sets [50] that each have their own roadmap [51] [52] for future changes, it is important to determine early which workflow to use. We wish to have a high fidelity setup, but photorealistic visuals are not required. Rather we would have it that all members of Tech United can run the DT. Therefore, the choice was made to use URP for this project.

#### 5.6.2.2 Loading the library in Unity

In Section 5.5 we created a C/C++ file that exposes the data from the PE using Tech United's RTDB. This file was then compiled into a library file, that was then added added to our Unity project. What remains for us to do now is to create a C# script in Unity that calls it such that we can actually use our data. Naturally many other entities within our digital environment would need the data pulled by this script. To this end it was decided to set this script up following the Observer design pattern [53].

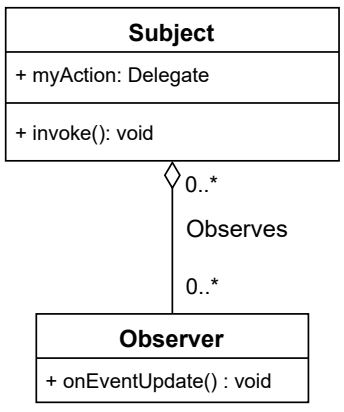



Figure 5.11: UML Class diagram for the Observer pattern

In the Observer design pattern one class is defined as the subject, and one or more other

classes as observers. Instead of having a tightly coupled setup where everything that requires information from the robots needs access to an object providing this information, all files that require data (the observers) simply listen to one file (the subject) that provides them with this data whenever an update is available. To achieve this, in our C# subject file we have some delegates defined which get invoked every frame after an update has been pulled from the PE. Other files dedicated to specific functionalities, like the movement of the robots or interfaces that display the status of the robots, can then subscribe to these delegates. When these delegates then get invoked, they can receive information and act on that data accordingly.

The advantage of this approach is that the system as a whole is more likely to remain functioning if one of the components is removed or fails, as this just leads to no action being undertaken with regards to those respective subject(s) and observer(s). Furthermore adding new functionalities later is easier to do as well, is it would require new scripts to just simply subscribe to the subject as well to receive any information that they may need. We will now go over the structure of this main communication file to showcase how the Observer pattern has been set up to handle the data exchange for our project. The full code can be seen in Listing A.5, although some snippets will be included below for ease of reference.

First we import the functions from the shared library file. These can be imported using the "DllImport" tag in C#. This then allows them to be matched with C# functions in this file. These functions can then by called and variables passed by reference to get the relevant data in them. An example of this can be seen in Listing 5.1.

```
1  [DllImport("TurtlePlugin", CallingConvention = CallingConvention.Cdecl,
       EntryPoint = "get_turtle_position")]
2  private static extern void get_turtle_position(int agent, ref double robot_x,
       ref double robot_y, ref double robot_phi);
```

Listing 5.1: Excerpt of the importing functions from the shared library file (TurtleCommManager.cs, Listing 5.1)

After this several *Action* events are defined, as can be seen in Listing 5.2. Actions are native implementations of delegates in C# that can take input variables but do not return anything. For example, a robot controller class listens to the `robotPosUpdateEvent`. When this event gets invoked all Observer classes listening to this event get notified and passed the necessary variables. This happens every frame after pulling fresh data from the RTDB. The diagram of the Observer pattern applied to this project in which these Actions are used can be observed in Figure 5.12.

```
1  // Actions for observer pattern
2  public static event Action<bool, double, double, double> ballUpdateEvent;
3  public static event Action<int, bool, double, double, double>
       robotPosUpdateEvent;
4  public static event Action<int, IntPtr, IntPtr, IntPtr, double, double, double>
        robotDataUpdateEvent;
```

Listing 5.2: Delegates defined to handle communication (TurtleCommManager.cs, Listing 5.1)

Next is one of Unity's native functions, the `Start` function. This function is called before the very first frame in the scene is rendered. Commonly, this is where initialization takes place. In the case of our `TurtleCommManager`, we call our first external function from the shared library file, `check_db_status`. This is a quite simple function that returns 0 if the RTDB has been successfully initialized (which happens when loading the shared library file), and 1 otherwise. This is a simple check that we can do before starting the DT to ensure that our data connection is working properly. If this is not the case, an error will be printed and the Unity Editor will stop play mode automatically.

Finally we have another major Unity function: `Update`. This function gets called for every frame that is rendered. Here we are going over every of our robots and request their information

from the RTDB using our external functions, `get_turtle_position` and `get_turtle_info`. After the updated data has been received, the respective Actions get invoked such that any listeners can receive the new information. An external function gets called each frame as well for receiving the current ball position, also invoking an Action to inform listeners about that update.

The majority of scripts that shall be used in Unity implement Mono [54]. Instead of creating objects of the classes in these scripts like is normally done in OOP by calling constructors, these scripts are instantiated by attaching them to an object in the scene. In Unity, these objects are referred to as *GameObjects* (to avoid some confusion with other definitions of the word object). Usually these are attached to GameObjects that the script itself is related to (as we will see in the next subsection). Since our `TurtleCommManager` is more of a general script for many things in the scene, we shall instead attach it to an *Empty* GameObject. This is a simple, invisible zero-sized GameObject in our scene.

All of the files that observe the main communication file and receive data from these delegates can be seen in the Class diagram in Figure 5.12. We will describe the functionality that each of these files serve in the next sections.

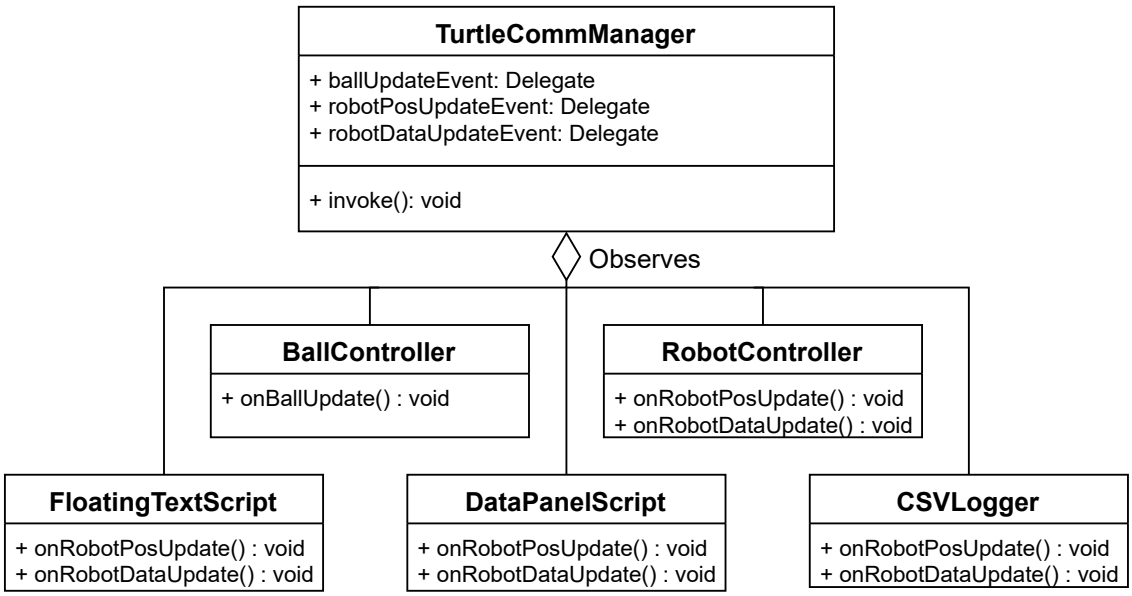



Figure 5.12: UML Class diagram for main observer-subject setup as applied in our project

To get more insights into how the calling order of these classes work, refer to the Sequence diagram of our project in Figure 5.13.

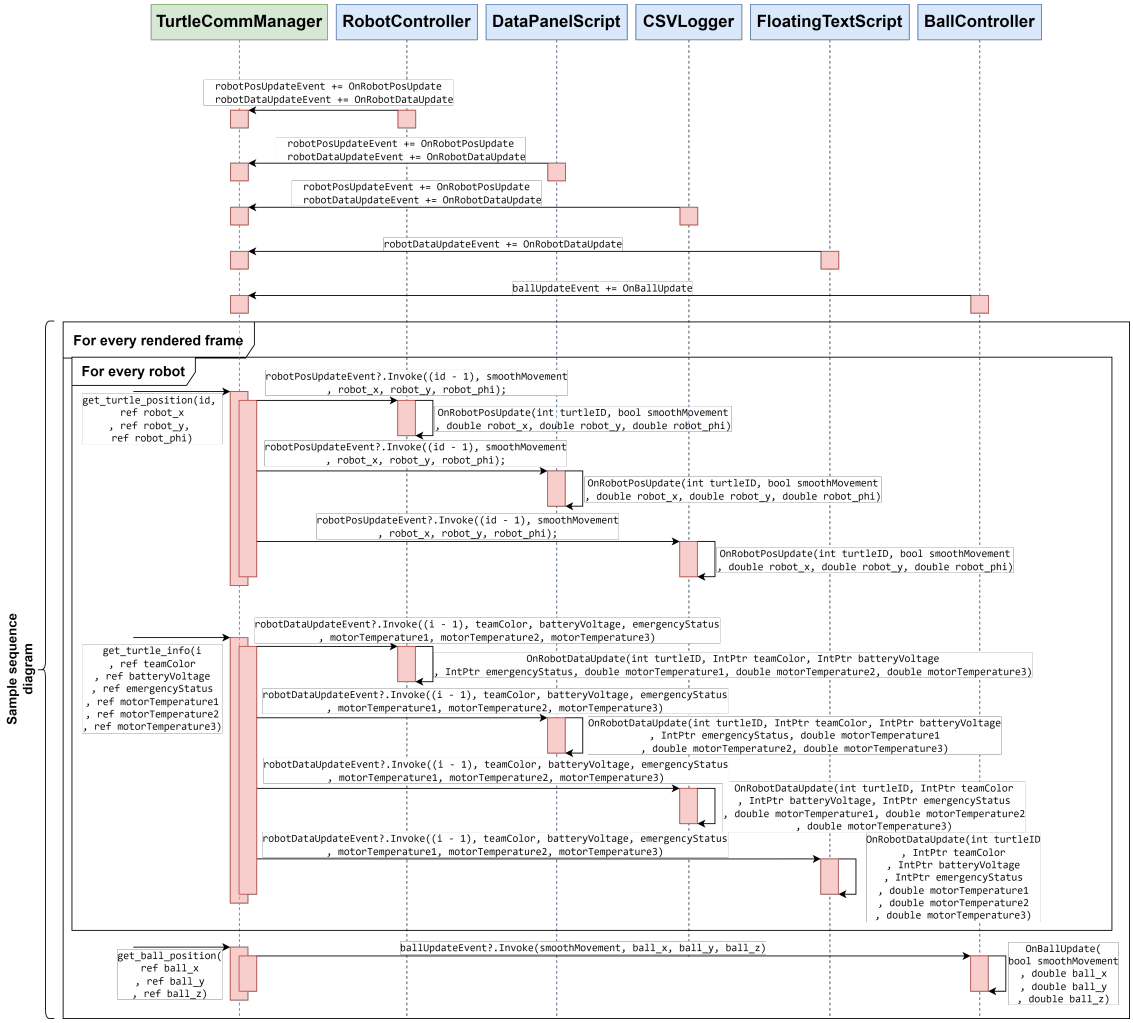



Figure 5.13: UML Sequence diagram of our project

#### 5.6.2.3 Creating baseline movement

With the connection between the Matlab Turtle robot models and Unity now established, the next step would be to establish the foundation of our DT's functionality: movement of the robots and the ball within the digital environment. This is achieved using dedicated scripts that are observers to the `TurtleCommManager` subject script mentioned above.

As can be seen in Listing A.6, in the two native Unity functions `OnEnable` and `OnDisable` we subscribe and unsubscribe to the events. These functions get called when the GameObject that the script is attached to is enabled (which happens upon starting the DT) or disabled, respectively. This ensures there is one function subscribed to the Action whenever it is necessary.

### 5.6.3 Newly introduced features

Now that the baseline data flow and movement of the robots has been created, next some new features will be created. We aim to demonstrate some of the possibilities that our setup has, and the ease of adding new functionalities. We can already create some functions that are currently

not available in the Turtle software ecosystem. This way our result will have benefits for Tech United, no matter how far we get in our total development.

#### 5.6.3.1 Camera controls

To allow for a clear view of the DT in action from a multitude of possible angles, proper camera controls will be necessary.

To this end it was decided to create an orbit camera setup. With this approach the camera will focus on a target, and the user can control the rotation and zoom of the camera using the mouse. This provides users with an easy to use method of observing the entities within the DT.

A dropdown that is automatically filled with objects in the scene with the "CamTarget" tag was added to the user interface as well. This allows for easy switching of the camera's orbit target. Current targets include every individual robot, the field as a whole, and the ball. Since this list is filled dynamically based on all objects in the scene with the CamTarget tag, it is thus easy to add more possible targets by assigning this tag to them.

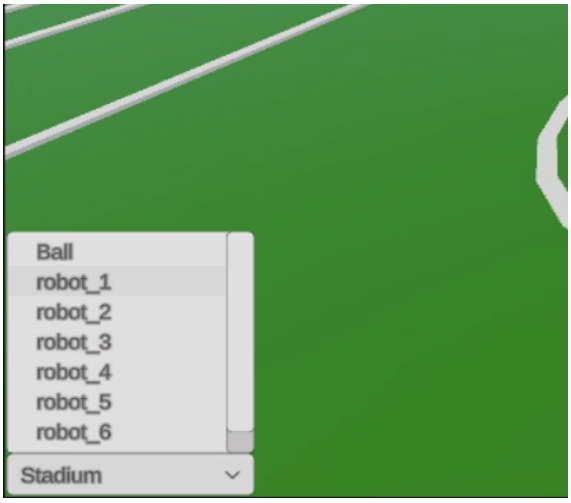



Figure 5.14: Dropdown available in the GUI of the DT used to change camera target

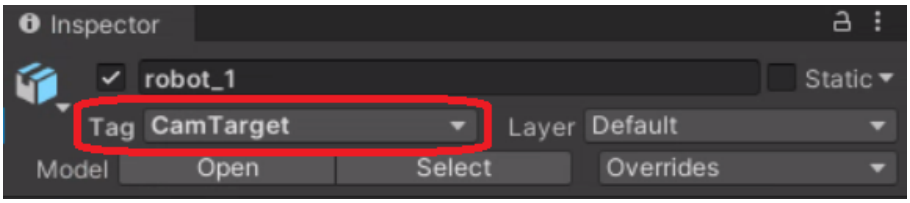



Figure 5.15: One of the Turtle robot objects shown with the CamTarget tag

#### 5.6.3.2 Motion Smoothing

Currently the digital environment is set up to pull data at least on every rendered frame in Unity. Depending on the user's device however, this experience may not be smooth enough. Even though steps have been undertaken to reduce the performance load (such as the low poly model), it still demands a noticeable effort from the host computer.

To provide a remedy for this, a smoothing functionality has been added. If enabled, the incoming positional data regarding the robots and the ball will be linearly interpolated over several frames, instead of plainly setting the transform values on every update. As a result of this the objects will more smoothly move through the scene. Since a downside of this approach can be that the shown positions are not always accurate, it may not always be desirable to have this functionality on. The smoothing can thus be simply toggled on and off at any time using Unity's inspector window. Code for the smoothing can be found in Listing A.6.

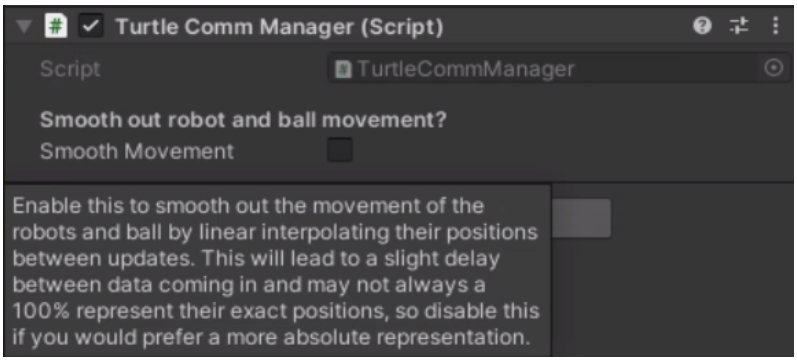



Figure 5.16: Inspector window of the `TurtleCommManager` script's smoothing toggle

#### 5.6.3.3 Data panel

To provide more information about specific robots in play, a data panel has been created. This panel is part of the UI, and shows some of the pulled data from the RTDB about a specific robot. The currently selected robot is determined in by the current camera target set via the earlier mentioned dropdown. This data currently includes the robot's ID, team it is on, position and rotation and battery status.

As this data panel also works according to the Observer pattern, adding new information is as simple as adding it to the shared library file and creating a text element in the panel for it. The elements in this panel are also anchored to corners and containers in the GUI, ensuring that the interface is readable in a wide range of screen resolutions.


Figure 5.17: Data panel image in the GUI of the DT

#### 5.6.3.4 Heatmap

With data now coming into Unity, it would now be expedient if it would be possible to introduce a functionality not available in the current Turtle robot software package. To this end it has been decided to introduce a heatmap functionality, based on the robot's position. The data for the heatmap is created by emitting a ray downwards from the currently selected robot. The collision of this ray with the field below the robot is recorded. The recorded collision is then transformed into a texture coordinate, which gets fed into a float array for a heatmap shader. This shader will draw these datapoints on top of a texture of the field. The more concentrated data points are, the more intense this region will be displayed on the heatmap. The heatmap is customizable, with many options being exposed in the inspector window when the heatmap UI element is selected, as can be seen in Figure 5.19.

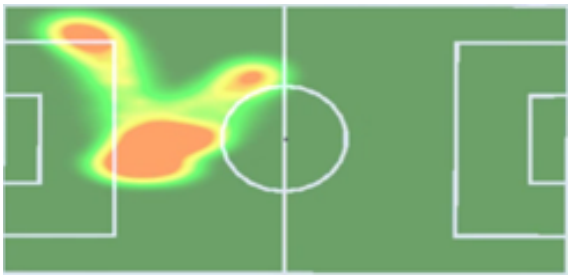


Figure 5.18: Example of one of the live generated heatmaps

This heatmap can then be used to get more insights into where a robot spends most of its time in the field. For example, one team's robots may spend a significant amount of time playing aggressively while on the offense, but may then be too slow at falling back during defense. Or perhaps when using one of the strategies, one robot taking on a certain role may be out of position for too long. This may especially prove to be of interest when some of the robots use a different iteration of the software than others. These findings can then be used to make changes in the robot's strategies, or change the weighing of certain actions that they take, in order to improve the overall performance of the team. The itself shader may also be adapted to use different data, such as where passes take place or where robots collide with other robots. The baseline of the shader used to create this heatmap was made by E. Albers [55].

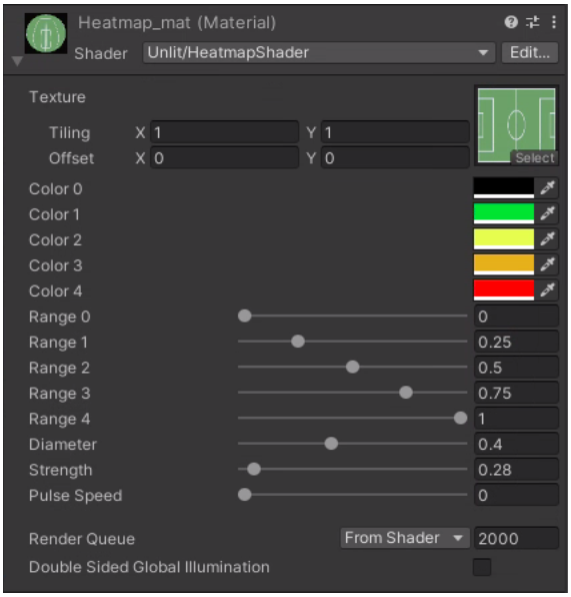



Figure 5.19: Inspector window of the heatmap shader

#### 5.6.3.5 Data export

Most of the current logging available for the robots, in the form of the historic data, is intended for the 3D simulator. The data can be loaded in the 3D simulator to get a sort of replay of a match. This data is not very readable and in a format generally only intended for Matlab. To provide a data endpoint that is human readible, can be used for different purposes and loaded into external programs as well, a CSV logger was created. This logger keeps track of each robot's data every time the data gets updated. It is then possible to export this data into a CSV file for each of the robots. To make this easily accessible and allow users to export the CSVs on demand instead of at all times, an entry in Unity's top bar menu has been added to provide quick access. It can be pressed at any time while the DT is running to export to CSVs the data of the robots up to that time.

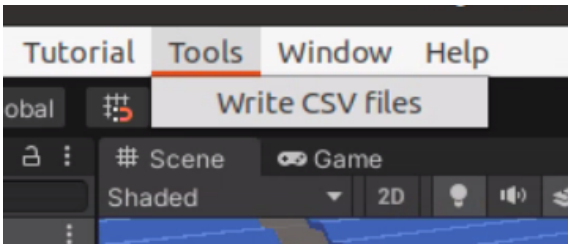



Figure 5.20: Menu item that can be used to export the robot data to CSV

An example excerpt of one of these log files can be seen in Figure 5.21

| time | ID | x | y | rotation | team | battery | motor1_temp | motor2_temp | motor3_temp |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2.1 | 1.047 | 1.663 | Magenta | 127 | 33 | 33 | 20 |
| 0.02 | 1 | 2.047 | 1.074 | 1.334 | Magenta | 127 | 33 | 33 | 20 |
| 0.12 | 1 | 0.512 | 2.091 | 1.568 | Magenta | 127 | 33 | 33 | 20 |
| 0.22 | 1 | 0.217 | 2.471 | 1.465 | Magenta | 127 | 33 | 33 | 20 |
| 0.3 | 1 | 0.135 | 2.573 | 1.492 | Magenta | 127 | 33 | 33 | 20 |
| 0.37 | 1 | 0.104 | 2.609 | 1.501 | Magenta | 127 | 33 | 33 | 20 |
| 0.47 | 1 | -0.031 | 2.768 | 1.511 | Magenta | 127 | 33 | 33 | 20 |
| 0.57 | 1 | -0.143 | 2.888 | 1.557 | Magenta | 127 | 33 | 33 | 20 |
| 0.63 | 1 | -0.191 | 2.93 | 1.573 | Magenta | 127 | 33 | 33 | 20 |

Figure 5.21: Example of one of the exported CSV files of the robots in Excel

These logs can be used to get an overview of the data and see how some values developed over time. They can also be used to generate e.g. graphs or fed into other data processing programs to gain more insight in their robot's behaviour. The code for this CSV logger can be seen in Listing A.10. Ideally this functionality this functionality could also be extended in the future to work live within Unity itself (similar to the heatmap), such as a live graph showing ball possession. Similarly to all other added functionalities, this CSV logger also makes uses of the benefits of the Observer pattern, so extending the data that is logged and exported should be relatively simple.

## 5.7 Resulting DT

With our work we set about to create a DT for the Turtle robot soccer project. Using Tao et al.'s DT definition, we investigated what artifacts were available for the project, and adapted them for use:

- For the physical entity we have not made any changes, as the robots were already satisfactory. The simulator was similar in this, allowing us to replicate the behaviour of the real robots accurately enough
- The virtual entity is where we did most of our work. The 3D CAD artifact was used to create new, up to date and lightweight 3D model. This model was place in our digital environment, Unity. The 3D model was than scripted to move according to the data that is being fed into Unity. This setup was then finally enriched with several other functionalities
- We reused the TRC control interface to control our result as well. This saved us time in creating a new interface, and saves Tech United's members from having to learn to use a new interface
- The live data available from the simulator was used to provide our digital environment with data. This data was used to move the robots and ball in the environment, but also to provide other metadata about the robots that can be used to infer other statistics

- We used the existing RTDB connection artifact in our setup, which we used to expose required data endpoints by compiling it into a library file acceptable by Unity. Reuse of this artifact allows for ease of scalability, use by Tech United, and integration with the rest of the Turtle soccer robot project

With this we have created a baseline platform that already provides a significantly improved visualization, and some new functionalities too. Tools to observe, investigate and analyse the robots have been added to showcase some of the possibilities of that a DT of the Turtle soccer robots may have.

While following the true definition of a DT however, our work cannot be considered a DT. Using Tao et al.'s 5D DT definition as guidance, we have managed to create nearly all of the 5D components. The most significant component that is still lacking though, is the "Connections" aspect of the 5D model. Namely, this component is described as bidirectional communication between each of the other 5D components. In our case however, we do not have this; there is communication between our PE and our VE (the simulator feeding Unity with data), but not the other way around (Unity providing data back to the simulator. The physical entity is currently unaware of any changes that happens in our digital environment. Therefore our final result is in reality not a true DT yet, but rather a digital shadow, a term that we introduced in Chapter 1.


Figure 5.22: The DT in operation

# Chapter 6

# Conclusions

In this chapter we will draw conclusions from the work that has been done in this thesis project, and suggestions will be made for future work. Section 6.1 provides a short recap of what steps have been undertaken in this graduation project. Next Section 6.2 answers the research questions that have been defined in Section 1.3. Finally in Section 6.3 suggestions for future research and continued work on this project will be given.

## 6.1 Summary

In this thesis project steps were taken to develop a DT for Eindhoven University of Technology's soccer robot team, Tech United. Being such a long standing project of over ten years, it forms a suitable use case for DT development with reuse of artifacts. More specifically, it allows for an investigation into the benefits and challenges faced in creating a DT for a project of this age.

We defined what our requirements for a DT would be in Chapter 4, which was based on Tao et al.'s 5D DT definition. Then after investigating the Turtle robot ecosystem and contact with Tech United, the available artifacts have been gathered and analyzed. We made them suitable for usage in a DT. This included creating a new lightweight model that is also textured and rigged, applying the RTDB to establish a connection with Unity and the Turtle Matlab models, and creating a baseline DT implementation. This implementation was then extended with several features to make the DT easier to use and provide some functions not currently available in the Turtle robot software ecosystem.

While our resulting work is a product that introduces new functionalities, improved visualization, and contains many of the elements that constitute a DT, by our earlier defined requirements it is not truly a digital twin. Instead, our result can rather be classified as a digital shadow. This is due to the lack of bidirectional communication between the physical entity and virtual entity. This manifests itself in the fact that currently Unity only receives data from the simulator, which as explained before, we have used instead of the physical robots using the same TRC interface, but Unity does not send anything back.

Despite this, we still obtained some interesting findings. The artifacts that were available to us in this project, most notably the simulator, the RTDB communication library, and the 3D models, were of great aid. The simulator and associated models driving the robot saved us a significant amount of work. Without them, we would need to have access to several of the robots in working condition for multiple hours, several days a week. This is already near impossible for us as outsiders to the project, especially with the pandemic conditions during which this project was executed. The simulator allowed us to work on the project at will without having the members of Tech United aid us at all times. The usage of the RTDB allowed us to quickly set up a connection between the PE (or simulator in our case) and the VE. The flexibility of this library also ensures us that swapping the simulator with the real robots later on is as painless as possible. Furthermore

using the same communication channels in our project as in the rest of the Turtle soccer robot project helps Tech United in maintaining and using our final product.

From these findings, we can generalize methods when faced with the development of a DT from (some) existing artifacts. The reuse of artifacts saved us a significant amount of work, so neglecting to use them seems like a great waste. Therefore when faced with creating a DT for an already long established project, it would be expedient to first investigate what artifacts are available. Documentation, software repositories, and, if the DT developers are not the same as the developers of the physical object, contact with the developers. The latter is especially of importance, as this will allow for the requirements engineering of the DT. Knowing what the desires and expectations of future users of the DT are helps in determining which artifacts are relevant, and what work is to be done to make them usable. The work following this will be highly dependant on a case by case basis. There may be situations where not many of the artifacts are close to being able to be applied, and require a lot of work to get them to this state. While other cases may have artifacts that are very flexible, such that they can be easily adapted for usage in a DT. Regardless of what the case may be, it would still be of interest to try and make use of the artifacts. Reuse of the artifacts minimizes any increase in complexity of the soft- and hardware of the overall project at hand, since many previously applied tools that are reapplied.

## 6.2 Research questions

**RQ1**: What are the requirements for the virtual entity of a digital twin for turtle soccer robots?

**Knowing what is required to make a DT provides us with a baseline indication of what steps we need to go through in our development, as well as a way of checking if our result actually constitutes as a DT.**
The requirements for the virtual component of the DT first stem from the definition of a DT. This was established in Section 1.2, and could be observed in some of the use cases in Section 2.1 as well.

- Naturally the first requirement of our DT would be a physical entity. For the turtle soccer robots, this entity is the robot itself and the software running on the robot. With the soccer robot project being this long standing it is quite mature already, satisfying all that we require of the physical entity for now.

- After this there is a virtual visualization aspect to consider. Since a DT would require a high fidelity environment to provide the necessary immersion to make all components recognisable, we have opted that 3D models will be required. Since the visualization by itself can already be seen as a benefit of the DT for this use case, effort was to be put into creating an accurate, high fidelity model. The provided model did not meet these demands right away, nor was it suitable for usage in Unity. This extended the requirement for the 3D model to also be lightweight, textured and rigged to fulfill our high fidelity demands.

- Next a digital environment in which the 3D model can be placed will be necessary. While a wide range of options are available, for this use case Unity was used. The advantage of game engines like Unity is that they provide a free, feature rich platform that are easy to extend, have wide range of resources available, and natively provide useful systems like physics out of the box.

- After creating a digital environment with a 3D model of the physical entity in it, next the data will be necessary. To create the utmost basics of the DT, data needs to be available on the position on the robots and the ball. This will be required to move our 3D models in

our digital environment. Extra data beyond this will be necessary to create more advanced features in the possible future.

- Finally to communicate this data, data connections will be necessary. These connections are needed to send data back and forth between the other components, and to send any feedback/commands to each other. The requirements for our DT is that these connections are two way, compatible with our digital environment of choice (Unity) and fast enough for DT. The latter is in our setup determined by data pulls that are done for every frame rendered by Unity, i.e. it should support at least 10 to 60 pulls per second (based off the average, common frame rates when using the DT). Note that this requirement we did not satisfy in our use case as we have no bidirectional communication. the RTDB that was used is capable of this however, we have just not been able to implement it yet.

**RQ2**: What currently existing artifacts can be used in creating the virtual entity of the digital twin?

**Knowing what interesting artifacts are available and what their current state is of great importance for our goals.**
The artifacts that have been reused in creating the virtual entity for this DT are the 3D CAD models, the Matlab models that form the driving software of the robots, the TRC interface, the simulators, the real time data and the RTDB. Note that while we ended up with a DS instead of a DT because of our lack of bidirectional communication, this is not to blame on our connection artifact, the RTDB. The RTDB is actually capable of back and forth data transfer, we have just not implemented it in our work.

Interestingly if looking at the definition of a DT according to Tao et al., this means that artifacts that would likely not be classified as directly part of the virtual entity have played a role in creating this virtual entity. The TRC and simulators are part of the services, and the RTDB mostly fits with the connections, as we have observed in Chapter 5. Tao provides no concrete insight in this aspect with regards to artifacts, so no there are no strict rules in place that dictate us that artifacts can only belong to one 5D component.

While these artifacts are not necessarily needed to construct the DT, they definitely did speed up development. This would imply that when investigating the available artifacts for constructing a DT, it is not sufficient to look at artifacts directly associated with the virtual entity. As Tao et al.'s 5D model would suggest, all components are interconnected, and therefore so are the artifacts.

**RQ3**: What tools and methods can we apply to create the virtual entity of a digital twin for the turtle soccer robot?

**There is no set recipe for creating a DT (or DS) as of yet, and many factors that differ on a case by case basis play a big part too. As we set out about creating our DT, we investigated several tools and methods that were available. This was done that our development process was as smooth and efficient as possible given the limited time frame that we had. It might also provide us with a more generalized way of working for DT/DS creation from existing artifacts.** Answering this RQ lies in the analysis of the artifacts, the possible goals, and the requirements of the DT. As was established in Chapter 5, the physical entity is already sufficient for creating our DT, so nothing had to be done for this component.

For our VE a 3D model was be necessary. To create a 3D model naturally 3D modeling techniques are necessary. The methods that will need to be applied will depend on the requirements of the DT. If models have been provided that are not too computationally impactful and not much else is necessary, then possibly little to no work has to be done here. Much more likely however, is that the available artifacts are not directly satisfactory. If the model is outdated or too dense, then a new, more lightweight model will need to be created. If the visualization aspect if of any

importance at all as well, then texturing the model, rigging it and creating animations will also be required. After all this there may also be some work involved in getting the model in the digital environment of choice. Textures, scaling, and rigging may all require a different workflow depending on the environment that is used.

For the digital environment we have decided to go with Unity, so knowledge of Unity and the programming language it uses, C#, will be necessary. To get our 3D model into Unity, we simply export the model from Blender in a format that is supported by Unity, such as FBX. The textures are to be exported in a file format that work with Unity's render pipeline as well, as was shown in Section 5.6.1.4.

The necessary data needs to come from the real time information feed containing positions of the robots, ball, and any other extra data necessary for desired functionalities. Since there are already examples available of the data that is available and how they should be used (including in which units they are) from the simulator, these can be used as reference to create the DT. This data is stored in variables in C/C++ since that is what the majority of the turtle software is written in. These data types and structures will need to be mapped to an equivalent pairing in C# that Unity uses.

Lastly to get this data into Unity we need a connection from the physical entity/simulator. For this the RTDB was used. Since the RTDB is also written in C/C++, a way of getting it to work in C# was needed. To this end it was compiled into a shared library file which could be loaded into Unity. For when we wish to turn our DS into a proper DT later, this can be done using the same approach, where the compiled library should then also have the 'put' functions exposed. Having to undertake more work to make the artifacts like the RTDB compatible is a possible challenge in other use cases where DTs are developed for a long standing project. Legacy or otherwise incompatible technologies are likely to be found in these projects, which may pose to be a source of conflicts in development.

Looking at the methods and tools that we have applied, we can observe that knowledge of 3D modeling, digital environment software, communication methods are important to have for DT/DS development. Many different options are available here, the choice of which depend on the preference of the creators and the goals of the DT/DS. However, not for all other use cases our experiences may represent what can be generally applied. For example, a 3D model is not a hard requirement to have in a DT/DS. Thus in use cases where a 3D model is not used, knowledge about 3D modeling is naturally not necessary either. Because of the many different factors that can play a part in creating a DT/DS, coming up with a general technique that can be applied when creating a DT/DS from artifacts is quite hard. For the methods we applied a similar consensus holds. While we can recommend e.g. the high to low poly method that we employed, this may not always be applicable, even if a 3D model is used (such as when the 3D model artifact is already lightweight enough).

What we can do however is provide a possible workflow plan. As we had observed in Section 6.1 already, it is expedient to look at what artifact are available, come up with requirements for the DT/DS, and then investigate what artifacts are relevant to these requirements and what work they require to make them applicable. The prioritization of each component of the DT/DS will differ, just like in any other software engineering project. However, identifying what is required to create some of the most basic functionality (in our case, movement of the robots and the ball) is a good starting point, as other functionalities will likely need to build on this anyway.

**RQ4**: What discrepancies between the virtual and physical entities are of significance in a digital twin for the turtle soccer robots, and how can they be negated if necessary?

**There will always be differences between the actual physical system and a digital counterpart. How significant the impact of these discrepancies are will not be the same for every project, but possible ways to mitigate or minimize those that are of significance is important if time and resources allow it. Some discrepancies are of**

**such magnitude that they prevent the operation of the DT as a whole, while other discrepancies are actually introduced on purpose**
Since a DT's aim is to create a replication of the physical counterpart that is fit for the purposes of the DT, there will realistically be some differences between a DT and the actual physical object. Similarly to human biological twins, digital twins also feature several discrepancies from each other, even if hard to distinguish. Even with access to a theoretical, perfect, 1 to 1 virtual replica of the physical robot with all the same functionalities and data, some sacrifices have to be made somewhere (for example, there will always be a delay in transmitting information, compared to just observing the physical entity with your own eyes). On a case by case basis some elements may also be purposely given less attention, as they do not play a major role in the use case at hand.

An example of this in this use case can be found in the 3D modeling aspect. The 3D CAD model provided a very highly detailed model, down to every bit and bolt used on the robot. In reality however, most of these tiny components are not of interest when using the DT. To this end several pieces were simply left out, or baked into the textures such that there is still semblance of them visible. The resulting lightweight model is not fully accurate either, with some non essential parts not included, such as small bolts, wires and connectors. Nevertheless, the created model is accurate enough and has a degree of fidelity such that the components are recognisable and look correct.

The biggest possible discrepancy between the digital entity and the physical entity in this use case is the usage of the simulator. Used as a replacement for the actual physical robot during development of the DT, in theory the whole physical entity has not played a part its creation. While certainly of great benefit, it is conceivable that, given the complexity of the whole robot, the simulator is not capable of fully simulating every aspect of the actual robot accurately. Some elements are even impossible to achieve in the simulator compared to real life operation. For example visual information from actual cameras is vastly different from simulated data in a 2D space. Despite this, care has been taken to cause as few problems as possible. The possible discrepancies are minimized by using data that is only available for both the simulator and the real robots, and by using the same communication channels. The usage of the RTDB should in theory allow anything that is capable of providing the relevant data to be connecting using the same channels, whether that be a simulator or actual robot. The mitigation of discrepancies between the digital and physical entities thus further ties into one of the advantages of reusing artifacts in creating DTs: the usage of already applied technologies to minimize technical issues, and making the DT easier to use and maintain.

When it comes to discrepancies in DTs/DSs and how to deal with them however, there is no black and white answer. DTs can include such a great amount of subsystems, sensors and data that differences between reality and the virtual will always exist. Not to mention that some discrepancies are introduced on purpose. Just like in our 3D model where some small pieces of the robots are not modeled on purpose, discrepancies were introduced just because they were not relevant enough to include in the DT.

This brings us to a two step approach to dealing with them. First is to determine whether the discrepancies are of any significant impact. Just like the discrepancies that we introduced on purpose, these had no impact on the actual requirements of the DT. In fact, trying to mitigate this discrepancy would have actually been negative for the DT, as it would impact the performance unfavorably. If a discrepancy is undesirable to have, then steps will need to be undertaken to do something about it. Here the approach will vary based on the type of discrepancy, and how severely it impacts the overall DT.

> **RQ5**: Given the available artifacts, what findings and functionalities can be realized using the digital twin?

**In the end, we wish that our use case not only provides a usable result, but also interesting generalized findings for DT development as a whole when working with a**

**long standing project and its artifacts.**
The answer to this RQ is rather broad. Theoretically, the artifacts can be seen as just an aid in creating a DT more easily. Once the DT has been fully established, all desired findings and functionalities can be realized that make use of the advantages of a DT (barring some case by case limitations). To formalize this answer more to this use case, the focus shall be put on findings and functionalities more directly associated with the used artifacts. Furthermore the answers can also be split up in two distinction categories; findings related to DTs as a whole and findings related to this specific use case.

The physical entity artifacts speak for themselves. In our use case the physical entity was already quite mature, with no work necessary from our side. When developing a DT from artifacts for an already long standing project such as this, this would likely be a common situation. The most notable work that could be necessary on the physical side would be the application of sensors and connections to supply the necessary data and communication back and forth between the physical entity and virtual entity.

For the virtual entity the 3D CAD model artifact provided to be a great benefit in the creation of the DT. Using it a to-scale model could be created a lot more quickly than starting from scratch. Furthermore it also provided a great insight in many of the components of the robot that were not visible at a first glance. The resulting new 3D model can then play a vital role in the overall functioning of the DT. To summarize this to a more general answer, the existence of 3D model artifacts can aid in increasing the understanding of the physical entity and speed up DT development. For any 3D artifacts it is preferable for them to be to scale, up to date, accurate, capable of animations, and equipped with the proper materials. If one of these aspects is not necessary (or at least not to the degree that is available), they can always be removed/reduced. This is often a lot less time consuming than making any missing properties from scratch.

With regards to services artifacts, the TRC and simulators were available. The TRC provided a good interface already for controlling much of our DT. Reusing it saves work on creating our own interface, and operating the DT will be easier for Tech United when we use the same control interfaces that they are already used to. The simulator was an extremely beneficial asset to have, as it meant that the DT could be developed at all times without the need of an actual physical robot nearby to provide data. Not only did this speed up development from a technical standpoint, but also from an organizational one. Getting access to the robot every day is simply not feasible. Being able to develop at all time from home is therefore of great benefit. The simulator also served as a comparison check for the DT. Data that is being used in both the simulator and the DT can be compared to make sure that e.g. all units conversions are done correctly. A possible finding here is that the reuse of service artifacts (such as interfaces) not only saves time in creating the DT, but also makes the DT easier for the intended audience to use. Other services that can mock the data and behaviour of the actual physical entity can be of great benefit when the physical entity is not readily available at all times. This is not an unrealistic scenario to consider, especially when creating DTs for sizeable entities such as aircraft, or collective entities like factories or cities.

Data wise, the real time data available was used in this use case. Since these data variables were already defined and in use in the simulator, there were ample examples to learn from on how they could be applied. The historical data that was available has no use as of yet and did not have many applications already available to learn from, so this could be a future avenue of research. The real time data was used to create the baseline functionalities necessary for the DT, such as the movement of the robots. This data was then used to create some new functionalities not currently available, such as the data panel, heatmap and CSV exporter. Since the data is so crucial to the functioning of the DT, it was beneficial to have these artifacts available. This should extend to DTs in general as well. While the scale, complexity, accuracy or availability of the data may differ on a case by case basis, creating a DT without data is naturally out of the question. An investigation into the data that may be available as an artifact, how they have been used before and any documentation that may exist for it is thus an expedient step to undertake when developing a DT.

Lastly there are artifacts with regards to the connections between all the components of our DT. In our use case this artifact was their method of utilising shared memory via a RTDB. This

artifact proved to be a great help in creating the DT, as it provided a tried and tested method of communication. The RTDB's data scheme is easy to understand, is efficient in use and can be applied for the communication between virtually any component due to being built on a black box basis. The biggest challenges in using this RTDB were actually caused by Unity, where some work had to be done to compile the RTDB in a library format acceptable by Unity. Nevertheless, a system capable of connecting different components already being available as an artifact meant that a lot of time was saved on our end. Instead of having to create e.g. a TCP connection of our own, we could simply reuse the data connections they already have in place. This not only saves us time, but also makes it easier for Tech United to use and maintain the connection, and possibly mitigate any discrepancies, as was already described in our answer to RQ4. Since the connections are so fundamental to bringing all the components of DT together, it is important to lay a good foundation from the start of DT development. We had contact with the Tech United team about this aspect to ensure that our communication matches up with their current setup as much as possible, making a possible future integration of our result into their software stack easier.

In the end from our use case's perspective, we can definitely say that the reuse of existing artifacts has helped us tremendously in our creation process. Some of the artifacts needed no work to apply at all, saving us a lot of time. Others needed some adjustment to make them work, but even then this was a lot less time consuming then making them from scratch. Artifacts also provided us with a good insight of the inner workings of the Turtle robot project already, allowing us to more quickly get to work on creating the DT. This leaves us to conclude that, even though the state of the artifacts can differ greatly on a case by case basis, it is still lucrative to investigate, and possibly integrate, any existing artifacts. It should still be noted however, that care has to be taken in this. As we will discuss in Section 6.3 as well, it could be possible that some artifacts differ too greatly from the intended goal to realistically make them work. In this case, the amount of work required to apply them in a DT for a specific goal may not be in line with the investment that needs to be made in order to create something for that goal from scratch. Taking our 3D CAD model as an example: say that the provided model is very outdated, not to scale, and is lacking many important components. In this case, it may be faster to create a new model from scratch based on recently taken images, compared to going through the provided artifact, verifying every single piece, and then recreating/adjusting many of the provided parts, while checking continuously whether what we are doing is still correct.

## 6.3 Future work

Regarding future work in relation to this project, there are two perspectives to consider: that of this use case, and the wider perspective of reusing artifacts in the creation of a DT.

For the case of the Turtle soccer robot use case, the main priority is to turn it into a proper DT, instead of a DS. This can be achieved by adding bidirectional communication to the project, such that not only a change of state in the physical will result in a change of state in the virtual, but vice versa as well. The project should also be tested more in depth with the actual robots, instead of just the simulator as we have done.

Furthermore the extra added functionalities, while new to the Turtle robot software stack, do not make optimal use of all the benefits of a true DT. In the future, it would be desired that this aspect is expanded further. Several avenues are available to explore here:

- One application could for example include swapping in certain components and testing how they behave. Since CAD models are created for new candidate components anyway, it would be possible to adapt them easily to the lightweight model, place the new model in Unity, and then use Unity's own physics system to test the behaviour. This would save on having to machine several components in real life and manually attaching them to the real robots in order to test what design is the best.

- Another application of the DT in this use case can be analysis of data flow in real-time.

When the DT data that is being communicated between the physical and virtual entity has been expanded, it would be possible to process and analyze this data as a match is ongoing. We introduced the heatmap functionality to provide an example analysis method, but many more analysis possibilities exist that make more use of the capabilities of a DT than this. For example, online real-time analysis can be employed to analyse data of the robots in play over a large period of time, which can then be used to update parameters of the robots on the go.

- Building off the concept of tournaments, another interesting application of the DT would be troubleshooting from afar. Since the robot soccer tournaments are organized in differing locations across the globe every year and the Tech United team is quite sizeable, it is not unlikely that not every member can attend every tournament at all times. Since each team member has their own specialization as well, this could lead to the team missing out on someone's knowledge and skills when the need arises. Using a DT however, a member could accurately monitor the performance, actions, and issues of the robots from far away. Hereby possible issues can be noticed by simply observing the robot as if that person was actually there. Findings can then be communicated with the team on site, or even software adjustments may be made from afar.

- Lastly, Unity itself has many different interesting technologies available as was noted in Section 5.6.2. VR could be used to observe individual components in detail while the DT is running, and the robot could be virtually taken apart without having to disassemble an actual machine. Machine learning could be applied to develop new strategies for the robot by applying reinforcement learning, something that Unity's ML agents have been applied for before. Technologies such as Prespective's DT tools could be integrated to allow for e.g. more in depth testing of mechatronic processes.

With respect to DTs/DSs as a whole, some possible conclusions can be made. In this use case the artifacts, while they required work to make them usable, definitely saved time. The overhead incurred by adapting the artifacts definitely outweighed our estimated necessary time investment to create the components from scratch. Applying the artifacts also led us to reusing many of the same methods and technologies that were already known to Tech United. This makes the DT easier for them to understand, maintain, and possibly integrate in their workflow in the future.

While we conclude that for this use case the artifacts were of great benefit in the development process, it is still just one use case in the end. To truly generalize these findings, it would be expedient if more case studies were to be conducted. In differing instances of reusing artifacts the experience may be different. Perhaps in some cases the work required to reuse the artifacts outweigh the benefit that they may bring. It would be especially interesting to consider not only different artifacts across these use cases, but also which stage of development these artifacts originate from.

For example, the 3D model that was used in our use case needed a bit of work since it was made during the construction stage of the project. Usage in a real time lighting setting was never considered at that time, and as such it was not taken into consideration when creating the 3D model. However, a different use case may have a 3D model from a later stage, such as during a release stage. Here the 3D model may have actually been created to produce renders, instructional videos or promotional material. A 3D model from this stage may be a lot more apt to directly apply in a DT without much work necessary, as it may already be lightweight, textured and animated when used for other purposes. It is thus recommended to not only make a distinction on the context of future research use cases, but also on the context of the artifacts themselves to truly generalize the findings in this thesis report.

# Bibliography


[1] F.Tao, M.Zhang, and AYC.Nee. *Digital twin driven smart manufacturing*. Academic Press, 2019.

[2] Tech United. Tech United. https://www.techunited.nl/. Accessed: 2021/04/08.

[3] Tech United. Turtle robots. https://www.techunited.nl/en/soccer_robots. Accessed: 2021/04/08.

[4] M. Grieves. Digital twin: manufacturing excellence through virtual factory replication. *White paper*, 1:1–7, 2014.

[5] M. Grieves. Virtually intelligent product systems: digital and physical twins. AIAA ARC, 2019.

[6] E. Glaessgen and D. Stargel. The digital twin paradigm for future nasa and us air force vehicles. In *53rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference 20th AIAA/ASME/AHS adaptive structures conference 14th AIAA*, page 1818, 2012.

[7] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn. Digital twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine*, 51(11):1016–1022, 2018.

[8] M. Matulis and C. Harvey. A robot arm digital twin utilising reinforcement learning. *Computers & Graphics*, 95:106–114, 2021.

[9] Y.Liau, H.Lee, and K. Ryu. Digital twin concept for smart injection molding. In *IOP Conference Series: Materials Science and Engineering*, volume 324, page 012077. IOP Publishing, 2018.

[10] K. Vijayakumar, C. Dhanasekaran, R. Pugazhenthi, and S. Sivaganesan. Digital twin for factory system simulation. *International Journal of Recent Technology and Engineering*, 8(1):63–68, 2019.

[11] P. Aivaliotis, K. Georgoulias, and K. Alexopoulos. Using digital twin for maintenance applications in manufacturing: State of the art and gap analysis. In *2019 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, pages 1–5. IEEE, 2019.

[12] S. Boschert and R. Rosen. Digital twin—the simulation aspect. In *Mechatronic futures*, pages 59–74. Springer, 2016.

[13] L. Wright and S. Davidson. How to tell the difference between a model and a digital twin. *Advanced Modeling and Simulation in Engineering Sciences*, 7(1):1–13, 2020.

[14] I. Krasikov and A.N. Kulemin. Analysis of digital twin definition and its difference from simulation modelling in practical application. In *III Annual International Conference" System Engineering".—Ekaterinburg, 2020*, pages 105–109. Knowledge E, 2020.

[15] V. Kuts, T. Otto, T. Tähemaa, and Y. Bondarenko. Digital twin based synchronised control and simulation of the industrial robotic cell using virtual reality. *Journal of Machine Engineering*, 19, 2019.

[16] J. Savolainen and M. Urbani. Maintenance optimization for a multi-unit system with digital twin simulation. *Journal of Intelligent Manufacturing*, pages 1–21, 2021.

[17] A. Ait-Alla, M. Kreutz, D. Rippel, M. Lütjen, and M. Freitag. Simulation-based analysis of the interaction of a physical and a digital twin in a cyber-physical production system. *IFAC-PapersOnLine*, 52(13):1331–1336, 2019.

[18] H. Lasi, P. Fettke, HG. Kemper, T. Feld, and M. Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.

[19] E. Negri, L. Fumagalli, and M. Macchi. A review of the roles of digital twin in cps-based production systems. *Procedia Manufacturing*, 11:939–948, 2017.

[20] W. Yang, Y. Tan, K. Yoshida, and S. Takakuwa. Digital twin-driven simulation for a cyber-physical system in industry 4.0. *DAAAM International Scientific Book*, pages 227–234, 2017.

[21] G. Schrotter and C. Hürzeler. The digital twin of the city of zurich for urban planning. *PFG–Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, 88(1):99–112, 2020.

[22] Q. Qiao, J. Wang, L. Ye, and RX. Gao. Digital twin for machining tool condition prediction. *Procedia CIRP*, 81:1388–1393, 2019.

[23] Eindhoven University of Technology. The messi of robot soccer is a tricycle with a fisheye. https://www.tue.nl/en/news/features/soccer-robots/. Accessed: 2021/06/21.

[24] RoboCup. Robocup federation official website. https://www.robocup.org/. Accessed: 2021/09/27.

[25] P. Barbosa and I. Castellanos. *Ecology of predator-prey interactions*. Oxford University Press, 2005.

[26] D. Pagliari and L. Pinto. Calibration of kinect for xbox one and comparison between the two generations of microsoft sensors. *Sensors*, 15(11):27569–27589, 2015.

[27] L. de Koning, Juan Pablo Mendoza, Manuela Veloso, and René van de Molengraft. Skills, tactics and plays for distributed multi-robot control in adversarial environments. In *Robot World Cup*, pages 277–289. Springer, 2017.

[28] L. Almeida, F. Santos, T. Facchinetti, P. Pedreiras, V. Silva, and L. Seabra Lopes. Coordinating distributed autonomous agents with a real-time database: The cambada project. In *International Symposium on Computer and Information Sciences*, pages 876–886. Springer, 2004.

[29] N.L. Webster. High poly to low poly workflows for real-time rendering. *Journal of visual communication in medicine*, 40(1):40–47, 2017.

[30] Polygon. Texture Baking. http://wiki.polycount.com/wiki/Texture_Baking. Accessed: 2021/06/21.

[31] Blender. Unwrapping. https://docs.blender.org/manual/en/2.79/editors/uv_image/uv/editing/unwrapping/index.html. Accessed: 2021/06/21.

[32] M. Pharr, W. Jakob, and G. Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[33] J.Petty. What is 3d rigging for animation & character design? https://conceptartempire.com/what-is-rigging/. Accessed: 2022/01/04.

[34] A. Aristidou, J. Lasenby, Y. Chrysanthou, and A. Shamir. Inverse kinematics techniques in computer graphics: A survey. In *Computer Graphics Forum*, volume 37, pages 35–58. Wiley Online Library, 2018.

[35] Sopiro. Procedural Animation in Unity. https://github.com/Sopiro/Unity-Procedural-Animation. Accessed: 2022/01/04.

[36] Crytek. The complete solution for next generation game development by Crytek. https://www.cryengine.com/. Accessed: 2021/06/21.

[37] Epic Games. The most powerful real-time 3d creation platform. https://www.unrealengine.com/en-US/. Accessed: 2021/06/21.

[38] Unity Technologies. Unity. https://unity.com/. Accessed: 2021/06/21.

[39] Blender Foundation. Blender. https://www.blender.org/. Accessed: 2021/11/10.

[40] Open Cascade. Cad Assistant. https://www.opencascade.com/products/cad-assistant/. Accessed: 2021/11/10.

[41] Adobe. Substance Painter. https://www.substance3d.com/. Accessed: 2021/11/10.

[42] Unity Technologies. Made with Unity. https://unity.com/madewith. Accessed: 2021/06/21.

[43] Unity Technologies. Architecture, engineering & construction. https://unity.com/solutions/architecture-engineering-construction. Accessed: 2021/06/21.

[44] B.B.M. van Ginneken. Design of a digital twin for the sorting workstation of a festo production line, 2020.

[45] Prespective. Prespective Digital Twin Software. https://assetstore.unity.com/packages/tools/utilities/prespective-digital-twin-software-161664. Accessed: 2021/11/09.

[46] Unity Technologies. Virtual Reality Development. https://unity.com/unity/features/vr. Accessed: 2021/11/09.

[47] Unity Technologies. Unity Machine Learning Agents. https://unity.com/products/machine-learning-agents. Accessed: 2021/11/09.

[48] Unity Technologies. High definition render pipeline overview: High definition rp: 7.1.8. https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@7.1/manual/index.html. Accessed: 2021/06/21.

[49] Unity Technologies. Universal render pipeline overview: Universal rp: 12.0.0. https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@12.0/manual/index.html. Accessed: 2021/06/21.

[50] Unity Technologies. Feature comparison table URP vs HDRP. https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@7.1/manual/universalrp-builtin-feature-comparison.html. Accessed: 2021/06/21.

[51] Unity Technologies. URP roadmap. https://portal.productboard.com/unity/1-unity-platform-rendering-visual-effects/tabs/3-universal-render-pipeline. Accessed: 2021/06/21.

[52] Unity Technologies. HDRP roadmap. https://portal.productboard.com/unity/1-unity-platform-rendering-visual-effects/tabs/18-high-definition-render-pipeline. Accessed: 2021/06/21.

[53] E. Gamma, R. Johnson, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[54] Mono Project. About Mono — Mono. https://www.mono-project.com/docs/about-mono/. Accessed: 2021/10/06.

[55] E. Albers. Unity Heatmap Shader. https://github.com/ericalbers/UnityHeatmapShader. Accessed: 2021/10/11.

# Appendix A

# Code listings

```
1 import bpy
2 import math
3
4 current_selected_obj = bpy.context.selected_objects
5 for x in current_selected_obj:
6     if x.type == 'MESH':
7         x.name = x.data.name
```



Listing A.1: Renames the Blender objects to the name value stored in the object's data (mesh-to-obj-name.py)

```
1 import bpy
2 import math
3
4 current_selected_obj = bpy.context.selected_objects
5 for x in current_selected_obj:
6     if x.type == 'MESH':
7         x.name = x.name + '.high'
```



Listing A.2: Appends ".high" to the selected object names (append-high.py)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <signal.h>
5  #include <math.h>
6  #include <utility>
7  #include "/home/robocup/svn/trunk/src/Turtle3/Libs/multicast/rtdb/rtdb_user.h"
8  #include "/home/robocup/svn/trunk/src/Turtle3/Libs/multicast/rtdb/rtdb_api.h"
9  #include "/home/robocup/svn/trunk/src/Turtle3/Libs/multicast/worldmodel/
       KinectShared.h"
10 #include "/home/robocup/svn/trunk/src/Turtle3/Libs/multicast/worldmodel/BallShared.
       h"
11 #include "/home/robocup/svn/trunk/src/Turtle3/Libs/multicast/worldmodel/
       StrategyShared.h"
12 #include "/home/robocup/svn/trunk/src/Turtle3/Libs/multicast/worldmodel/
       TRC2SimLocal.h"
13 #include "/home/robocup/svn/trunk/src/Turtle3/Libs/multicast/worldmodel/SimLink.h"
14 #include "/home/robocup/svn/trunk/src/Turtle3/Libs/multicast/worldmodel/Sim2Turtle.
       h"
15 #include "process.h"
16 #if _MSC_VER                // this is defined when compiling with Visual Studio
17 #define EXPORT_API __declspec(dllexport) // Visual Studio needs annotating exported
        functions with this
18 #else
19 #define EXPORT_API // XCode does not need annotating exported functions, so define
       is empty
```

```
20 #endif
21
22 // ------------------------------------------------------------------------
23 // Plugin itself
24
25 // Link following functions C-style (required for plugins)
26 extern "C"
27 {
28 #define DEVPC 0
29 #define TURTLE 1
30
31   // Declaring variables
32   int end2 = 0;
33   int t_BS = 0;
34   int t_SS = 0;
35   int t_SL = 0;
36   int t_ST = 0;
37   int age_ST = 0;
38
39   struct BallShared S_BS;
40   struct StrategyShared S_SS;
41   struct TRC2SimLocal TSL;
42     struct SimLink SL_in;
43   struct Sim2Turtle ST;
44
45   // Initialize RTDB
46   int ret = DB_init();
47
48   static void signal_catch(int sig)
49   {
50     end2 = 1;
51   }
52
53   EXPORT_API int check_db_status() {
54     if (ret != 0)
55     {
56       return 1;
57     }
58     else
59     {
60       return 0;
61     }
62   }
63
64   EXPORT_API void get_turtle_position(int agent, double* robot_x, double* robot_y,
65    double* robot_phi)
66   {
67     t_SS = DB_get(agent, STRATEGY_SHARED, &S_SS);
68     *robot_x = (double)S_SS.current_xyo[0]*0.001;
69     *robot_y = (double)S_SS.current_xyo[1]*0.001;
70     *robot_phi = (double)S_SS.current_xyo[2]*0.001;
71   }
72
73   EXPORT_API void get_ball_position(double* ball_x, double* ball_y,
74   double* ball_z)
75   {
76     t_ST = DB_get(Whoami(), SIM2TURTLE, &ST);
77     *ball_x = ST.ball_xyz[0]*0.001;
78     *ball_y = ST.ball_xyz[1]*0.001;
79     *ball_z = ST.ball_xyz[2]*0.001;
80   }
81
82   EXPORT_API void get_turtle_info(int agent, char* teamColor, char* batteryVoltage,
83   char* emergencyStatus, double* motorTemperature1, double* motorTemperature2,
84    double* motorTemperature3)
85   {
86     t_SS = DB_get(agent, STRATEGY_SHARED, &S_SS);
```

Checks DB status

Exposes turtle position data

Exposes ball position data

Exposes turtle misc. data

```
87      *teamColor = S_SS.teamColor;
88      *batteryVoltage = S_SS.batteryVoltage;
89      *emergencyStatus = S_SS.emergencyStatus;
90      *motorTemperature1 = (double)S_SS.motor_temperature[0];
91      *motorTemperature2 = (double)S_SS.motor_temperature[1];
92      *motorTemperature3 = (double)S_SS.motor_temperature[2];
93    }
94  } // end of export C block
```

Listing A.3: Main communication file that exposes data endpoints for Unity (TurtlePlugin.cpp)

```
1  # Define compiler and flags
2  CC = gcc
3  CFLAGS = -c -g -fPIC -O3
4
5  # Create turtle plugin
6  turtle-plugin:
7    @# Create object file
8    $(CC) $(CFLAGS) -o build/TurtlePlugin.o src/TurtlePlugin.cpp
9    @# Create shared library linked with RTDB library
10   $(CC) -shared -o build/TurtlePlugin.so build/TurtlePlugin.o -lrtdb
11   @# Create debug file
12   objcopy --only-keep-debug build/TurtlePlugin.so build/TurtlePlugin.debug
13   strip --strip-debug build/TurtlePlugin.so
14   @# Copy resulting files to Unity Assets
15   cp -t "../Unity/TurtleDigitalTwin/Assets/Plugins/Linux/" build/TurtlePlugin.so
         build/TurtlePlugin.debug
16
17 # Clean up the output dir
18 clean:
19   $(RM) -r ./build
```

Make command to build plugin

Listing A.4: Used to compile the communication plugin to a shared library (MakeFile)

```
1  using System;
2  using System.Runtime.InteropServices;
3  using UnityEngine;
4
5  public class TurtleCommManager : MonoBehaviour {
6      /**
7      * Importing all necessary functions from shared library
8      */
9      [DllImport("TurtlePlugin", CallingConvention = CallingConvention.Cdecl,
           EntryPoint = "test")]
10     private static extern int test(int agent);
11
12     [DllImport("TurtlePlugin", CallingConvention = CallingConvention.Cdecl,
           EntryPoint = "get_turtle_position")]
13     private static extern void get_turtle_position(int agent, ref double robot_x,
           ref double robot_y, ref double robot_phi);
14
15     [DllImport("TurtlePlugin", CallingConvention = CallingConvention.Cdecl,
           EntryPoint = "get_ball_position")]
16     private static extern void get_ball_position(ref double ball_x, ref double
           ball_y, ref double ball_z);
17
18     [DllImport("TurtlePlugin", CallingConvention = CallingConvention.Cdecl)]
19     private static extern int check_db_status();
20
21     [DllImport("TurtlePlugin", CallingConvention = CallingConvention.Cdecl,
           EntryPoint = "get_turtle_info")]
22     private static extern void get_turtle_info(int agent, ref IntPtr teamColor, ref
            IntPtr batteryVoltage, ref IntPtr emergencyStatus, ref double
           motorTemperature1, ref double motorTemperature2, ref double
           motorTemperature3);
23
24     /**
```

```
25      * Declaring variables (make them public or append them with
26      * [SerializeField] to make them visible in the inspector)
27      */
28      [Header("Smooth out robot and ball movement?")]
29      [Tooltip("Enable this to smooth out the movement of the robots and ball by "
30      + "linear interpolating their positions between updates. This will lead to a "
31      + "slight delay between data coming in and may not always a 100% represent "
32      + "their exact positions, so disable this if you would prefer a more absolute "
33      + "representation.")][SerializeField] bool smoothMovement;
34      double robot_x = 0.0;
35      double robot_y = 0.0;
36      double robot_phi = 0.0;
37      IntPtr teamColor;
38      IntPtr batteryVoltage;
39      IntPtr emergencyStatus;
40      double motorTemperature1;
41      double motorTemperature2;
42      double motorTemperature3;
43      double ball_x = 0.0;
44      double ball_y = 0.0;
45      double ball_z = 0.0;
46      const int MAX_ROBOTS = 6;
47
48      // Actions for observer pattern
49      public static event Action<bool, double, double, double> ballUpdateEvent;
50      public static event Action<int, bool, double, double, double>
            robotPosUpdateEvent;
51      public static event Action<int, IntPtr, IntPtr, IntPtr, double, double, double>
             robotDataUpdateEvent;
52
53      // Start is called before the first frame update
54      void Start() {
55          // Check if the RTDB has been correctly initialized
56          if (check_db_status() != 1) {
57              Debug.Log("RTDB succesfully initialized");
58          } else {
59              Debug.LogError("RTDB was not correctly initialized! Aborting");
60              UnityEditor.EditorApplication.isPlaying = false;
61          }
62      }
63
64      // Update is called once per frame
65      void Update() {
66          // Iterating over all the turtles
67          for (int i = 2; i < MAX_ROBOTS; i++) {
68              // Get all Turtle positions and rotations
69              get_turtle_position(i, ref robot_x, ref robot_y, ref robot_phi);
70              // Notify robot position listeners
71              robotPosUpdateEvent?.Invoke((i - 1), smoothMovement, robot_x, robot_y,
                    robot_phi);
72
73              // Get data from the Turtles
74              get_turtle_info(i, ref teamColor, ref batteryVoltage, ref
                    emergencyStatus, ref motorTemperature1, ref motorTemperature2, ref
                    motorTemperature3);
75              // Notify robot data listeners
76              robotDataUpdateEvent?.Invoke((i - 1), teamColor, batteryVoltage,
                    emergencyStatus, motorTemperature1, motorTemperature2,
                    motorTemperature3);
77          }
78
79          // Get ball position
80          get_ball_position(ref ball_x, ref ball_y, ref ball_z);
81          // Notify ball listeners
82          ballUpdateEvent?.Invoke(smoothMovement, ball_x, ball_y, ball_z);
83      }
84  }
```

Listing A.5: The subject file in the Observer pattern that handles all the data communication for the robot's data (TurtleCommManager.cs)

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

public class RobotController : MonoBehaviour {
    [SerializeField] private int turtleID;
    private float speed = 7.0f;
    private IntPtr currentTeam = (IntPtr)0;

    private void OnEnable() {
        // Subsribe to the RobotDataUpdateEvent
        TurtleCommManager.robotPosUpdateEvent += OnRobotPosUpdate;
        TurtleCommManager.robotDataUpdateEvent += OnRobotDataUpdate;
    }

    private void OnDisable() {
        // Unsubsribe to the RobotDataUpdateEvent
        TurtleCommManager.robotPosUpdateEvent -= OnRobotPosUpdate;
        TurtleCommManager.robotDataUpdateEvent -= OnRobotDataUpdate;
    }

    private void OnRobotPosUpdate(int turtleID, bool smoothMovement, double robot_x
        , double robot_y, double robot_phi) {
        if (turtleID == this.turtleID) {
            // Set Turtle positions and rotations in Unity
            // Invert depending on team, values are relative to center of field,
            // own side determining positive and negative values
            if (currentTeam == (IntPtr)0) {
                if (smoothMovement) {
                    Vector3 currentPos = this.transform.position;
                    Vector3 newPos = new Vector3((float)-robot_x, this.transform.
                        position.y, (float)-robot_y);
                    this.transform.position = Vector3.Lerp(currentPos, newPos,
                        speed * Time.deltaTime);
                } else {
                    this.transform.position = new Vector3((float)-robot_x, this.
                        transform.position.y, (float)-robot_y);
                }
                this.transform.eulerAngles = new Vector3(0, (float)-robot_phi * 10,
                     0);
            } else {
                if (smoothMovement) {
                    Vector3 currentPos = this.transform.position;
                    Vector3 newPos = new Vector3((float)robot_x, this.transform.
                        position.y, (float)robot_y);
                    this.transform.position = Vector3.Lerp(currentPos, newPos,
                        speed * Time.deltaTime);
                } else {
                    this.transform.position = new Vector3((float)robot_x, this.
                        transform.position.y, (float)robot_y);
                }
                this.transform.eulerAngles = new Vector3(0, (float)robot_phi * 10,
                    0);
            }
            CSVLogger.Instance.Log(this.transform.position.x);
        }
    }

    private void OnRobotDataUpdate(int turtleID, IntPtr teamColor, IntPtr
        batteryVoltage, IntPtr emergencyStatus, double motorTemperature1, double
        motorTemperature2, double motorTemperature3) {
```

```
52          if (turtleID == this.turtleID) {
53              this.currentTeam = teamColor;
54          }
55      }
56  }
```

Listing A.6: Moves the robots in Unity to the correct places according to the received data (RobotController.cs)

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class BallController : MonoBehaviour
6   {
7       private float speed = 7.0f;
8       private void OnEnable() {
9           // Subsribe to the RobotDataUpdateEvent
10          TurtleCommManager.ballUpdateEvent += OnBallUpdate;
11      }
12
13      private void OnDisable() {
14          // Unsubsribe to the RobotDataUpdateEvent
15          TurtleCommManager.ballUpdateEvent += OnBallUpdate;
16      }
17
18      private void OnBallUpdate(bool smoothMovement, double ball_x, double ball_y,
            double ball_z) {
19          if (smoothMovement) {
20              Vector3 currentPos = this.transform.position;
21              Vector3 newPos = new Vector3((float)ball_x, 2.439f, (float)ball_y);
22              this.transform.position = Vector3.Lerp(currentPos, newPos, speed * Time
                    .deltaTime);
23          } else {
24              this.transform.position = new Vector3((float)ball_x, 2.439f, (float)
                    ball_y);
25          }
26      }
27  }
```

Listing A.7: Handles the movement of the ball (BallController.cs)

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4   using System.Linq;
5   using UnityEngine;
6
7   public class OrbitCameraScript : MonoBehaviour
8   {
9       [Header("Camera and Target")]
10      [SerializeField] private Camera cam;
11      [SerializeField] private Transform target;
12
13      [Header("Zoom")]
14      [SerializeField] private float minZoom = 1.5f;
15      [SerializeField] private float maxZoom = 30.0f;
16      [SerializeField] private float zoomSpeed = 10.0f;
17      [SerializeField] private float currentZoom = 8.0f;
18
19      private Vector3 currentPosition;
20      public GameObject[] camTargets;
21
22      // Start is called before the first frame update
23
24      void Start() {
```

```
25          // Retrieve all possible targets by the "CamTarget" tag
26          camTargets = GameObject.FindGameObjectsWithTag("CamTarget");
27          // Sort this array
28          camTargets = camTargets.OrderBy (p => p.name).ToArray();
29      }
30
31      // Update is called once per frame
32      void Update()
33      {
34          Vector3 direction = new Vector3(0, 0, 0);
35          // Get the current position on the initial click of LMB
36          if (Input.GetMouseButtonDown(0)) {
37              currentPosition = cam.ScreenToViewportPoint(Input.mousePosition);
38          }
39
40          // Calculate the new position on draggin with LMB down
41          if (Input.GetMouseButton(0)) {
42              Vector3 newPosition = cam.ScreenToViewportPoint(Input.mousePosition);
43              direction = currentPosition - newPosition;
44              currentPosition = newPosition;
45          }
46
47          UpdateCamera(target, direction);
48          HandleZoom();
49      }
50
51      /// <summary>Handles the zooming in and out of the camera</summary>
52      private void HandleZoom() {
53          // Scrolling up loads to zooming in
54          if (Input.mouseScrollDelta.y > 0) {
55              currentZoom -= zoomSpeed * Time.deltaTime;
56          }
57          // Scrolling down leads to zooming out
58          if (Input.mouseScrollDelta.y < 0) {
59              currentZoom += zoomSpeed * Time.deltaTime;
60          }
61
62          Mathf.Clamp(currentZoom, minZoom, maxZoom);
63
64      }
65
66      /// <summary>Updates the camera position, rotation and translation</summary>
67      /// <param name="target">The camera's orbit target</param>
68      /// <param name="direction">The direction the camera should move</param>
69      private void UpdateCamera(Transform target, Vector3 direction) {
70          cam.transform.position = target.position;
71
72          cam.transform.Rotate(new Vector3(1, 0, 0), direction.y * 180);
73          cam.transform.Rotate(new Vector3(0, 1, 0), -direction.x * 180, Space.World)
                ;
74          cam.transform.Translate(new Vector3(0, 0, -currentZoom));
75      }
76
77      /// <summary>Sets the camera target to passed parameter</summary>
78      /// <param name="target">The target the camera should orbit</param>
79      public void SetCameraTarget(GameObject target) {
80          this.target = target.transform;
81      }
82
83      /// <summary>Gets all camera orbit targets</summary>
84      /// <returns>Array of GameObjects containing cameras</returns>
85      public GameObject[] GetCameraTargets() {
86          return this.camTargets;
87      }
88  }
```

Listing A.8: Allows the user to control the camera using the mouse (OrbitCameraScript.cs)

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

public class PopulateDropdownScript : MonoBehaviour
{
    public OrbitCameraScript orbitCameraScript;
    public GameObject camController;
    public GameObject[] camTargets;
    public TMPro.TMP_Dropdown targetDropdown;
    // Start is called before the first frame update
    void Start()
    {
        // Get the dropdown component on the current gameobject
        targetDropdown = this.gameObject.GetComponent<TMPro.TMP_Dropdown>();
        // Get the orbit camera script
        orbitCameraScript = camController.GetComponent<OrbitCameraScript>();
        // Get the targets from OrbitCameraScript
        camTargets = orbitCameraScript.GetCameraTargets();
        PopulateDropdown(targetDropdown, camTargets);

        // Add listeners
        targetDropdown.onValueChanged.AddListener(delegate {
            DropdownValueChanged(targetDropdown);
        });
        targetDropdown.onValueChanged.AddListener(delegate {
            DataPanelScript.OnDropdownChange(targetDropdown);
        });
        targetDropdown.onValueChanged.AddListener(delegate {
            HeatmapScript.OnDropdownChange(targetDropdown);
        });

        // Set the target initially to Stadium if possible
        foreach(TMPro.TMP_Dropdown.OptionData option in targetDropdown.options) {
            if (option.text == "Stadium") {
                targetDropdown.value = targetDropdown.options.IndexOf(option);
            }
        }
    }

    private void PopulateDropdown(TMPro.TMP_Dropdown dropdown, GameObject[]
        optionsArray) {
        List<string> options = new List<string>();
        Debug.Log("nr of targets: " + optionsArray.Length);
        foreach (var option in optionsArray) {
            options.Add(option.name);
        }
        dropdown.ClearOptions();
        dropdown.AddOptions(options);
    }

    private void DropdownValueChanged(TMPro.TMP_Dropdown change) {
        orbitCameraScript.SetCameraTarget(camTargets[change.value]);
    }
```

Listing A.9: Dynamically fills the camera target dropdown list with possible targets (PopulateDropdownScript.cs)

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEditor;
using System;
using System.Linq;
```

```
8
9  public class CSVLogger : MonoBehaviour
10 {
11     // This class is used for writing CSV log files of the robot
12     // note that this approach may not be the most efficient, it saves quite a
13     // lot of data in memory without limit, it is more to demonstrate and lay
14     // the foundation for future implementations
15
16     // Folder path for log files
17     private static string directoryPath = "Assets/Logs";
18     // List to keep track of robot data
19     private static List<DataEntry>[] dataList = new List<DataEntry>[6];
20     // Sanity check to make sure data ends up in correct list
21     private bool[] entryDone = new bool[6];
22
23     private void OnEnable() {
24         // Subsribe to the RobotDataUpdateEvent
25         TurtleCommManager.robotDataUpdateEvent += OnRobotDataUpdate;
26         TurtleCommManager.robotPosUpdateEvent += OnRobotPosUpdate;
27     }
28
29     private void OnDisable() {
30         // Unsubsribe to the RobotDataUpdateEvent
31         TurtleCommManager.robotDataUpdateEvent -= OnRobotDataUpdate;
32         TurtleCommManager.robotPosUpdateEvent -= OnRobotPosUpdate;
33     }
34
35     private void Start() {
36         // First check if logging folder exists
37         if (!Directory.Exists(directoryPath)) {
38             // If not, then create it
39             Directory.CreateDirectory(directoryPath);
40         }
41         // Initialize first list entry
42         for (int i = 0; i < dataList.Length; i++) {
43             dataList[i] = new List<DataEntry>();
44             dataList[i].Add(new DataEntry());
45         }
46     }
47
48     [MenuItem("Tools/Write CSV files")]
49     private static void WriteCSVFiles() {
50         Debug.Log("Start writing out CSV log files");
51         // Write out files for all robots
52         for (int i = 0; i < dataList.Length; i++) {
53             string filePath = directoryPath + "/robot" + i + "_logs_" + DateTime.
                   Now.ToString("yyyyMMdd_hhmmss") + ".csv";
54
55             Debug.Log("Writing CSV log file for robot " + i + ": " + filePath);
56             StreamWriter writer = new StreamWriter(filePath);
57             writer.WriteLine("time,ID,x,y,rotation,team,battery,motor1_temp,
                   motor2_temp,motor3_temp");
58             foreach (DataEntry entry in dataList[i]) {
59                 writer.WriteLine(entry.GetEntryCSV());
60             }
61             writer.Flush();
62             writer.Close();
63         }
64         Debug.Log("Writing out CSV log files finished");
65     }
66
67     private void OnRobotDataUpdate(int turtleID, IntPtr teamColor, IntPtr
           batteryVoltage, IntPtr emergencyStatus, double motorTemperature1, double
           motorTemperature2, double motorTemperature3) {
68         string timeStamp = Time.timeSinceLevelLoad.ToString("0.00");
69         dataList[turtleID].ElementAt(dataList[turtleID].Count - 1).SetRobotData(
               timeStamp, turtleID, teamColor, batteryVoltage, motorTemperature1,
```

```
            motorTemperature2, motorTemperature3);
70          if (entryDone[turtleID]) {
71              dataList[turtleID].Add(new DataEntry());
72              entryDone[turtleID] = false;
73          } else {
74              entryDone[turtleID] = true;
75          }
76      }
77
78      private void OnRobotPosUpdate(int turtleID, bool smoothMovement, double robot_x
            , double robot_y, double robot_phi) {
79          string timeStamp = Time.timeSinceLevelLoad.ToString("0.00");
80          dataList[turtleID].ElementAt(dataList[turtleID].Count - 1).SetRobotPos(
                timeStamp, turtleID, robot_x, robot_y, robot_phi);
81          if (entryDone[turtleID]) {
82              dataList[turtleID].Add(new DataEntry());
83              entryDone[turtleID] = false;
84          } else {
85              entryDone[turtleID] = true;
86          }
87      }
88
89      private class DataEntry {
90          private string timeStamp { get; set; }
91          private int turtleID { get; set; }
92          private string team { get; set; }
93          private IntPtr batteryVoltage { get; set; }
94          private double motorTemperature1 { get; set; }
95          private double motorTemperature2 { get; set; }
96          private double motorTemperature3 { get; set; }
97          private double robot_x { get; set; }
98          private double robot_y { get; set; }
99          private double robot_phi { get; set; }
100
101         public void SetRobotData(string timeStamp, int turtleID, IntPtr teamColor,
                IntPtr batteryVoltage, double motorTemperature1, double
                motorTemperature2, double motorTemperature3) {
102             this.timeStamp = timeStamp;
103             this.turtleID = turtleID;
104             if (teamColor == (IntPtr)0) {
105                     this.team = "Magenta";
106                 } else if (teamColor == (IntPtr)1) {
107                     this.team = "Cyan";
108                 } else {
109                     this.team = " ";
110                 }
111             this.batteryVoltage = batteryVoltage;
112             this.motorTemperature1 = motorTemperature1;
113             this.motorTemperature2 = motorTemperature2;
114             this.motorTemperature3 = motorTemperature3;
115         }
116
117         public void SetRobotPos(string timeStamp, int turtleID, double robot_x,
                double robot_y, double robot_phi) {
118             this.timeStamp = timeStamp;
119             this.turtleID = turtleID;
120             this.robot_x = robot_x;
121             this.robot_y = robot_y;
122             this.robot_phi = robot_phi;
123         }
124
125         public string GetEntryCSV() {
126             return this.timeStamp + "," + this.turtleID + "," + this.robot_x + ","
                    + this.robot_y + "," + this.robot_phi + "," + this.team + "," +
                    this.batteryVoltage + "," + this.motorTemperature1 + "," + this.
                    motorTemperature2 + "," + this.motorTemperature3;
127         }
```

```
128         }
129 }
```

Listing A.10: Records the robot data and allows the user to export the results to CSV files (CSVLogger.cs)