Department of Informatics
Technical University of Munich

Bachelor's Thesis in Informatics: Games Engineering

Application of Game Engines for Environment Randomization in Simulation-trained Computer Vision Tasks

Anwendungen von Game Engines für die Randomisierung von Umgebungen in simulationsgeschulten Computer Vision Aufgaben

Supervisor Prof. Dr.-Ing. habil. Alois C. Knoll

Advisor Josip Josifovski, M.Sc.

Author Florian Panzer

Date October 14, 2020 in Garching

Disclaimer

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Garching, October 14, 2020

(Florian Panzer)

Acknowledgments

I would like to thank my supervisors Professor Alois C. Knoll for the thorough supervision and my advisor Josip Josifovski for his advices during the whole period of the thesis and his help in performing and evaluating the experiment.

Abstract

Transferring deep-learning models trained in simulation to a real-world application is a challenging task due to the mismatch between the two domains, known as the reality gap problem. To address this problem for object recognition tasks, in this thesis we employ the benefits provided by game engines to develop a generic tool for domain randomization. The tool allows the user to generate a large number of synthetic training images by randomizing the objects and their properties in the simulated environment. During the development process, special care is taken to enable easy integration and extension of the tool when designing computer vision experiments.
To evaluate the usefulness of the tool for computer vision tasks, we perform an experiment and measure the accuracy of a state-of-the-art model for object detection. The same model is trained with two different training sets: the first training set contains only real images, while the second is a mixture of real images and synthetic images generated by the developed tool. The results show that the mixed dataset significantly improves the performance of the model on the real test images.

Zusammenfassung

Die Übertragung von simulationsgeschulten deep-learning Modellen auf eine Anwendung in der realen Welt ist aufgrund der Diskrepanz zwischen den beiden Bereichen eine schwierige Aufgabe, bekannt als das Reality Gap Problem . Um dieses Problem für Objekterkennungsaufgaben zu lösen, nutzen wir in dieser Arbeit die Vorteile von Game-Engines, um ein generisches Werkzeug für die Domain-Randomization zu entwickeln. Das Tool ermöglicht es dem Benutzer, durch Randomisierung der Objekte und ihrer Eigenschaften in der simulierten Umgebung eine große Menge an synthetischen Trainingsbildern zu erzeugen. Während des Entwicklungsprozesses wird besonders darauf geachtet, eine einfache Integration und Erweiterbarkeit des Werkzeugs zu ermöglichen, insbesondere bei der Gestaltung von Computer Vision Experimenten.
Um die Nützlichkeit des Tools für Computer-Vision-Aufgaben zu evaluieren, führen wir ein Experiment durch und messen die Genauigkeit eines modernen Modells zur Objekterkennung. Dasselbe Modell wird mit zwei verschiedenen Trainingssätzen trainiert: Der erste Trainingssatz enthält nur reale Bilder, während der zweite eine Mischung aus realen Bildern und synthetischen Bildern enthält, die durch das entwickelte Werkzeug erzeugt werden. Die Ergebnisse zeigen, dass der gemischte Datensatz die Leistung des Modells auf realen Testbildern deutlich verbessert.

Contents

Acknowledgments **v**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives . . . 1
1.3 Overview . . . 2

2 Background and Related Work **3**
2.1 Background . . . 3
2.1.1 Game Engine . . . 3
2.1.2 Simulation in Industry and Research . . . 4
2.1.3 Reality Gap and Domain Randomization . . . 4
2.2 Related Work . . . 5

3 Implementation **7**
3.1 Overall structure . . . 7
3.1.1 Unity's Software Architecture . . . 7
3.1.2 Model-View-Controller Architecture . . . 8
3.1.3 Interface, Inheritance and Factory Pattern . . . 10
3.1.4 ApplicationManager . . . 10
3.1.5 Physical Based Rendering . . . 12
3.2 MenuScene . . . 13
3.3 EditorScene . . . 14
3.3.1 Item Presentation and Placement . . . 15
3.3.2 SceneObject Selection . . . 20
3.3.3 Changing Parameter . . . 23
3.3.4 Randomization . . . 27
3.3.5 Additional Features . . . 29
3.4 Image Generation . . . 30

4 Experimental Evaluation **33**
4.1 Experiment Setup . . . 33
4.2 Results . . . 35

5 Conclusion **39**

Bibliography **45**

1 Introduction

1.1 Motivation

It was never so easy as today to develop a video game. With game engines like Unity[1], Unreal Engine[2], or Cry Engine[3], that support the development process and provide a wide range of functionality everyone from absolute beginners to large developer teams can design and create their own games.
However, game engines are no longer exclusively used to design games or mobile applications. Through their integrated and extensible physics engines and high definition rendering pipelines, they have became increasingly popular for scientific, robotic, or even industrial tasks. Unity has recently been used to present the new BMW 8er Coupé through an interactive, high-resolution demo [DL19]. It has also been used to create an external system that monitors the manufacturing of one-of-a-kind products without interfering the actual production process [Sit+17].
Game engines provide a good basis for the simulation of robotic tasks in industry. The integrated editor and the option to easily import 3D models enable the possibility to quickly create and design environments as well as small, functional prototypes. In addition, the created environments and their properties can be changed and adjusted very rapidly, which allows multiple runs in a short time without much effort. Moreover, it reduces the general costs, since not everything has to be tested in the real world, as most of the tasks can be simulated in a virtual environment. This makes it easy to run several agents simultaneously, without the need for multiple, expansive robots.
However, it is not simple to transfer the trained model from a simulation into the real world [Zha+15]. Even with the given functionality of a game engine, there are several common problems, like imprecise physics simulations or inaccurate defined models, that cannot be solved easily. These differences between the reality and the simulated world are known as the reality gap problem.
To counteract the reality gap problem, we aim to create an application that tries to reduce its consequences through domain randomization, a technique of randomizing the parameters and properties of the simulation environment.

1.2 Objectives

The goal of this thesis is the development of a software tool (application) in the field of domain randomization that can be used to improve the training procedure for industry-related

[1] Unity https://unity.com/de (Accessed 04.03.2020)
[2] Unreal Engine https://www.unrealengine.com/en-US/ (Accessed 04.03.2020)
[3] CryEngie https://www.cryengine.com/ (Accessed 04.03.2020)

robotic tasks in simulation. This tool should support the users to generate a set of randomized images, providing them with the possibility to adjust the randomization parameters, and allowing them to use the generated data to reduce the reality gap of simulation-trained models.
To achieve the goal, we aim to develop an application that is understandable and easy to use with an intuitive user interface (UI) and configurable parameters.The application should also be extendable and allow an easy implementation of new object categories and randomization parameters by using best practices from software development and architecture.
In addition, we want to use the data generated with this tool to augment the training data set for a computer vision task and to determine whether the additional data leads to improvements of the results.

1.3 Overview

In Chapter 2 related work will be described such as the UnrealCV project or experiments that show the relation between the realism degree of the provided data and the accuracy of the AI agent. Background information about Unity and game engines as well as domain randomization and the reality gap will also be provided.
The following chapter describes the implementation details of the application, the software architecture used, and the resulting advantages. Moreover, it includes a detailed description of the communication flow and relationship between the different modules.
Chapter 4 presents the experimental setup and evaluation, the findings, and a short discussion.
In the last chapter, a conclusion about the work done and an outlook on possible future work are given.

2 Background and Related Work

2.1 Background

The following chapter describes the early beginnings of gaming, which is described in [Wik], the development of the first game engines up to Unity, in particular. Subsequently, applications of the simulations in industry and research are shown. Then the advantages, but also the problems of transferring synthetic data into the real world are discussed. Furthermore, a method is presented which is used in this thesis to counteract these problems. The last section is about related work, which has already dealt with sim2real transfer. In addition, it describes also existing tools, which simplify both, the creation of simulations and the training of AI model or try to solve the above-mentioned problems.

2.1.1 Game Engine

The first interactive video game, Tennis for Two that could be played by two players was developed in 1958 by William Higinbotham[1]. This game was a two-dimensional tennis simulation, which ran on an analog computer and was played on an oscilloscope.
In 1962 Spacewar! was developed at the Massachusetts Institute of Technology (MIT) and is considered to be the first digital video game. In contrast to Tennis for Two, Spacewar! ran on a digital computer and could be transferred onto other computers. The goal of the game was to destroy the spaceship of the opponent with projectiles that are affected by a gravity well of a star in the middle of the field.
But these games were not made for commercial use since they needed many resources and expensive machines that were only affordable for universities. This changed, when in 1972 the Magnavox Odyssey was released, the first commercial home console. Based on a version of table tennis, included in the console, the game Pong was developed by Atari. Pong was the first commercially successful video game. It established the foundation of the game industry and paved the way for many more video games.
The complexity of the games grew and soon games like Space Invader, Pac Man, Mario Bros. and Tetris became very popular and usher in the golden era of arcade games. With the appearance of home computers in the late 1970s and early 1980 owners could now develop their own video games and share them with others. Unfortunately, this led to a video game crash in 1983 because the market was flooded with poor-quality games. The game industry recovered a few years later when Nintendo released the NES (Nintendo Entertainment System) and the game Super Mario Bros. In the early 1990 games like Sonic The Hedgehog or Super Mario Kart were released and increased the success of the new consoles SNES (Super Nintendo Entertainment System) and Sega Mega Drive. But till now developers could only use simple graphics and had to hard-code every part of the game almost entirely from scratch since only very little could be reused from other games.

[1] TennisForTwo https://en.wikipedia.org/wiki/Tennis_for_Two (Accessed 20.09.2020)

This changed with the appearance of game engines. They provide a platform for developers, where they can build their game from game assets that can be reused in every other game that is made in this engine. Thereby the game engine takes over many core functionalities like physics and collision detection, rendering the game or memory management. This makes it possible for almost everyone to develop their games even without being familiar with such topics.
In 1993 id Software released the Doom Engine, one of the first engines that was able to successfully create the visual illusion of 3D levels, even if 2D sprites were used instead of 3D Models [Van; LJ02; PGB12]. Three years later, in 1996, also Id Software published the game Quake,and the associated Quake Engine[2] which was a big milestone in the development of video games. It was the first real 3D engine that actually used real 3D information, models, and texture-mapped polygons. Additional new features were dynamic light sources and the opportunity to change the player's view with a combination of mouse and keyboard input. The level editor, that was included in the Unreal Engine was unique in 1998, where Epic Games released their engine, and allowed the user to modify levels on the fly. A considerable step in real-time graphics was made with the development of Crysis[3] and the new render engine in CryEngine in 2007.
The first Version of Unity (1.0) was released in 2006 [Dav]. Since then, the engine significantly improved in terms of physics simulation and its rendering capabilities, real-time and baked lighting, high definition rendering, or particle systems, just to name a few examples. Today Unity is internationally the most used game engine [Sha; Kol]. One reason is the large number of supported platforms. There are over 20 platforms, like PlayStation 4, iOS, Windows, or Nintendo Switch[4], for which games or applications can be created. Also, there are a lot of tutorials, guides, and support from the large community as well as from the developers themselves. Moreover, Unity has a user-friendly interface and offers some assets that add a visual scripting feature to the editor that even user without prior experience in programming can create their own game.

2.1.2 Simulation in Industry and Research

However, virtual worlds are nowadays not only used to create video games. Their usage has almost no limitations and therefore they are also used in fields like industry or research. Simulations can help to simplify a system and get results on an abstract level. Furthermore, simulations are also used to represent whole systems and observe whether an improvement of the process can be archived by changing certain conditions. Also, it can reduce costs, since it can simulate procedure that would typically need a lot of materials, like crash tests or safety engineering. They allow further the simulation of real world scenarios that would be too dangerous or unacceptable to engage, like experiments with autonomous cars, which might be too risky to test them on real streets, can be done under safe conditions.

2.1.3 Reality Gap and Domain Randomization

In addition, simulations can also be used to train an artificial intelligence. However, to train a data-driven AI, a enormous amount of training data is needed to get sufficient precision. To

[2]Quake Engine https://de.wikipedia.org/wiki/Quake-Engine (Accessed 09.03.2020)
[3]Crysis https://en.wikipedia.org/wiki/Crysis_(video_game) (Accessed 29-09-2020)
[4]Overview Unity https://unity3d.com/de/public-relations (Accessed 09.03.2020)

gather such a large amount of data in real-world training needs a lot of resources like money and time and is therefore very impractical [Lev+18]. The gathering of data from real-world training becomes even more complicated when the environments are dynamic or plain and simple, not feasible. There are sources that provide large sets of real image training data like ImageNet[5], but especially when it comes to AI in industry, the needed data becomes very specific. The process of labeling the set of million of needed images yourself would require a excessive amount of time to complete this in a reasonable pace.
To avoid such time-consuming tasks, simulation became a cheap solution to generate, in theory, an unlimited amount of data. It is possible to run many thousand agents at once and at a higher speed to get much faster results. Also, it allows the creation of millions of synthetic images in shortest time that are correctly labeled and usable as training data with almost no human interaction.
As game engines become more and more extensive, they are well suited to create such simulations. Depending on the type of simulation, the researcher can use the features a game engine already provides, like rendering, easy import of 3D models, or a physics and fluid engine to create a suitable environment for the given task.
However, the generated data have to be transferred into the real world to be used in the designated environment. Yet, many aspects make it difficult to transfer gathered data from a simulated world into the real one. This mismatch between the simulation and the real world and the occurring difficulties to apply simulation-trained models in a real-world scenario is known in the literature as the reality gap problem. This includes aspects like the calculation of the light ambiance, the correct application of the physical laws, or properties like friction or mass.
Many attempts try to avoid or minimize the effects of the reality gap. A possible solution we try in this thesis is domain randomization. The objective of this approach is to train a model with a multitude of variations of the simulation environment and make it robust against different scenarios in the real world. Thereby several parameter and properties of the environment are randomized like the lightning, position, or textures of objects.

2.2 Related Work

It has already been shown that it is unnecessary to use real images as training data to train a deep neuronal network in simple tasks like grasping [Tob+17]. The randomized rendering provides a wide variation within the simulated images. This enables robots to perform well even in the real-world as it perceives the real images only as a further abstraction. Also, automatically generated synthetic images can be used for pose estimation of a 3D model [Jos+18]. This training data is sufficient to perform good performance even if the trained CNN is applied in a real-world setting.
There are now many tools and projects that support the user in tasks like training agents, generating large amount of data, or creating an environment. One of these tools is the OpenAi Gym [Bro+16], a platform to train and compare different types of algorithms and agents. Users can provide their self-made environment or choose one of the given ones like old Atari games or one based on the Doom game engine (VizDoom) [Kem+16], train their agents, and upload and compare their results and performance with other users. The toolkit deliberately does not offer any agents since it should not limit the users and enable them to implement different styles of agent interfaces.

[5] imageNet http://imagenet.stanford.edu/index (Accessed 20-07-2020)

Several scientific projects already use a game engine to develop such tools. The open-source project UnrealCV [Qiu+17] is created for Unreal Engine 4.0, another popular game engine, which is often used for modern game development [08]. UnrealCV is a plugin, which allows to transform content in Unreal into a virtual world and create external access to this virtual world. The access is created via a server-client architecture and makes it possible to receive some detailed information about the single elements and change their properties like position, rotation, or light intensity within the world. The server is automatically integrated during the compilation of the program, and the client can be integrated into external programs. The client can then be used to establish communication between the virtual world and an AI agent.
Another open-source project that shows that Unity becomes increasingly relevant when it comes to AI-Technology is the ML-Agents Toolkit[Uni]. This project is developed by Unity and made the easy usage of different machine learning methods possible. Its easy setup allows hobbyists as well as game developers, who are not familiar with AI, to use the features such as creating and manipulating the learning environments or training and applying some of the included algorithms. Moreover, it provides the possibility to use their algorithms or even turn the environment into a gym. Also, the opportunity to enable communication through a Python-API with an external algorithm makes the tool kit attractive for research or industry[Jul+18].
In parallel to the work in this thesis, Unity released their own Perception Package with tools to generate synthetic data sets for training a machine learning model. This tools creates a scene with predefined "foreground, background, and occluding objects along with a randomization of lighting, object hue, blur and noise" [AT20].

3 Implementation

This chapter will describe the software architecture, the different modules as well as the design choices that were made during the implementation of the tool. The first section presents the overall used structure and concepts of the application. Afterwards, the menu scene and the editor scene will be described in more detail. In the last part, the generation of the images will be explained.

3.1 Overall structure

3.1.1 Unity's Software Architecture

Unity uses a component-based software architecture. This means that the functionality is decoupled into compact and self-contained components. The component combines similar or related features. It provide a well-defined interface with methods and properties that enable communication between different components. This makes the individual components context-independent and reusable since they are designed to be used in different environments and situations.
These components can then be applied onto GameObjects, concrete instances of objects in a scene in Unity, and thereby represent its properties as shown in Figure 3.1. For example, each object contains the Transform component which represents the position, rotation and scaling of the game object in the scene. The Light component can turn each simple cube into a light source, the Collider component provides the opportunity to detect collisions between objects and the Rigidbody component applies physical properties like gravity or friction, simply by adding the correlated component to the game objects. Scripts can also be attached to objects so that these objects properties can then be easily referenced and changed in the script itself.

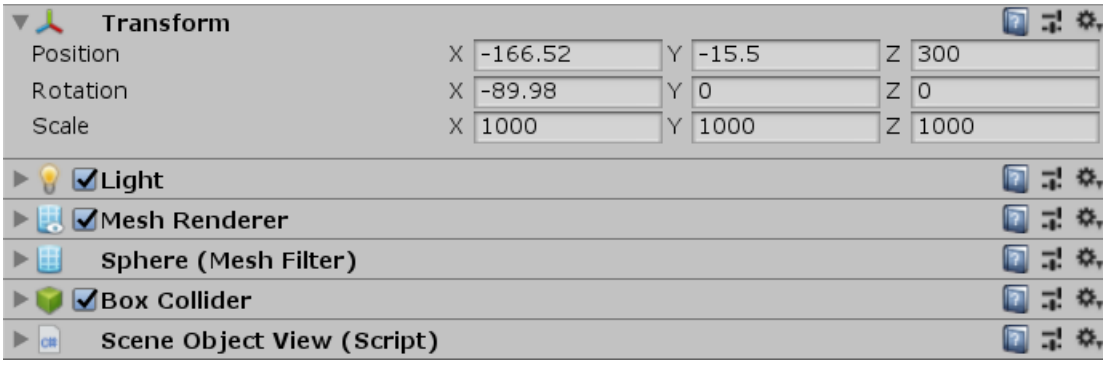

Figure 3.1: A GameObject with different attached components

3.1.2 Model-View-Controller Architecture

On top of the component-based, we chose a model-view-controller (MVC) structure as the main architecture, which was adapted to the component-based structure and is based on [Dun15]. The main feature of the MVC structure is the distribution of the software into three parts and the resulting division of data presentation and processing. As the name suggests, the three parts consist of the model, the controller and the view as shown in Figure 3.2. The tasks of this application are distributed over several different MVC architectures, which are logically connected and can therefore delegate tasks to each other. Model, view and controllers, which have the same task area and therefore form an MVC architecture, are further referred to as a module.

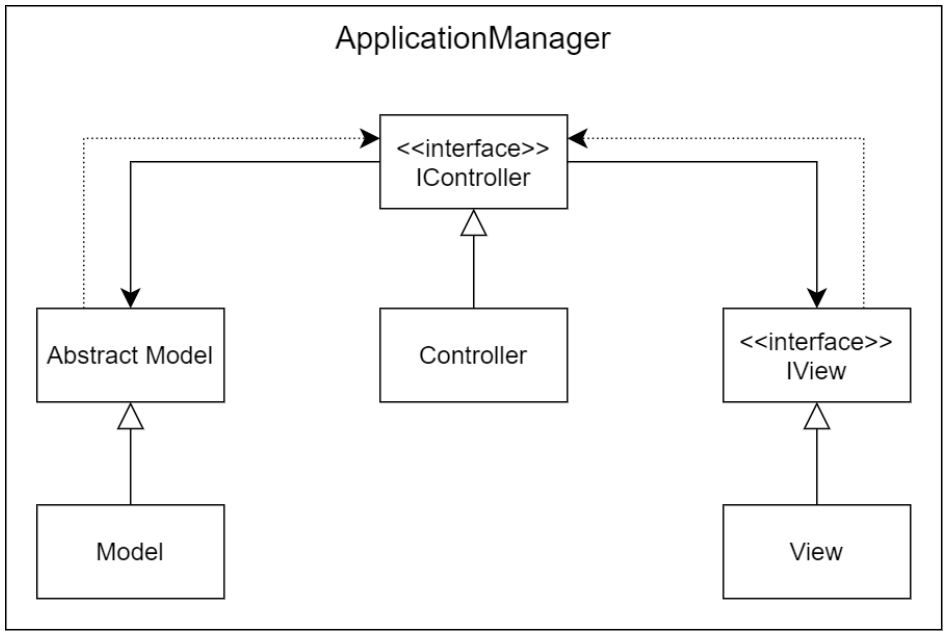

Figure 3.2: The general layout of the MVC structure

In an MVC structure, the model acts as data storage that can store for example Item properties or also other models. Additionally, depending on the type of model, it is responsible for changing or applying different properties onto a given GameObject. All necessary data of the application are stored in a tree-like hierarchy structure with superordinate models. The root model is stored in the ApplicationManager, which is explained in more detail at the end of this section.
The view handles the representation of the given data. It reacts to changes of the model, recognizes user input and forwards it to the controller. The division of responsibilities allows the data of a model to be represented by more than only one view. Due to the already provided architecture of Unity, the views in the application are represented by GameObjects and attached to them via a script component. This way the view can access the connected GameObject and pass it to the controller if necessary.
The data flow between the two components is regulated by the controller. The controller has a direct reference to the data model as well as to the representing views and can therefore communicate directly with them. In contrast, the model and the view have no direct connection to each other or the controller and communicate only via Unity's event system.
This allows both, the model and the view to trigger an event, which can then be perceived and processed by the corresponding event listener of the controller. The event listener must

be explicitly linked to the event in order to be able to intercept it. This procedure is also used to enable communication between two different controllers. Events allow the transmission of data via specially created event arguments, which are attached to the event during its creation. In some cases, it is not necessary to send data to the event listener, because the event should only trigger a certain logic. Therefore, actions are used, which are a simpler form of events because they only represent the call itself without any added arguments.

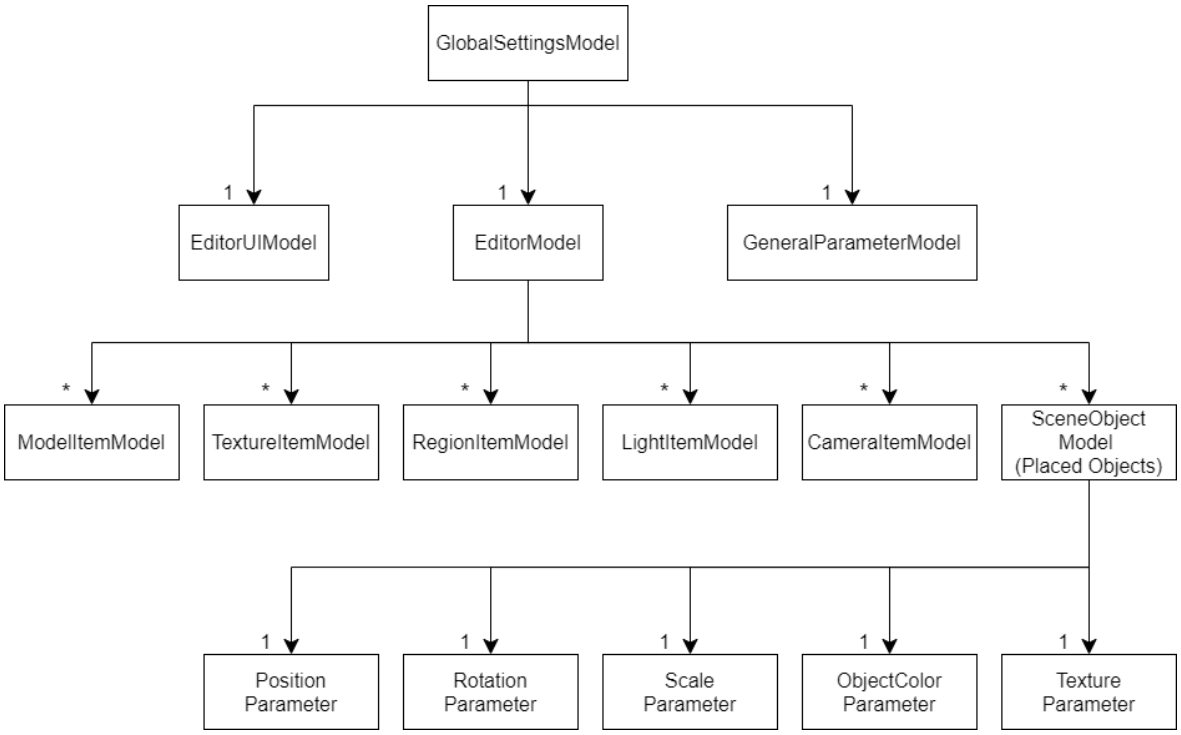

Figure 3.3: The tree-like hierarchy of the models

However, this led to certain difficulties during development while using Unity's event system. Since the event listener is linked with an event, they must also be decoupled from the corresponding events. That was not done properly in the beginning of the development. Otherwise, a permanent reference of the event listener to the event remains. When Unity changes scenes, it destroys all objects in the old scene, so all views are destroyed as well. Since the controllers still serve as event listeners and have a fixed reference to specific events, they are not destroyed when changing the scene, unlike the views. If the original scene is now called up again, these controllers still exist and still react to certain events, which are e.g. triggered by other controllers. If they try to access their destroyed view, this causes to a NullPointerException and the termination of the actual event.
One possible solution would be to reuse the already existing controllers. However, the application can be extended to allow data to be exported and re-imported. Only the data itself would be loaded and the controllers would have to be recreated when the scene is loaded for the first time. In order to keep the code simple, it would be advantageous if the controllers would be destroyed when the scene is changed. This way, a uniform procedure can be used, which does not have to differentiate between imported data and data from the current session.
Another simple solution would be to check the view before it is called. This would not delete the controller itself and would continue to occupy memory. This would result in a memory leak. Depending on the number of objects in the scene and the number of scene changes,

the number of inactive controllers in the memory increases until the memory is full and the program crashes. To avoid this problem, this solution was used.
However, this is not in the sense of events. The intended goal is to first remove all registered events of the controller when the scene is changed, so that the controller can then be deleted by the garbage collector. Nevertheless, this also involves certain difficulties, since, for example, no direct reference to the controller of the GameObjects is stored. This difficulty will be discussed in more detail in the EditorScene section.

3.1.3 Interface, Inheritance and Factory Pattern

Additionally to the MVC structure, interfaces are provided for both the views and the controllers. This makes it possible to provide a uniform interface, which thus demands certain methods from the individual implementations. Thereby a regulated communication between the individual modules is ensured, independent of the concrete implementations of the individual components. Thus, each instance of an interface can individually implement the given structure. Another reason why the use of interfaces was chosen is the associated extensibility. By separating the definition from the concrete implementation, additional views or controllers with specific interfaces can easily be added later. For example, it does not have to be ensured that a newly added view can be processed by a certain controller since the method calls are already defined by the interface.

For the models, the concept of abstract classes and the associated inheritance from object-oriented programming is used. Abstract classes themselves cannot be instantiated and serve as a basis for further subclasses. Abstract methods, similar to interfaces, provide certain functionalities that have to be implemented by the subclasses. This makes it possible for each subclass to implement the function differently and thus represent the respective type. Thereby it is ensured that all subtypes can be called safely at any point where a supertype is expected. This is used, for example, when creating a concrete GameObject or when displaying the Parameters. More details will be provided in the respective sections. In contrast to the interface, the superclass can provide useful functions that can be used by the subtypes. Additionally, the subclasses can add new methods to their class.

In addition to the division of storage and display of data, a division is also used when creating the views. Most of them were created based on the factory-pattern. One factory is used per module. By using this pattern, the creation of the object is done in a separated class, the so-called factory. This separation of usage and creation prevents the code of duplication and makes the view reusable. Besides the good testability of the correct creation and behavior of single views, it also makes the application very extensible. If a new view is needed it can easily be added to the corresponding factory without the need to adapt the calling code or already existing unit tests. To create a view the factory often loads the needed GameObject using prefabs, a feature from Unity that allows the pre-creation and configuration of templates that can be used to generate concrete instances from.

3.1.4 ApplicationManager

Most of the administrative functionalities are handled by the ApplicationManager. It is responsible for importing the assets, stores all data in the GlobalSettingsModel and holds a

reference to the main controllers, the GeneralParameterController, the EditorController and the EditorUIController. To be able to perform its functions, it is hierarchical above the model-view-controller structure and is maintained even when the scenes are changed. Furthermore, the ApplicationManager is a singleton, which means that it is ensured that only one instance of the object exists, and no more instances can be instantiated. This allows getting easy access to the generally used Parameter like the resolution or the path of the target folder, which are used during the generation of the images.
The ApplicationManager handles the import of the different types of objects, which must be placed in a predefined folder to be found by the application. The imported objects are converted into the model of an Item, which serve as blueprints of the imported objects and represent their properties. For each category of objects, one list exists that contains all unique Item models. These lists are stored in the model of the Editor module. Based on these, SceneObjects are created later on that represent a concrete instance of an Item in the scene with concrete and changeable properties. Each property and its values are represented by a Parameter entity.

The ApplicationManager can import 3D models, which can then be loaded into ModelItems or RegionItems. ModelItems represent the objects in the scene that are getting randomized. RegionItems also represent objects but are not taken into account during the randomization process. To import these Items, the asset ObjectLoader [Dum] is used. This allows to import and create GameObjects during run-time from provided.obj-files and their corresponding material files (.mat). From these GameObjects all necessary properties can be extracted like the name, the mesh data and the materials and saved as an Item.
Another Item that can be imported is a texture. Unity supports different shader and workflows and therefore needs different texture maps. For this thesis, the albedo and the normal map can be provided and imported. The imported textures are used in a Standard(Specular) shader, which uses a specular and a gloss map in addition to the normal and color map. The effects of each map are listed in the Physical Based Rendering section. The name and format of the texture are crucial during the import. Only files in PNG or JPG format are supported. Besides, the maps that belong together must have the same name, apart from the extension, in order to be correctly assigned later.
At the beginning of the import, the file is loaded and converted into a Texture2D, one of the formats Unity provides for processing textures. Afterwards, the name of the map is filtered out and checked if a TextureItem with this name already exists. If no model of this Item exists, a new one is created and stored in the list. Otherwise, the already stored one will be used and updated. The imported texture is then added to the stored material of the model either as normal or as color map according to its word ending. The ones for the color map are "_diff", "_diffuse", "_albedo", "_color" as well as all of them with a big initial letter. To be processed as normal map, the imported file has to end with "_normal", "_nor", "_Normal" or "_Nor".
The last Items that are imported are lights and cameras. Since Unity only provides one camera component and a small number of different light types, they are represented through several, predefined models. These models are hardcoded and are therefore always loaded with fixed default values. These are then stored in the list of the respective Items. The properties of the individual Items can then be changed later on after the Items have been placed in the scene.

In addition, the ApplicationManager is also responsible for switching between the scenes. When a scene is loaded for the first time, the main controllers of the corresponding scene are instantiated and a direct reference to them is stored. In the case of the MenuScene, the

GeneralParameterController is loaded and in case of the EditorScene the EditorUIController and the EditorController. These are each responsible for the higher-level tasks in the scene.
As already mentioned, the view of the controller will be destroyed when switching scenes. For this reason, the ApplicationManager ensures that the view that is no longer needed is properly separated from the controller and that the new views are correctly created and attached to them. The SceneManager, which is provided by the Unity Engine, is used for this purpose. It triggers an event to which the ApplicationManager can be appended as EventListener and thus is notified about every change of the scene.
Another main feature of the Application Manager is the generation of the images. This process will be described in more detail in the Image Generation section.

3.1.5 Physical Based Rendering

Unity uses Physically Based Rendering (pbr) for their materials to create realistic surfaces. For that, different textures are used to calculate the different lighting conditions and how they interact with each surface. The two standard workflows used by Unity are the Metallic/Roughness and the Specular/Glossiness. These require different kinds of texture maps to calculate the base color as well as the properties of diffuse and specular reflections. The basic ones are shortly described afterwards as well as in [McD18].
The albedo map or also called diffuse map represents the color information of the texture. It is stored without any external influence like reflections, light or shadow so that the true color of the surface is displayed.
The normal map is used to let the material appear more detailed (see Figure 3.4). With the normal map, also the details of a high-resolution model can be projected onto a low-resolution model. The surface is not changed, but the values of the normal map are included in the calculation of the light that bounces of the surface. This way it creates shadow and bright spots as well as the illusion of small heights and depths like scratches in a metal plate or the fine structure in a wood plank.
The specular map is used to represent the reflectance values for metal and non-metal objects. It controls the strength and color of specular reflections.
The gloss map is in grey-scale and in Unity, its values are stored in the alpha channel of the specular map. It describes how the light is scattered on the surface irregularities and changes the direction of the light. Very glossy points will have bright, small light highlights while very rough points have larger and dimmer highlights.
The metallic map tells the shader, which areas of the material appears as raw metal. The shader can then use this information to interpret the colors stored in the diffuse map. The map is in grey-scale although it is often binary since objects are either metal (white values) or not-metal (black values). If an area appears to be metal the shader takes the color map into account to get the corresponding reflectance values. In contrast to that, specular reflections of non-metallic areas have the same color as the incoming light.
The roughness map is related to the gloss map since it stores the same information than the gloss map but in inverted colors. Comparable to the gloss map it can be stored in the alpha channel of the metallic map.

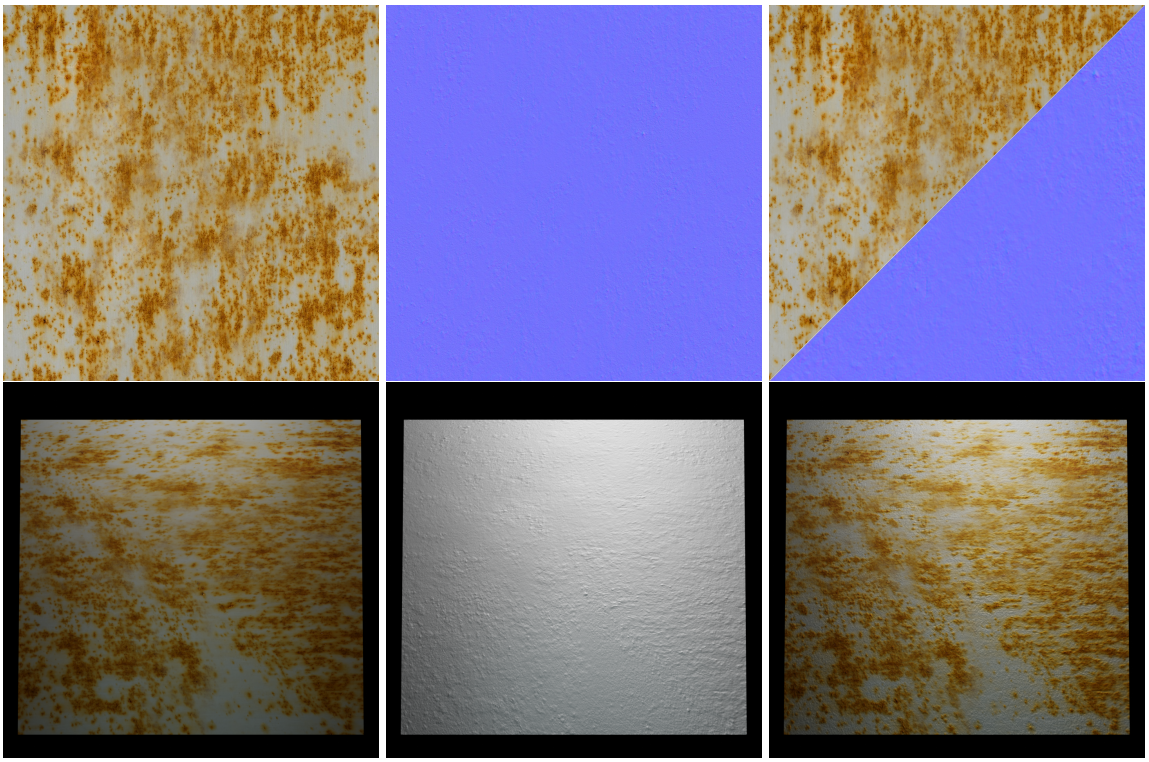

Figure 3.4: The same texture with different degree of details
Top: Used texture maps ltr: Diffuse, Normal, Diffuse & Normal. Bottom: The resulting surface

3.2 MenuScene

The MenuScene is the main screen that appears when the application is started and therefor the first screen the user will see (see Figure 3.5). Before the scene is visualized, the already mentioned import of models and textures will be done. After the completed import, the user can start and type in general Parameters. The user can define the number of images each camera should generate, the resolution of the images as well as a target folder where the images are stored afterwards. The user can also choose which type of images should be created. The different types will be described in the Image Generation section. Another Parameter that can be checked, affects the creation of the bounding box information of the objects during the generation process. Its influence will also be described in the just mentioned section. The last Parameter that can be provided is a randomization seed. This seed is useful when the user wants to reconstruct a set of images in the same way as before as long as all object are arranged as in the old scene. If no randomization seed is provided during the setup, a new one will be created at the beginning of the image generation.

The MenuScene uses only one set of MVC, the GeneralParameter module, to control the behavior in this scene. The only purpose of the GeneralParameterModel is to store all above-named Parameters. The GeneralParameterView handles the input that is inserted by the user. If some input is typed into the Input Fields, the field starts an event and sends the value together with the type of Parameter via event arguments to the GeneralParameterController. The controller stores the new values in the corresponding places in the model. The view also has buttons to pass on the command to switch scenes, start the image generation or quit the application. The target folder can either be typed in or be selected directly from the explorer. To enable this feature the Standalone File Browser for Unity [Ric] is used since the solution

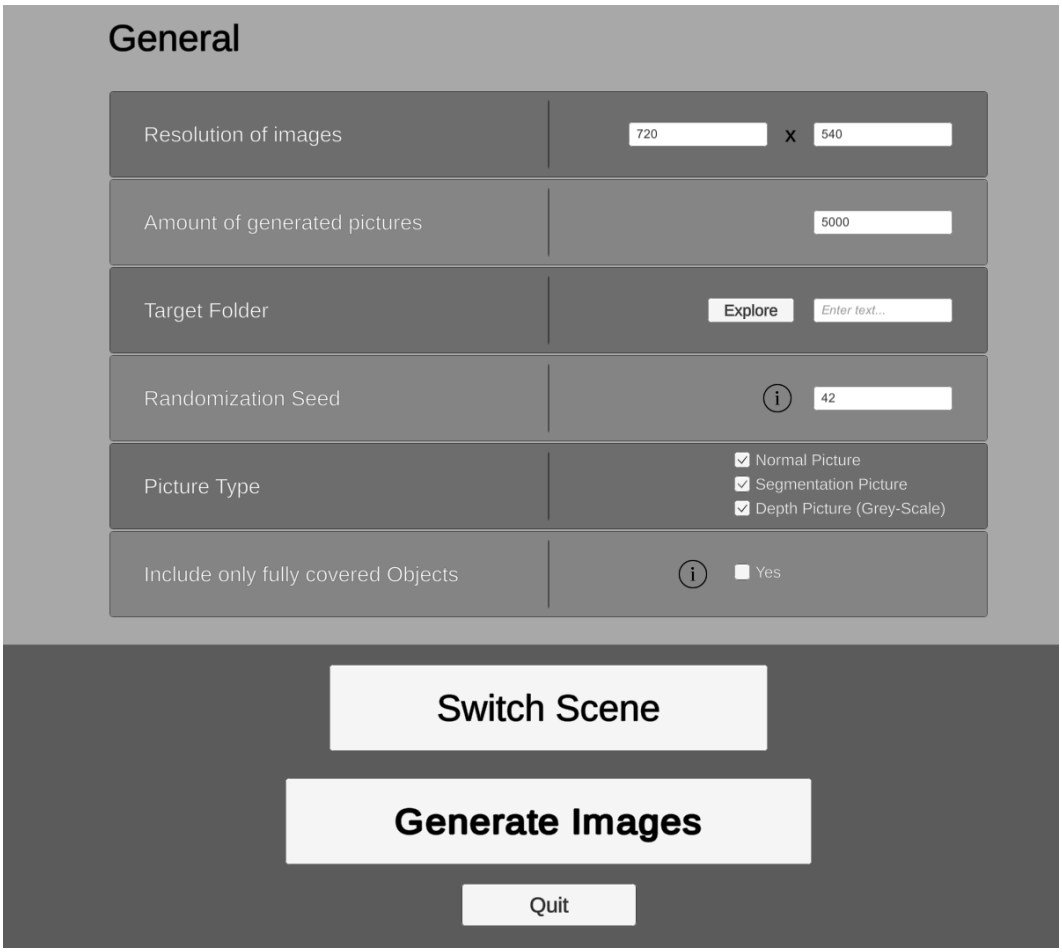

Figure 3.5: MenuScene

Unity provides only works inside the Editor. It cannot be used after the application is compiled and build into a standalone program.

3.3 EditorScene

The second scene the application provides is the EditorScene. It supplies features that allow the user to instantiate an Item and place them in the scene, change their properties and apply textures onto it. Several different modules with different areas of responsibility are used for this.
The user can move around the main camera to facilitate the placement of SceneObjects, to see the entire scene or to get an advantageous overview when there are several overlapping objects. For this purpose, the arrow keys as well as the keyboard keys w, a, s and d, an alternate representation of the arrow keys, are mapped onto Unity input axis. Using these input axis has the advantage that the keys mapped to it can be changed at a central location. So in case of a key mapping change, only the mapping has to be changed. This way, not every usage of it in the code has to be changed. The input of these axes is used to calculate the movement of the camera. In addition, the user can zoom in and out using the scroll wheel on the mouse, what will move the camera forwards or backwards in the current viewing direction. By clicking the middle mouse button and moving the mouse the camera can be

rotated around the actual position. If the Shift key is held down during one of the described actions, the current movement is slowed down, allowing the user to control the camera more precisely.
The scene is represented by two different cameras. One camera only displays objects that are already placed in the scene, while the other camera only renders the UI. The second camera will be rendered after the first one. Thus, the UI camera will overlay the other one and always be in the foreground. Otherwise, placed objects could be seen on top of the UI, if the camera would be in an unfavorable position.

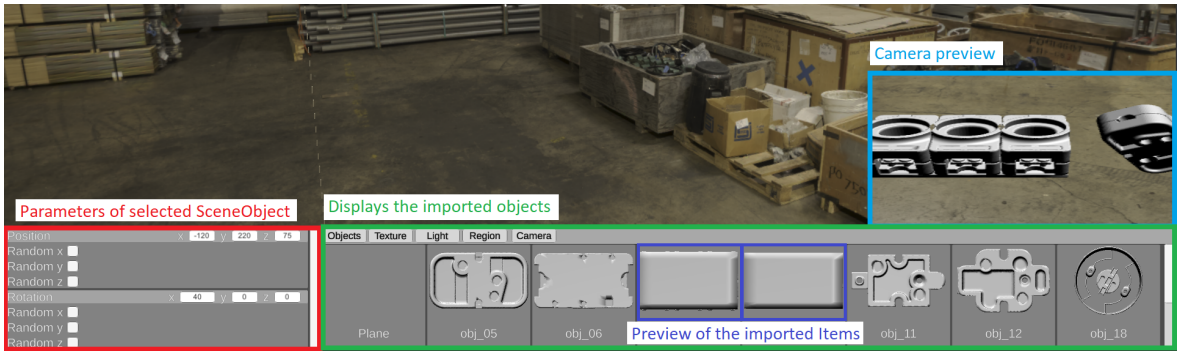

Figure 3.6: UI of the EditorScene

Most of the UI is displayed at the bottom of the screen as seen in Figure 3.6. If a SceneObject is selected in the scene, the respective Parameters of the SceneObject are displayed in the bottom left area. If a placed camera is selected, a preview image of the camera is displayed on the right side of the screen in addition to the Parameters. The preview image should support the user to place the camera at the correct position and thus help to generate the correct images. The rest of the lower screen edge is filled by the display of the imported Items. These are divided into categories, which are represented by different tabs, such as light, camera or texture. The length of the tabs adapts dynamically to the number of imported Items.
The basic UI functionality is covered from the EditorUI module. It is responsible for switching between the individual tabs. These are permanently connected to the view script via the Unity UI and stored in a list. Then a button component is added to the tabs. This triggers an OnClick event as soon as the user presses the button. In addition, the component offers the advantage of connecting further scripts and the triggering of their methods with the start of the event via simple drag and drop. In this way the tab button can be connected to the script of the EditorUIView easily. The script receives the index of the clicked tab and can activate it while all other tabs are hidden.
Besides, the EditorUI module is used to present the Parameters of the SceneObject. This task is explained in the section SceneObject Selection.

3.3.1 Item Presentation and Placement

Together with the SceneObject module, the Editor module has the most tasks in this scene, which are described in the following section.
The user can only select the Items if they are visualized after the scene started. Therefore the EditorController provides a function to display the different type of Items. This function receives a list of ItemModels and the name of a UI field. The individual Items are then displayed through a DisplayItemView. To produce this view a factory, as well as a DisplayItemController

are created. The controller receives the passed list of Items as his model. Then a view is created using the factory. Within this view, a UI prefab is loaded, which includes a text field as well as an empty field. Afterwards, a preview (see Figure 3.6 or Figure 3.7) of the Item is created. Here the advantage of the already mentioned inheritance comes into play. The types of the passed models are irrelevant as long as they inherit from an abstract ItemModel. Therefore, the controller does not need to know how the respective preview can be generated, as this is managed by the implementation of the model.
The preview of the camera and the lights are represented by pictograms while, the TexturesItems are represented by an image of their imported color map. The preview of the RegionItems is created in the same way as it is done for the ModelItems. For that, a new GameObject is created and the stored data of the 3D model is transferred onto it.
After creating the preview, it is assigned to a specific "UI Model" layer. Firstly, the UI camera is configured to only recognize and display objects that are on the UI layers. Additionally, some previews are 3D models, which means that they are also influenced by light. Without light, the individual elements would only be displayed in black. To make the elements in the UI easily recognizable, they are illuminated by an additional light. In order to prevent the light from affecting the lighting conditions in the environment, the light is adjusted so that only the UI layers are affected by it.
Afterwards, the preview is also adjusted to the size of the empty field, which is provided for it. The images can easily be scaled by a fixed factor to the size of the field. For the generated GameObjects, this factor must first be calculated from the size of the imported 3D model. It is crucial to ensure that not every dimension is scaled individually but the model is scaled with a uniform factor so that the object is not distorted.
The preview is subsequently loaded into the empty field of the instantiated UI prefab. After the adjustment of the name, the UI element is appended as a child under the given UI field. The UI field has a component of the type Grid Layout Group to distribute the displayed Items evenly as well as a custom variant of the Scroll component. The modification triggers an action with each scroll movement and sends it to the individual elements. If the action is registered, each Item checks its position within the UI and is hidden if it has exceeded a certain range. If the Item returns in the specified range, it will be displayed again. This ensures that the displayed previews are not displayed beyond the edge of the UI when scrolling, as they cannot be hidden by the scroll component.
At the end of the function, the created view is passed to the controller. A special feature is that the controller can save more than just one view. Therefore, a controller has control over all Items of its category at once. This process is repeated for each list of stored Items, creating an independent instance of the DisplayItem module for each category.

Now the user can click on one of the Items to create a concrete instance of it and place it in the scene. The created object can be moved around freely with the mouse. With an additional click, the object will then be placed. A right-click cancels the process and deletes the object. While the user presses the button, its background color changes slightly. This serves as visual feedback and indicates the user that his action has been registered.
The creation process is triggered by a click on an Item. The DisplayItemView registers the click via a button component. Afterwards, the view sends an event to the DisplayItemController. Using the provided index of the view the controller can filter the matching ItemModel from its list and append it to the event arguments. After that, these are forwarded to the EditorController. First, the controller creates a new GameObject, which is then passed on to the received ItemModel. Apart from creating the preview, the ItemModel is also responsible for creating the actual SceneObject. Again, the implementation depends on the individual model and not on the controller. The LightItemModel, for example, loads a corresponding prefab of

Figure 3.7: EditorScene. Each SceneObject has a SceneObjectView script attached via a component

the selected light type, while the ModelItemModel adds the components MeshRenderer and MeshFilter to the GameObject since these components are responsible for the correct presentation of the 3D model.
Afterwards, the first part of another module is created, specifically the model of a SceneObject. This task is also performed by the ItemModel. The created SceneObjectModel is kept simple and contains only a list of its assigned Parameters and a reference to the used ItemModel. The reference becomes important when the scene is reloaded. Also, each Parameter forms an module and is therefore independent of the SceneObject and other Parameters. To add a new Parameter to a SceneObject, it only needs to be added in the constructor of the corresponding model class as shown in Figure 3.8. This assumes that the view and the model of the Parameter have been correctly implemented.

```
4 Verweise
public class RegionSceneObjectModel : AbstractSceneObjectModel
{
    1-Verweis
    public RegionSceneObjectModel(RegionItemModel itemModel) : base(itemModel)
    {
        Parameter.Add(new PositionModel());
        Parameter.Add(new RotationModel());
        Parameter.Add(new ScaleModel());
        Parameter.Add(new ObjectColorModel());
        Parameter.Add(new TextureModel());
    }
}
```

Figure 3.8: Complete Implementation of a SceneObjectModel

After the SceneObjectModel is built, the GameObject previously created and modified is used to read its current properties and initialize the various Parameters with them. After-

wards, the associated SceneObjectView is being created. This is a script component, which can be used individually and does not differ for the different SceneObjects. However, it must be ensured that each object contains only one such component. During the initial stages of the development process, this causes some errors. For example, the loaded light prefab already had such a component and therefore another one was added during the creation process. As a result, the wrong view was later linked to the controller via events. In consequence, another view, which was therefore not connected to the controller, triggers the events and changes made for instance to the Parameters were not applied. This error has been fixed by checking whether a component of the type SceneObjectView is already existing before adding a new one onto the GameObject.
Now both the model and the view are passed to a new SceneObjectController. The new controller links the events of the view and the Parameters, stored in the model, with corresponding functions within the controller. The events of the controller, in turn, are linked to the EditorController. The SceneObjectModel, which now contains the concrete properties of the GameObject, is then stored in the list of placed objects. This list is needed to reload and display the already placed SceneObjects when the scene is reloaded.
In the end, the created SceneObject is selected. In addition, the current state of the scene is changed, which is represented and saved by an Enum in the EditorView. The state influences the process flow significantly and can vary between the following states: Idle, ItemPlacement, TexturePlacement, ItemSelection, SelectionMovement. At this point, the state changes from Idle to ItemPlacement. This avoids the creation of an additional SceneObject when the user clicks again on one of the Items at the lower screen edge.
The differentiation between the several states takes place within the EditorView. A reference to the current selection, which is now the created SceneObject, is also stored in the EditorView. Due to the new state, this selection can now be moved around by using the mouse. Unless the mouse is over a UI element, the screen coordinates of the current mouse position are readout. Afterwards, a raycast is sent out in a vertical line to the camera starting from the received coordinates. If the raycast hits an object in the scene, the collision point is calculated and the current selection is set to this position. In the case, that the raycast does not hit any object, the selected object is placed in front of the camera at a predefined distance. The position is updated every frame, allowing the object to be moved smoothly within the scene.

The EditorView is also capable of detecting further user input. If the EditorView detects a right-click during the placement phase, the current object is deleted. Therefore the model is first removed from the list of current SceneObjects. Since the saved selection only references the GameObject in the scene, there is no direct reference to the model connected. For this reason, an event is triggered in the SceneObjectView, which is received by the respective controller. The controller then sends the model to the EditorController, whereupon the controller deletes the SceneObjectModel from the list. Then, the deletion process is continued and another event is triggered by the view. By doing so, the SceneObjectController separates all its EventListeners from the respective events. By carefully separating all references, the controller's reserved memory can be freed by the garbage collector. This procedure prevents the problems already mentioned in the Model-View-Controller Architecture section. Both, the associated view and model are being deleted at the end.

The second option available to the user after the Item has been selected is to click left again to place the object at the desired position. Here as well, an event is triggered by the EditorView to trigger the corresponding functionality in the EditorController. The so-called tags are a classification system from Unity that allows objects to be divided into groups. In contrast to layers, tags are not used by Unity in any functional way. They can only be used

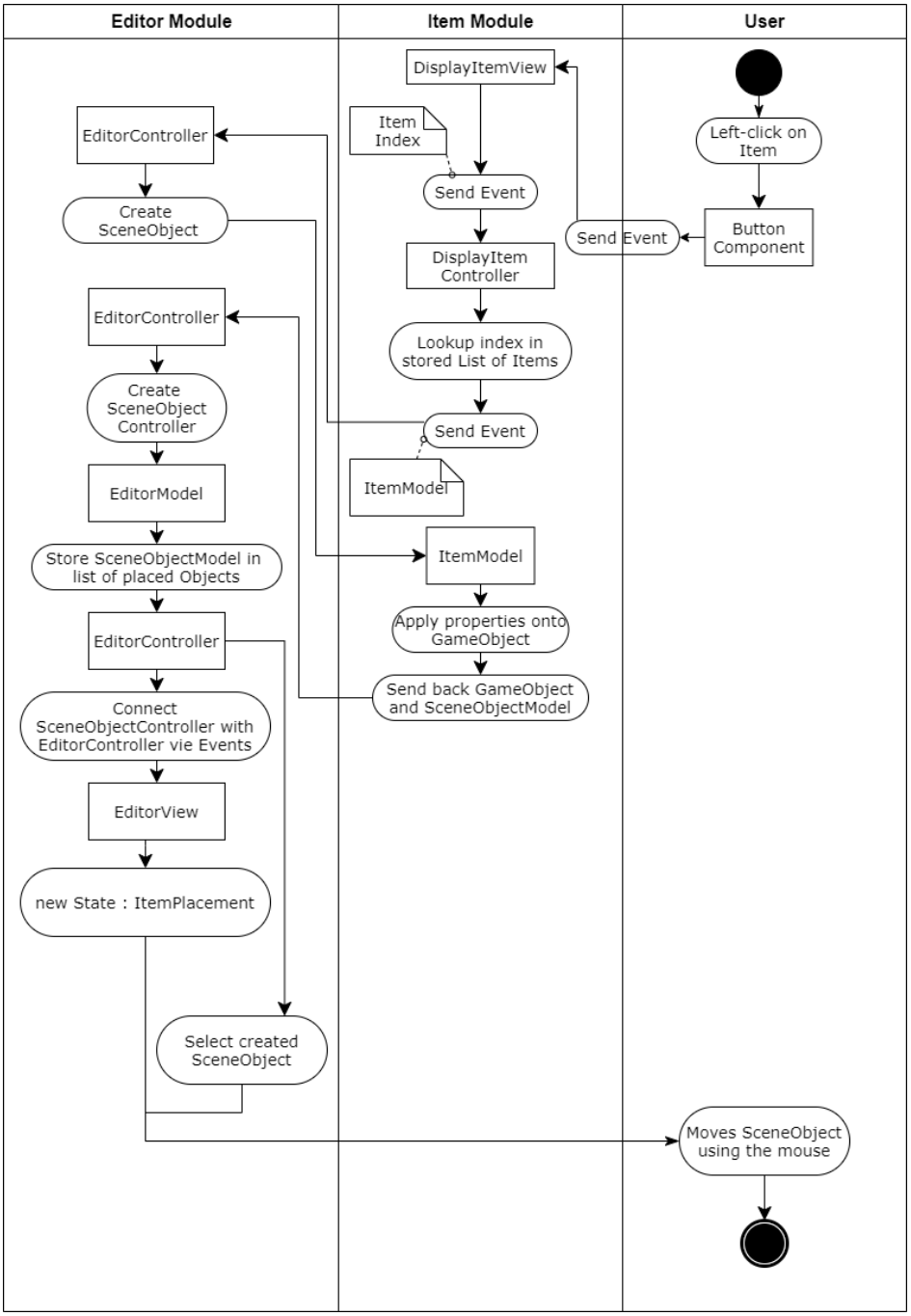

Figure 3.9: Activity diagram that visualizes the process during the Item creation

to map GameObjects in scripts to specific object groups. Thereby, GameObjects can be processed differently, according to their tag. Furthermore, it allows searching for objects in the scene via script using methods like FindGameObjectsWithTag.
During the placement of the SceneObject, its tag is readout and assigned to a layer based on it. A distinction is done between "Visualization" and "SceneObject". This distinction is relevant during the generation of the images. Objects such as the camera or lights are by default not represented by a fixed 3D object within Unity while the application is running. Only their properties, like the displayed image of the camera or the emitted light, can be perceived by the user. This would have the consequence that the user would not be able to know and change the position of the objects after they have been placed for the first time. Therefore, these Items have a 3D model, which serves as visualization. Therefore, all placed objects in the scene can be seen and clicked by the user while setting up the environment. The different layers ensure that the visualizations are not shown in the later images. The CameraItem is set up in such a way that only objects that are in the "SceneObject" layer are rendered. However, the properties of light and cameras, such as the emitted light, are still visible in the final images.
After the object has been assigned to the correct layer, the state of the EditorView changes to ItemSelection. As a consequence, the position of the created object is no longer modified by the movement of the mouse. So it remains at its current position.
If the Item the user clicked on at the beginning is a TextureItem, the procedure changes fundamentally. A plate-shaped GameObject is created in the EditorController and the selected texture is mapped onto it. In this way, the user receives visual feedback and sees which texture has been selected. After changing the state of the EditorView to TexturePlacement, the plate can now be moved freely by moving the mouse. The calculation of the current position is done in the same way as described above for other GameObjects. If the user now clicks on a SceneObject, this texture is applied to the object.
This is done using the same procedure as when changing a Parameter. The selected texture is sent as an event argument to the SceneObjectView of the clicked SceneObject and passed on to the corresponding ParameterModel. See the Changing Parameter section for more information.
When a texture is selected, the process is also aborted by a right-click. Since the texture is applied to already existing SceneObjects, it must not be removed from the list of current objects. Since no SceneObjectController is created, no references need to be deleted. It is sufficient to destroy the plate object and set the state of the EditorView back to Idle.

3.3.2 SceneObject Selection

The properties of objects placed in the scene can still be modified by the user. Therefore the user has to select the object first. To do so, the user can simply click on the SceneObject and get the values of the respective Parameters displayed (see Figure 3.10). However, not more than one element can be selected at a time. If another object is clicked on, the selection changes to this object or is completely deselected if the empty space is clicked on.

The selection of a SceneObject is also recognized by the EditorView. If the view is in the state Idle or ItemSelection, a mouse click, similar to the placement of a SceneObject, fires a raycast from the position clicked. The mouse must not be over the UI, otherwise, the click does not refer to the objects in the scene. This ensures that the user does not accidentally select an object hidden by the UI. Additionally, the raycast is limited to the layers "SceneObject" and "Visualization" to minimize the number of collision checks required. If a matching

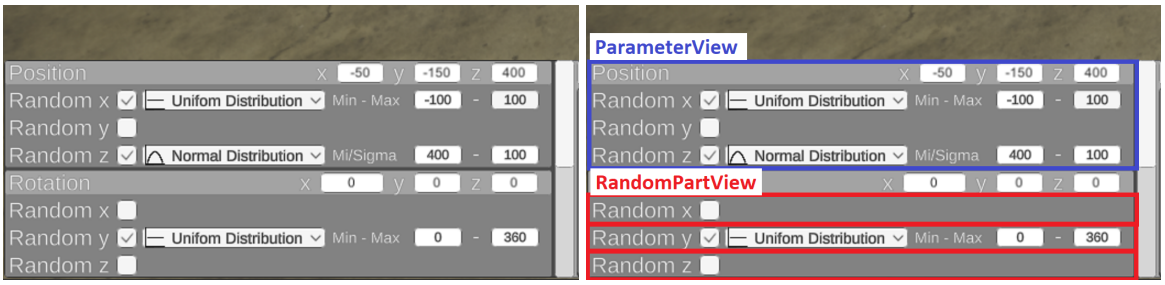

Figure 3.10: Left: Parameter of an object Right: Shows the used ParameterView and RandomPartViews

object is hit with the raycast, data about the hit object is returned by the raycast. Then the EditorView sends this data to the EditorController via event. If no object is hit by the raycast, an event is also sent to the EditorController. This event then causes the deselection of the current selection. For each selection, the currently selected object is also properly deselected first. This process is described after the following selection process.
In the beginning, the EditorController checks whether the passed object matches the current selection. If it is the same object, it is already selected and the process is aborted. Otherwise, the selection is changed to the object passed by event arguments. The state is then changed to ItemSelection and the outline component of the selected object is activated.
When creating the SceneObject, the AbstractItemModel also provides the ability to add the asset Outline [Nol] as a component to the object. This component is deactivated by default for SceneObjects and is activated when a SceneObject is clicked. When enabled, the component draws a light blue outline around the clicked Item (see Figure 3.14). This outline is also displayed through other objects. This way, the user can always see which object is currently selected and where it is located.
If the selected object is a camera, the preview image on the right border is activated. In order to display the values of the individual Parameters, the EditorController now requests the list of ParameterModel from the SceneObjectController. After it has received these via event arguments, they are forwarded to the EditorUIController. They are then passed on to the EditorUIView via Events. There, the received data of the Parameters are now processed and displayed one after the other.

The Parameters form another entity in the application and are each represented by a Model-View-Controller module. In addition, when such a module is created, the type of the Parameter is defined so it can be uniquely determined.
The EditorUIView creates an independent module for each ParameterModel in the received list. For this purpose, the model is taken directly from the respective entry in the list. For the creation of the ParameterView, the corresponding model is instructed. The model forwards the request to a corresponding factory. There, the UI element for displaying the values is created. Since each Parameter represents different properties, they cannot all be displayed in the same way. For example, the Parameter for the position must represent the x, y, and z coordinates of the object, while the light intensity is only represented by a single number. Therefore, for each type of parameter, a UI prefab is available, which is adapted to the individual representation. The factory loads one of the prefabs to create an instance of a view. Afterwards, the factory returns the created view to the EditorUIView.
Both objects, the ParameterView together with the ParameterModel, are then passed to the ParameterController during the creation. The new controller is then stored in a list of all current Parameter controllers. This list is needed to make it easier to remove the individual views later. If similar to the SceneObjectController, no direct reference to the controllers of the Parameter modules would be kept, this would result in a laborious series of event calls,

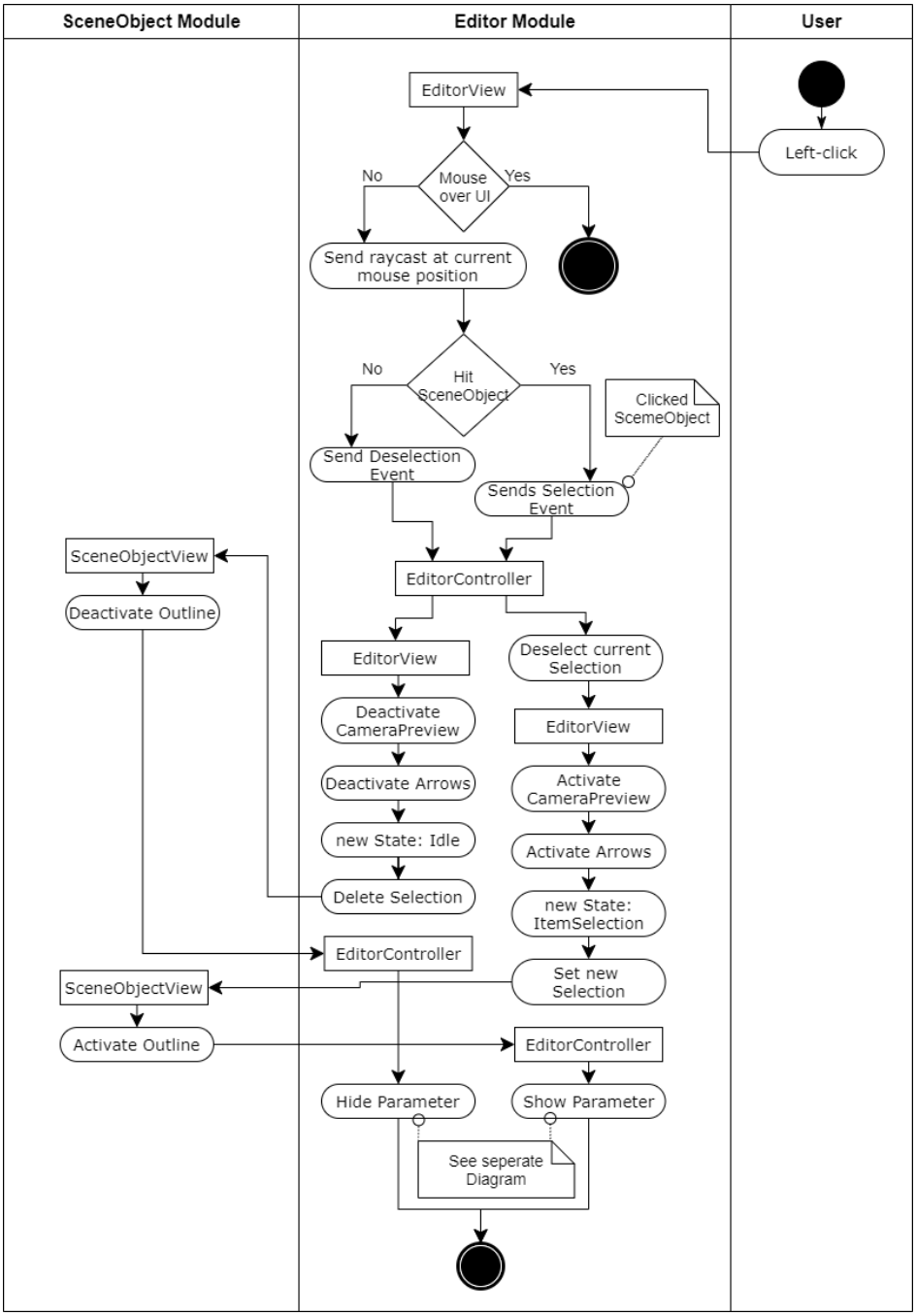

Figure 3.11: Activity diagram that visualizes the selection and deselection process of a SceneObject

whenever these should be reached.

Nevertheless, some EventListeners are still registered afterwards, which are needed for communication between the modules during Parameter changes. In the end, the view is placed and scaled correctly.
This whole process is repeated for each ParameterModel in the list. When this process is completed, the left field of the UI displays all properties of the clicked object as a scrollable list.
Again, the advantages of inheritance come into play. The implementation of the individual functionalities depends on the individual model. Thus, the application can easily be extended by further Parameters. Since the ParameterModel also determines how the properties of the SceneObject are changed concerning the parameter, Parameters can be attached to the individual types of SceneObject without having to make major changes in the code.
The following section describes the steps that are performed when the current object is deselected. After the controller has intercepted the appropriate event, it first checks whether deselection is necessary at all. If no selection is currently active, nothing needs to be done at this point. If a selection is active, the camera preview will be deactivated first, if the selection is a camera. Then the outline component of the selection is deactivated so that it is no longer highlighted.
Afterwards, parallel to selecting an object, the controller forwards the currently selected SceneObject to the EditorUIView via event. There, all existing event listeners of the selection are separated from the stored ParameterControllers. Besides, the list of stored ParameterControllers is emptied. Then, the single view elements of the UI field are removed and deleted. Finally, the EditorController resets the state of the EditorView to Idle.

3.3.3 Changing Parameter

If the user has selected an object, its Parameters can now be modified. For that, the displayed values of the Parameters must be overwritten. Again, the code of the application is kept in such an abstract way that the basic procedure is the same for each Parameter.
If the user enters a new value, this is recognized by the input fields of the UI. The input fields are UI elements provided by the Unity API. Thus the fields offer a variety of functions provided by their interface. One of them is the event OnEditEnd, which is triggered whenever the user completes his input. In concrete terms, this means that the event is triggered whenever the input field loses the focus or the user presses the enter key. This event allows the input fields to be linked to a certain method of the ParameterView.

If this method is called, the view reads the corresponding values from the input fields. These values are generally strings. They are then converted into the respective types for further processing. For instance, the three values of the color Parameter must first be converted to an integer and as RGB values to the corresponding color. After that, the new values and the type of the Parameter are sent to the SceneObjectController via the SceneObjectView using event arguments. For this purpose, the events are used, which were registered during the creation of the ParameterView. The controller iterates through the list of its stored ParameterModels and compares their type with the type just received. If the types are of equal type, the matching Parameter is found. The new values are then passed on to this Parameter.
The Parameter stores the new values and then requests the GameObject of the SceneObjectView from the SceneObjectController. After receiving it, the ParamaterModel adjusts the property of the GameObject according to the new values.

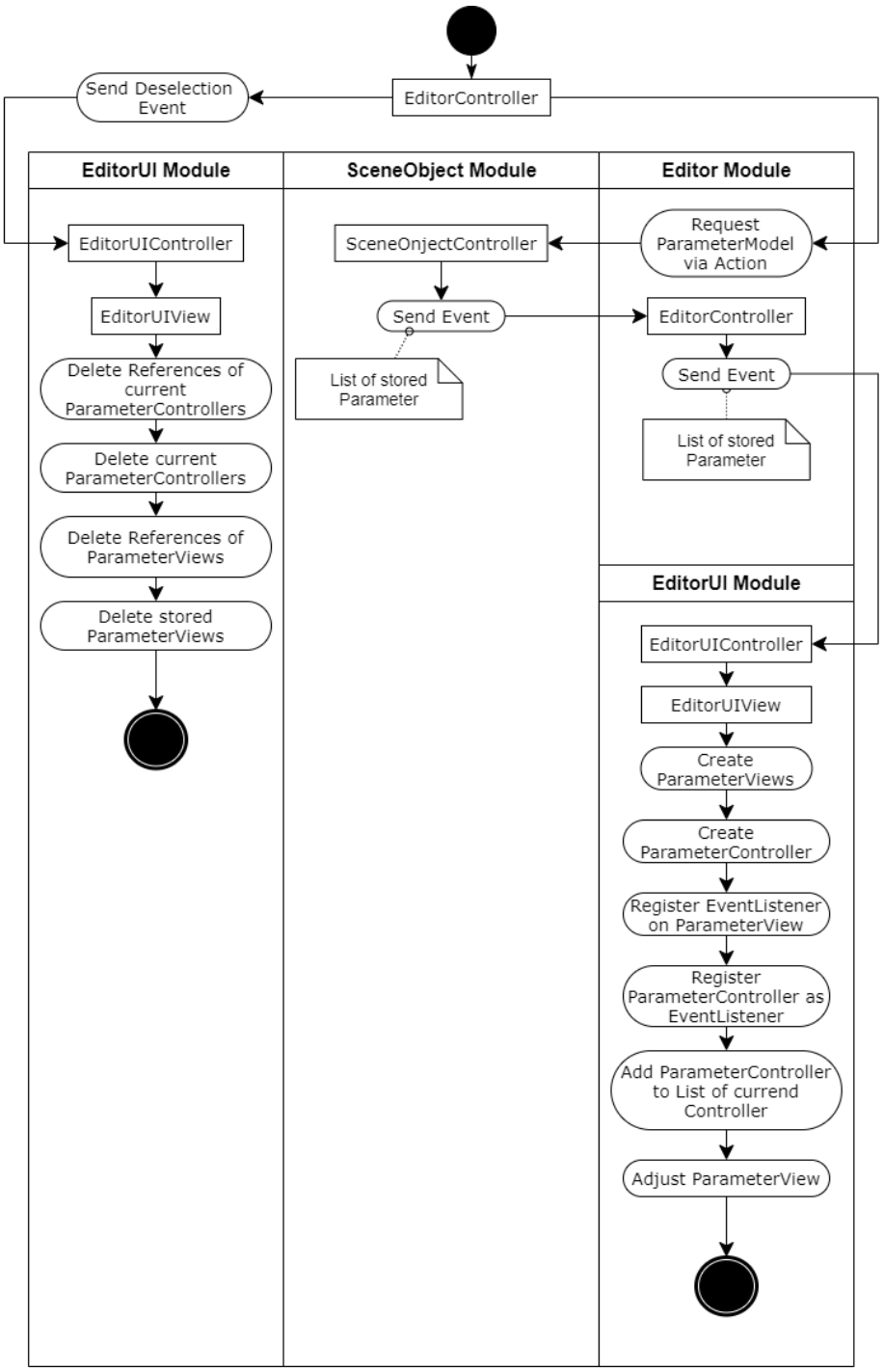

Figure 3.12: Activity diagram that visualizes the process of displaying and hinding the Parameter

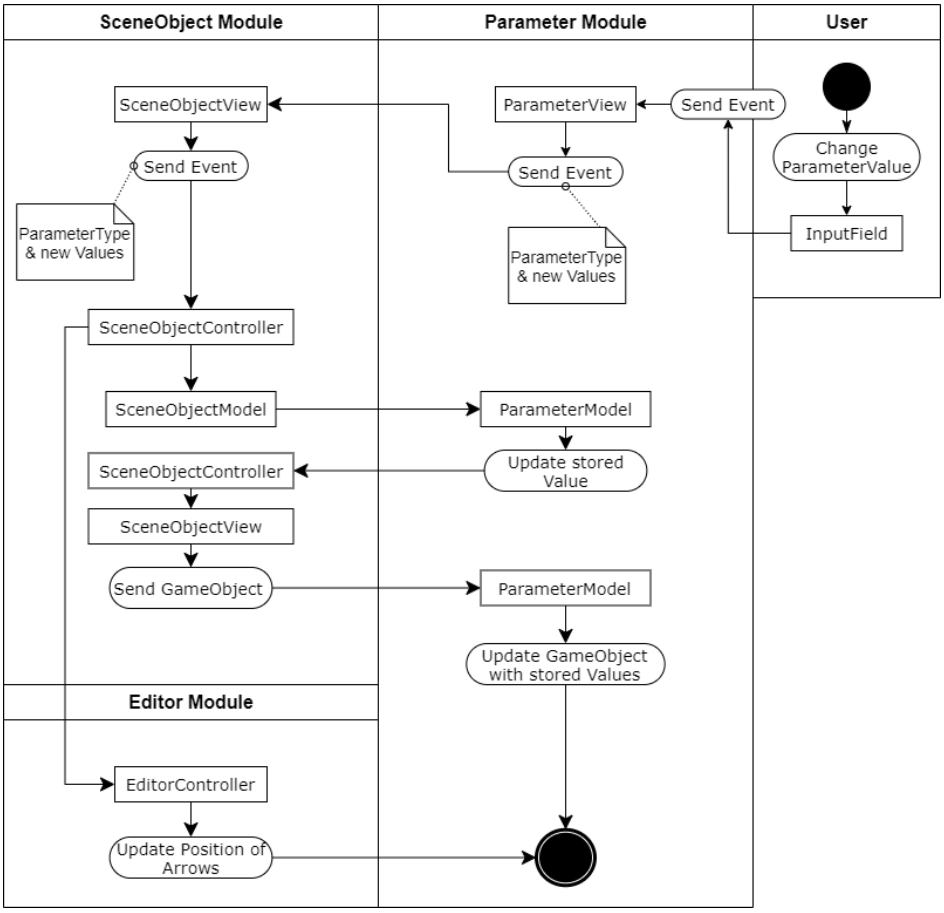

Figure 3.13: Activity diagram that visualizes the running process when a Parameter is changed

As already mentioned when selecting a TextureItem, the texture of a SceneObject is adjusted in the same way. After clicking on a texture in the bar at the bottom of the screen, the texture will be instantiated. If another object in the scene is clicked on, the selected texture will be packed into event arguments and texture will be selected as the Parameter type. Next, the event arguments are sent to the SceneObjectView via event and the process just mentioned is executed.

ParameterType	Description	Characteristics
Position	Represents the location in 3D space	Attached to every object
Rotation	Represents the orientation in Degree	
Scale	Represents the size of each dimension	Uneven scaling leads to a distorted object
Object Color	Defines the general color in RGB Format	Can be combined with the object texture
Texture	Defines the appearance of the surface and its structure	Can be combined with the object color
Light Intensity	Defines the brightness of the emitted light	Represented through a decimal number
Light Range	Defines how wide	Objects outside the range are not affected
Light Angle	Defines the width of the light cone	Only attached to the Spot Light
Light Color	Shows the color of emitted light in RGB Format	

Table 3.1: All implemented Parameter types with a short description and its particularities

In Table 3.1 the already implemented Parameter types are described in detail. In addition to the already listed particularities, the position offers another special feature. Besides the normal modification of the Parameters, it can be changed by additional help. For this purpose, a visualization of a coordinate system is used, which is represented by three arrows as shown in Figure 3.14. These arrows are displayed as soon as a SceneObject is selected. If the object is deselected, the arrows are hidden again or will be shifted to a new selection. The arrows always appear at the same position in relation to the object. To calculate this position, the BoxCollider component is used, which can be found on every SceneObject. A cuboid is created around the object with its sides parallel to the three-dimensional axes. Furthermore, the position of the sides is chosen in such a way that they lie as close as possible to the surface of the object and thus enclose it minimally. This type of element is also called an Axis-Aligned Bounding Box.

To calculate the position of the arrows, the EditorView uses the data of the BoxCollider. Based on the coordinates of the BoxCollider, the position of the lower right corner is calculated. Then arrows are placed at these coordinates with a slight offset so that they do not overlap with the selected 3D model. The EditorView has stored a fixed reference to the arrow object to avoid searching for it in the scene for each action to be performed.

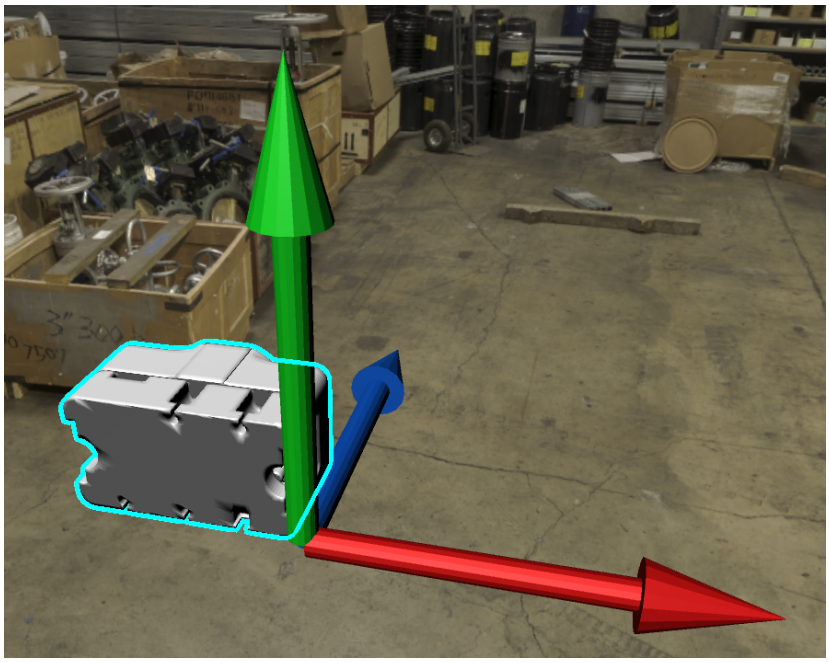

Figure 3.14: The arrows supports the user to position the objects. The highlighted outline visualizes the selected object

Each of these arrows also has its own ArrowView that stores the direction in which the SceneObject should be moved while using this arrow. It also contains the color in which the arrow should be displayed. If the user hovers over one of the arrows, its color changes slightly, similar to the button. Again, the visual feedback should confirm whether the user has selected the correct arrow.
If the user clicks on one of the arrows, it sends its direction, stored as a three-dimensional vector, to the EditorView via event. The EditorView saves the resulting vector and changes its state to SelectionMovement. As long as the user keeps the mouse button pressed, the view remains in this state. If the user releases the button, it changes back to ItemSelection. During the SelectionMovement state, the difference between the current position of the mouse and the position of the previous frame is calculated. The faster the mouse is moved, the greater the distance and the greater the change in position of the SceneObject. This movement can also be slowed down even further by pressing the Shift key simultaneously. This allows the user to change the position of the objects more precisely. Each time the arrows are moved, the Parameter display of the position is updated in parallel.

3.3.4 Randomization

Until now, the user can place objects in the scene and change their Parameters. However, to achieve the goals of domain randomization, it must also be possible to modify them by randomization. For this purpose, the user can now select a type of randomization for each Parameter and define the range of desired values.
In order to enable this, each ParameterView contains not only the display of the values but also at least one range that is responsible for the randomization of these values. This RandomPartView contains different randomization types depending on the Parameter type as visualized in Figure 3.10. Currently, three different types of randomization are supported.
The first randomization type is uniform distribution. Therefore two values are needed. A

minimum value and a maximum value. The new number is randomly generated between the two provided values. Every number within the given range has the same probability to be selected.
An additional approach to create new values is normal distribution. This procedure also uses two values, Mi and Sigma. Mi determines the midpoint of the values. The probability of a value to be drawn increases the more close it is located to Mi. Sigma defines the range in which values can be generated. Since the probability can be displayed as a bell curve, the chance for extreme values would never drop to zero. Even if the probability is very small, there is still the possibility of such an extreme value. To exclude such extreme values, no generated value can be more than three Sigma ranges away from Mi. Therefore all values, which are beyond that, are set to the boundary values. These two kinds of random number generations are used to generate a numeric number, such as a coordinate of position or a RGB value of the color.
The last type of randomization is TextureRandomizatin. As the name suggests, this type is used to randomly change the texture of an object. However, new realistic textures are not randomly created for this purpose. This process would be very complex for the Albedo Map alone. Therefore this kind of generation uses the textures imported by the user. During randomization, one of the textures is randomly selected and then applied to the SceneObject. Like the uniform distribution, each texture has the same chance to be selected.
The generation of the values is done within the individual randomization types. These are all derived from an abstract randomization class. Thus the basic structure is provided, by which the individual randomization types have to be implemented. This makes the call of the randomization independent of the given type. Only a random number generator is needed for the generation of new numbers. This generator is provided to the randomization types by the ApplicationManager. The random generator uses the Random class of the.NET framework. At the beginning of the generation, an instance of the generator is created and maintained for the remaining runtime of the application. However, this type of random generator only generates pseudo-random numbers. This means that the generated numbers are not random, because they are generated by a fixed algorithm. Thus, every single instance of the random generator would always deliver the same sequence of numbers. To prevent this, the randomization seed mentioned in the beginning can be used. This seed can be used when instantiating a random number generator to influence its algorithm. On the one hand, the output of the randomization seed changes for different seeds each time. On the other hand, randomizers, which were created with the same seed, always return the identical number sequence.
To generate such a seed, a Globally Unique Identifier (GUID) is used. Then an integer number based on the hash code of the GUID is returned. Each created GUID is unique, as described in [Mic], and due to the integer representation, the resulting seed can take 1 of 2^{32} different possibilities. So there is enough variance to use this number as a suitable seed for the random number generator in the sense of this application.
In order to be able to use the RandomizationTypes, the provided types are stored in the RandomPartView when it is initialized. An interface is also used for the view so that certain functionalities are provided and the calling class does not need to know the exact implementation.
The RandomPartView of the texture has only one checkbox since no further input is required for randomization. With this checkbox, the randomization can be activated and deactivated. The RandomParView for the generation of numbers contains additional fields to the checkbox. If the checkbox is checked, the second part of the UI element becomes visible. There you can select one of the stored randomization types via a dropdown menu. By default, the first element of the list is already selected. Afterwards, the values needed for the randomization

can be entered into the provided input fields. Depending on the type the fields have different labels to tell the user which values are expected.
Some of the values of the Parameters are composed of several single values, like the coordinates of the position or the RGB values of the object color. Each of the values can be individually randomized and has its area to define the values and type. For example, the height of an object can be changed using a normal distribution, while the position within the horizontal plane can be specified using a uniform distribution. The mapping of values and the corresponding RandomPartView is done in the respective ParameterView.
If the user changes the randomization, the changes are saved in the respective RandomizationType. If the selected type is changed, the previously selected type is deactivated and the new type is activated. Since one reference to a RandomPartView is stored for each value of the parameter, the stored RandomizationType can be readout during the randomization phase.

3.3.5 Additional Features

The application provides some additional functionality, which has not been covered yet. One function is to load the already existing SceneObjects. During the creation of such an object, its model and thus also its Parameter properties are stored in a list in the EditorModel. All models are stored in the ApplicationManager. This way they are retained when the scene is changed, unlike the controllers and the view. If the user switches now back to the EditorScene, this list is used to restore the last state.
The procedure is similar to the first creation of a GameObject. First, the EditorController iterates through the list of stored SceneObjectModels. Then a new GameObject is created. The basic properties like 3D mesh or loading the corresponding prefab are now transferred to the created GameObject by the ItemModel stored in the model. After the view and controller are created, the SceneObject is assigned to the correct layer based on the type of the SceneObjectModel. Afterwards, all necessary EventListener of the EditorController are assigned to the appropriate events. Finally, the stored Parameter properties such as position or light intensity are transferred to the GameObject. This process is repeated for each element in the list.

When the user has set all Parameters, the user can test the randomization in advance. For this purpose, the UI provides a button that randomly changes the values of the current selection once according to the specifications. For this purpose, however, a SceneObject must be selected, since the test randomization is only triggered for this object.

The user can not only delete SceneObjects during placement but also remove them afterwards. To do this, the corresponding object must first be selected. If the Delete button is now pressed, the object is removed. The deletion process described in the Item Presentation and Placement section is also used here. In addition, the state of the EditorView is set back to Idle and the help function in the form of arrows is hidden.

By default, a directional light is already placed in the EditorScene. This should help the user at the beginning of the setup and supply the scene with light. The user can then insert his own light sources and delete the help. Once the light is deleted, it remains deleted and is not recreated when the scene is reloaded.

3.4 Image Generation

For the generation of the images first, all cameras that have been placed in the scene, except the main camera, among with the objects in the scene that have to be randomized during the process have to be found and saved. To achieve that an iteration through all saved SceneObjects is made, which filters the objects by their tag and adds them to the right list. A camera itself can also be a randomizable object.
After filtering the actual generation process starts. During each run first, each affected object is getting randomized, where the randomization is handled in the ParameterModel of the individual Parameters of the SceneObjects. The values of the single Parameter are changed depending on the respective implementation of the ParameterModel. The new values are generated in the RandomizationType that was selected and deposited in the EditorScene. This approach encapsulates not only the generation of new values from their usage as already mentioned before, but it also makes the application of those values and the corresponding Parameter independent from the object, the Parameter is attached to.
The ParameterModel saves the created values and sends an event to the SceneObjectController, which then causes an update of the view and on this way an application of the new properties on the SceneObject.
After the finished randomization, each camera takes a number of different images of the current scene and its objects (see Figure 1.1). This is handled by the SceneObjectController of the individual camera since it can easily access the provided functions of the components of the connected view.

At first, a normal screenshot is taken using a new texture and saving the current view of the camera in it. After the normal one, a grey-scale image is created representing the depth of the image. Here the distance between the camera and the object is taken into account and used to calculate a grey color between black and white. The brighter the point in the image appears the further away it is from the camera that has taken the image. To achieve this a special shade is used [vex].
The third image that can be created is a segmentation image, which separates all objects by a unique color using also a special shader [KK19]. This image shows every pixel covered by the object in the same color without any edges, light or shadow, as long as it is not covered from another object. Through the individual colors, the model can later identify which part of the image belongs to the object and which one is background or another object.

After the run, the created textures are destroyed and the used memory is freed up as the textures are part of the UnityEngine and thereby not affected by the garbage collector. Doing so avoids a memory leak and the crash of the program in case of a large number of runs. This whole process will be repeated until the given number of runs was made. When the generation is finished the opportunity is given to open the target folder and spectate the images in the explorer or to return back to the menu, make some changes or adjustments and start the generation again. Additionally, the number of already generated images is updated after each run. This is then used to display the current progress in steps of ten to inform the user.

During the generation, a document is created that stores entries with information about the bounding box for each object per image. This entry consists of the length and width of the Bounding Box and the screen coordinate of the upper left corner. The exact procedure depends on the Parameter mentioned above.
If the Parameter is set, only objects that are fully placed inside the border of the image, will be considered. Therefore it does not matter if an object is partially covered by another one as

long as it is fully inside the image. To check this condition every single vertex of the mesh of an object is converted from world space into corresponding screen coordinates [eri]. If one of the vertices is outside the borders, which are represented through the predefined resolution, the object is not taken into account. To determine the minimal bounding box of an object the individual screen coordinates of the vertices are compared with the current stored values. If a coordinate is bigger or smaller then the stored one it is saved as new extremum.
The second variant also considers objects that are only partially contained in the image. Again, it is irrelevant whether the object is covered by another one. Parallel to the first method, all vertex points are converted into screen coordinates and compared with the currently stored extreme values. However, if a vertex is outside the screen, its values are set to the border values of the image (0, width or height). Before the data of the bounding box is calculated at the end of the complete iteration, the extreme values are compared with each other. If the minimum and maximum of an axis are equal, the considered object lies completely outside the generated image. In this case, no entry is created in the document.
After a successful iteration, these stored values are used to calculate the left upper coordinate together with the width and length of the bounding box. Besides this information also the id of the object, which is fixed for each object during the whole generation process, is included in the entry.
For every individual image, one set of entries is written into the file. This file is later used by the AI-Agent to retrace the objects in the given images using the uniform data representation. This helps the AI-Agent to extract and gather useful data about every object and improve its precision in object detection.

4 Experimental Evaluation

After the application has been fully developed, it can be used to generate a data set for an experiment. Here the effect of domain randomization on the performance of an object detection model is investigated. The T-LESS data set is used for the experiment to train the model in the first run with real data. In a second independent run, synthetically generated images are added and the performance of the two models is measured and compared.

4.1 Experiment Setup

In this experiment, we want to find out if the generated images influence the performance of an object detection model. In the first step, the developed tool is used to generate a set of synthetic images. The T-LESS data set is used as a starting point. This data set consists of 3D models and RGB images which can be used to evaluate object detection and 3D pose estimation. The data set is freely available and licensed with Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) [1]. The objects of the data set are imported into the tool without modification. During the generation of the images, only the color of some objects is changed.

The data set contains 30 3D objects of different quality. For the generation of the images, the set with the highest quality is used. The provided models are based on real objects, all of which are related to the industry. The objects have no textures and are all displayed in the same grey color. To create the synthetic images, 9 of the 30 available objects are used and imported into the application. The objects are displayed in the upper part of Figure 4.1. Additionally, 11 textures including a normal map were loaded. The imported objects are used to create a virtual environment, where all 9 objects are placed at the same time. The position, the rotation and the color of the object are randomized. Additionally, several different light sources are used in the scene. Their position and orientation, as well as the intensity and range of the light, are also randomized. As a result, the individual images have different lighting conditions in addition to the differently positioned objects. The textures are used to integrate another random parameter into the environment, i.e., using them, the background of the objects is also randomized. During the generation of the images, it is ensured that they have the same resolution as the other images in the training set.
As already mentioned above, the data set provides not only the 3D models but also a large number of images already created. A distinction can be made between sets for training the model and sets for subsequent testing of the model. The training data consists of one set per 3D model. In each set, one 3D model is displayed in front of a black background. The pictures also show the object from every possible perspective, so that each set consists of

[1] Attribution-ShareAlike 4.0 International https://creativecommons.org/licenses/by-sa/4.0/ (Accessed 07-10-2020)

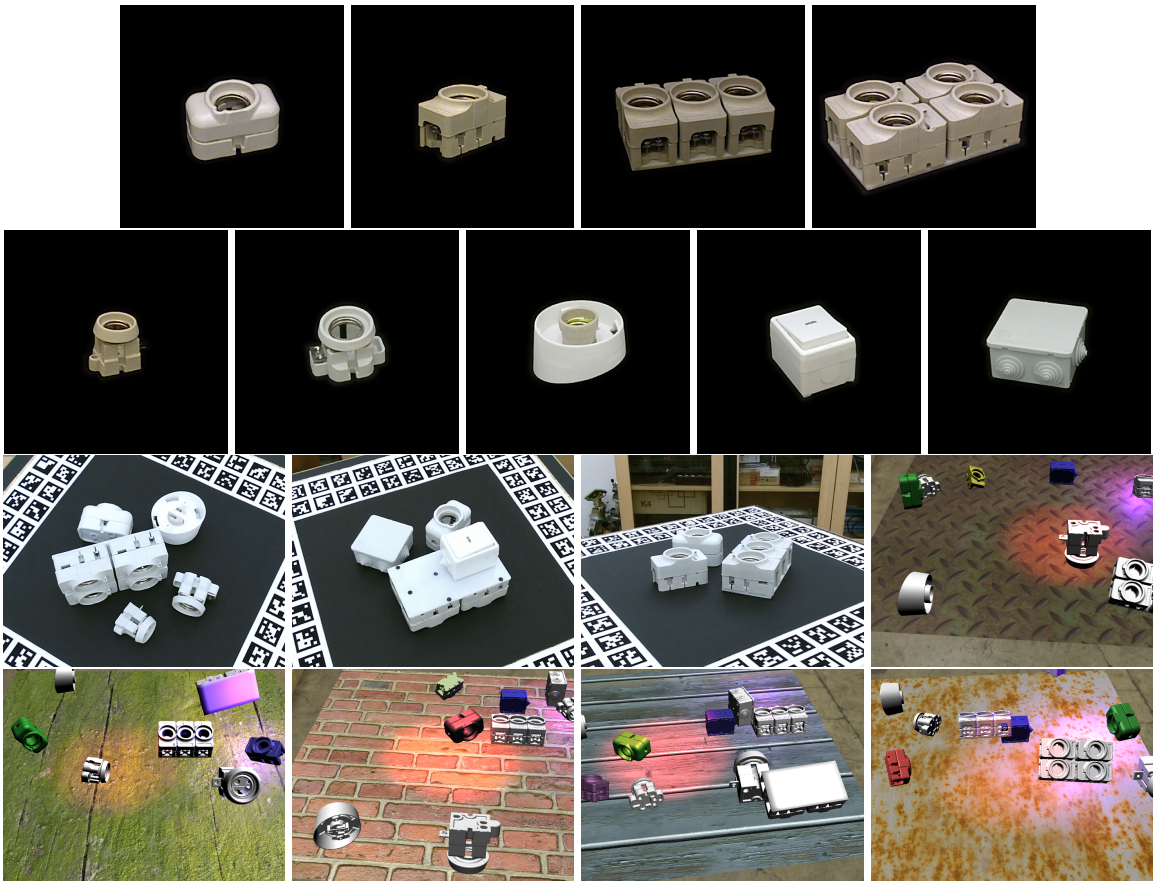

Figure 4.1: Sample images from the T-LESS dataset [Hod+17]. First two rows: Objects used for the training data. Last two rows: sample images of the three sets of real training data as well as some random generated ones.

more than 1.200 pictures. The test sets, on the other hand, contain more than just one object. These show compositions of different objects. The pictures also show the scene from all perspectives. In contrast to the generated images, the images of the T-LESS data set are images of real objects. Additionally, each data set contains information about the respective bounding boxes of the individual images, so that they can be used during the training or the subsequent testing of a model.
However, the training data sets are not suitable for the experiment, as current state-of-the-art deep-learning models require multiple objects per image as training data. Therefore, data from the test sets are used to train the models. The first model is provided with the images of sets 2, 3 and 4. The second model receives 5,000 additional synthetic images generated with our tool in addition to the above-mentioned sets. During training, both models receive information about the matching bounding boxes of the objects in the image.

After the preparation of the images, the actual experiment takes place. The basic structure is similar to the one described in [Jos+18]. For object detection an implementation of the Faster-RCNN model [Ren+15] is used, which is provided by the Tensorflow Object Detection API [ten]. This API is an open source framework, which was built on top of TensorFlow. It supports the user to build and train an object detection model. The API provides a collection of various detection models, which differ in speed and accuracy. These models are pre-trained on the COCO data set [Lin+14], which contains thousands of real images of commonly found objects. As metric, we use the mean Average Precision (mAP), as it has been defined for the PASCAL Visual Object Classes (VOC) Challenge [Eve+10].

To get comparable results, both models are trained in the same way. The models both start in a pre-trained state and are fine-tuned with the respective data set. The first model has about 1500 real images of the objects available, while the second model is trained on a mixture of these real images and the 5000 synthetic ones. After 5000 steps the performance of the models is measured on real images. The T-LESS test sets 11, 15, 16, 18 and 19 are used for this, which together make up about 2500 images. However, the test images have some difficulties. They also contain objects that are not included in the training data but look very similar to some of the objects used, as seen in row 4 in Table 4.3. In addition that the test data also contain completely different objects which only have the propose of disturbing the object detection. Furthermore, the background, at least that of the real training data is always black, which is a good contrast to the used objects. Thus, these are highlighted and can be identified more easily by the model. The test data have a cluttered and highly textured background, so the easy background-object distinction is no longer present.

4.2 Results

Figure 2 shows the loss function of both models. It can clearly be seen that the first model (orange), which was trained with real images only, achieves a lower loss value than the second model (blue), which received both real and synthetic images. A lower value represents a higher accuracy of the model. These differences during the training phase are due to the variance in the generated images. Since the data for the first model is composed of different images of only three fixed situations, the model can recognize them more easily during training. In addition, there is much less training data than performed steps, so the same images are presented to the model several times. In contrast, each of the generated images shows a unique constellation of the objects used. Therefore the chances are very small that the model receives similar images during the training and recognizes the shown objects.

However, this changes, as it can be seen in Table 4.1. In the test phase, the second model has a higher, or at least the same accuracy as the first model. The only exception to this is the recognition of obj_26. The position, as well as the color of obj_26, was randomized, but this object had a fixed rotation so that it is always displayed from the same side. Therefore, the second model never sees the front of the object, which appears in some of the test images (Figure 4.3 Row 1: The object in the middle). This kind of error can also easily occur with a data set of real images. The objects shown there are usually on one side, making the side always hidden and no information about it can be collected. This can be prevented by using generated images and with randomized rotation so that each side can be seen by the model. While the fixed rotation was oversight during the generation of the training data, it turned out to be interesting result that highlights the importance of randomizing rotation for synthetic training data. Due to the long time needed to generate new data and retrain the model and since the results were conclusive, the experiment was not repeated with randomized rotation of obj_26..
Another conspicuous feature of the values is the accuracy of 0% for obj_06.This is due to the fact that this object was included in the training data but not in the test data. Therefore, no accuracy can be calculated for this object.

In Figure 4.3 row 2 and 4 you can see that both models have problems with the many different objects in the scene. The models have received training data only for some of those objects shown in the scene. Nevertheless, both models often recognize other objects and try

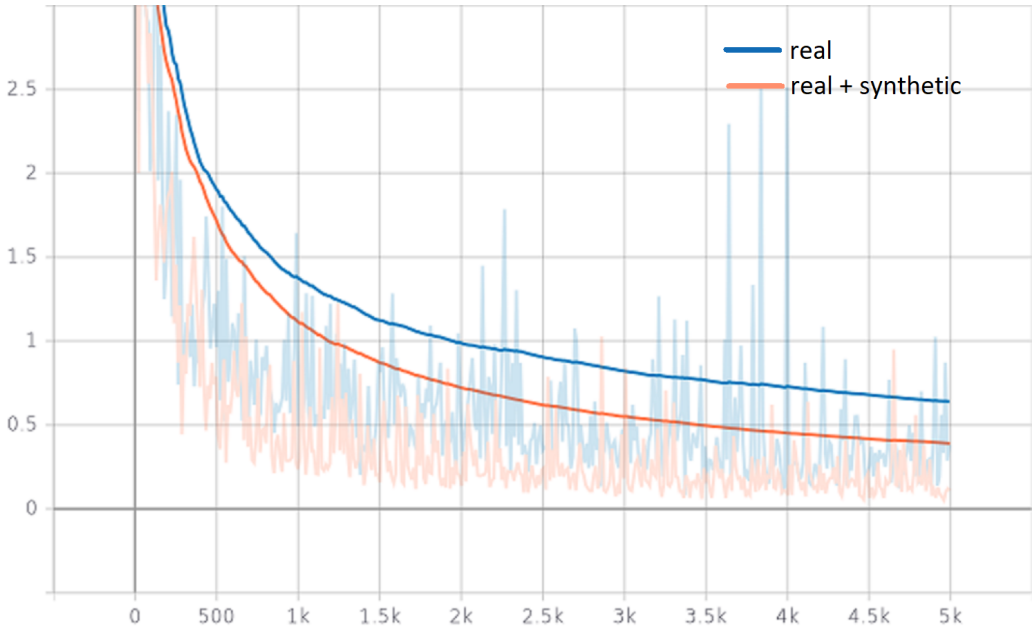

Figure 4.2: The smoothed TotalLoss function during the training phase
x-axis represents the number of steps done; y-axis represents the loss value
The orange graph visualizes the run with only real images while the blue one visualizes the run with both, real and synthetic images

	obj_5	obj_6	obj_7	obj_8	obj_11
Real Images	0.805	0.000	0.245	0.0516	0.049
Real & Synthetic Images	0.929	0.000	0.334	0.0889	0.071
	obj_12	obj_18	obj_26	obj_28	average
Real Images	0.036	0.392	0.504	0.254	0.292
Real & Synthetic Images	0.034	0.389	0.277	0.598	0.340

Table 4.1: Accuracy results of the object detection experiment.

to assign them to a known label. Often the real objects are very similar to the objects the model assigns to. It is also noticeable that especially in cluttered scenes with many objects, the first model generally recognizes more objects with high confidence (greater than 0.5) than the second model (see Figure 4.3 row 3). In general, it can be said that the second model is more accurate in object detection than the model that was trained only with real images. This is due to the large number of different situations represented by the generated images. The first model may be overfitted due to the many similar images and the simplistic background of the real training data. As a result, the model is too fixed to a certain viewing angle of the object that it will not recognize it if it does not appear in the usual way, or if it is not easily distinguishable from the background. This can also be seen with obj_26. To counteract this, Domain Randomization can be used, which was applied to the data set for the second model. This creates many different views of the object and enough variance to make good predictions even in new, unknown situations. Unfortunately, the problem with synthetic images is that they do not represent real situations. Therefore, environmental influences and disturbances like reflections on the surface are often missing on these images. However, if you look at the T-LESS data set, it must be mentioned that it is very laborious and time-consuming to create such a data set and prepare it for object detection. Not only must

the objects be imaged from every perspective, but also the information about the appropriate bounding box must be created afterwards. In return, tools like the one we have developed can be used to create a large number of randomly generated images in a very short time. The tool brings a huge time saving compared to the manual method. It also offers the advantage that the detection and labelling of the bounding boxes are done automatically.

Figure 4.3: Examples with predicted bounding boxes as well as a label and confidence. Left column: run with only real images, Right column: run with real and synthetic image. The used real images are from the T-LESS data set.

5 Conclusion

In this thesis, we applied the Unity game engine to develop a tool that can generate synthetic images and annotations for training deep-learning-based computer vision models. The tool was specifically designed to minimize the effects of the reality gap problem using domain randomization techniques.
Several key functionalities were implemented in order to ensure generality, ease of use and easy integration and extension of the tool for computer vision tasks:

- The user can customize the objects of interest and import their 3D models and textures.
- The imported models can be used in the application GUI to build a suitable environment.
- Different properties of the objects, like position, rotation or color, as well as scene properties like lighting and camera positions can be randomized.
- Several types of output images (RGB, depth) and annotations (object bounding box, pixel-wise segmentation image) were implemented
- The user can configure general parameters like image resolution, number of generated images or target location through simple and easy to understand interface

During the development of the tool, we also paid attention to enable easy extensibility of the existing functionalities. This was achieved by using a suitable software architecture and a combination of several design concepts. By the abstract design of the individual elements, new elements can be added, without having to make large changes to the core code. Thus, new categories, new parameters or new randomization properties can be added with minimal effort.

After the implementation of the tool was completed, it was used to demonstrate its applicability and investigate the influence of the generated images in object detection task. For this purpose, a state-of-the-art model for object detection (FRCNN [Ren+15]) was evaluated using a benchmark dataset for object detection (T-LESS[Hod+17]). The same model was trained to detect several objects of the T-LESS dataset by predicting the objects' bounding box and class. Two independent raining runs were performed: one run included only a set of real images while the other included a set of real images and an additional synthetic images generated by the developed tool. The model from each run was then evaluated on a test set of real images. The experiment results showed that the model trained with the additional synthetic images achieved higher accuracy than the model trained only with real images. We assume that the better results are due to the greater variance provided by the synthetic images, which prevents the model to overfit on the small number of real training images and perform poorly on the test data.

The general trends in deep learning turn toward training in simulation due to the unlimited training data that can be produced and the savings in terms of resources needed to train a model in simulation. Because of this, tools that enable reality gap reduction are expected to play an important role both in research and industry-related applications. The work of this thesis provides such a tool and demonstrates the easy integration and benefits of it in the training process for computer vision tasks. As future work, we want to address several options that can enable the wider application of the tool. For example, an option can be added to change the HDRI background, enabling the users to adjust the background images to their needs. We also consider adding more randomization and general parameters, as well as annotations like 6DOF to enable a wider range of computer vision problems to be addressed. Finally, the current implementation of the tool allows only offline generation of training data useful mostly for supervised learning, while in the future we would like to integrate it directly into a training environment such that it can be useful for a wider range of machine-learning approaches, for example, reinforcement learning.

List of Figures

3.1 A GameObject with different attached components 7
3.2 The general layout of the MVC structure 8
3.3 The tree-like hierarchy of the models 9
3.4 The same texture with different degree of details Top: Used texture maps ltr: Diffuse, Normal, Diffuse & Normal. Bottom: The resulting surface 13
3.5 MenuScene 14
3.6 UI of the EditorScene 15
3.7 EditorScene. Each SceneObject has a SceneObjectView script attached via a component 17
3.8 Complete Implementation of a SceneObjectModel 17
3.9 Activity diagram that visualizes the process during the Item creation 19
3.10 Left: Parameter of an object Right: Shows the used ParameterView and RandomPartViews 21
3.11 Activity diagram that visualizes the selection and deselection process of a SceneObject 22
3.12 Activity diagram that visualizes the process of displaying and hinding the Parameter 24
3.13 Activity diagram that visualizes the running process when a Parameter is changed 25
3.14 The arrows supports the user to position the objects. The highlighted outline visualizes the selected object 27

4.1 Sample images from the T-LESS dataset [Hod+17]. First two rows: Objects used for the training data. Last two rows: sample images of the three sets of real training data as well as some random generated ones. 34
4.2 The smoothed TotalLoss function during the training phase x-axis represents the number of steps done; y-axis represents the loss value The orange graph visualizes the run with only real images while the blue one visualizes the run with both, real and synthetic images 36
4.3 Examples with predicted bounding boxes as well as a label and confidence. Left column: run with only real images, Right column: run with real and synthetic image. The used real images are from the T-LESS data set. 38

List of Tables

3.1 All implemented Parameter types with a short description and its particularities 26

4.1 Accuracy results of the object detection experiment. 36

Bibliography

[eri] eric. Source: https://stackoverflow.com/questions/51905936/unity-function-to-access-the-2d-box-immediately-from-the-3d-pipeline (Accessed 07-09-2020).

[Bro+16] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[DL19] Dany Ayoub, K. M. and Lagarde, S. *Reality vs illusion*. https://blogs.unity3d.com/2019/04/11/reality-vs-illusion/ (Accessed 04.03.2020). Apr. 2019.

[vex] vexe. *Depth Shader*. Source: https://answers.unity.com/questions/877170/render-scene-depth-to-a-texture.html (Accessed 06-09-2020).

[Van] VanKroenen. *Doom Engine*. https://de.wikipedia.org/wiki/Doom-Engine (Accessed 09.03.2020).

[Eve+10] Everingham, M., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. "The pascal visual object classes (voc) challenge". In: *International journal of computer vision* 88.2 (2010), pp. 303–338.

[Wik] Wikipedia. *History of video games*. https://en.wikipedia.org/wiki/History_of_video_games#First_generation_of_home_consoles_and_the_Pong_clones_(1975%E2%80%931977) (Accessed 27-09-2020).

[Mic] Microsoft. *Guid.NewGuid Method*. https://docs.microsoft.com/de-de/dotnet/api/system.guid.newguid?view=netcore-3.1 (Accessed 07-10-2020).

[Hod+17] Hodaň, T., Haluza, P., Obdržálek, Š., Matas, J., Lourakis, M., and Zabulis, X. "T-LESS: An RGB-D Dataset for 6D Pose Estimation of Texture-less Objects". In: *IEEE Winter Conference on Applications of Computer Vision (WACV)* (2017).

[Dun15] Dunstan, J. *A Model-View-Controller (MVC) Pattern for Unity*. https://www.jacksondunstan.com/articles/3092 (Accessed 06-07-2020). 2015.

[Jos+18] Josifovski, J., Kerzel, M., Pregizer, C., Posniak, L., and Wermter, S. "Object detection and pose estimation based on convolutional neural networks trained with synthetic data". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 6269–6276.

[Jul+18] Juliani, A., Berges, V.-P., Vckay, E., Gao, Y., Henry, H., Mattar, M., and Lange, D. "Unity: A general platform for intelligent agents". In: *arXiv preprint arXiv:1809.02627* (2018).

[Kem+16] Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. "Vizdoom: A doom-based ai research platform for visual reinforcement learning". In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2016, pp. 1–8.

[Lev+18] Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., and Quillen, D. “Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection”. In: *The International Journal of Robotics Research* 37.4-5 (2018), pp. 421–436.

[LJ02] Lewis, M. and Jacobson, J. “Game engines”. In: *Communications of the ACM* 45.1 (2002), p. 27.

[Lin+14] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. “Microsoft coco: Common objects in context”. In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[08] *List of Unreal Engine games*. https://en.wikipedia.org/wiki/List_of_Unreal_Engine_games (Accessed 04.03.2020). 2008.

[Uni] Unity-Technologies. *ML-Agents Toolkit Overview*. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md (Accessed 10.09.2020).

[ten] tensorflow. *Model GitHub*. https://github.com/tensorflow/models/tree/master/research/object_detection/models (Accessed 04.06.2020).

[Dum] Dummiesman. *ObjectLoader*. Source: https://assetstore.unity.com/packages/tools/modeling/runtime-obj-importer-49547 (Accessed 20-08-2020).

[PGB12] Paul, P. S., Goon, S., and Bhattacharya, A. “History and comparative study of modern game engines”. In: *International Journal of Advanced Computed and Mathematical Sciences* 3.2 (2012), pp. 245–249.

[McD18] McDermott, W. *THE PBR GUIDE*. https://academy.substance3d.com/courses/the-pbr-guide-part-2 (Accessed 12-10-2020). 2018.

[Qiu+17] Qiu, W., Zhong, F., Zhang, Y., Qiao, S., Xiao, Z., Kim, T. S., Wang, Y., and Yuille, A. “UnrealCV: Virtual Worlds for Computer Vision”. In: *ACM Multimedia Open Source Software Competition* (2017).

[Nol] Nolet, C. *Quick Outline*. https://assetstore.unity.com/packages/tools/particles-effects/quick-outline-115488#content (Accessed 10-07-2020).

[Sha] Shah, V. *Reasons Why Unity3D Is So Much Popular In The Gaming Industry*. https://medium.com/@vivekshah.P/reasons-why-unity3d-is-so-much-popular-in-the-gaming-industry-705898a2a04 (Accessed 09.03.2020).

[Ren+15] Ren, S., He, K., Girshick, R., and Sun, J. “Faster r-cnn: Towards real-time object detection with region proposal networks”. In: *Advances in neural information processing systems*. 2015, pp. 91–99.

[KK19] Kelly, A. and Kelly, K. *Unity Image Segmentation*. Source: https://www.immersivelimit.com/tutorials/unity-image-segmentation (Accessed 05-09-2020). 2019.

[Sit+17] Sita, E., Horváth, C. M., Thomessen, T., Korondi, P., and Pipe, A. G. “Ros-unity3d based system for monitoring of an industrial robotic process”. In: *2017 IEEE/SICE International Symposium on System Integration (SII)*. IEEE. 2017, pp. 1047–1052.

[Ric] Ricardo Rodrigues, G. G. *Unity Standalone File Browser*. Source: https://github.com/gkngkc/UnityStandaloneFileBrowser (Accessed 15-08-2012).

[Tob+17] Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. “Domain randomization for transferring deep neural networks from simulation to the real world”. In: *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2017, pp. 23–30.

[Dav] David-Helgason. “Release of Unity”. In: (). https://forum.unity.com/threads/unity-1-0-is-shipping.56/ (Accessed 09.03.2020).

[AT20] Adam Crespi Cesar Romero, J. H. and Thaman, A. “Use Unity’s perception tools to generate and analyze synthetic data at scale to train your ML models”. In: (2020). https://blogs.unity3d.com/2020/06/10/use-unitys-perception-tools-to-generate-and-analyze-synthetic-data-at-scale-to-train-your-ml-models (Accessed 10.09.2020).

[Kol] Kolambe, H. *Why is Unity (Game Engine) so successful?* https://www.quora.com/Why-is-Unity-Game-Engine-so-successful (Accessed 09.03.2020).

[Zha+15] Zhang, F., Leitner, J., Milford, M., Upcroft, B., and Corke, P. “Towards vision-based deep reinforcement learning for robotic motion control”. In: *arXiv preprint arXiv:1511.03791* (2015).