Escuela
Politécnica
Superior

Synthetic Data Generation for Deep Learning-based Semantic Segmentation

Bachelor's Thesis

Degree in Computer Engineering

Bachelor's Thesis

Author:
Álvaro Jover-Álvarez

Advisors:
José García-Rodríguez, Alberto García-García,
Pablo Martínez-González

June 2019

Universitat d'Alacant
Universidad de Alicante

UNIVERSITY OF ALICANTE

BACHELOR'S THESIS

Synthetic Data Generation for Deep Learning-based Semantic Segmentation

Author:
Alvaro JOVER-ALVAREZ

Supervisor:
Dr. Jose
GARCIA-RODRIGUEZ

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Bachelor's Degree in Computer Engineering
Department of Information Technologies and Computing

May 30, 2019

Declaration of Authorship

I, Alvaro JOVER-ALVAREZ, declare that this thesis titled, "Synthetic Data Generation for Deep Learning-based Semantic Segmentation" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Optimal."

Alberto Garcia-Garcia

UNIVERSITY OF ALICANTE

Abstract

University of Alicante
Department of Information Technologies and Computing

Bachelor of Science

Synthetic Data Generation for Deep Learning-based Semantic Segmentation

by Alvaro JOVER-ALVAREZ

The semantic segmentation of a scene is one of the basic components towards the total understanding of this scene that make up a robotic perception system. Currently, systems based on deep learning, specifically convolutional networks, dominate the state of the art with highly accurate results. However, these systems rely on datasets of unprecedented scale and variability in order to properly generalize into the potentially infinite number of situations in which they can be deployed. Current datasets often have problems in achieving this scale and variability as they rely on human operators both for the capture of the data itself and for its labelling, which is essential for this type of supervised learning techniques. The high cost in time and resources of this task makes it difficult to obtain large-scale and highly representative data sets for specific situations.

In this work we propose the exploration of photorealistic synthetic data as a source to train new systems, to improve the capacity of generalization of those already trained with real data or to facilitate training when a small amount of them is available. To do this we will resort to Unreal Engine 4 to create UnrealROX[1] with the objective of generating an extremely photorealistic data set. We will implement a series of tools to generate this data by creating a simulator capable of doing this work.

[1] https://github.com/3dperceptionlab/unrealrox

Acknowledgements

This project would not have been possible without the direct collaboration of the Department of Information Technologies and Computing (DTIC) of the University of Alicante of the Faculty of Computer Science. I would like to thank my tutor Jose Garcia-Rodriguez and co-tutors for the great show of interest in the work done. I would especially like to thank Pablo Martinez-Gonzalez and Albert Garcia-Garcia for the continuous revisions and the constant perseverance to get an optimal and well-done project.

In addition, I would like to thank the department for giving me the opportunity to work on such a significant project as UnrealROX.

List of Acronyms

CDO Class Default Object

UE4 Unreal Engine 4

HUD heads-up display

CUDA Compute Unified Device Architecture

IDE integrated development environment

SSD Solid State Drive

HDD Hard Disk Drive

SSHD Solid State Hybrid Drive

GPU graphics processing unit

6DOF 6 degrees of freedom

FSM finite state machine

VR virtual reality

FPS frames per second

JSON JavaScript Object Notation

RGB Red Green Blue

PBR Physically Based Rendering

DLSS Deep Learning Super Sampling

BVH Bounding Volume Hierarchy

GI Global Illumination

GAN Generative Adversarial Networks

SDR Structured Domain Randomization

DR Domain Randomization

VKITTI Virtual KITTI

ADR Active Domain Randomization

RL Reinforcement Learning

SVPG Stein Variational Policy Gradient

URDF Universal Robotic Description Format

FABRIK Forward And Backward Reaching Inverse Kinematics

NDDS NVIDIA Deep learning Dataset Synthesizer

DTIC Department of Information Technologies and Computing

UCV UnrealCV

THOR The House Of inteRactions

MINOS Multimodal Indoor Simulator

COMBAHO COMe BAck HOme

Contents

List of Figures

List of Tables

Dedicated to my parents, who supported me through this adventure.

Chapter 1

Introduction

The first chapter describes the main topic of this work. This chapter is divided in four sections: Section 1.1, where we will see an overview of the complete thesis. Section 1.2, where we will explain the motivation of this work. Then Section 1.3, describes the main objectives of this work. And finally, section 1.4 delineates the main structure of this thesis.

1.1 Overview

In this work we propose the exploration of photorealistic synthetic data generation as the main source of data to train deep learning architectures for various computer vision tasks (object detection, image segmentation, object pose estimation...). For that, we propose UnrealROX, a simulator able to generate ground truth employing artificial data. To do this we will use UE4 with the objective of generating an extremely photorealistic data set. This simulator covers different unsolved problems of the current state of the art and opens future exploration avenues thanks to the flexibility of its systems.

1.2 Motivation

This document represents the knowledge acquired both inside and outside the degree of Computer Engineering at the University of Alicante.

This thesis arose due to a collaboration with the DTIC department, specifically with the *3D Perception Lab* group. The main research area of this group focuses on the intersection of Machine Learning (Deep Learning specifically), 3D Computer Vision, and graphics processing unit (GPU) Computing.

This collaboration addresses several projects, one of which was presented at the IROS technology conference in 2018, *The RobotriX: An eXtremely Photorealistic and Very-Large-Scale Indoor Dataset of Sequences with Robot Trajectories and Interactions* [3]. RobotriX is funded by *Ministerio de Economia y Competitividad* of the Spanish Government as part of the project COMe BAck HOme (COMBAHO): system for enhancing autonomy of people with acquired brain injury and dependent on their integration into society (TIN2016-76515-R), and counts with the participation of Jose Garcia-Rodriguez and Miguel Angel Cazorla- Quevedo as main researchers, both being professors at the University of Alicante. UnrealROX is the main tool used to generate the data of such dataset, which will be detailed later in this document.

Personally, I believe that generating synthetic data to train new systems is a necessity, since there is a current human dependency to collect data for datasets. Game engines with advanced graphics capabilities can accelerate this task by working with synthetic data. In order to achieve this, there is the need to create a framework capable of recording and generating synthetic data, that is why UnrealROX was born.

UnrealROX was already created when I entered the RobotriX project. It was my task to coordinate the main refactor of the source code in order to improve its flexibility, effiency, and to bring it into a production state; progressively porting it to C++ and providing advice to the rest of the team members based on my previous experience with UE4.

The main purpose of this project is to create a simulator that will serve to reduce the gap between synthetic and real data. Future works would have to demonstrate that the simulator is capable of generating samples that are easily transferred to a real-world domain. My main objective on this project is to ensure an efficient framework able to record at a minimum framerate of 60+ frames per second (FPS) and generate data as efficiently as possible since we are going to deal with huge amounts of data.

One proposal to test the performance of UnrealROX simulator is binary categorization by using UnrealGrasp [13], which allows us to grasp dynamic objects from a scene, extending it to classify when a user is touching or not a particular object. Therefore, the problem would be divided into two major classes, interaction and non-interaction. Once the dataset is obtained, we can train an architecture able to classify within this synthetic data both situations. The next step would be testing this same architecture with real data to see how it performs.

(A) *Unreal Paris 2018* by Dereau Benoit.

(B) *Archviz for UE4* by SOA Academy.

(C) *Summer House Archviz Project* by Ervin Jesse.

(D) *Post soviet bathroom* by White Noise Team.

FIGURE 1.1: Synthetic scenes on UE4

Thanks to this work, we will be able to prove that synthetic datasets, such as *The RobotriX* [3], are valid for real-world problems, which means that the gap between real data and synthetic data (as seen in Figure 1.1) will be even smaller.

1.3 Objectives

In this section we are going to define the main objectives of this project. To do this, we must first describe the state of the system prior to this work in Section 1.3.1. Then, Section 1.3.2 will describe the main objectives once seen the initial architecture and the process we followed to achieve the completion of the extensible framework.

1.3.1 Initial architecture

UnrealROX was designed to compensate the main absences of the current state of the art in terms of simulators. The first version of the environment consisted on a UE4 project with a premature grasping system (UnrealGrasp [13]) that allowed an operator to interact with different objects in the scene by controlling an interchangeable 3D agent using the Oculus Rift headset and controllers. The project also allowed to record the state of the scene and the movements of the operator with a primitive version of our Tracker Actor, documented in Section 4.7.

FIGURE 1.2: Some samples of the RobotriX dataset.

UnrealROX had a single camera embedded in the head of the controlled agent, this camera would be used to generate data for our dataset. In order to create said data, the simulator would reproduce the recorded movements offline to generate ground truth and raw data making use of UnrealCV [15] to capture the scene from a specific camera. The weight of the algorithm fell on a prototype made in Python that made use of UnrealCV with an HTTP server, as seen in Figure 1.3, and was therefore slow and limited to the features of the plugin. This first version of the project was the one used to generate The RobotriX[1] [3] dataset (see Figure 1.2), which motivated the continuity of the simulator.

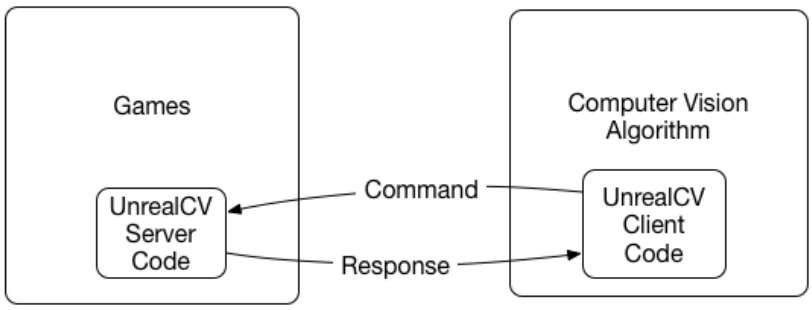


FIGURE 1.3: Architecture of UnrealCV. Figure from the documentation of UnrealCV[2].

[1] https://github.com/3dperceptionlab/therobotrix

1.3.2 Final version

Once the architecture of UnrealROX had been studied, a series of objectives were set to make the environment more flexible and efficient. One of the first priorities was to add support for on boarding cameras, since most robots in the public market integrate multiple cameras in different parts of their bodies (see Section 4.2). In addition, we implemented external cameras to provide the robot with information that it is not able to perceive directly.

FIGURE 1.4: Multi-camera setup on UnrealROX.

Adding more cameras increased considerably the computation time when we executed the algorithm to generate data, which meant that we needed to do something regarding the generation algorithm. The final decision was to get rid of the client Python application to communicate with the UnrealCV server, executing everything locally. This decreased considerably the execution time of the algorithm. Then, in order to not bound ourselves to UnrealCV, we decided to create a ground truth generator specific for UnrealROX, which allowed us to have more control over the generated data. We also added a way to select which type of data is relevant for the user, so we only generate the data that the user specifies on the generation settings.

Furthermore, we were able to integrate support for the HTC Vive headset, which was possible thanks to the cooperation of the DTIC department. Adding support for other devices on UE4 is easy thanks to the input system and the flexible API that the engine provides.

Another of the great challenges that UnrealROX posed was to completely refactor the animation system in order to simplify the import of new 3D agents (skeletal meshes), for that, we implemented a custom animation node for the animation system of UE4 that enabled us to bulk a series of bones by name accompanied by the modified transform; this node is the responsible of setting the translation and rotation of all the bones that we record with the Tracker Actor. Another of my responsibilities in the project was to convert all the code to C++ to decrease compute time.

1.4 Thesis Structure

This thesis is organized as follows. In Chapter 1 we give an introduction to the project and its goals. In Chapter 2 we review the current state of the art as well as all the inspirational works that have been instrumental in the creation of this project. Then, in Chapter 3 we describe the methodology followed in this work ranging from all the hardware equipment to each software tool we used. Next, in Chapter 4, we describe every subsystem that composes UnrealROX. And finally, in Chapter 5 we we draw the conclusions to which we have been able to arrive after this work.

[2] http://docs.unrealcv.org/en/latest/reference/architecture.html

Chapter 2

State of the Art

The second chapter reviews the current state of sim-to-real. First, we will explore the two main avenues, photorealism and domain randomization. Then, we will put some of the most relevant simulators in context. And finally, we will present the advantages of UnrealROX.

2.1 Sim-To-Real

One of the main problems when training a semantic segmentation model is the dataset. Specific large-scale sets of images are needed to solve different problems since we need to include a considerable amount of variability and complete ground truth for each one of them. Collecting data that satisfies these requirements is hard due to the main difficulties of capturing and processing this data manually.

Traditional data collection techniques consist on capturing the environment with the right device. For instance, if we want to capture 2D data, an RGB camera is needed. Following next we have to process the data based on the problem requirements, this means that we need to define a set of classes to represent every entity we want to classify and assign an identifier to each class. In the case of 2D Semantic Segmentation, this codification can be done with colors so we would paint every pixel from each captured image with the chosen color for that class. This is a complicated and costly process since it requires human operators to paint the images, which can introduce errors in the dataset due to the resolution and the quantity of classes some of these datasets handle. Figure 2.1 showcases a manually labeled image from the Mapillary Vistas Dataset[1], where each color represents a different entity type.

FIGURE 2.1: Manually labeled scene from the Mapillary Dataset.

[1] https://www.mapillary.com/dataset/vistas?pKey=aFWuj_m4nGoq3-tDz5KAqQ&lat=20&lng=0&z=1.5

The importance of large-scale data when working with data-hungry learning algorithms is critical in order to achieve proper results and generalization. As we saw before, collecting and labeling real-world data is a tedious and costly process partly due to the main complications explained above.

This problem is partially solved thanks to synthetic datasets, which are created without the need for real data. However, for a synthetic dataset to be useful, it must resemble reality as much as possible. Physics simulators like Bullet[2], FleX[3] or PhysX[4], and 3D rendering engines in Unity, UE4, and Physically Based Rendering (PBR) engines have played a substantial role in this process.

To recapitulate, synthetic datasets are sets of computer generated or enhanced images for which ground truth can be extracted automatically without the need for human intervention, which alleviates the laborious and complex task of collecting real world data.

In this section, we will document sim-to-real and see the different influxes that dominate the state of art. In section 2.1.1, we explain how photorealism has been a very important factor when generating synthetic datasets, as well as how some modern tools, such as ray-tracing, have improved the rendering pipeline to achieve more photorealistic results. In section 2.1.2 we describe the alternative to photorealism, Domain Randomization (DR), and we will expose some related works. Section 2.1.3 describes different simulators that have helped on this matter. Then, Section 2.1.4 showcases UnrealCV (UCV) and NVIDIA Deep learning Dataset Synthesizer (NDDS), two generators able to generate ground truth given a dataset. And finally, Section 2.1.5 will put UnrealROX, our sim-to-real simulator, in context.

2.1.1 Photorealism

As we commented previously, a synthetic dataset is useful when it is capable of significantly resembling reality in a considerable manner. In this regard, the use of photorealistic rendering solutions is a necessity.

The state of the art in relation to photorealism is evolving vertiginously. One of the biggest advancements today is the inclusion of real-time ray-tracing technology in NVIDIA GeForce RTX[5] graphics cards. Ray-tracing allows to simulate the physical behavior of light to provide a cinematographic quality rendering in real time. This implies a great advance in the field of real-time rendering, since real-time ray-tracing has not been possible until the current date. NVIDIA RTX technology introduces some new features to the graphics pipeline:

- **Ray-Triangle Intersection:** This technique lets us decide what to do when a ray is shot into a scene. It is up to the user to determine if it intersects with geometry and what to do with it regarding reflections, occlusion, shadows and things of that nature.

- **Bounding Volume Hierarchy (BVH):** A BVH is a tree-based data structure that contains multiple hierarchically-arranged bounding boxes that surround different amounts of scene primitives (see Figure 2.2). In this type of data structure each ray only needs to be tested against the BVH using a depth-first tree traversal process instead of against every primitive in the scene, this makes doing high-performance ray tracing into a scene in real time possible.

[2] https://pybullet.org/wordpress/
[3] https://developer.nvidia.com/flex
[4] https://www.geforce.com/hardware/technology/physx
[5] https://www.nvidia.com/en-us/geforce/20-series/rtx/

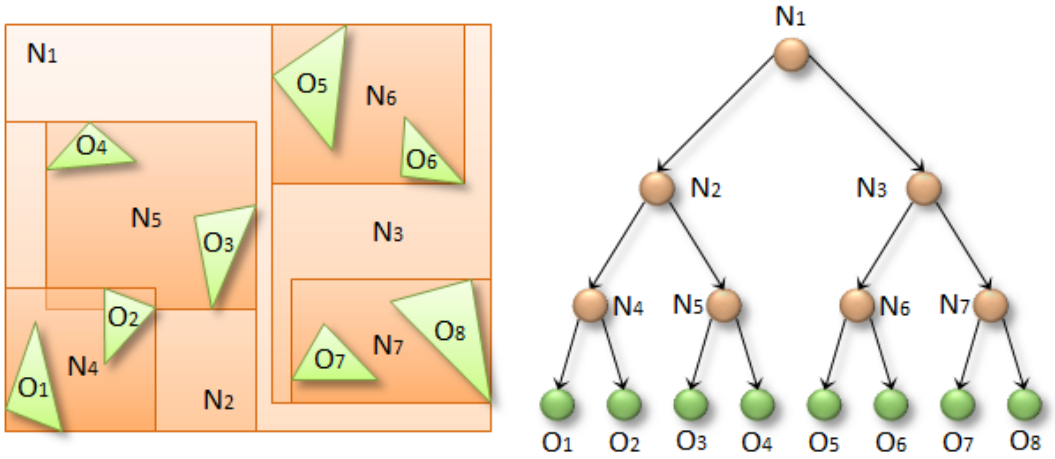


FIGURE 2.2: Visual representation of a BVH data structure. Source: *NVIDIA Devblogs*[6].

- **Denoising Filtering:** Ray tracing results in a grainy image for real time solutions. Denoising filtering can produce high fidelity images from ray tracers that appear visually noiseless, as we can see in Figure 2.3. Denoisers need to interpret and merge these pixels in the most appropriate way, that is why many approaches do not work quite well in motion, since pixel caching is not a possibility. Some denoising implementations choose to blur the final result by design to eliminate any kind of artifact, this is the case of Metro Exodus Global Illumination (GI) Ray Tracing approach, which does not take into account normal maps, just stores the scene data in a voxel grid as spherical harmonics in world space which encodes some color and directional properties[7].

FIGURE 2.3: Denoise filter applied over a real time ray-traced image. Image from: *What's the Latest? DirectX and the New Rise of Ray Tracing*[8].

The RTX graphics cards also feature a new technology called Deep Learning Super Sampling (DLSS) that uses artificial intelligence to generate crisp images while running up to 2 times faster than previous generation GPUs using conventional anti-aliasing techniques. In short, DLSS render pass, renders the scene at a lower resolution and then uses an AI algorithm to make it look as rendered at a higher one, but without the overhead of rendering it at that resolution. Thats where the performance enhancement comes from, and ideally, where the maintained visual quality does too. Figure 2.4 displays a comparison between DLSS enabled and disabled.

[6] https://devblogs.nvidia.com/thinking-parallel-part-ii-tree-traversal-gpu/
[7] https://www.eurogamer.net/articles/digitalfoundry-2019-metro-exodus-tech-interview
[8] https://www.youtube.com/watch?v=476N4KX8shA

(A) Metro with DLSS disabled.

(B) Metro with DLSS enabled.

FIGURE 2.4: DLSS provides better images without using as much processing power of the graphics card as traditional anti-aliasing techniques.Images are courtesy of Patrick Glynn from Bubble Pony INC.

However, the final output image that DLSS produces is far from perfect, since the super-sampling algorithm generates blur in some cases, which is not very ideal for a photorealistic setup. Added to the fact that DLSS is an NVIDIA exclusive technology that requires new hardware and compatible software as of today.

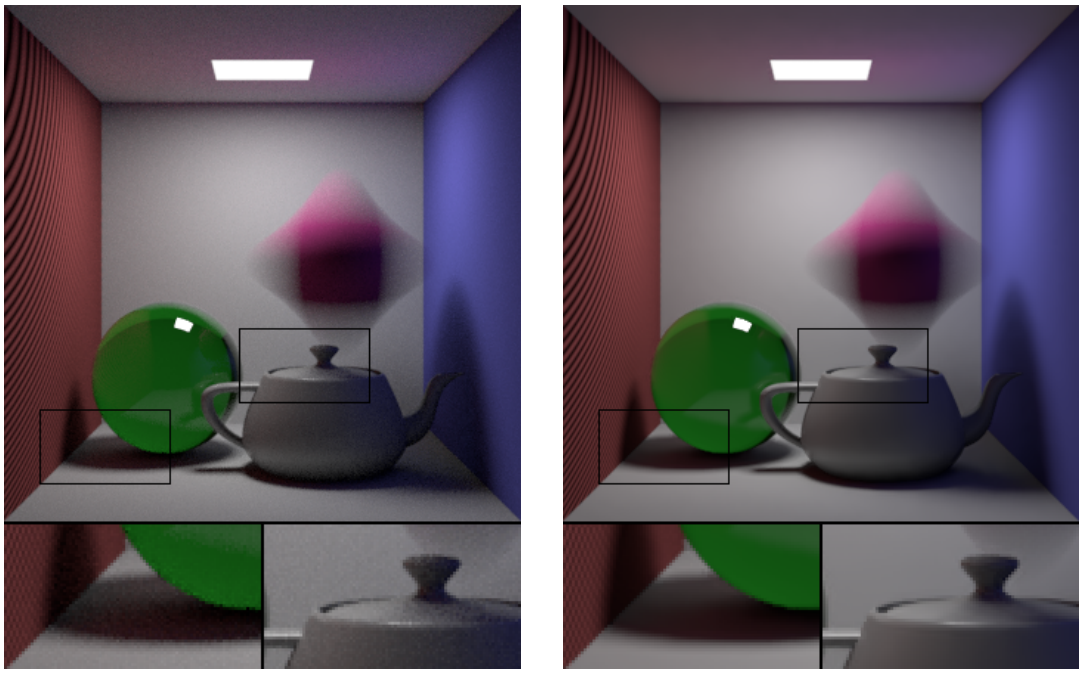
Even with all these inconveniences, real-time ray tracing is usable to generate photorealistic synthetic datasets, since the quality of the output images is decent enough. Selective ray tracing is a technique that DICE has developed and demonstrated on Battlefield 5[9]. This approach allows them to know which surfaces have reflective properties, so they trace more rays from the most reflective surfaces, and less rays from the less reflective ones. Which allows them to have ray tracing reflections where it matters, which lets them deliver a really great result, while still delivering a good frame rate.

To sum up, if it is desired to use real time ray tracing in its greater potential as of today, it is necessary to make an intelligent use of the technology. Nowadays, our hardware resources do not allow us to trace an infinite number of rays in real time, that is why hybrid solutions must be considered.

[9] https://www.ea.com/es-es/news/battlefield-5-real-time-ray-tracing

We have talked about hyper-realistic rendering in real time and how for the present time, these solutions are limited to high end devices even using advanced optimization techniques. However, offline solutions are more affordable for the average consumer since the main trade-off is rendering time. These systems take longer to compute because they are looking for the most accurate result. In this case, instead of having a hybrid ray-tracing approach in real time to get high resolution and good frame-rate, we focus on achieving the same but offline. This means that we can multiply significantly the amount of rays we trace, meaning that each frame will take a considerable amount of time to render.

Although we increase the number of rays per pixel, it is necessary to apply a denoising filter over the scene to hide minimum artifacts that have been produced. Obviously this denoising will not have to assume as much information as an aggressive denoiser and will be able to return more precise results as we can see in Figure 2.5 taken from the RenderMan[10] render system.



(A) 96 rays per pixel, before denoising. (B) After denoising.

FIGURE 2.5: RenderMan denoising algorithm created by Walt Disney Animation Studios and Disney Research.

Offline rendering is used in animation films, where they can afford to trace billions of rays per frame. However, rendering a single frame using this technique is very slow, that is why film studios, such as PIXAR or LucasFilms, employ rendering farms where they can distribute the work-load and speed up the production of the film. It is clear that there is a relationship between visual quality and computational time, where the user utilizing the rendering engine is the one who decides how many traces make a specific scene look right for the imposed quality requirements.

As we can see, the rendering cost is equivalent to the amount of rays we want to trace. The more rays we add, the more accurate and expensive to render the scene will be. The less rays we add, a more aggressive denoising filter will be needed, and as we said previously, an aggressive denoiser decreases the overall quality of the output image.

[10] https://renderman.pixar.com

One of the main problems when it comes to render hyper-realistic scenes is resolution. If we take a modern high-budget animated ray-traced film as an example, we could speak of an average of 1,584 rays per pixel. That would be 3,284,582,400 rays for a single 1920 x 1080 image, which makes 13,138,329,600 rays for a 4K frame, without having any lights that add more complexity to it (this data has been extracted from Arnold render system[11] on production quality).

The problem is evident, the computational cost increases for the same amount of rays per pixel the higher the desired resolution is. We spoke earlier about NVIDIA's DLSS technology, which upscales the input image using Artificial Intelligence. This technique suggests to render first at a lower resolution, which means that we trace less rays (maintaining the same amount of rays per pixel), to then post-process the image upscaling it. NVIDIA has demonstrated that this technique is applicable in real time with DLSS [12]; however, the algorithm is not available to the public as of today.

On the other hand, a paper presented in 2017, proposes the use of Generative Adversarial Networks (GAN) [8] to solve this problem. In this paper they discuss and compare the performance of SRResNet and SRGAN to NN, bicubic interpolation, and four state-of-the-art methods [5]. The idea is similar: rendering the image at a lower resolution (off-line or on-line), and then post-process it to achieve the desired resolution. Figure 2.6 represents the distinction between different upscaling methods in a practical way.

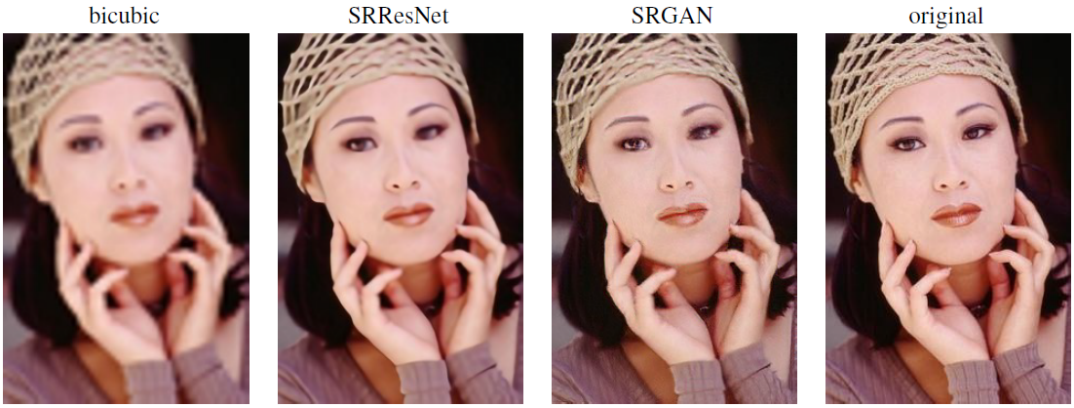


FIGURE 2.6: Results of a sample from **Set5** using bicubic interpolation, SRResNet and SRGAN. [4× upscaling]. Image taken from the paper referenced.

The main objective of that work is to improve the overall quality of the target images rather than computational efficiency. However, they prove that shallower networks have the potential to provide very efficient alternatives at a small reduction of qualitative performance, nonetheless, this comes at the cost of longer training and testing times.

This type of architecture consists of two neural networks that compete with each other in a zero-sum game framework. The generative network generates candidates while the discriminative network evaluates them. The generative network training objective is to increase the error rate of the discriminative network [17]; this is why GANs deliver a powerful environment for generating realistic looking natural images with high perceptual fidelity.

[11] https://docs.arnoldrenderer.com

In conclusion, the best rendering technique in terms of photorealism out of the traditional ones[12] [25] is non-hybrid ray-tracing; however, this method comes with the disadvantage of a very great performance cost if it is desired to render at a high resolution with a great sample-per-pixel rate. In order to circumvent this issue, the current state of the art is trying to optimize the process using upscaling techniques; however, these approaches affect the quality of the final image. Nonetheless, they can provide accurate enough output images to represent a close-to photorealistic setup. In conclusion, it is conceptually impossible to get a lossless output using these methods, but thanks to this problem, upscaling techniques have evolved to a point where the upscaled result is very close to the original image, hence we could consider that getting close to a photo-realist environment should be the main consideration when creating a photorealistic synthetic dataset.

2.1.2 Domain randomization

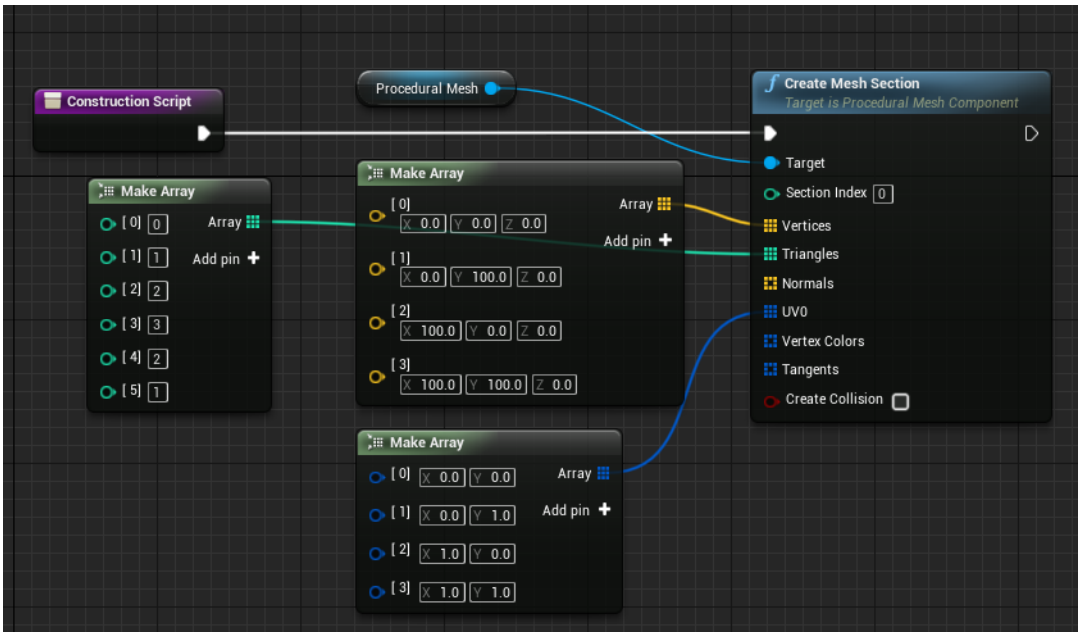
As we said in the introduction of this section, domain randomization is the main alternative to photorealism. Domain randomization is a technique for training models with synthetic data that gets generated by randomizing input in a simulator (meshes, materials, lighting...). With enough variability, the real world may appear to the model as just another variation [22]. There are a lot of approaches to generate synthetic data to construct a domain randomized dataset. The simplest approach is object randomization. If we take NVIDIA's *Isaac Sim*[13] domain randomization plugin as an example, displayed in Figure 2.7, we can study simple randomization techniques:

- **Randomize Meshes:** Each class has an array of possible meshes. The selected mesh gets determined thanks to the seed mechanism included in the plugin used to initialize a pseudorandom number generator.
- **Randomize Lighting:** Randomizing how lighting behaves on a specific scene is crucial if it's desired to achieve a data-set able to perform great at different lighting conditions. This Plugin allows the user to tweak how the light looks.
- **Randomized Materials:** Aside randomizing meshes and lighting, we can take existing meshes and modify their materials to increase the samples of a single class without the need of modeling a new object of the same type.

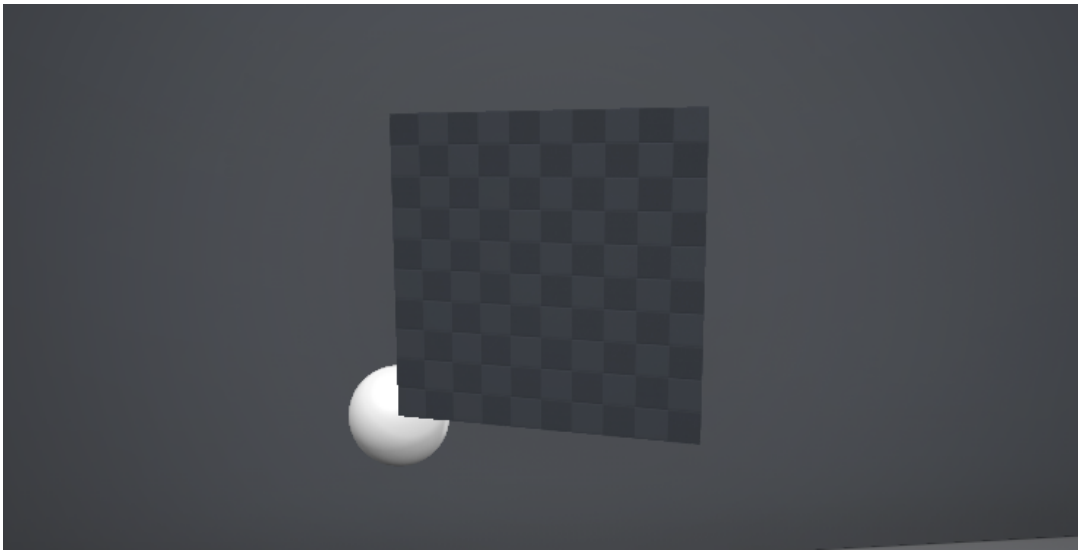
FIGURE 2.7: Isaac Sim Randomized scene.

[12] https://en.wikipedia.org/wiki/Rendering_(computer_graphics)#Techniques
[13] https://docs.nvidia.com/isaac/isaac_sim/index.html

Another interesting approach not observed on this project, is generating and deforming meshes in runtime. UE4 provides a powerful set of tools to generate and edit meshes in real time[14]; however, the main complication when using these tools is that we need to work with fully procedural content, meaning that we have to parametrize everything to create objects that make sense and belong to our domain. Figure 2.8 shows a mesh created using these tools.



(A) Blueprint graph to generate a simple squared plane with a procedural mesh component.



(B) Result of the parameters applied on Figure 2.8a.

FIGURE 2.8: Using UE4's procedural mesh component to generate a mesh in real time.

Defining a mesh in runtime is costly and complicated, that is why it is not considered on most of the cases. However, deforming a mesh could help for future non-rigid objects deformations, which is one of the current limitations of most of the grasping simulators, like UnrealGrasp [13], included on UnrealROX.

[14] https://api.unrealengine.com/INT/BlueprintAPI/Components/ProceduralMesh/index.html

On October of 2018, NVIDIA's research group published a paper that presents Structured Domain Randomization (SDR) as an alternative of DR. In contrast to DR, which places objects and distractors randomly according to a uniform probability distribution, SDR places objects and distractors randomly according to probability distributions specific to the presented problem [14]. The main strength of SDR is that takes into account the context and structure of the scene; hence the parameters exposed for the random system to modify are constraint to real world possible data. Meaning that we won't have nonsensical randomizations as the one exposed in Figure 2.9 for the Virtual KITTI (VKITTI) dataset [2].

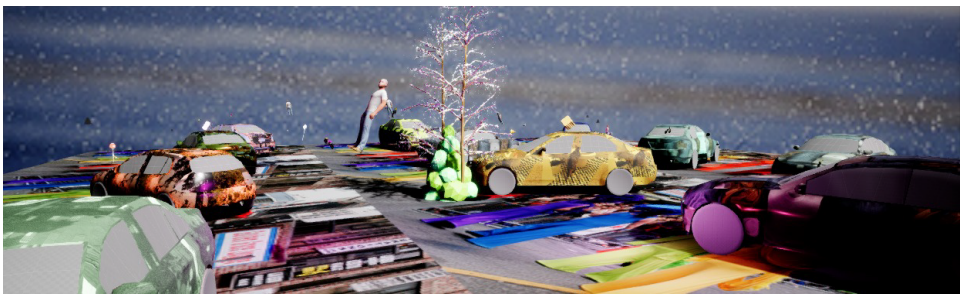


FIGURE 2.9: Domain randomization intentionally avoids photorealism for variety, which leads to weird generated synthetic data. Image extracted from the same paper.

SDR strikes a balance between photorealism and domain randomization, producing images that are realistic in many respects but nevertheless exhibit large variety. SDR focuses its attention on improving the VKITTI dataset with sensical domain randomization; for that, a scenario is chosen at random, then global parameters (road curvature, lighting, etc.),which cause context splines (road lanes, sidewalks, etc.) to be generated, upon which objects (cars, pedestrians, cyclists, houses, buildings, etc.) are placed. Figure 2.10, shows a comparison between different synthetic datasets used for training object detection models.

FIGURE 2.10: Synthetic data coming out from similar datasets. Image extracted from the same paper.

SDR outperforms the GTA-based synthetic data of Sim 200k [6], because SDR provides more variability in the geometry of the scenes. VKITTI is a replica of the KITTI dataset [4] so it is highly correlated with KITTI, which means not enough variability. Also, DR generates nonsensical data as commented above. According to the authors, synthetic SDR data combined with real KITTI data outperforms real KITTI data alone, thanks to the context based generation.

As we mentioned before, domain randomization sacrifices realism to include a lot of variety in its samples. SDR partially solves this problem by manually controlling the generation parameters with a predefined set of scenarios.

Bhairav Mehta et al. proposed on April of 2019 Active Domain Randomization (ADR) [10], a type of DR that looks for randomized environments that maximize utility for the agent policy within a given randomization range. The task of searching randomized environments is casted as a Reinforcement Learning (RL) problem [20], where the samples get parameterized using Stein Variational Policy Gradient (SVPG), which balances exploitation and exploration [9]. ADR method learns an adaptive randomization strategy that finds problematic environments within the given randomization ranges. They found that training on these instances led to better agent generalization. Also, according to the authors, ADR can provide insight into which dimensions and parameter ranges are most influential, which can aid the tuning of randomization ranges before expensive experiments are undertaken. Figure 2.11, displays the main functioning scheme of Active Domain Randomization.

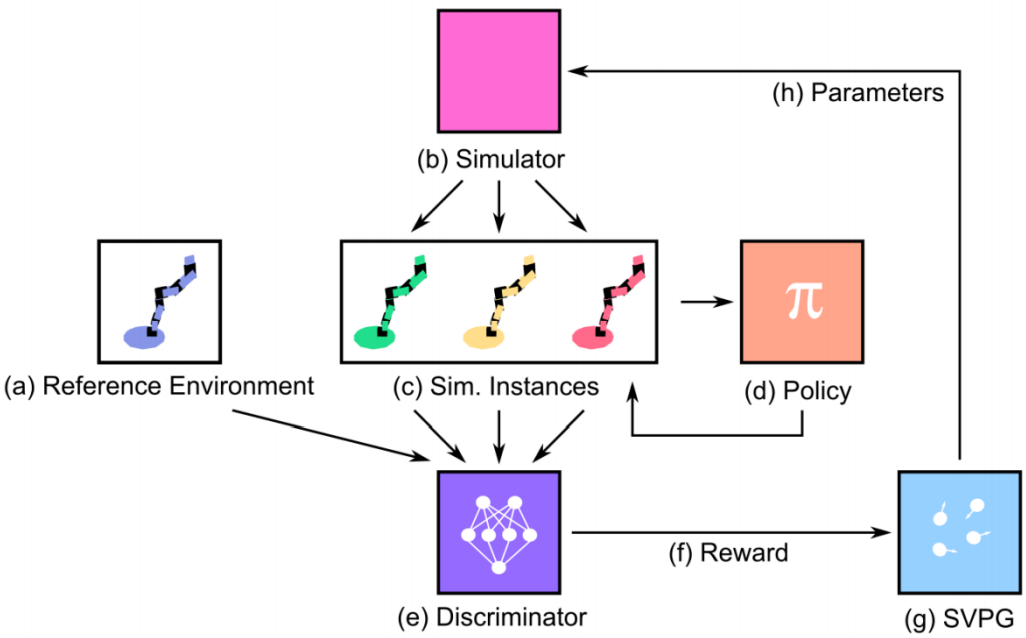


FIGURE 2.11: "ADR proposes randomized environments (c) or simulation instances from a simulator (b) and rolls out an agent policy (d) in those instances. The discriminator (e) learns a reward (f) as a proxy for environment difficulty by distinguishing between rollouts in the reference environment (a) and randomized instances, which is used to train SVPG particles (g). Enforced through the SVPG formulation, the particles propose a diverse set of environment dynamics, and try to find the environment parameters (h) that are currently causing the agent the most difficulty" (Bhairav Mehta et al.).

The main issue of ADR is that the reward calculation and discount factor need to be defined manually for each proposed environment due to the RL approach. However, this solution expedites the overall process of obtaining a randomized set that works for a specific problem by automating the parameter lookup.

2.1.3 Simulators

Traditionally, computer vision problems are approached using real world datasets. However, most of the current vision problems need to develop perception models for agents that are physically active in the world, like robots. Commercial datasets are passive and cannot achieve this task, since the content or camera location cannot change according to agents actions. This issue can be solved by placing a physical agent in a world with a series of on-board cameras. The main trade-off of this approach is that learning speed is bounded to real time; also, important critical events, like a car crash, cannot be freely reproduced; plus the fact that working with hardware can become very tedious and expensive. An alternative approach is learning in simulators; however, we encounter two new challenges about generalization:

- **Photorealism:** In Section 2.1.1 we documented some of the main issues photorealism has as of today. However, these problems can be alleviated as hardware and computer graphics get better followed along the improvement of domain adaptation methods.

- **Semantic distribution mismatch:** The models used in simulators are usually hand designed and artificial so they do not really reflect the semantic complexity of real-world. This issue can be improved with domain randomization.

In this section we review some of the most popular simulators that alleviate the commented issues. First, we describe Gibson, followed by AI2-THOR. Next, we discuss how Minos and House3D have been relevant on the current state of the art.

On August 2018, Fei Xia et al. proposed the Gibson Environment [27], a real-world perception simulator for embodied agents. Figure 2.12 displays and describes the Gibson Environment at a glance.

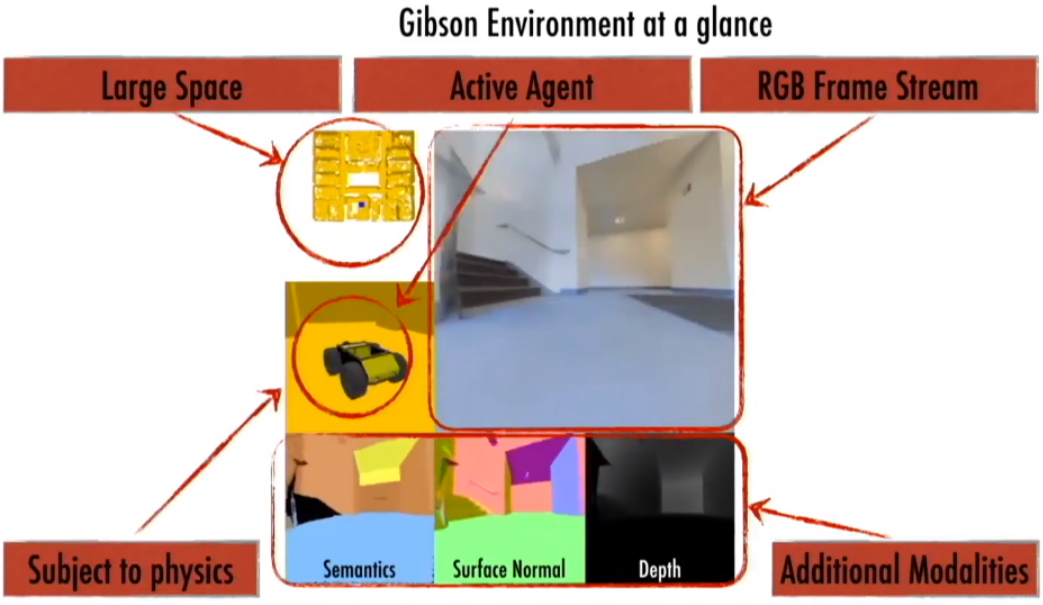


FIGURE 2.12: On this representation, there is an active agent subject to physics constraints simulated by PyBullet (gravity and collision) that can move around in a large space. Said agent receives a stream of RGB frames, captured with a virtual on-board camera, plus some additional modalities like semantics or depth. Image from: *GIBSON ENVIRONMENT: a real-world perception simulator for embodied agents*[15].

Gibson provides a dataset that counts with 572 full buildings, which consist on real spaces scanned with 3D scanners, that can be fully explored on the simulator. Gibson also gives the possibility to import arbitrary agents using the Universal Robotic Description Format (URDF)[16] as we can see in Figure 2.13.

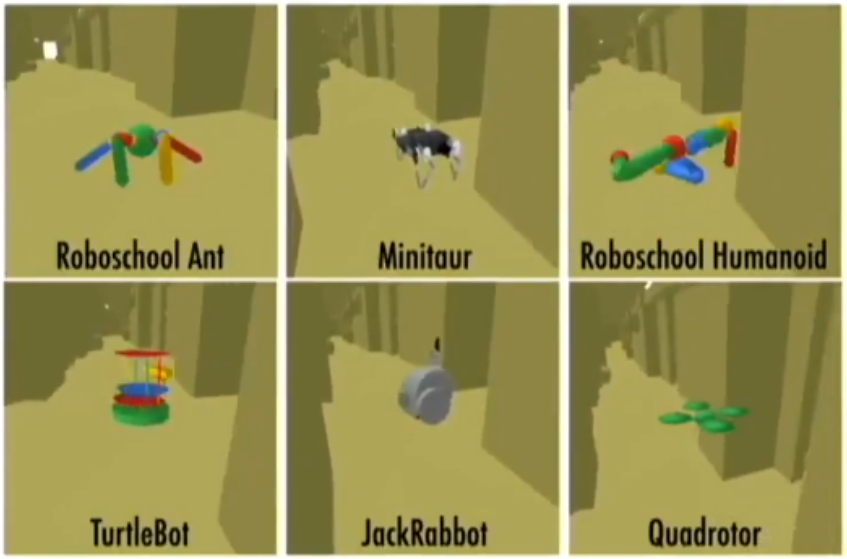


FIGURE 2.13: Multiple agents on Gibson thanks to the import feature.

In Gibson, the physics constraints are enforced by integrating PyBullet3D[17] engine, which encompasses all the collision and gravitational data of the simulator.

In order to provide RGB frames from arbitrary viewpoints, they developed a neural view synthesizer called *Goggles*, that has a baked-in adaption mechanism for transferring to real world. This approach geometrically renders a base image for the target view, which is resorted to a neural network to correct artifacts. Said neural network fills the dis-occluded areas, along with jointly training a backward function for mapping real images onto the synthesized ones, as we can see on Figure 2.14.

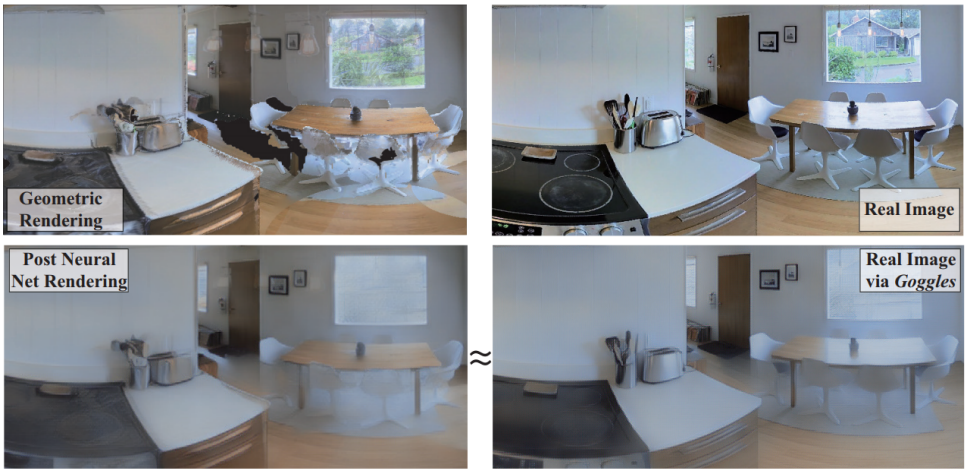


FIGURE 2.14: Goggles can be seen as corrective glasses of the agent. Image source: Gibson paper referenced above.

[15] https://www.youtube.com/watch?v=1T8-PLy5mVo
[16] http://gazebosim.org/tutorials/?tut=ros_urdf
[17] https://pybullet.org/wordpress/

One of the main issues of Gibson is the lack of interaction with the scene, AI2-The House Of inteRactions (THOR) [7] proposes an environment centered mainly in interaction, as we can see in Figure 2.15. AI2-THOR is an open-source visual AI platform that provides a rich actionable 3D environment controlled by a Python interface to communicate with the engine through HTTP commands.

AI2-THOR v1.0 consists of 120 near photorealistic scenes covering four different categories: bedrooms, bathrooms, living rooms and kitchens, within which an agent can interact. Interaction takes various shapes in THOR: the agent can open and close different objects, pick up and place them in various locations, turn on and off lights...

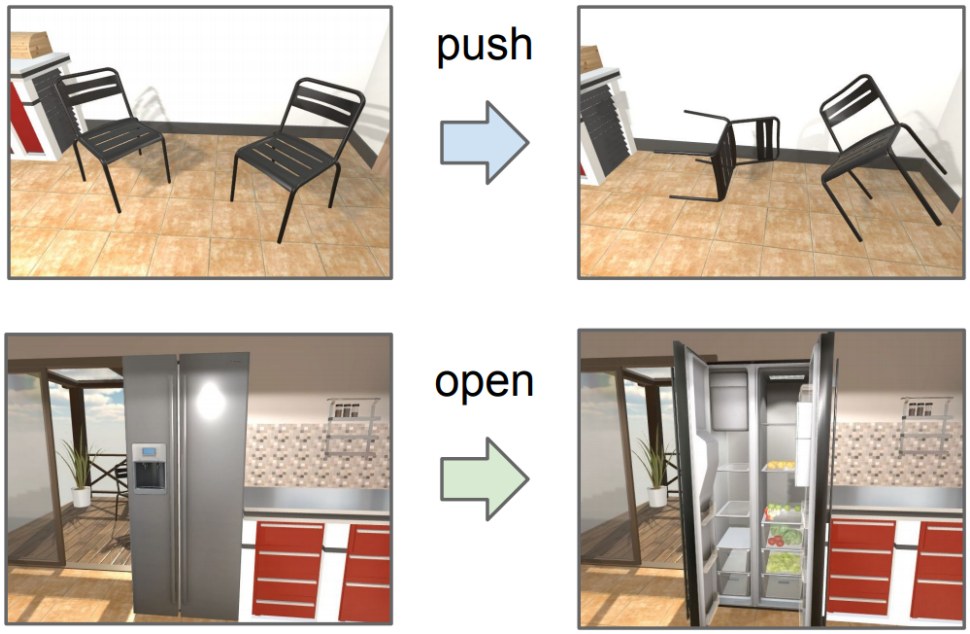


FIGURE 2.15: Interaction example of AI2-THOR. Figure from: *AI2-THOR: An Interactive 3D Environment for Visual AI.*

Photorealism is one of the main features of THOR, since it allows better transfer of learned models to the real world. THOR is built on top of the Unity game engine, which allows for observing pixel level results to actions performed by the agents, as well as support on a wide variety of platforms. Some of the drawbacks of this environment are the absence of a controllable 3D agent, the lack of multiple points of view, and the binary categorization for interactables, e.g., a door can be only open or closed. However, the fact that THOR os a Unity project, makes easier its extension.

Multimodal Indoor Simulator (MINOS) [18] is a simulator for navigation in complex indoor environments. The framework provides access to a large number of realistic synthetic scenes and reconstructed indoor spaces such as SUNCG [19] and Matterport3D [1] respectively. MINOS provides a 3D agent represented by a cylinder proxy geometry with parameterized radius, height and ground offset; able to navigate through the scenes, obtaining (if desired) information from multimodal sensory inputs, including vision, depth, surface normals, contact forces, and semantic segmentation. The simulator provides two client APIs: a Python wrapper designed to support efficient RL, and a web client for interactive exploration and data collection. Figure 2.16 showcases an overview of the MINOS framework and APIs.

However, this simulator lacks some features such as photorealism, a more complex geometry model instead of a cylindrical proxy, configurable points of view, and scene interactions.

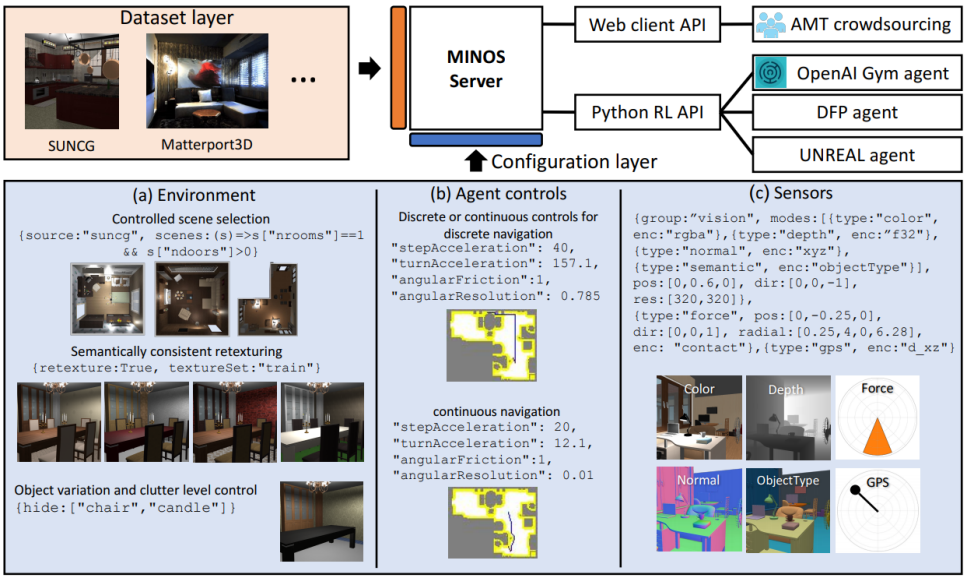


FIGURE 2.16: Overview of the MINOS framework and APIs. Figure from: *MINOS: Multimodal Indoor Simulator for Navigation in Complex Environments*.

House3D [26] is a rich, extensible and efficient environment that contains over 45,000 synthetic 3D scenes of visually realistic houses, equipped with a diverse set of fully labeled 3D objects, textures and scene layouts, sourced from the SUNCG dataset. Each scene in SUNCG is fully annotated containing 3D coordinates, object type and the room these objects are in.

There is a simplistic agent able to access the following data: the visual RGB signal of its current first person view, semantic/instance segmentation masks for all the objects visible in the same view, and depth information (see Figure 2.17).

FIGURE 2.17: House3D agent ground truth. Figure from: *House3D : A Rich and Realistic 3D Environment*.

To render SUNCG scenes they have employed OpenGL, which can run on both Linux and MacOS, and provides RGB images, semantic segmentation masks, instance segmentation masks and depth maps.

Nevertheless, House3D has some disadvantages, such as the lack of a realistic 3D agent, the absence of interactive elements and the non-existence of many perspectives to capture the data.

2.1.4 Generators

When it comes to generate a dataset, not only simulators have been relevant for this task. Generators decouple the ground truth creation from the problem paradigm. A generator consists of a decoupled interface to generate ground truth given synthetic data. These generators work with environments able to hold said samples, such as Unity or UE4. The main difference between a generator and a simulator that generates, is that the generator is agnostic to the problem paradigm, while the simulator can induce data into their generation process, to solve problems such as the semantic interaction proposal mentioned in Section 1.2.

Although generators are agnostic to the problem to resolve and can only be used in the environment for which they were created, they have the great advantage of being decoupled entities. This implies that we will be able to use the generator in any project of the environment for which they have been created without major complications.

In this section we are going to review two generators that have inspired the creation of our simulator, UnrealROX. Concretely, we are going to describe chronologically UCV and NDDS, both working under the UE4 environment.

UCV [15] [16] is a generator for UE4 that extends the engine to create virtual worlds and facilitate communication with computer vision applications. UnrealCV is comprised of two components: server plugin and client application.

- **Server plugin:** The server consists of a UE4 plugin that runs embedded into a UE4 project. It uses the built-in socket system of the engine to listen to UnrealCV commands sent by a client, executing them using UE4's C++ API[18]. For example: change the current view mode.

- **Client application:** The client is a Python application which communicates with the server. It sends commands to the server and waits for a response. All the commands can be consulted in the official documentation of the generator.

UnrealCV generates synthetic images and its ground truth as we can see in Figure 2.18. The generator can produce RGB, object instance mask, depth and surface normal images.

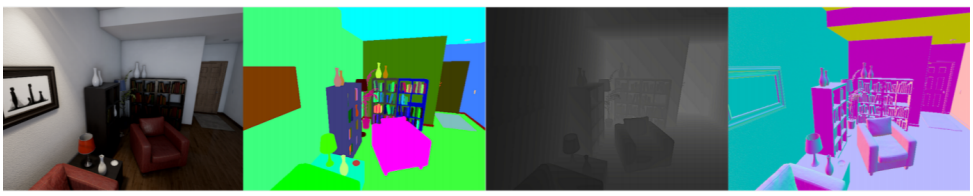


FIGURE 2.18: The room is from the demo RealisticRendering, built by Epic Games. From left to right are the synthetic image, object instance mask, depth, surface normal. Image taken from the UnrealCV paper.

Besides the fact that UnrealCV is an open-source tool decoupled and agnostic to the problem to resolve, it has some inconveniences. The main disadvantage of UnrealCV is the delay induced by the client-server communication. This delay slows down the dateset generation process in spite of gaining a good flexibility thanks to the proposed paradigm.

[18] https://docs.unrealengine.com/en-us/Programming

Thang To et al. released on 2018 NDDS [21], a UE4 based simulator which solves the problematic issue of generating hand-labeled data. This is troublesome when the task demands expert understanding or not-so-obvious notations (e.g., 3D bounding box vertices). To solve these limitations, using simulators is one of the most recurrent state of the art strategies.

NDDS is an NVIDIA UE4 plugin that allows computer vision researchers to export high-quality labeled synthetic images. NDDS supports images, segmentation, depth, object pose, bounding box, key points and custom templates. In addition to the exporter, the plugin includes different components to generate random images. This randomization includes lighting, objects, camera position, poses, textures and distractors, as well as camera path tracking, and so on. Together, these components allow researchers to create random scenes to train deep neural networks.

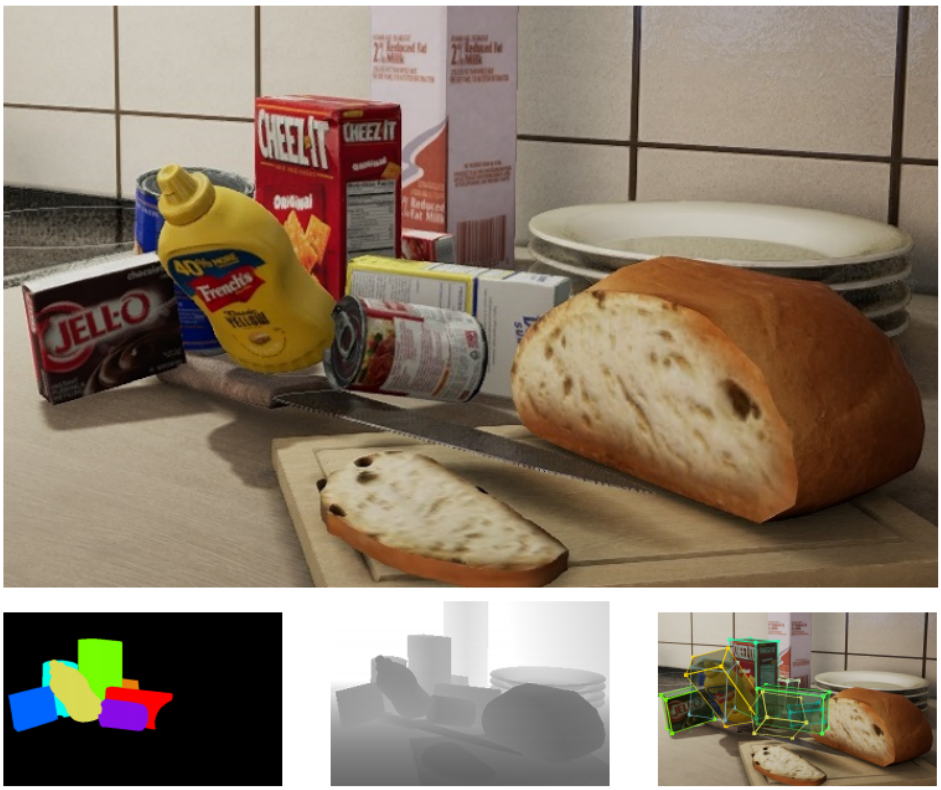


FIGURE 2.19: Example of an image generated using NDDS, along with ground truth segmentation, depth, and object poses. From the referenced paper.

This simulator has been used to generate the *Falling Things* dataset: *A Synthetic Dataset for 3D Object Detection and Pose Estimation* [23]. Which has been successfully tested and used to solve a deep object pose estimation problem [24], so this proves that networks trained on synthetic data operate correctly when exposed to real-world data.

We are proving slowly that these networks perform well in the real world when trained with synthetic samples, this means that these environments are effectively helping to close the gap between synthetic and real data. Continuous research on simulators depend on how rendering solutions and hardware evolve in the future. Domain randomization has proven to work if the data is sanitized, however, photo-realism is as well part of the formula in which simulators have yet to improve.

2.1.5 UnrealROX in context

In this section we have talked about a good portion of the current advancements of the Sim2real field. To recap, in Section 2.1.1, we discussed some of the improvements, advantages and disadvantages of synthetic photorealism. Then, in Section 2.1.2, we cover the main alternative to synthetic photorealism: domain randomization, which solves partially the variability problem. Simulators have played an important role in generating synthetic datasets. However, the biggest problem with the majority of them, is that they focus their resources on either getting many FPS, or achieving a decent resolution.

UnrealROX tries to alleviate these problems by balancing the resource consumption, offering that way high resolution photorealism (1080p) at 60 FPS, combining the strenghts of previous simulators and reducing their weaknesses (strenghts like resolution, framerate, and realism) and reducing weaknesses (now objects are interactable, hands, grasping...). In addition, UnrealROX is developed on top of the UE4 engine, which means that the State of Unreal[19], can translate seamlessly to what the simulator will be able to offer in the future at a higher level, including ray-tracing (see Figure 2.20).

FIGURE 2.20: State of Unreal (2019) shows UE4's take on ray-tracing.

UnrealROX is open-sourced, this means that it's not a single purpose simulator, since the final user can modify its goal. As of today, UnrealROX is being extended in some projects on the University of Alicante, this means that it is constantly under development to make the whole framework more user friendly. The base of UnrealROX covered on this work has the following features:

- **Virtual Reality:** UnrealROX is an environment that records the movements of an operator and plays them back off-line generating ground truth (instance segmentation, normal maps, depth and RGB data). This means that the user is the main responsible of the movements of the robot in which we play back this motion. This introduces a layer of natural movement from which the dataset can benefit.

- **Complex kinematic system:** Thanks to the built in Forward And Backward Reaching Inverse Kinematics (FABRIK) system, the agents can resolve bone hierarchy movements respecting their degrees of freedom.

[19] https://www.youtube.com/watch?v=s55Uob494Do

- **Grasping:** UnrealROX comes with UnrealGrasp, a grasping system that allows an agent to interact and grasp dynamic objects from the scene, as we can see on Figure 2.21.

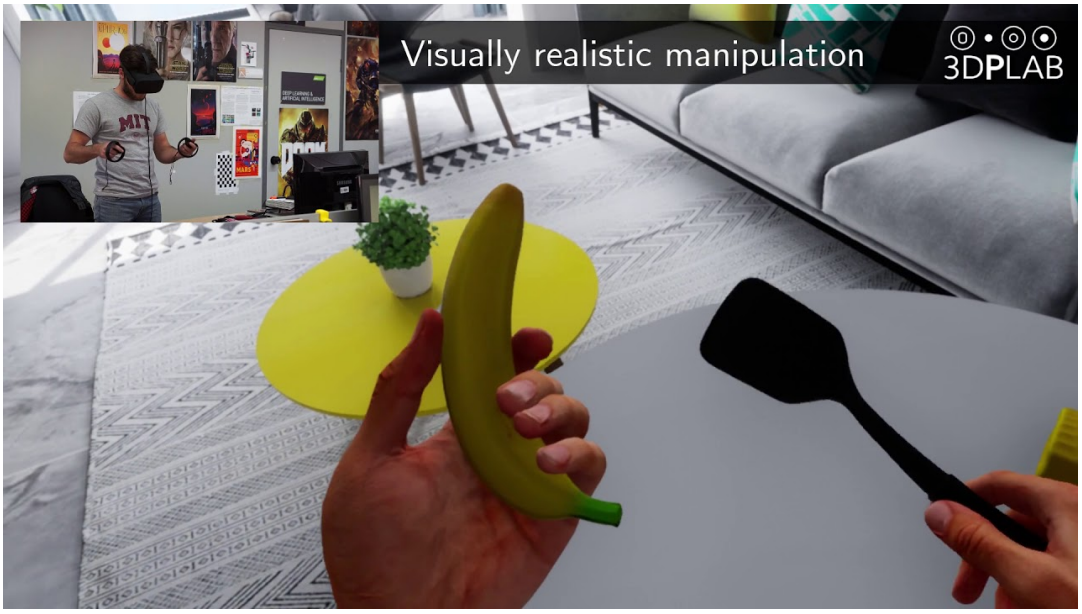


FIGURE 2.21: With UnrealGrasp you can grasp two dynamic objects simultaneously.

- **UE4:** Being a UE4 project brings an incredible amount of advantages to this work. For example, thanks to the Skeletal Mesh system, we can re-target new skeletons easily by mapping bone names; this means that we can import photorealistic arms to generate a synthetic dataset with UnrealGrasp. Furthermore, UE4 counts with a very powerful community that generate content constantly. This has been a critical point for UnrealROX, since all the scenes we gathered come from its community. We can also find URDF import plugins and several tools that can speed up the development.

- **Multicamera System:** UnrealROX allows the user to define as many on board cameras as desired. It supports aswell static cameras. This makes the dataset more rich in content and context thanks to the multiple points of view. Ground truth is also generated for each one of these cameras.

- **Open to domain randomization:** As we said previously, UnrealROX is an open source simulator that supports extensive customization. This means that there is an open via to cover both of the Sim2Real avenues, photorealism and domain randomization. One example of this is Isaac Sim, as we commented in Section 2.1.2, it is an environment created on UE4 that supports extensive domain randomization.

In short, UnrealROX is an open source framework that allows users to extend (or modify) its base using Blueprints and/or C++. Aside working with virtual reality (VR), the possibilities of UnrealROX go beyond that. Some of these points are further elaborated on chapter 4.

Chapter 3

Methodology

In this third chapter we describe the different materials and methods we used in this Thesis. The chapter is divided into three sections: Section 3.1, where we put in context the different materials based on their nature. Section 3.2, where we decompose the software used to make UnrealROX. And finally Section 3.3, where we reveal the different physical resources we had to employ to make posible this Thesis.

3.1 Introduction

To do this work we have chosen the most appropriate tools based on our requirements. This environment encompasses several tools, such as UE4, *Visual Studio*, *Visual Assist*, *Python*, *irfanView* and *Sublime Text Editor*. All these tools have been used to develop, post process and validate the generated data.

Furthermore, we need hardware that adheres to our needs. In this case it is recommended to have a powerful setup with a mid-end/high-end GPU that supports complex scenarios composed by high resolution textures and detailed models. In addition, due to the nature of this work, we need a virtual reality headset compatible with UE4.

3.2 Software

It is essential to have the necessary tools to carry out an UE4-based project for data generation. The choice to work with UE4 and not other game engine is mainly because of the virtual reality support and the potential to represent photorealistic scenes in real time. In addition, I count with previous experience working with UE4, which has helped on the development of the tool. UE4 disposes of Blueprints, a visual scripting tool that we have used to prototype the base functionality of the project. Then, we iterated to C++ for efficiency and flexibility, so we needed an IDE; that is where Visual Studio comes into play as well as Visual Assist. We also needed to program a set of scripts to process and post-process the data, so we used Python. Finally, we used irfanView and SublimeText to verify the generated samples.

Unreal Engine 4

If we want to define UE4, we can look directly at the definition that Epic Games give in their official web page[1]: *"Unreal Engine 4 is a complete suite of development tools made for anyone working with real-time technology"*. UE4 is the tool we use to generate the first layer of data, in order to get to that point we had to build a project named *UnrealROX* that counts with a set of tools which allows the user to record

[1] https://www.unrealengine.com/en-US/features

their actions in a realistic scene, and then be able to reproduce them frame by frame exporting the desired images (depth, rgb, normals and/or mask).

UE4 also counts with a custom scripting language called **Blueprints** that allows us to define custom behaviors in a very easy way, we will see some of its features later in Section 4.4.2. Blueprints have played a substantial role when developing UnrealROX, since they allow quick iterations to create a prototype, which can be later converted and extended in C++.

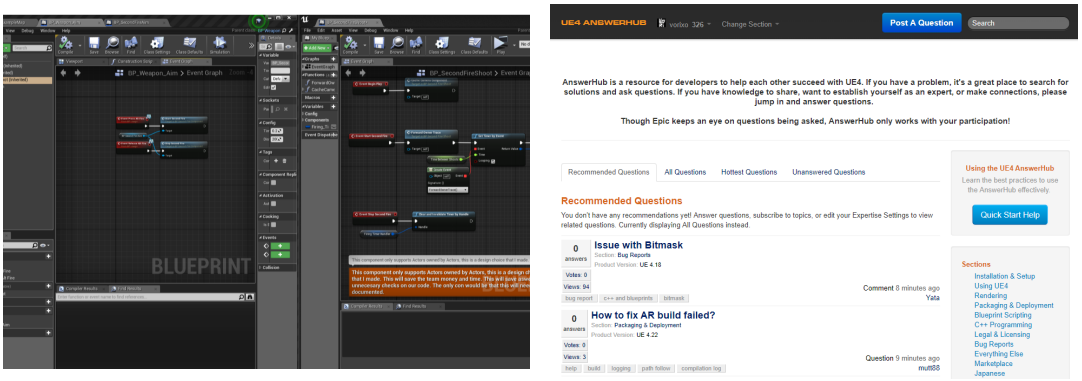
Some of the main advantages of Unreal Engine that have been beneficial for the complete development of UnrealROX are the following:

- **Active development:** UE4 is on continuous development due to how demanding the gaming industry is. This makes the engine perfect for current state of the art studies in modern rendering techniques. One recent example is the inclusion of ray-tracing technology in real time, as we commented in Section 2.1.1. Also, the staff tries to fix as soon as possible all the bugs that might appear between versions.

- **Big community:** UE4 disposes of a very active community located in the official forums[2], the unofficial *UnrealSlackers* Discord[3] and the *UE4 AnswerHub*[4], where continuous questions are answered of several problems from various topics. UE4 marketplace is where creators sell or give out UE4 assets for free , which is the case of some of the scenes we employed in the creation of *The RobotriX* dataset [3].

- **Blueprints:** This is one of the most powerful tools that the engine disposes. It started as *Unreal Kismet*[5] and has been evolving since the first version of Unreal Engine 4. The set of tools available in Blueprints makes possible to prototype quickly any project using only this visual scripting language. The only inconvenience is that it is binary based, which hampers version controlling.

Figure 3.1 displays how Blueprints look and the UE4 AnswerHub community.



(A) Blueprints.

(B) UE4 AnswerHub.

FIGURE 3.1: This figure showcases some of the advantages of UE4. In Figure 3.1b we can see some members of its community.

[2] https://forums.unrealengine.com/
[3] https://unrealslackers.org/
[4] https://answers.unrealengine.com
[5] https://www.youtube.com/watch?v=IReehyN6iCc

In Figure 3.2 we can see the capacity of UE4 to render photorealistic scenes.

FIGURE 3.2: Snapshots of the daylight and night room setup for the Realistic Rendering released by Epic Games to showcase the realistic rendering capabilities of UE4.

However, it's not all about advantages, since UE4 has some inconveniences and inconsistencies we had to deal with:

- **Regressive bugs:** UE4 has some inconsistencies when it comes to Blueprint-C++ communication. From cyclical dependencies that produce Class Default Object (CDO) resets, to blueprintable C++ structs that get corrupted after inserting members on the type. As I mentioned before, these bugs get addressed when the staff is able to reproduce them, but they eventually come back between versions, which makes updating projects of UE4 harder.

- **C++ compile times:** This is not exactly a UE4 problem, but since UE4 uses C++ we can include it to the list. When not using a very powerful setup, compiling a project can take minutes, which slows down considerably the development process. Current versions of UE4 have improved the situation with Live++[6].

Fortunately, the advantages far outweigh the disadvantages for the scope of this project, which is why we have decided to go ahead with Unreal Engine despite the inconveniences discussed.

Visual Studio

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop computer programs as well as websites, web applications, web services and mobile applications.

Like any other IDE, it includes a code editor that supports syntax highlighting and code completion using IntelliSense (see Figure 3.3). It also includes a debugger that works both as a source-level debugger and as a machine-level debugger. In addition, Visual Studio counts with a great quantity of plugins to ease development.

UE4 is designed to integrate seamlessly with Visual Studio[7], allowing the user to quickly and easily make code changes in its projects to immediately see results

[6] https://molecular-matters.com/products_livepp.html

[7] https://docs.unrealengine.com/en-us/Programming/Development/VisualStudioSetup

upon compilation. Configuring Visual Studio to work with Unreal Engine can help improve the efficiency and overall user experience of developers using Unreal Engine. Also, Visual Studio has support for extending the debugger with visualizers that allow easy inspection of common Unreal types such as FNames and dynamic arrays.In *UnrealROX* we've complemented the usage of Visual Studio with Visual Assist, a tool we will describe bellow.

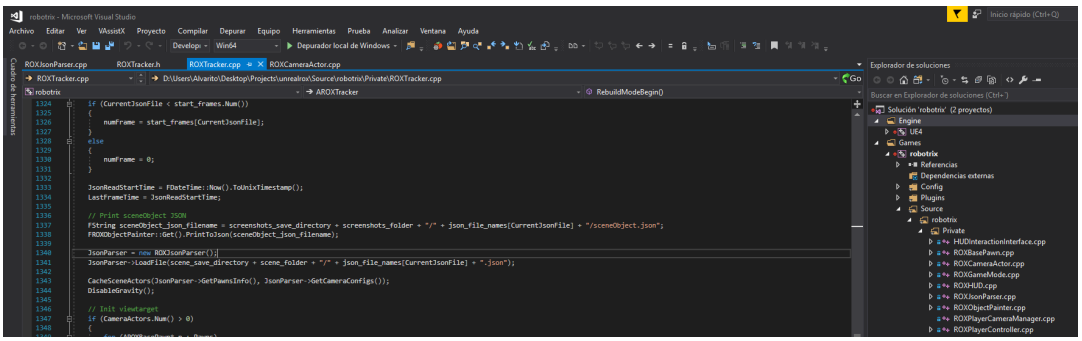


FIGURE 3.3: Microsoft Visual Studio environment.

Visual Assist

Visual Assist is a plugin for Microsoft Visual Studio developed by Whole Tomato Software. The plugin mainly improves IntelliSense[8] and syntax highlighting. It also enhances code suggestions, provides refactoring commands, and includes spell-check support for comments. It can also detect basic syntax errors such as the use of undeclared variables. Visual Assist includes features specific to development with UE4, including support for UE4 keywords, preprocessor macros, and solution setup. It also includes a series of bindings that allow us to navigate more easily through the project and engine code. These are some of the features we have used:

- **Find Symbol in Solution:** It allows the user to list all the symbols of the project filtered by a string. It can be fast acceded pressing Shift+Alt+S.
- **GoTo Implementation:** Jumps to the declaration or implementation of the current symbol. Its shortcut is Alt+G.
- **Open File in Solution:** Opens a dialog of filenames filtered by a string with the shortcut Shift+Alt+O.

Python

Python is an interpreted programming language whose philosophy emphasizes a syntax that favors readable code.

It is a multiparadigm programming language, as it supports object-oriented, imperative programming and, to a lesser extent, functional programming. It is an interpreted language, uses dynamic typing and is multiplatform.

It is administered by the Python Software Foundation. It has an open source license, called Python Software Foundation License[9], which is compatible with the GNU General Public License from version 2.1.1, and incompatible in certain earlier versions.

[8] https://docs.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2015
[9] https://docs.python.org/3/license.html

Python has been used to extend the data generation pipeline with an additional ground truth generator module to produce extra data from the raw images and information. One of the multiple tasks in which Python has been used is to generate visible bounding boxes of the objects in a scene (as seen in Figure 3.4).

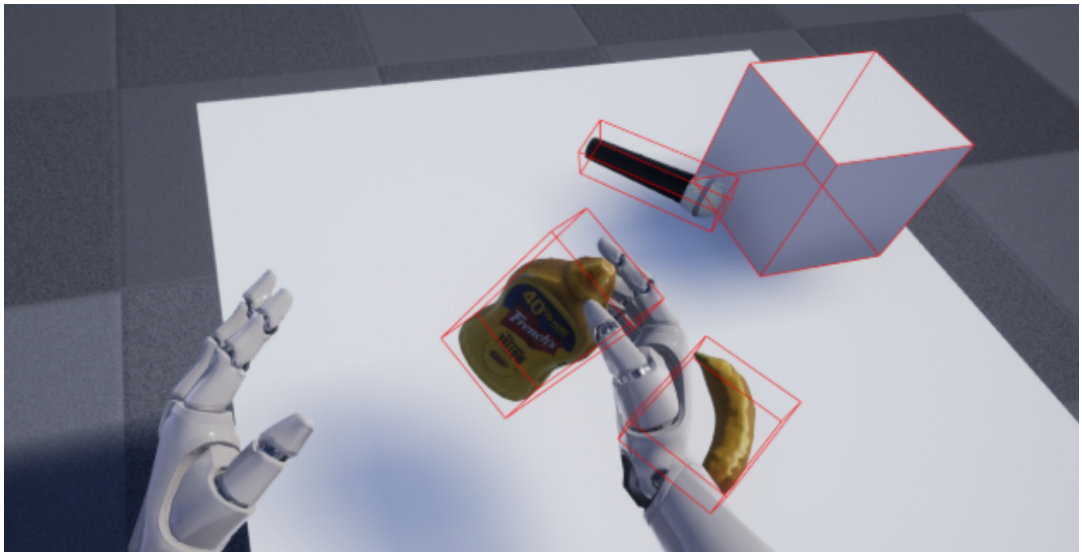


FIGURE 3.4: Bounding box generation using Python. These images are part of the UnrealROX dataset.

irfanView

IrfanView is an image viewer, editor, organiser and converter program for Microsoft Windows. IrfanView is specifically optimized for fast image display and loading times. It supports viewing and saving of numerous file types including image formats such as *BMP*, *GIF*, *JPEG*, *JP2* and *PNG* between others.

IrfanView allows us to verify that the images are generated properly by observing the RGB value of every pixel using the color picker tool from the paint dialog, which can be enabled by pressing F12, as we can see in Figure 3.5.

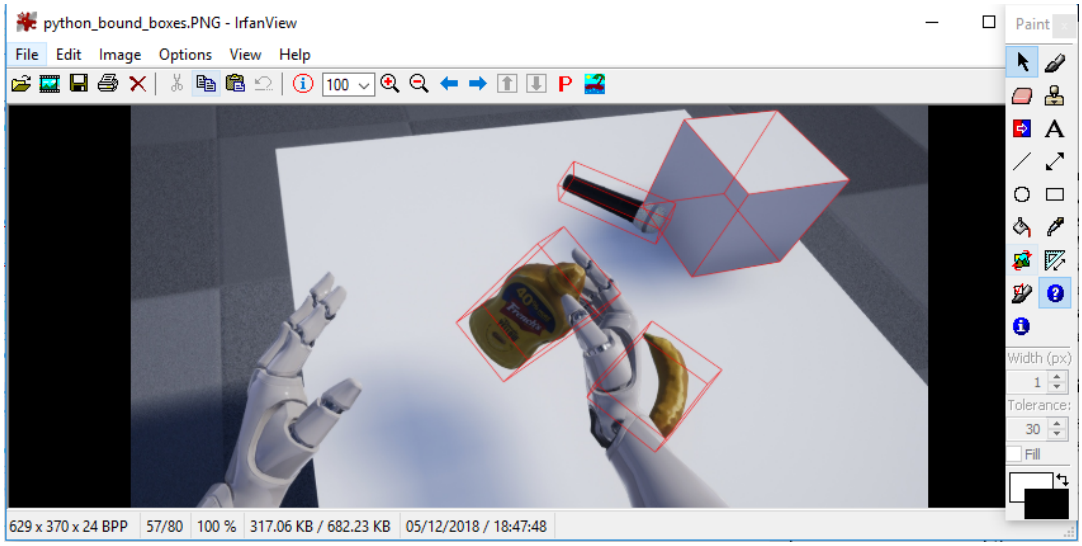


FIGURE 3.5: irfanView and infanView paint dialog.

Sublime Text Editor

Sublime Text is a sophisticated text editor for code, markup and prose. The following description represents the main features of Sublime Text editor:

- **GoTo anything:** Quick navigation to files, symbols or lines. With extremely long JSON files, this feature has been one of the most important ones when it comes to verify the data. This enabled us quick navigation between our files.
- **The Command Palette:** Uses adaptive correspondence for quick invocation of arbitrary commands from the keyboard.
- **Simultaneous editing:** Simultaneously make the same interactive changes in several selected areas. This feature allowed us to fix different JSON fields after main project refactors, meaning that we could fix previously recorded datasets to work with the up to date version.
- **Extensive customization capability:** Through JSON configuration files, including project-specific and platform-specific configuration.
- **Cross-platform:** Cross-platform (Windows, macOS and Linux) and cross-platform support plugins.

Thanks to that set of features, Sublime Text has been the main text editor used when developing the Python Scripts for the dataset. It also has been a great tool for text logging verification due to the extensive lookup functions that Sublime Text disposes, such as the regular expresion tool.

3.3 Hardware

As we have already mentioned in Section 3.1, hardware is one of the most important parts when dealing with large amounts of data and complex UE4 scenarios. Without powerful enough hardware, recording at a constant FPS rate becomes more difficult. Also, when we play the data back to generate ground truth, we need the algorithm to execute as fast as possible, this becomes difficult when the computer has difficulties processing the scene to switch from one viewmode to another. In addition, it is convenient to have a good computer to work comfortably in UE4.

To solve these problems, there have been several hardware PC architectures that have allowed a remarkable acceleration generating and processing the data. It is also important to have a comfortable and capable personal computer to develop and run reduced tests, we will divide this PC architectures in Subsection 3.3.1.

We will also see in Subsection 3.3.2 the additional hardware that has been used in *UnrealROX*: the virtual reality headsets *HTC Vive Pro* and *Oculus Rift*.

3.3.1 PC Resources

The project has been developed and tested on several PCs, however we are going to remark the most notable and used ones: my personal PC, *Challenger* and *Asimov*, both provided by the *3D Perception Lab*[10]. In order to make the scheme simpler, we will divide the resources from less to more power in this Section.

[10] https://labs.iuii.ua.es/3dperceptionlab/

Personal Computer

My personal computer has been used mainly for the learning process and the development of several projects implied in this thesis. The Table 3.1 shows the complete configuration of the hardware of my personal computer.

My personal PC	
Motherboard	Z370 GAMING PLUS (MS-7B61) [0] Intel Z370 Chipset 2× PCIe 3.0 × 16 slots 4× PCIe 3.0 × 1 slots
CPU	Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz [1] 3.7 GHz (4.7 GHz Turbo Boost) 6 cores (12 threads) 95 W TDP
GPU	NVIDIA GeForce GTX 1070 Ti [2] 2432 Compute Unified Device Architecture (CUDA) cores 8 GB of GDDR5 Video Memory 16× PCIe 3.0 180 W TDP
RAM	G.Skill Ripjaws V Red DDR4 2400 PC4-19200 16GB 2× 8GB CL15
Storage (Data)	Seagate ST2000DX001 Solid State Hybrid Drive (SSHD)
Storage (OS)	Crucial MX500 CT500MX500 Solid State Drive (SSD)

TABLE 3.1: Hardware specifications of my personal PC.

Challenger

Challenger has been used to develop *UnrealRox*, it is a computer with a powerful processor and GPU, so it has also been used for data generation. It has an NVIDIA GPU with CUDA support and we can see its specifications in table 3.2.

Challenger	
Motherboard	MSI Z370 PC Pro [0] Intel Z370 Chipset 2× PCIe 3.0 × 16 slots 4× PCIe 3.0 × 1 slots
CPU	Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz [1] 3.7 GHz (4.7 GHz Turbo Boost) 6 cores (12 threads) 95 W TDP
GPU	NVIDIA GeForce Titan Xp [2] 3840 CUDA cores 12 GB of G5X Video Memory 16× PCIe 3.0 250 W TDP
RAM	G.Skill Ripjaws V Red DDR4 2400 PC4-19200 16GB 2× 8GB CL15
Storage (Data)	Seagate ST2000DX001 SSHD
Storage (OS)	Crucial MX500 CT500MX500 SSD

TABLE 3.2: Hardware specifications of Challenger.

Asimov

Asimov has been used to generate data and hold it in its RAID configuration. As can be seen in table 3.3, *Asimov* has a total of 3 GPUs each dedicated to a specific task, the Titan X, has been allocated to all the deep learning part, having another graphics card for all graphics processing, the GT730, which relieves the work of the Titan X. At the same time, the server also has an extra GPU that is used mainly for computing.

	Asimov
Motherboard	Asus X99-A [0]
	Intel X99 Chipset
	4× PCIe 3.0/2.0
CPU	Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz [1]
	3.3 GHz (3.6 GHz Turbo Boost)
	6 cores (12 threads)
	140 W TDP
GPU (visualization)	NVIDIA GeForce GT730 [2]
	96 CUDA cores
	1024 MiB of DDR3 Video Memory
	PCIe 2.0
	49 W TDP
GPU (deep learning)	NVIDIA GeForce Titan X [3]
	3072 CUDA cores
	12 GiB of GDDR5 Video Memory
	PCIe 3.0
	250 W TDP
GPU (compute)	NVIDIA Tesla K40c [4]
	2880 CUDA cores
	12 GiB of GDDR5 Video Memory
	PCIe 3.0
	235 W TDP
RAM	4 × 8 GiB Kingston Hyper X DDR4 2666 MHz CL13
Storage (Data)	Seagate Barracuda 7200rpm 3TiB SATA III Hard Disk Drive (HDD) [5]
Storage (OS)	Samsung 850 EVO 500GiB SATA III SSD [6]

TABLE 3.3: Hardware specifications of Asimov.

3.3.2 Virtual Reality headsets

One of the fundamental requirements when working in a VR environment is the headset. In this case and due to the nature of this project, we have counted with two very different devices in order to increase compatibility of the project. There are differences in function between these devices, such as the way they calculate head positions.

In this case we have mainly worked with the Oculus Rift headset and the HTC Vive Pro headset, both provided by the 3D Perception Lab. The use of two different technologies has helped us to test and debug the project with different perspectives, which is essential when it comes to expanding the number of potential users who will be able to use UnrealROX in the future.

In this subsection we will proceed to describe all the additional materials that we have used for the realization of this thesis.

Oculus Rift

The *Oculus Rift* is a virtual reality headset developed and manufactured by Oculus. It has a total resolution of 2160x1200 (1080x1200 per eye) and its refresh rate is at 90 Hz. They have integrated 3D audio headphones. In addition to this, it can be controlled with the normal *Xbox One* controller or with the *Oculus Touch* motion tracked controllers.

According to tracking input specifications, it has 6 degrees of freedom (6DOF) (3-axis rotational tracking + 3-axis positional tracking) through USB-connected *IR LED* sensor, which tracks via the *constellation* method. The headset has a series of external sensors that help establish the position of the user's head and the controllers. *Oculus* called this tracking technique constellation, which gave their sensors its name. We can see the complete set in Figure 3.6.

FIGURE 3.6: Full Oculus Rift Set.

HTC Vive Pro

The HTC Vive is a virtual reality headset developed by HTC and Valve Corporation. The headset uses tracking technology that allows the user to move in a 3D space and interact with it using the motion-tracked controllers. It has a total resolution of 2880x1600 (1400x1600 per eye) and its refresh rate is at 90 Hz. They have integrated 3D audio headphones. The controllers have multiple input methods including a track pad, grip buttons, and a dual-stage trigger. The Vibe Base Stations emit timed infrared pulses that are picked up by the headset to create a 360 degree virtual space. The set of these accessories (as seen in Figure 3.7) makes it possible to navigate quite precisely in a simulated environment.

FIGURE 3.7: Full HTC Vive Pro Set.

Chapter 4

UnrealROX

The fourth chapter describes UnrealROX project architecture. Each section describes a subsystem that takes part on the project.

4.1 Introduction

In Section 2.1.3, we delineated various simulators that try to minimize the gap between synthetic and real data using different techniques, such as: physics-based robots on scanned indoor scenes, basic interaction systems, reconstructed houses from various datasets, ect. We witnessed and studied the main advantages and disadvantages of each one. And while the features of some complement others, there is no middle ground between the discussed simulators.

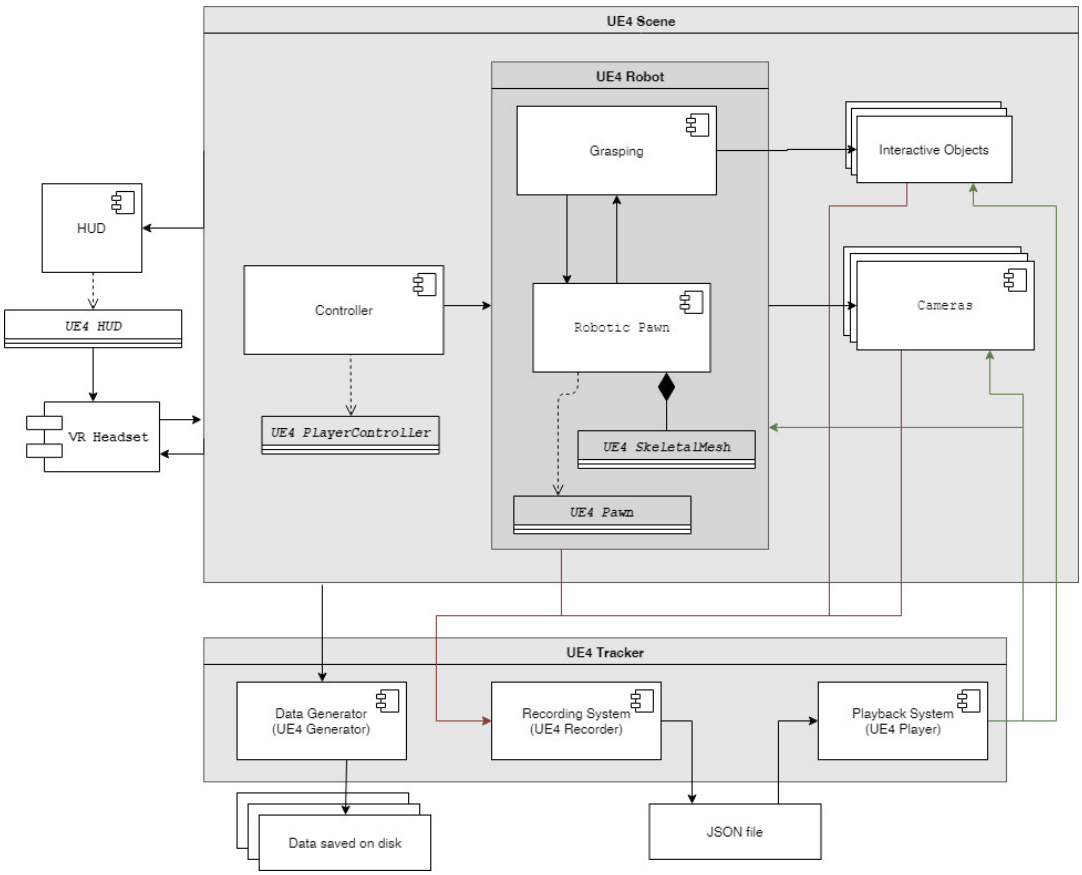


FIGURE 4.1: UnrealROX decouples the recording and data generation processes so that we can achieve high framerate when gathering data.

UnrealROX is an open-source simulator that tries to bring together and improve all the advantages of the aforementioned environments and unite them in a single framework. This simulator is based on UE4 since it conforms to the state of the art in terms of rendering techniques, meaning that making visually believable photo-realistic scenes becomes a possibility. As we mentioned in Section 2.1.1, photorealism is one of the ways to reduce the gap between synthetic and real data, which is the main basis of our engine choice. In addition, we observed that among the simulators that we have analyzed in this work, none of them had virtual reality support, which inclusion in UnrealROX was possible thanks to UE4. Since UnrealROX is a UE4 project, the amount of possibilities our environment offers are subject to the users extending it, thanks to the fact that Unreal Engine source code is modifiable.

In our environment, UE4 renders a scene and a robot into a VR headset (see Section 3.3.2), so that a user can move the robot and interact with objects with the robotic hands; storing scene information on a text file every frame so that it can be reproduced offline to generate ground truth from multiple points of view predefined by the user. This virtual reality environment enables robotic vision researchers to generate visually plausible data with full ground truth for a wide variety of problems: class and instance semantic segmentation, object detection, depth estimation, visual grasping, navigation, etc. Figure 4.1 displays the main architecture of the complete framework.

In this chapter, we will describe every component and subsystem that shapes UnrealROX; starting with Section 4.2, which defines our multi camera setup. Then, Section 4.3 describes the debugging tools implemented on the user interface of UnrealROX. Next, in Section 4.4 we will see all the animation implementation details, followed by a detailed explanation of the controller class in Section 4.5. Finally, Section 4.6 and 4.7 delineate the main controllable robot class and our data dumper in depth.

4.2 Multi-camera Support

If we look at any of the robots available on the market, we can see that almost all of them integrate multiple cameras in different parts of their bodies. These cameras can be used for any purpose the developers require. In addition, external cameras are usually added to the system to provide external (non-egocentric) data to he robot. In *UnrealROX*, we want to simulate the ability to add multiple cameras in a synthetic environment with the goal in mind of having the same or more amount of data that we would have in a real environment. For instance, in order to train a data-driven grasping algorithm it would be needed to generate synthetic images from a certain point of view: the wrist of the robot. To simulate this situation in our synthetic scenario, we would like to give the user the ability to place cameras attached to sockets in the robot's body, e.g., the wrist itself. Another plausible application scenario would be a pose estimation algorithm which needs static cameras placed in a room. In a real environment, it would be necessary to place them manually to gather data, which can become a tedious and slow task; however, in *UnrealROX* we offer the possibility to place cameras in a user friendly way using the **UE4** editor.

One of the examples to be highlighted in *UnrealROX* that makes use of this practice is the social robot Pepper (Figure 4.2), thanks to the multi-camera support, we can generate a dataset for Pepper in which it is seen how the user grabs an object from multiple points of view (being one of them the robot's wrist). This dataset can be used to teach Pepper to pick up objects.

FIGURE 4.2: User interacting as Pepper in the scene.

To make this system of multiple cameras possible, both static and attached to bones or sockets, we will use *CameraActor* as the camera class and the *Pawn* class as the entity to which we will attach them. By default, UE4 does not allow us to precisely attach components in the editor so it is necessary to define a socket-camera relationship in the *Pawn* class. This is due to the fact that it has direct access to the skeleton which we will be attaching some of the cameras.

In this section, we will describe the implementation of the multi-camera subsystem. Firstly, in Section 4.2.1 we will describe the *CameraActor* class itself. Next, in Section 4.2.2, we will explain how to make those actors to move by attaching them to sockets. At last, Section 4.2.3 provides all the implementation details needed to put all the components together.

4.2.1 The Camera Actor

The objective of the *CameraActor* class is to render any scene from a specific point of view. This actor can be placed and rotated at the user's discretion in the viewport, which makes them ideal for recording any type of scene from any desired point of view. The main goal in *UnrealROX* is to use these cameras to collect data from arbitrary points of view. The output of this camera is the one that we will use to generate part of the dataset as seen in Figure 4.3.

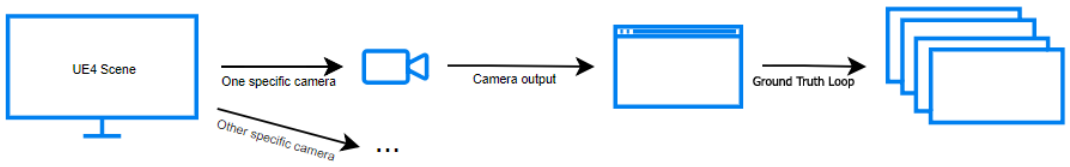


FIGURE 4.3: Main execution flow depending on the cameras.

The *CameraActor* is represented in UE4 by a 3D camera (shown in Figure 4.4) icon and like any other actor, it can be moved, rotated and scaled in the viewport, however scaling the camera will not have any effect on the output rendering.

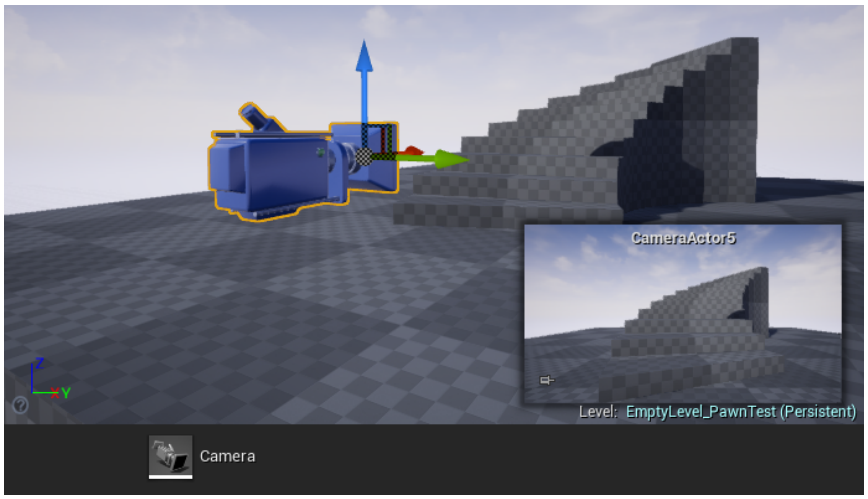


FIGURE 4.4: A *CameraActor* in the viewport rendering a stair.

This class has an extensive collection of parameters that the user can manipulate to adjust the camera rendering in the editor. Among these variables[1] we can discriminate between different categories:

- **Camera Settings:** In this category we have parameters such as the projection mode, the field of view and the aspect ratio.
- **Color grading:** In this category we can modify the post-processing of the camera, correcting the gamma or modifying the white balance among other options.
- **Tonemapper:** Here we can modify the values of Slope, Toe, Shoulder, Black clip and White clip of the camera.
- **Lens:** The Lens section exposes various effects, such as chromatic aberration, grain or vignette intensity. We can also modify the bloom, add masks, modify the auto exposure, ect.
- **Other rendering features:** In this section we will be able to adjust the post processing materials, the environmental occlusion, the blur and the global illumination in addition to other similar parameters.

As can be seen, UE4 has a very complete camera class, which we will use throughout the project to render our scenes. In addition UnrealROX will expose the camera options most demanded by the user as is the case of the field of view or the aspect ratio, as well as additional features not included by default in the actor, such as the possibility of creating stereo vision [11], done by placing automatically in generation time an additional camera next to the original one with the same settings.

4.2.2 Camera Movement

Since some of the points of view are dynamic, e.g., robot's wrists, we need to make the cameras move with them. To achieve that, an attaching operation where the camera actors will be parented to the component's point of view that we want to record will be needed. In this regards, understanding the attaching operations the engine provides (as can be observed in Listing 4.1) is essential.

[1] https://api.unrealengine.com/INT/API/Runtime/Engine/Engine/FPostProcessSettings/index.html

LISTING 4.1: Attaches an actor to a parent actor.

```
void AttachToActor
(
        AActor * ParentActor,
        const FAttachmentTransformRules & AttachmentRules,
        FName SocketName
)
```

The *AttachToActor* function is in charge of parenting one actor with another following some attachment rules. In addition, we can specify on which socket we want to attach the object. This means that when the selected socket changes its transform, the attached object will change it too according to the *AttachmentRules*. These rules specify how this new attached actor will behave when the socket it is linked moves or rotates.

The *AttachmentRules* can be defined separately for location, rotation, and scale (we can choose between these three options defined on the *EAttachmentRule* enumeration[2]). Each entry will modify the attached actor (or component) behaviour in the following way:

- **KeepRelative:** Keeps current relative transform as the relative transform to the new parent.
- **KeepWorld:** Automatically calculates the relative transform such that the attached component maintains the same world transform.
- **SnapToTarget:** Snaps transform to the attach point.

4.2.3 Implementation

This problem lead us to define an implicit relatioship between the *CameraActor* and the socket it is attached to, in order to implement this relationship, the *Pawn* class implements a *USTRUCT* (the Unreal Engine struct) that has two parameters: the camera itself and the socket name. These properties are accompanied by the *EditAnywhere* meta[3], which makes possible the edition of the properties not only on the CDO but also on the instance of the object.

LISTING 4.2: Struct used at UnrealROX to resolve the Camera-Socket relationship.

```
USTRUCT()
struct FBoneCam
{
        UPROPERTY(EditAnywhere)
        ACameraActor* CameraActor;

        UPROPERTY(EditAnywhere)
        FName SocketName;
};
```

[2] https://api.unrealengine.com/INT/API/Runtime/Engine/Engine/EAttachmentRule/index.html

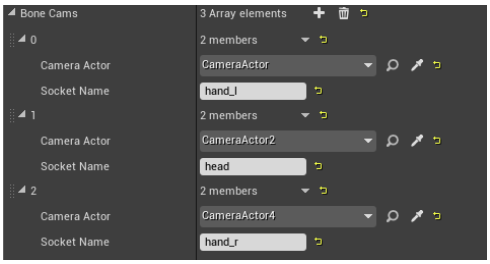
[3] To fully understand the metas, it is necessary to understand the *UPROPERTIES* and the reflection system of UE4 which can be studied on the following URLs https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection https://wiki.unrealengine.com/UPROPERTY

Once the relationship between these two components has been sorted out (see Listing 4.2), it only needs to be adapted for a scenario in which we can possibly have more than one camera. For this, we will use an array of structs as seen in Listing 4.3.

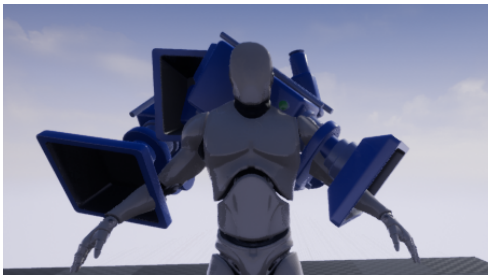
LISTING 4.3: Array of structs of the previously defined type.

```
UPROPERTY(EditAnywhere, Category = Tracker)
TArray<FBoneCam> BoneCams;
```

The user will be in charge of filling the array specified in Listing 4.3. To make the process easier, we exposed these properties to be editable in the editor (see Figure 4.5).



(A) Representation of the array.



(B) Pawn Actor with cameras.

FIGURE 4.5: In-engine representation of the array of structs. In (a) we can see the representation of the array struct in the instance of the object, while in (b) we see its visual representation in the engine.

In order to do this, we will override the *OnConstruction* function (represented in Listing 4.4) of our *Pawn* class, which is executed every time we alter the Pawn in the viewport. This handle is managed by the engine automatically and it is used to program custom behaviour for a concrete actor in the editor world (OnConstruction function will not execute if we move the actor while playing).

LISTING 4.4: Called when an instance of this class is placed (in editor) or spawned.

```
virtual void OnConstruction
(
        const FTransform & Transform
)
```

The *OnConstruction* algorithm will have to automatically manage all the *BoneCams* entering and exiting the array. For this, we will use an extra variable of the same type (*CachedBoneCamsPT*), which will contain the *BoneCams* of the previous execution of the *OnConstruction* function. The objetive under caching this array is to control the cameras that existed in the array in the previous execution that no longer exist (and vice-versa).

```
BoneCams := unique_cameras(BoneCams);

C := BoneCams.intersect_cameras(CachedBoneCamsPT);
for x in CachedBoneCamsPT do
    if x NOT in C then
        x.Camera->DetachFromActor(...);
    end
end

for b in BoneCams do
    b.Camera->AttachToComponent(b.socketName);
    c := b.found(CachedBoneCamsPT);
    if c != nullptr && c.socketName != b.socketName then
        b.SetActorRelativeTransform(FTransform());
    end
end

CachedBoneCamsPT = BoneCams;
```

Algorithm 1: Main algorithm to attach and detach camera actors to the relative socket

As can be seen in Algorithm 1, the first step is to make sure that there are no repeated cameras in the array, since conceptually we cannot assign the same camera to two different sockets. The next step is to detach the cameras that were in the array in the previous execution that are no longer in the array. For that, we need to know which cameras are still on the array using an intersection operation, the cameras that are in *CachedBoneCamsPT* but not in the intersection will need to be detached. Finally, the last step is to attach the *BoneCams* cameras to the appropriate socket. If the camera already existed in the array and the socket has changed, we will set its relative transform to FTransform()[4], since it is possible that the user has defined an offset. Once the process is completed, we will prepare *CachedBoneCamsPT* for the next execution, as can be seen at the end of Algorithm 1.

[4]FTransform default constructor sets all the values to unitary or zero as it follows: FTransform(FVector(0,0,0), FRotator(0,0,0), FVector(1,1,1)) for Location, Rotation and Scale respectively

4.2.4 Recapitulation

To recap this episode, we will do the complete process described above, starting with placing a camera, seen in Figure 4.6. We will have to ignore the camera placed behind the pawn, since it is the one that comes with the pawn and is not a *CameraActor*, it is a *CameraComponent*.

FIGURE 4.6: Placing a camera.

Next, we will associate this camera to a bone. In order to do that, we have to select the Pawn and find in the exposed properties of the instance of the Pawn a Property called BoneCams as we can see in Figure 4.7. This Property is a *TArray* type, so we will see a + symbol on it which needs to be clicked to add a new entry.

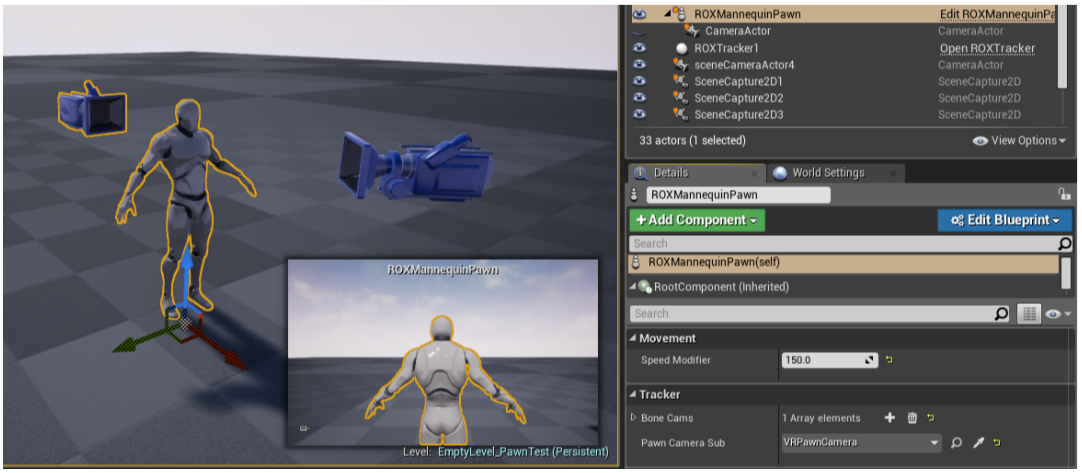


FIGURE 4.7: BoneCams Property.

In the added entry we will assign in *CameraActor* the camera we created previously, and in *SocketName*, the bone or socket to which we want to attach it, as Figure 4.8 shows, we have chosen to attach this camera to the *hand_l* bone of the skeleton of the selected pawn.

FIGURE 4.8: Adding a new entry in the array.

Then the OnConstruction algorithm executes doing the convenient operations following the execution flow represented in Figure 4.9.

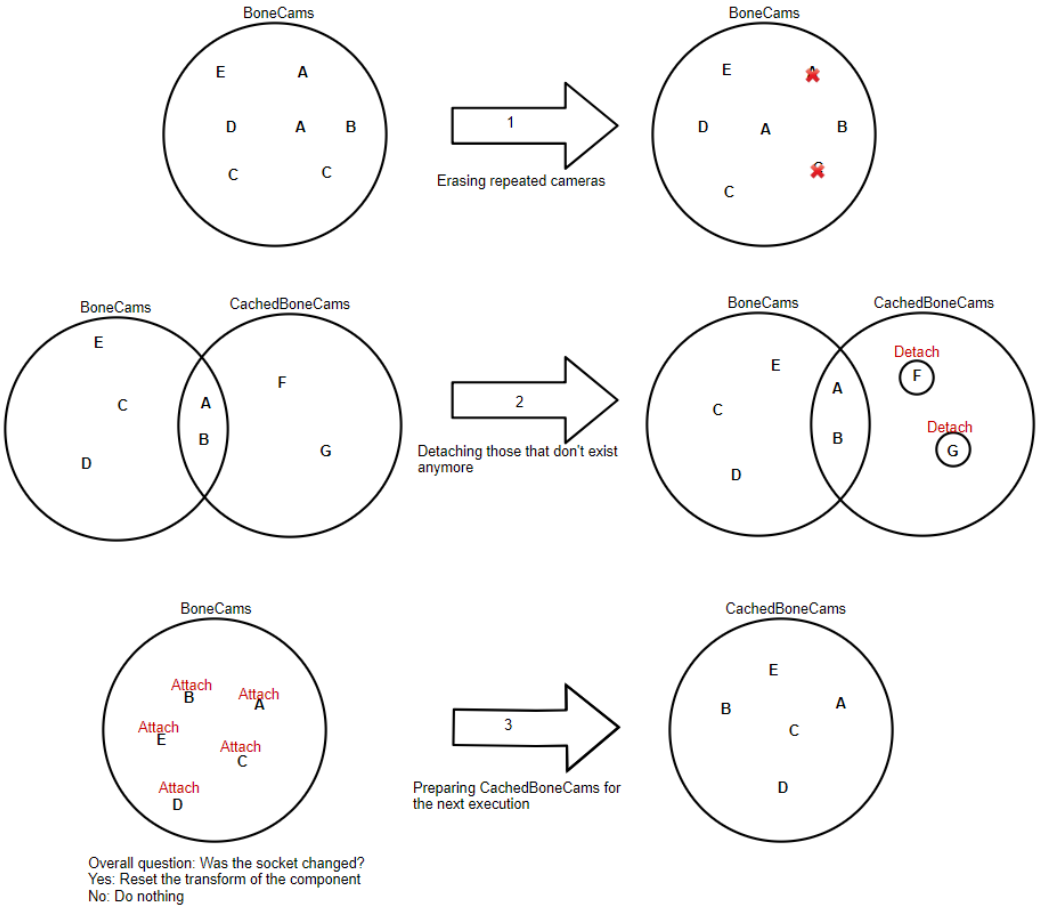


FIGURE 4.9: Execution flow of the OnConstruction function.

Once the *OnConstruction* function has been executed we will have our camera attached to the socket that we have assigned to it, in this case *hand_l*.

We will see this relationship in the editor as it follows: the camera will set its location and rotation following the socket orientation and position, this can be seen in Figure 4.10.

FIGURE 4.10: Camera actor assigned to a socket.

Usually this first setup does not suffice to have a clear viewpoint from a desired location, that is why we allow the user to add a relative offset (location and rotation) to this position. In our case we have to adjust the camera to the right position to replicate our desired viewpoint (possible point of view example represented in Figure 4.11).

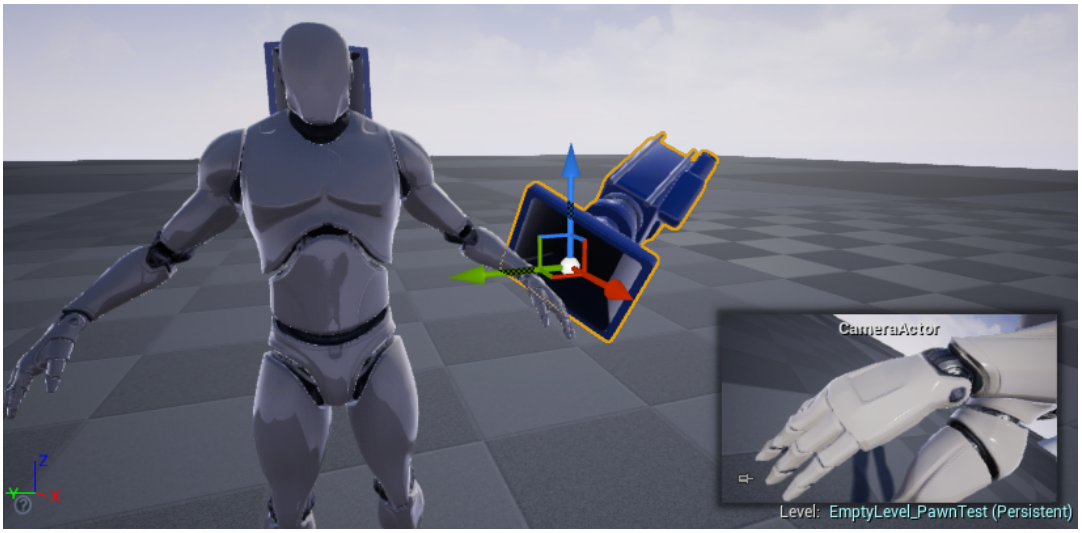


FIGURE 4.11: Camera actor exaggerated offset from original position.

Once we have completed all this process, we are ready (if required) to record from the new defined point of view. To add other points of view we would simply have to add a new entry in the array mentioned before and follow the steps defined above.

4.3 User Interface

It is convenient for any system to have a debug subsystem that provides feedback to the user about the application states. In UnrealROX we let the user know several points if this requires it. UnrealROXs heads-up display (HUD) can be turned off or on to the users will, making it completely decoupled from the rest of the systems. It can even be removed/erased if maximum performance is required. The main points covered in HUD are as follows:

- **States:** Notifies the user through a message on the screen with the relevant buttons pressed, the joints in contact with an object, the recording state, etc.
- **Error:** Prints a red message indicating an error. An example of this would be trying to record without the tracker on the scene (as seen in Figure 4.12).
- **Profiling:** The HUD also allows us to easily profile the system, allowing activation and deactivation with a single button. In addition, the user will be notified with a state message.
- **Scene Capture:** It allows us to establish a debugging point of view so that we can see our pawn from a different perspective from the editor. This scene capture will be described in Section 4.3.1.

FIGURE 4.12: Error message representing the lack of the tracker.

UnrealROX allows to make use of this class in a very simple way thanks to the UE4 interfaces[5], in case the user needs to debug a custom feature.

[5] https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Interfaces

4.3.1 The HUD class

In UnrealROX we have used the HUD class that provides UE4. This class has a canvas and a debug canvas on which primitive shapes can be drawn. Provides some simple methods for rendering text, textures, rectangles and materials which can also be accessed from blueprints. An example of texture drawing in practice in our project is the Scene Capture, which consists in drawing a texture in the viewport captured from a camera, as can be seen in Figure 4.13. This will be useful for the user to see if the animations are being played correctly in a VR environment.

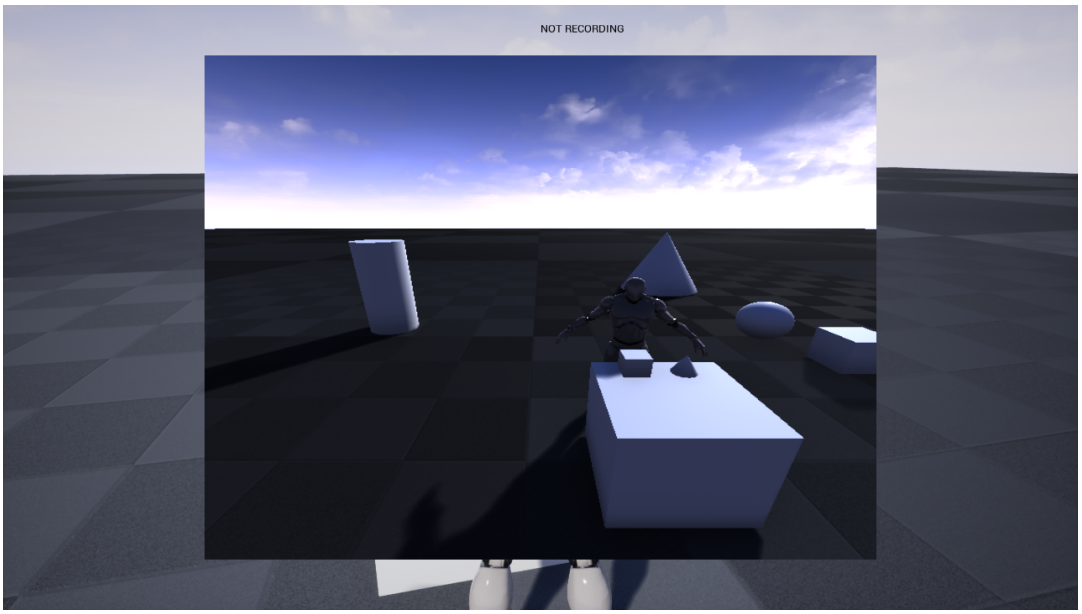


FIGURE 4.13: Scene Capture drawn in the viewport.

4.4 Animation System

It is essential to have an animation system to transfer all the movements captured with the virtual reality peripherals to the skeleton we are controlling. In order to do that we will use Persona, which is the set of features of UE4 for working with skeletal animations and Skeletal Meshes.

In Section 4.4.1 we will describe Persona, followed by a study of the order of execution of the animations in Section 4.4.2, next, in Section 4.4.3, we will see the diverse methods of inverse kinematics used, and finally in Section 4.4.4, we will observe and analyze a special animation node that has been created for *UnrealROX*.

4.4.1 Persona

Persona is the name UE4 gives to the main animation framework of the engine. This framework is composed by several features that are listed bellow:

- **Skeleton Editor:** Here we can examine and modify the Skeleton of a Skeletal Mesh. In this editor is where we will be able to add Sockets to our skeletons, it is showcased in Figure 4.14.
- **Skeletal Mesh Editor:** In this editor we can assign default materials to your Skeletal Mesh.
- **Animation Editor:** Here we will be able to work with Animation Sequences and other animation assets, such as Blend Spaces and Animation Montages.
- **Animation Blueprint Editor:** Is where we can create the rules for when and how the animations are played. Here we'll be able to use complex state machines and different blends to make the skeleton of our pawn move.

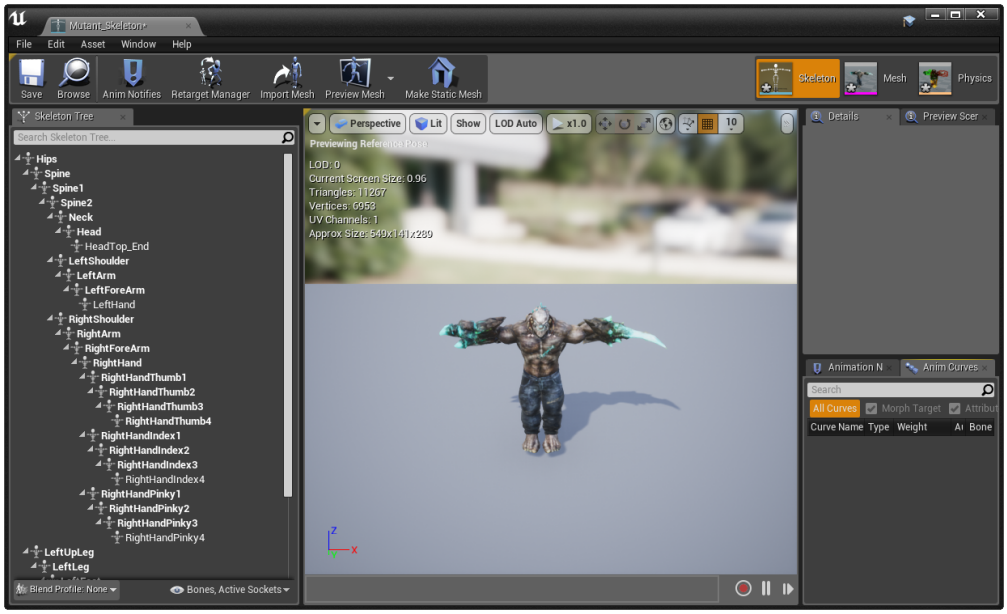


FIGURE 4.14: UE4 Skeleton Editor. Source: UE4 documentation.

The **Animation Blueprint Editor** will be the main point of study in the following subsections because it is where we will drive all the logic for our animations to play, that is why it's so important to describe the full environment.

4.4.2 Execution Flow

The **Animation Blueprint Editor** has two principal features, the Event Graph and the Animation Graph. Each graph is in charge of a different task, however they are coupled in a way that one needs from the other to be coherent.

Event Graph

The **Event Graph** is in charge of retrieving all the variables and functions belonging to our owning pawn using Blueprints as can be seen in figure 4.15.

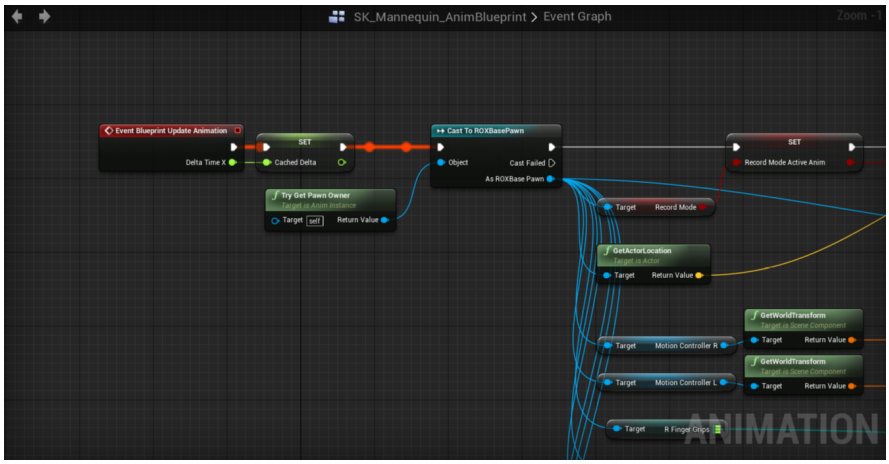


FIGURE 4.15: We can see how we use Blueprints to retrieve from the *ROXBasePawn* variables like *RecordMode* (showcased in red).

This graph holds all the desired logic we want to give our animations. We can create events, functions, sub-graphs and macros, like in any other UE4 Graph.

A peculiarity to comment about this Event Graph is the native event that we use to handle all our ticking logic, the *Event Blueprint Update Animation*. As we can see in Figure 4.15, this node returns the Delta Time X, which is the time between animation updates that gets cached on a variable for ease of use. However, this delta time is different than the global Tick (Update frame) time.

Another relevant node that we can observe on the same Figure is *TryGetPawnOwner*, which is exclusive to the Animation Blueprint Editor Event Graph. This node gets the owning pawn of this animation system, however if the owning pawn is invalid, it will retrieve *nullptr*. We take advantage of this node and use it to retrieve every useful value stored on our pawn, for that we will have to Cast to its type (since it inherits from pawn, it is a valid case).

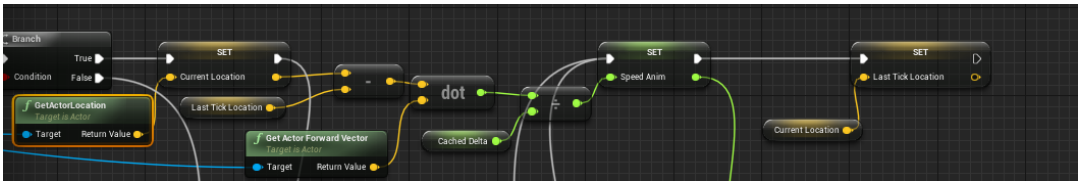


FIGURE 4.16: Velocity Calculation using Blueprints

Since we have access to the owning pawn, we can get its location and calculate a speed (Speed = Space/Time) as we can see in Figure 4.16. We calculate the Space by subtracting the Current Location with the previous location (stored on *LastTickLocation* variable).

Animation Graph

The **Animation Graph** will handle all the logic referred to the animations, it is totally relevant that we analyze deeply the execution order and how several conditions can affect and differ the output. The Figure 4.17 shows UnrealROX's Animation Graph along a reduced version of our main FSM.

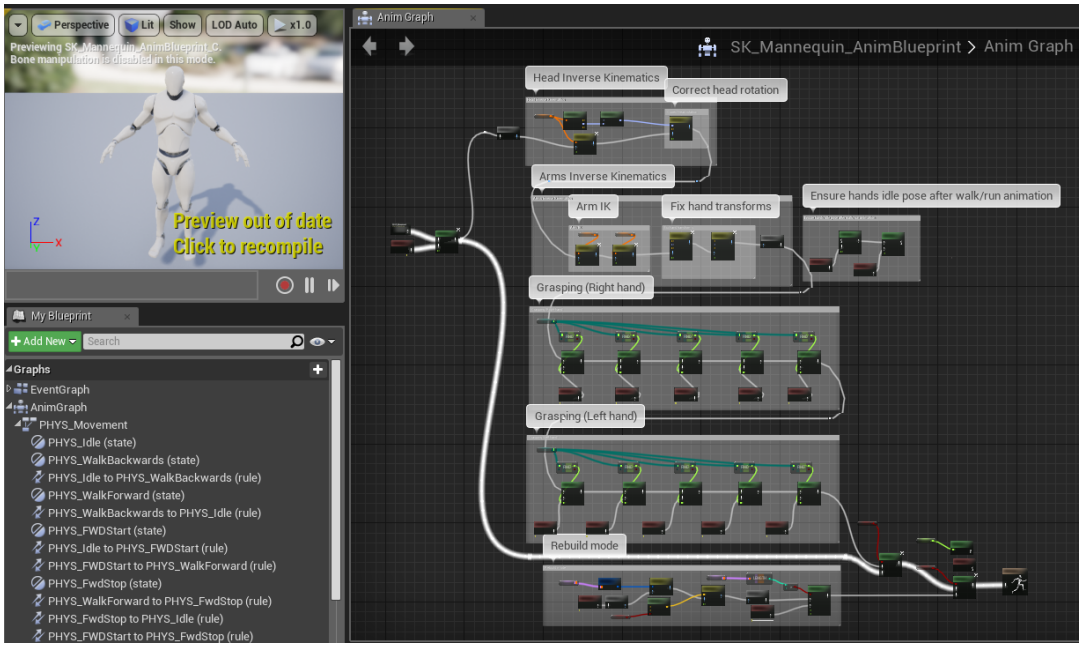


FIGURE 4.17: UnrealROX full Animation Graph, on the left side of the image we can see the main FSM, each circle corresponds to a single animation state, while the arrows refers to the transitions of these states.

As we can see in the previous Figure, the logic splits in two main branches that relate to the animation mode. UnrealROX counts with two custom modes, *record mode* and *rebuild mode*.

The **record mode** reads the retrieved variables on the event graph and applies these transformations to the bones through inverse kinematics. This mode plays the animations while the operator controls the VR peripherals described in Section 3.3.2.

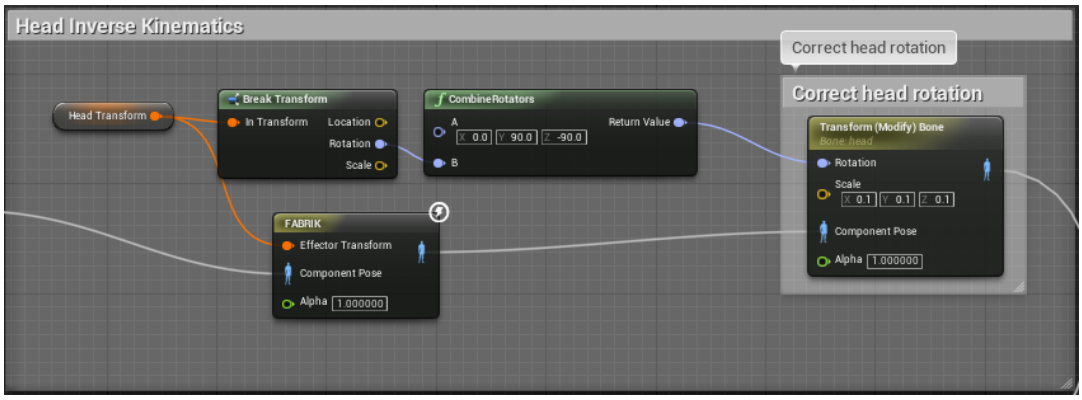


FIGURE 4.18: *HeadTransform* variable applied to the head skeleton bone. It makes the head rotate applying inverse kinematics.

An example of this can be seen in Figure 4.18 where we get the head transform, that has been calculated on the Event Graph (as we can see in Figure 4.19), and use it to apply a transform to the head bone and relatives, using the *FABRIK*[6] and the *Transform (Modify) Bone*[7] nodes.

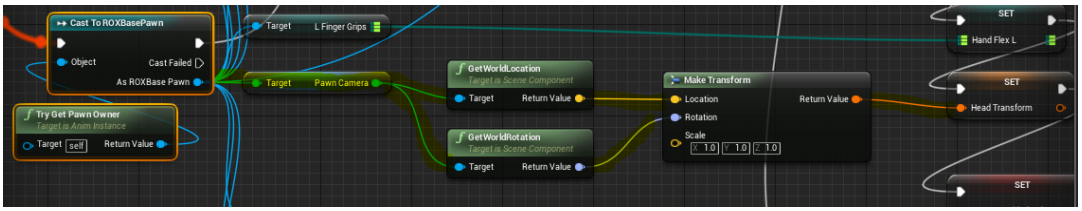


FIGURE 4.19: HeadRotation calculation based on the Pawn Camera.

The **rebuild mode** simply plays back all the movements we recorded previously. These movements, which were stored frame by frame on a text file as transformations, are retrieved in rebuild mode and applied to each related bone with a custom bulk operation (see Figure 4.20) that will be described in Section 4.4.4.

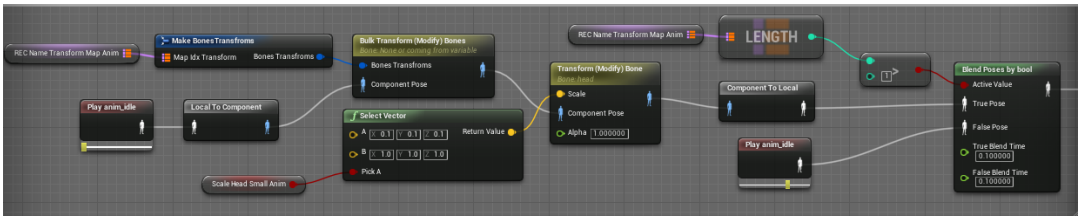


FIGURE 4.20: Bulk Transform (Modify) Bones applies transforms in bulk to specific bones.

One detail to comment about the previous Figure is the *ScaleHeadSmallAnim* variable, this variable determines if we should reduce the head size to 10% of the original value. We do that on first person mode, because some artifacts can be seen if the head is in place, so to avoid these artifacts, we reduce the size of the head notably.

We have two separated modes that we need to control separately, for that, we created the *RecordMode* variable we spoke about previously, this variable is in charge of deciding which execution branch will reach to the final Animation Pose.

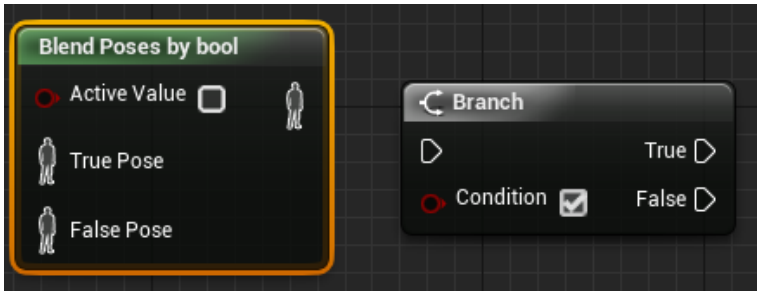


FIGURE 4.21: Left to Right: Blend Poses by bool and normal branch.

In order to do that we use the *Blend Poses by Bool* node. This node behaves like a normal branch node, the only difference is that it outputs a pose instead of a execution signal as we can see in Figure 4.21.

[6] https://docs.unrealengine.com/en-us/Engine/Animation/NodeReference/Fabrik

[7] https://docs.unrealengine.com/en-us/Engine/Animation/NodeReference/SkeletalControls/TransformBone

Apart from the record mode condition, in UnrealROX we also differentiate the situation in which the user does not have the headset on (*IsHMDEEnabledAnim*), so we can avoid the inverse kinematic and grasping logic that is not needed without VR. These branches are located at the end of our animation graph as the Figure 4.22 shows.

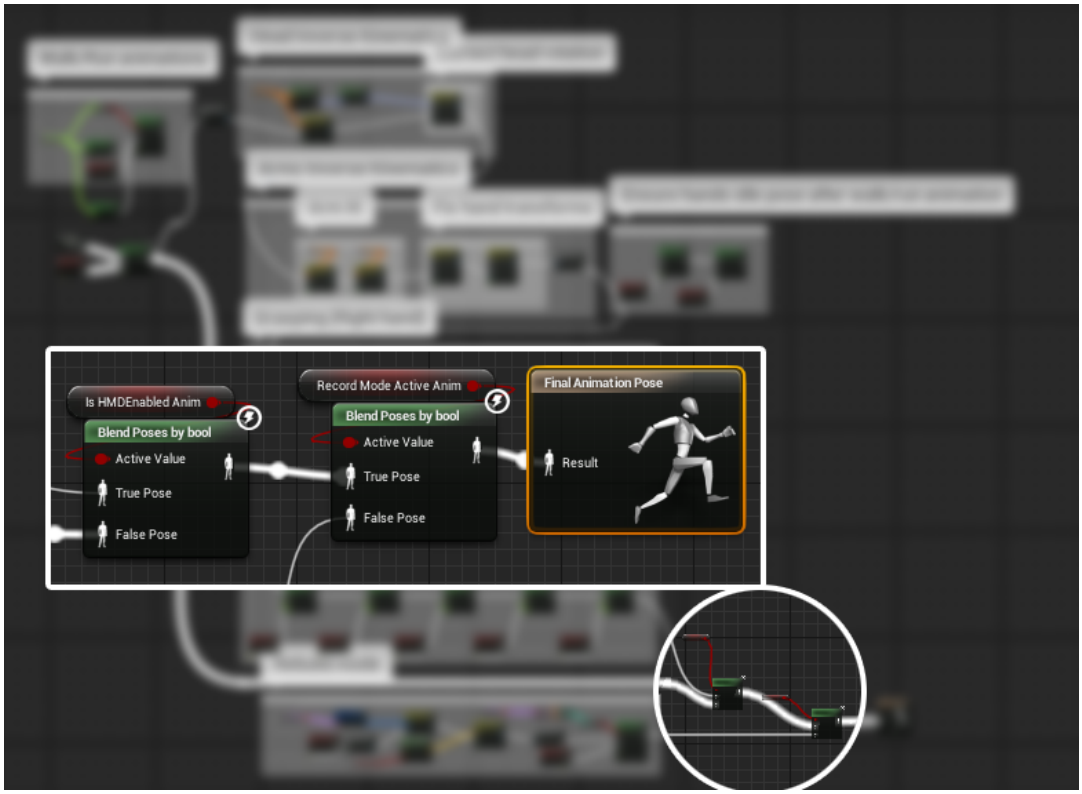


FIGURE 4.22: Main animation graph branches.

Figure 4.23 shows the main state machine used in UnrealROX, it can be extended with more states and transitions up to the user will. In this case the previously calculated Speed (Figure 4.16) drives the most part of the transition logic. Each state has a final animation pose that feeds our main animation graph, these states can be considered sub-animation graphs.

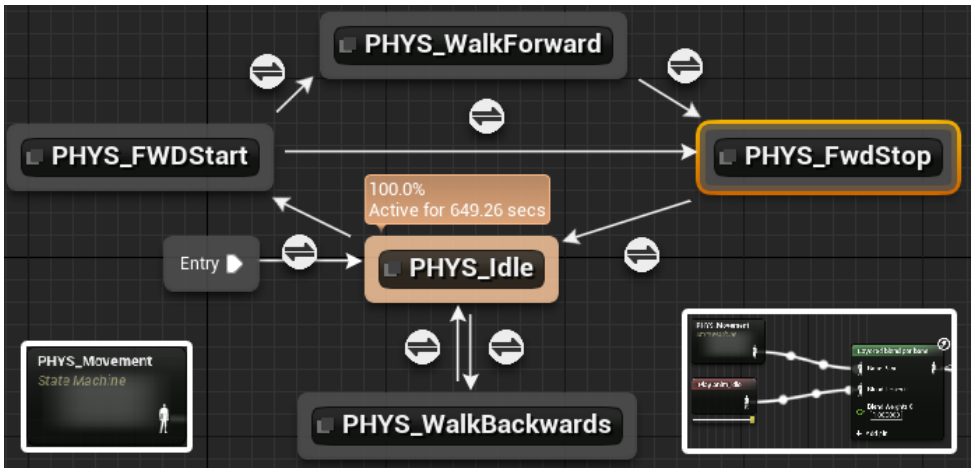


FIGURE 4.23: UnrealROX main animation finite state machine.

4.4.3 Inverse Kinematics

In order to apply several transformations to the skeleton of our pawn coherently, inverse kinematics techniques have had to be employed. Specifically, we have worked with chains of bones to simulate a more natural movement of the skeleton. The only inputs we have available are the position of the hands and head, all of this thanks to the virtual reality controllers. With only these positions we have to make all the body move accordingly in a cohesive way.

FIGURE 4.24: The poses that the human operator makes get translated to the mannequin appropriately thanks to the inverse kinematic techniques applied.

In order to get the effect achieved in Figure 4.24, we need to open our Animation Graph and analyze each specific skeleton bone that we are controlling using inverse kinematics.

The FABRIK node

The bone system of UE4 classifies the bones of the skeleton through a hierarchy of inheritance, this means that the bones of the fingers belong to the bone of the hand which at the same time belongs to the bone of the arm and so forth. We can use this hierarchy to define chains of bones affected by the movement of a single bone. This is done by the FABRIK node, which allows us to define a tip bone and a root bone. FABRIK will automatically handle the translations and rotations of the bones that are included in this defined chain as we can see in Figure 4.25.

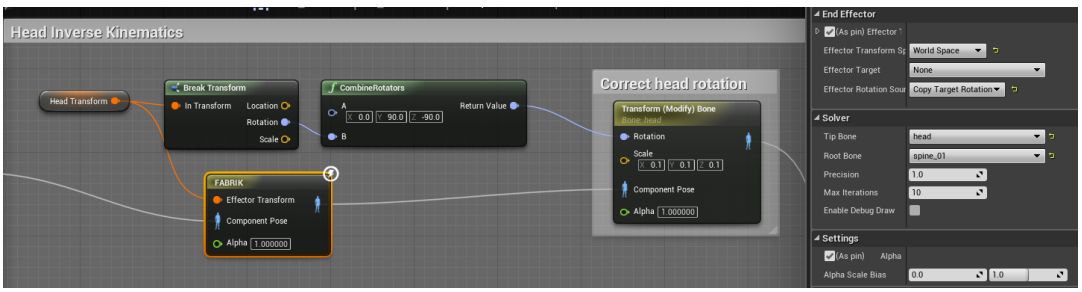


FIGURE 4.25: FABRIK node and its detail window in context.

The following visual representation in Figure 4.26 represents these two elements, the tip and the root. As we can see we have a direct relationship between the root bone that acts as a support bone and the tip bone, which is the *moving* bone on our bone chain, every movement made by the tip bone will adjust the other bones of this chain, setting their translations and rotations using the root bone as a base.

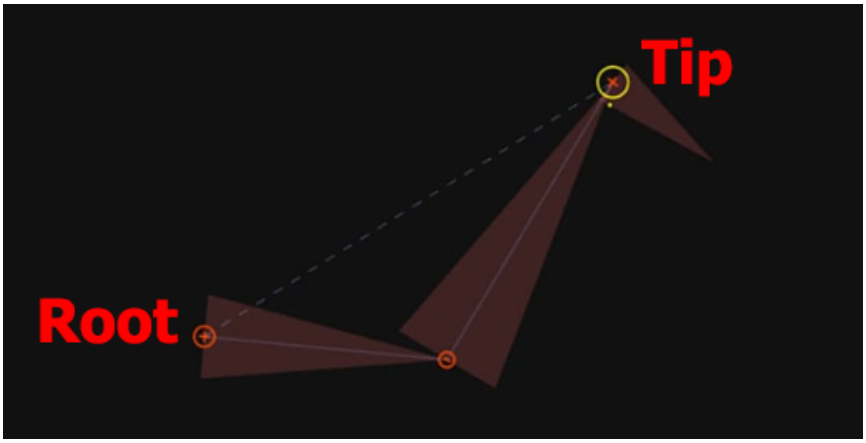


FIGURE 4.26: Basic inverse kinematics scenario. Source: Luis Martinez IK solver[8].

The example visualized at Figure 4.25 is using the head as the tip bone and the spine as the root bone. This means that if we incline our head forward, all the bones between the head and the spine will rotate accordingly, however it is impossible to make a total representation of the reality without more tracking points, so the system only tries to represent in the best possible manner the operator's pose.

4.4.4 Bulk Transform (Modify) Bones

Bulk Transform (Modify) Bones is a node that has been created to assign multiple transforms at multiple bones. To understand this node, we have first to take a look at the original *Transform (Modify) Bone* node.

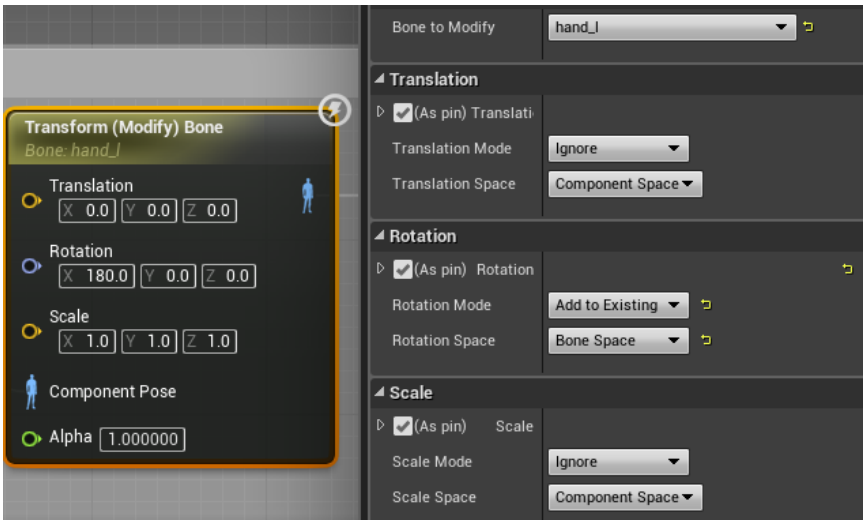


FIGURE 4.27: Transform (Modify) Bone node.

[8] https://vimeo.com/246238063

This node (represented in Figure 4.27) takes a pose as an input, on the detail panel of the node we can assign the bone we want to modify (in the case of the image *hand_l*), and then we can define the rules to apply for the new translation, rotation and scale applied to said bone, which modification gets returned on the output pose.

The main problem of this node is that it only allows us to select one single bone, so at the first stages of development it was very complicated to handle every bone in the *rebuild mode*, since the code was enormous. This led us to come up with a solution in which we could assign all these transformations at once without having to create an endless chain of nodes to handle each bone in the skeleton.

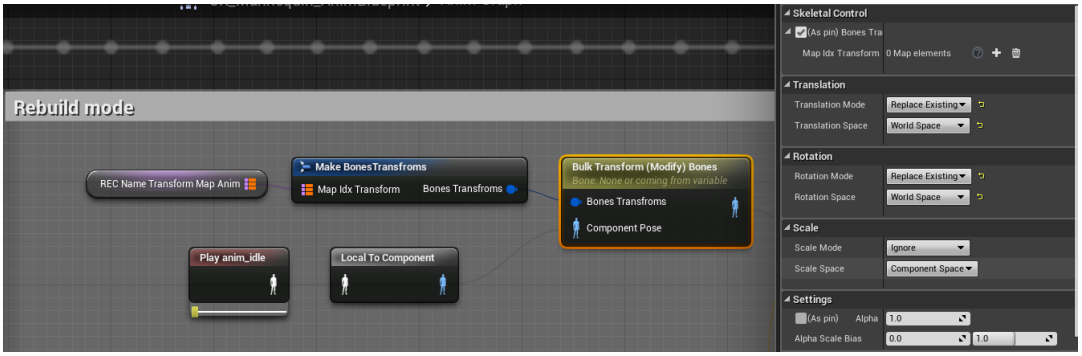


FIGURE 4.28: Bulk Transform (Modify) Bones in detail.

In Figure 4.28 we can see this node applied on the rebuild mode along its detail panel that doesn't differ too much from the original *Transform (Modify) Bone* node. The node receives a pose and a struct as parameters and returns a pose with the transforms applied. The struct holds a Hash Map that has as a key the name of each bone accompanied by the transform of that same bone as a value. This statement is valid since there cannot be bones on the UE4 skeleton that have repeated names.

The functionality of this node consists of modifying each bone to the transform assigned to it, which is why we have opted for a hash-map to create this relationship. That hash-map will be filled at runtime and will change every single *movement iteration* according to the previously *recorded* txt file.

The struct is necesary due to a limitation of UE4 that doesn't allow the user to have a hash map as an input for this kind of node, but it is totally redundant. The functionality of this node is as simple as described in Algorithm 2.

```
for pair in REC_NameTransformMap_Anim do
    if pair.bone in Pose.Bones then
        BoneId := Pose.GetId(pair.bone);
        Pose.Bones[BoneId].transform = pair.transform;
    end
end
```

Algorithm 2: Very simplistic version of the bulk modify bones algorithm.

UE4 animation system disposes of specific handles to extract the bone id given a bone name along another useful features, however that is way too specific for the purpose of this document. Algorithm 2 is a very naive and simple version of the real algorithm applied to modify all the transformations for all the bones.

4.5 The Player Controller

UE4 describes the *PlayerController* as the interface between the Pawn and the human player controlling it. The *APlayerController* class inherits from the *AController* class which is a non-physical Actor that can possess a Pawn (or Pawn-derived class) to control its actions.

Controllers receive notifications for many of the events occurring for the Pawn they are controlling. This gives the Controller the opportunity to implement the behavior in response to this event, intercepting the event and superseding the Pawn's default behavior.

In UnrealROX, the Player Controller (ROXPlayerController) is used as the main input handler class. This is where all the input propagates to the specific classes that need an entry point behavior. Another use we give the Controller is handling the movement of the controlled pawn.

In Section 4.5.1 we list all the inputs, then in Section 4.5.2 we will briefly describe the functionality of each one of those inputs as well as other details that will help for further implementations.

4.5.1 Input Definitions

In order to add input functionality to a *PlayerController* in UE4, we need to override the **SetupInputComponent()** function defined on the base *PlayerController* class. This function adds a *UInputComponent* to the Controller, and this component allows us to define inputs by name and add behavior to them as shown in the figure 4.29.

```
void AROXPlayerController::SetupInputComponent()
{
    Super::SetupInputComponent();
    InputComponent->BindAxis("MoveForward", this, &AROXPlayerController::MoveForward);
    InputComponent->BindAxis("MoveRight", this, &AROXPlayerController::MoveRight);
    InputComponent->BindAxis("Turn", this, &AROXPlayerController::AddControllerYawInput);
    InputComponent->BindAxis("TurnRate", this, &AROXPlayerController::TurnAtRate);
    InputComponent->BindAxis("LookUp", this, &AROXPlayerController::AddControllerPitchInput);
    InputComponent->BindAxis("LookUpRate", this, &AROXPlayerController::LookUpAtRate);
    InputComponent->BindAxis("MoveCameraUpDown", this, &AROXPlayerController::MoveCameraUpDown);
    InputComponent->BindAxis("GraspRightHand", this, &AROXPlayerController::GraspRightHand);
    InputComponent->BindAxis("GraspLeftHand", this, &AROXPlayerController::GraspLeftHand);

    InputComponent->BindAction("MoveVRCamControllersModifier", IE_Pressed, this, &AROXPlayerController::ToggleMoveVRCamControllersModifier);
    InputComponent->BindAction("MoveCamModifier", IE_Pressed, this, &AROXPlayerController::ToggleMoveCameraModifier);
    InputComponent->BindAction("StartRecording", IE_Pressed, this, &AROXPlayerController::OnStartRecording);
    InputComponent->BindAction("ResetVR", IE_Pressed, this, &AROXPlayerController::OnResetVR);
    InputComponent->BindAction("ShowCamTexture", IE_Pressed, this, &AROXPlayerController::OnShowCamTexture);
    InputComponent->BindAction("RestartLevel", IE_Pressed, this, &AROXPlayerController::OnRestartLevel);
    InputComponent->BindAction("StartProfiling", IE_Pressed, this, &AROXPlayerController::OnStartProfiling);
    InputComponent->BindAction("ShowDebugging", IE_Pressed, this, &AROXPlayerController::OnShowDebugging);

    InputComponent->BindAction("ChangeLit", IE_Pressed, this, &AROXPlayerController::Lit);
    InputComponent->BindAction("ChangeVertex", IE_Pressed, this, &AROXPlayerController::Object);
    InputComponent->BindAction("ChangeDepth", IE_Pressed, this, &AROXPlayerController::Depth);
    InputComponent->BindAction("ChangeNormal", IE_Pressed, this, &AROXPlayerController::Normal);

    InputComponent->BindAction("CameraNext", IE_Pressed, this, &AROXPlayerController::CameraNext);
    InputComponent->BindAction("CameraPrev", IE_Pressed, this, &AROXPlayerController::CameraPrev);
    InputComponent->BindAction("TakeScreenshot", IE_Pressed, this, &AROXPlayerController::TakeScreenshot);
    InputComponent->BindAction("TakeDepthScreenshot", IE_Pressed, this, &AROXPlayerController::TakeDepthScreenshot);

    InputComponent->BindAction("SetRecordSettings", IE_Pressed, this, &AROXPlayerController::SetRecordSettings);
}
```

FIGURE 4.29: UnrealROX input definition.

The Input component exposes two functions that helps the user to define a specific behavior for a specific axis or action.

- **Bind Axis:** Binds a delegate function an Axis defined in the project settings.
- **Bind Action:** Binds a delegate function to an Action defined in the project settings.

Action Mappings are for key presses and releases, while Axis Mappings allow for inputs that have a continuous range. These mappings can be configured on UE4 under *"ProjectSettings/Input"* as can be seen in Figure 4.30.

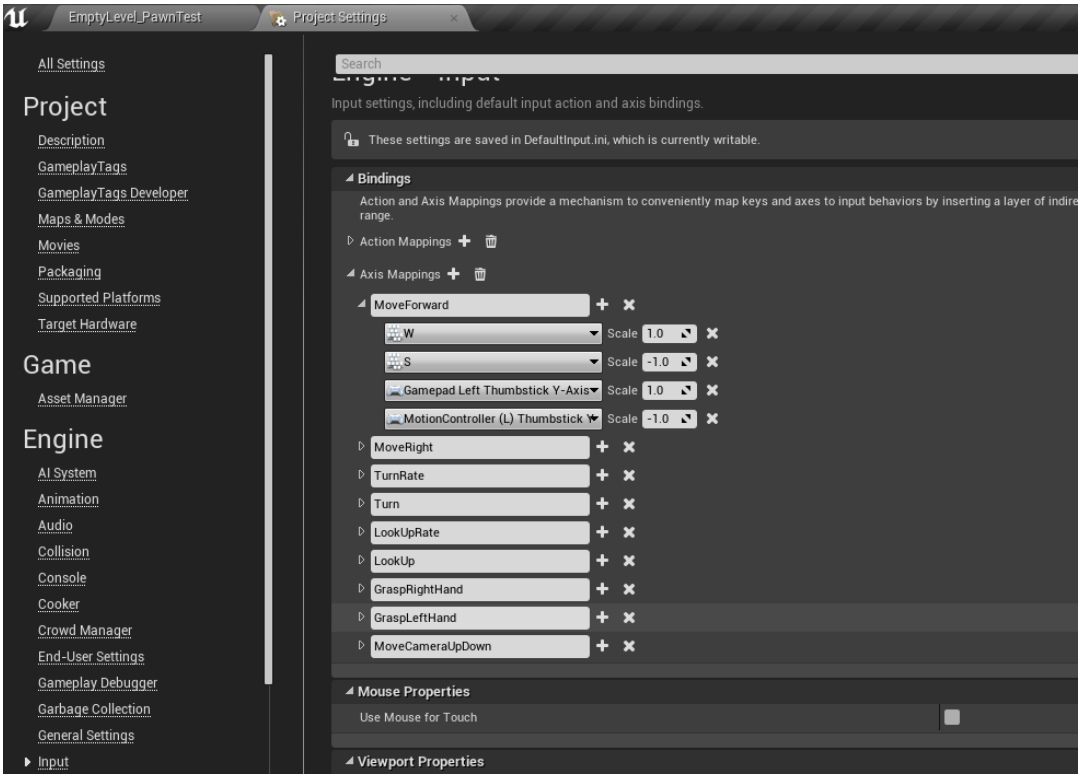


FIGURE 4.30: Inputs defined at Project Settings.

This is where we define the default keys-action/axis relationships of our project, as we can see these names need to match with the action/axis names we set in the controller on the **SetupInputComponent()** function as we saw in Figure 4.29.

4.5.2 Main Behavior

Before looking at all input functions seen above, it is necessary to comment on the BeginPlay function of *ROXPlayerController*. BeginPlay is a function that gets called when an Actor object got properly instantiated and initialized. Since APlayerController belongs to the AActor inheritation chain and it's an available function to use, we can override it to implement any behavior when the Actor is created. On algorithm 3 we can see a simplistic version of the *ROXPlayerController* BeginPlay.

```
CachedPawn = Cast<AROXBasePawn>(this->GetPawn());

if XRSystem->IsHeadTrackingAllowed() then
    isHMDEnabled = true;
    HMDDvcType = XRSystem->GetHMDDevice()->GetHMDDeviceType();
end
```

Algorithm 3: On the first line we cache the casted Pawn in order to avoid casting every time we need to access any specific feature of our subclass (GetPawn() returns APawn type). Next, we gather information about the VR device in use.

Once we have all this information cached, we can look at the rest of the code that is handled by the input. In order to follow the subsequent points it is recommended to look at Figure 4.29.

- **Move functions:** Move forward and move right simply move the pawn onto the desired direction applying an acceleration formula, this acceleration formula is defined at **JoystickAxisTunning(float x)**.
- **Turn and Look:** These four functions are in charge of rotating the camera according to the user view.
- **MoveCameraUpDown:** This function moves on the Z axis the camera location at the user will.
- **Grasp functions:** Propagate input to the pawn, which will handle the grasping of each hand.
- **Move Modifiers:** These functions toggle the functionality of moving the camera with the axis.
- **Start Recording:** It toggles the recording function of the tracker by accessing the tracker from the cached pawn.
- **Reset VR:** Executes the console command "vr.HeadTracking.Reset" which resets the VR virtual headset position to the origin.
- **ShowCamTexture:** It is the input handler to activate the cam texture, as seen in Figure 4.13.
- **Restart Level:** It restarts the current level, meaning that all the objects we manipulated in runtime will return to their original state.
- **Start/Stop Profiling:** Delegates input to the HUD to handle this profiling function as described in Section 4.3.
- **ShowDebugging:** Toggles on and off the UE4 stats handled in the HUD class.
- **Change Lit/Vertex/Depth/Normal:** Propagates input to the tracker through the cached pawn in order to change the view mode.
- **Camera Next/Prev:** Manual handlers for the camera iterator to set a camera for the controlled pawn, this is all controlled by the tracker so we need to access it through the cached pawn.
- **Take Screenshot:** Calls this function on the tracker through the cached pawn.
- **Set Record Settings:** Prepares the map for the recording state. This is partially handled by the controller and the tracker. The controller sets the viewmode to unlit and the FPS to 60 to record at that rate, while the tracker will erase every single light actor of the scene to get the most performance friendly recording environment.

4.6 The Pawn

The Pawn class, as UE4 defines it, is the base class of all Actors that can be controlled by the players or the AI. A Pawn is the embodiment of a player or AI actor in the world. This not only translates into the Pawn visually defining how the AI actor or player looks, but also how it interacts with the world. By default, there is a one-to-one relationship between Controllers (see Section 4.5) and Pawns; meaning, each Controller controls only one Pawn.

In UnrealROX, the Pawn (ROXBasePawn) handles the Bone-Camera subsystem that we saw in Section 4.2. This class is also responsible for defining the Skeletal Mesh that the pawn uses, we can also choose the Animation Graph that this mesh will use, the responsibilities of the Animation Graph are defined in Section 4.4.

In Section 4.6.1 we will see and explain in detail the CDO of the ROXBasePawn as well as the initialization functions like *BeginPlay*, then in Section 4.6.2 we will describe a number of functions of the pawn that serve to return information to the classes that require it, like the Tracker class, which we will describe in Section 4.7. Finally in Section 4.6.3 we will explain the grasping system.

4.6.1 Initialization

The initialization of an Actor in UE4 has various phases. Once it gets instantiated, the very early phase consists on retrieving the values of the CDO of the Actor. These values can be defined on the *Details Panel* of a child Blueprint Actor as seen in Figure 4.31. However, if we instantiate the parent C++ class of that Blueprint child, the values will be retrieved directly from the constructor of that class.

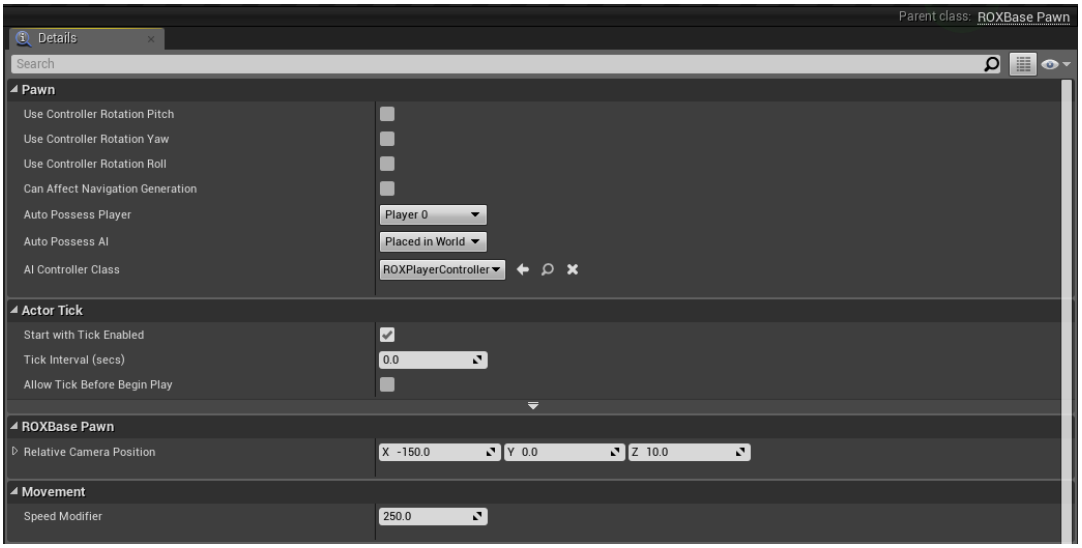


FIGURE 4.31: Excerpt of the editor details panel of the ROXMannequinClass (Child of ROXBasePawn).

Variables need to be marked in the C++ class as UPROPERTY with the specifier EditDefaultOnly to be able to access them from the details panel in the child Blueprint class (Figure 4.31). These specifiers work along with the UE4 reflection system and are defined on their documentation about UProperties[9]. Besides this, if we create a variable directly in Blueprints, UE4 will add it to the list of properties in the details panel of that Blueprint class and their childs.

[9] https://wiki.unrealengine.com/UPROPERTY

UnrealROX Pawn variable definition is all contained on the parent C++ AROXBasePawn class, which already contains all the properties inherited from its parent APawn. The following figure 4.32 is a section of the constructor where the non-component variables are initialized.

```
// Sets default values
AROXBasePawn::AROXBasePawn()
    : bRecordMode(true)
    , SpeedModifier(250.f)
    , isHMDEnabled(false)
    , bScaleHeadSmall(false)
{
```

FIGURE 4.32: Specific variables initialized at the Pawn constructor.

Next we proceed to describe the properties seen in the image:

- **bRecordMode:** Used to know if we are in recording mode in the current execution.
- **SpeedModifier:** Pawn Movement Speed.
- **isHMDEnabled:** Whether or not switching to stereo is enabled; if it is false, then EnableStereo(true) will do nothing. Defines if the game is in VR Mode.
- **bScaleHeadSmall:** Defines if the head of the mannequin should be scaled down, as we can see in Figure 4.20.

```
{
    PrimaryActorTick.bCanEverTick = true;

    RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("RootComponent"));

    MeshComponent = CreateOptionalDefaultSubobject<USkeletalMeshComponent>(TEXT("Mesh"));
    MeshComponent->SetupAttachment(RootComponent);
    MeshComponent->SetRelativeRotation(FRotator(0.0f, -90.0f, 0.0f));
    MeshComponent->bEnableUpdateRateOptimizations = false;
    MeshComponent->MeshComponentUpdateFlag = EMeshComponentUpdateFlag::AlwaysTickPoseAndRefreshBones;

    VRTripod = CreateDefaultSubobject<USceneComponent>(TEXT("VRTripod"));
    VRTripod->SetupAttachment(RootComponent);
    //VRTripod->SetRelativeLocation(RelativeCameraPosition);

    VRCamera = CreateDefaultSubobject<USceneComponent>(TEXT("VRCamera"));
    VRCamera->SetupAttachment(VRTripod);

    PawnCamera = CreateDefaultSubobject<UCameraComponent>(TEXT("PawnCamera"));
    PawnCamera->SetupAttachment(VRCamera);
    //PawnCamera->SetRelativeLocation(RelativeCameraPosition);

    VROrigin = CreateDefaultSubobject<USceneComponent>(TEXT("VROrigin"));
    VROrigin->SetupAttachment(VRTripod);

    MotionController_R = CreateDefaultSubobject<UMotionControllerComponent>(TEXT("MotionController_R"));
    MotionController_R->SetupAttachment(VROrigin);
    MotionController_R->Hand = EControllerHand::Right;

    MotionController_L = CreateDefaultSubobject<UMotionControllerComponent>(TEXT("MotionController_L"));
    MotionController_L->SetupAttachment(VROrigin);
    MotionController_L->Hand = EControllerHand::Left;

    AutoPossessPlayer = EAutoReceiveInput::Player0;
    AIControllerClass = AROXPlayerController::StaticClass();

    SetupHandsCapsuleColliders();
}
```

FIGURE 4.33: ROXBasePawn Constructor body.

The rest of the ROXBasePawn class constructor initializes some of the properties defined in the header, especially components as we can observe in Figure 4.33.

MeshComponent is a variable of type *USkeletalMeshComponent* which receives a USkeletalMesh as a variable, which we can assign in one of the children of this Blueprint. We can set this mesh in C++, but it's inconvenient since we have to work with relative routes to find the editor asset, that's why an inherited Blueprint class is used for these purposes as the Figure 4.34 shows.

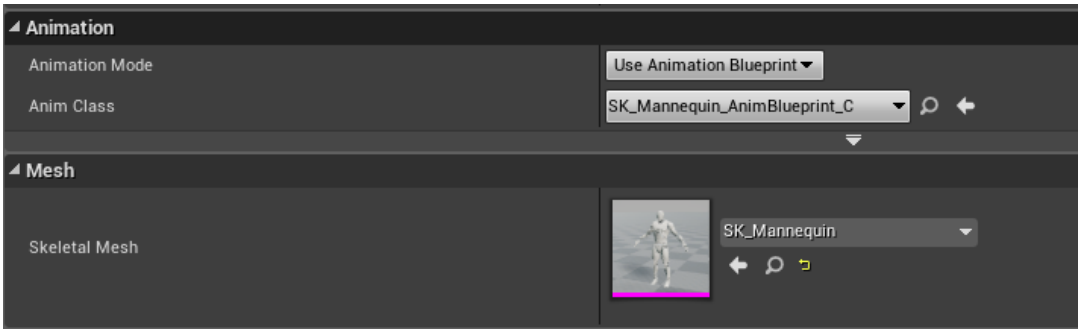


FIGURE 4.34: Skeletal Mesh and Blueprint graph assets for the USekeletalMeshComponent.

VRTripod will be used as the root component for all the VR subcomponents. Thanks to the hierarchy of unreal components, if we move the root component, all subcomponents associated with it, will move too. This can be seen visually better in the editor as Figure 4.35 shows.

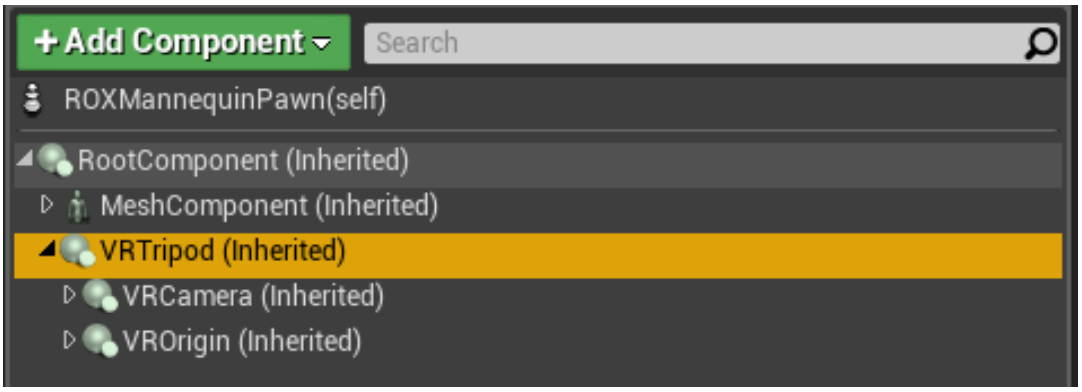


FIGURE 4.35: VRCamera and VROrigin are attached to the VRTripod which is acting as the root for this component chain.

At the same time, **PawnCamera**, which is the first person camera of this pawn class, is attached to **VRCamera**. The Motion controllers are children of VROrigin. These motion controllers are the components in charge of communicating the hand controllers with UE4, giving it input information as well as transform information.

The last relevant bit in the constructor class is the *SetupHandsCapsuleColliders* function, which will setup all the colliders for the hands of the Pawn. This function creates as many colliders as we need and places them in the fingers of the mannequin we are working with. It also defines the event dispatchers we need for the collision of a specific collider with a world object.This is specific to the grasping method used in the project. In our case, grasping is based on collision with capsules in the hand of the mannequin, as we will elaborate in Section 4.6.3.

Once we understand how the CDO works, we can explain the latest initialization phase utilized on this specific Actor: The *BeginPlay*.

As we described in Section 4.5.2, this event is triggered for all Actors when the game is started, any Actors spawned after the game is started will have this called immediately.

```
Super::BeginPlay();

if GEngine->XRSystem.IsValid() then
    if GEngine->XRSystem->IsHeadTrackingAllowed() then
        isHMDEnabled = true;
    end
end

if !isHMDEnabled then
    PawnCamera->SetRelativeLocation(RelativeCameraPosition);
end

for const EHandFinger Finger : EHandFingerAll do
    R_FingerGrips.Emplace(Finger, 0.0f);
    R_FingerOverlapping.Emplace(Finger, false);
    R_FingerBlocked.Emplace(Finger, false);
    L_FingerGrips.Emplace(Finger, 0.0f);
    L_FingerOverlapping.Emplace(Finger, false);
    L_FingerBlocked.Emplace(Finger, false);
end

for i = 0; i < MeshComponent->GetNumBones(); ++i do
    REC_NameTransformMap.Emplace(MeshComponent-
     >GetBoneName(i), FTransform());

end

UpdateBoneCamsArray();
SetupHandsCapsuleCollidersAttachment();

CachedPC = Cast<AROXPlayerController>(GetController());
```

Algorithm 4: Updating finger block and calling SmoothGrasp if conditions are met.

Next, we will describe Algorithm 4 in order. The first thing we do in BeginPlay is to check if we have some kind of virtual reality device connected. If the answer is yes, we will put isHMDEnabled to true. However, if this is not met, we will put the camera in third person mode by just setting the relative location of it. Following next, we initialize the hash maps for our grasping algorithm by just iterating through the Finger enum (which will be described in Section 4.6.3). The next loop initializes the NameTransformMap which stores the transformations that must be applied to each bone, it is used in Rebuild mode to receive the joint transformations from the Tracker and send them to the AnimGraph. UpdateBoneCamsArray initializes the *BoneCams* array documented in Section 4.2.1. Next *SetupHandsCapsuleCollidersAttachment* attaches all the finger colliders to the skeletal mesh so they follow the skeleton animations. Finally, we cache the Controller to avoid extra casting.

4.6.2 Main Handlers

The Pawn is constantly accessed from external classes, that's why we have certain public handlers that let the user retrieve information about it or update a specific variable. In this section we won't discuss the handlers used for the Grasping algorithm, which will be documented in Section 4.6.3.

The first handler we find, which has to do with the initialization section, is **InitFromTracker**, this function is called from the Tracker object and sets the recording mode in which we are, also caches the tracker to access it from the Pawn. In addition, if we are in playback mode, we deactivate the user input and disable the HUD.

The second handler is **MoveCameraRelative**, this function is accessed by the *ROXPlayerController*, and it simply moves the camera in the desired direction by using a vector as an input, which will work as a position offset. This handler is called from the camera movement functions defined in Section 4.5.2.

The next function is **MoveVRCameraControllersRelative**, which is in charge of moving the **VRTripod** we saw on Figure 4.35. This is called on the Controller when we want to relocate the VR subcomponents in a new location, we do this specifically when we move the camera to a new location, so the VR handlers are placed appropriately in relation to the camera (which represents our head on VR).

Tick is a function included in the AActor class which gets called once per frame. The tick interval can be altered, but for the scope of this project we don't need to change the tick rate. In ROXPlayerController, the Tick will set the proxy PawnCamera (PawnCameraSub) position to where the CameraComponent of the Pawn is located at by just getting its transform. We use this camera Actor proxy because the real Pawn camera is a UCameraComponent, and the PlayBack algorithm works only with ACameraActors, recapitulating Section 4.2.4.

The fifth handler is **CameraPitchRotation**, this function is called from the Controller to modify the PawnCamera pitch rotation.

ChangeViewTarget is a function called from the Tracker object that changes the current active point of view (camera) of the controller. This function is used on the Tracker algorithm to swap between the cameras that are placed in the scene or in the pawn skeleton.

The next handler is **CheckFirstPersonCamera**, which is called from the Tracker and it checks if the input CameraActor is the proxy camera we use to represent the first person camera (PawnCameraSub). This function will scale down the head and return true if the input camera is the one used to represent the first person pawn camera. The Tracker algorithm calls this function every time a new camera is swapped, that's why we reduce the head when the input camera matches, we don't want artifacts on a first person view.

The last function we will be documenting in this section is **PrintHUD**. PrintHUD is a function that communicates with the HUD interface to show a screen message. This interface has been documented in Section 4.3, where we explain in detail the user interface system of UnrealROX.

In addition to these handlers, ROXBasePawn implements inlined getters for several variables: the record mode, the tracker, the speed modifier, the name transform hash map and the mesh component.

4.6.3 Grasping

As we stated previously, UnrealROX grasping algorithm is based on the collision produced by the capsules attached to the hand of the skeletal mesh of the Pawn with a world object eligible for this interaction. These capsules need to be placed manually by the user according the skeleton hands. Figure 4.36 shows the hands colliders attached to the Mannequin Pawn, which is the Pawn example UnrealROX provides.

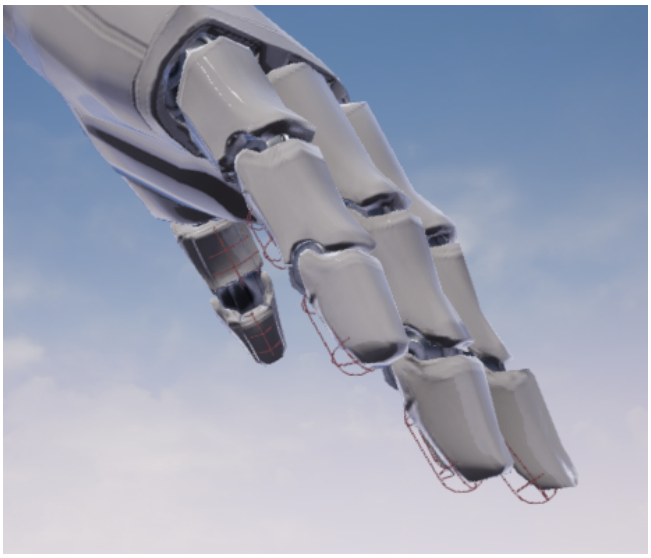


FIGURE 4.36: Capsule colliders placed manually on the hand.

Next, we will explain the algorithm in a more precise way, which is implemented in ROXBasePawn.

As we explained in Section 4.6.1, the constructor of the Pawn will be in charge of creating and initializing each of these capsules, as well as placing them in the correct position for the Pawn. The initialization method is the same for each of these colliders, as we can see on Figure 4.37.

```
Thumb_3R = CreateDefaultSubobject<UCapsuleComponent>(TEXT("Thumb_3R"));
Thumb_3R->SetupAttachment(MeshComponent);
Thumb_3R->SetRelativeTransform(FTransform(FRotator(0.0f, 100.0f, 92.0f), FVector(-1.8f, 1.0f, 0.1f)));
Thumb_3R->SetCapsuleSize(0.6f, 1.0f);
Thumb_3R->bGenerateOverlapEvents = true;
Thumb_3R->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);
Thumb_3R->SetCollisionResponseToChannel(ECollisionChannel::ECC_WorldDynamic, ECollisionResponse::ECR_Ignore);
Thumb_3R->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn, ECollisionResponse::ECR_Ignore);
Thumb_3R->SetCollisionObjectType(ECollisionChannel::ECC_WorldDynamic);
Thumb_3R->OnComponentBeginOverlap.AddDynamic(this, &AROXBasePawn::OnThumb_3ROverlapBegin);
Thumb_3R->OnComponentEndOverlap.AddDynamic(this, &AROXBasePawn::OnThumb_3ROverlapEnd);
Thumb_3R->SetHiddenInGame(false);
```

FIGURE 4.37: Initialization of the Tumb_3R capsule collider.

The first line creates the component then we set its parent using the **SetupAttachment** function. **SetRelativeTransform** is the function we use to properly place these colliders, and with **SetCapsuleSize** we can adjust the size of this collider. Next, we activate the Overlap events for this specific collider setting *bGenerateOverlapEvents* to true, and we set up some collision rules. Finally, we can bind the overlapping events to whatever function we have in our class; as soon as it meets the function requirements[10] (same signature as the Overlap engine functions).

[10] https://api.unrealengine.com/INT/API/Runtime/Engine/Components/UPrimitiveComponent/OnComponentBeginOverlap/index.html

As we documented previously, the **BeginPlay** will iterate through the *EHandFinger* to initialize the grasping maps for both hands. These maps control the finger grips, the overlapping and the block based on certain conditions that we will study later in this document. The initialization is simple, as we can see in Algorithm 4.

Once everything is initialized, the next step is analyzing the entry points, which are GraspRightHand and GraspLeftHand, as we documented in Section 4.5.2. We'll only cover GraspRightHand, since the behavior for both Grasping functions is the same.

GraspRightHand works like an axis, it takes a value and it calls a function which will do the grasping algorithm. Since the axis are ticking, we can have a continuous control of the user input, hence the *Grasp* function will be called every tick, the only difference will be the *R_CurrentGrip* value, which is set to the axis value (how much the user presses a button, like a joystick).

The Grasp function is reused for the both hands thanks to the input parameters. The following list includes all the input parameters as well as an explanation of each of them.

- **float CurrentGrip:** Current axis value.
- **TMap<EHandFinger, float> &FingerGrips:** Map that stores the fingers bending value.
- **TMap<EHandFinger, bool> &FingerOverlapping:** Map that holds the overlapping state of each finger.
- **TMap<EHandFinger, bool> &FingerBlocked:** Map that determines if a finger should be blocked from bending or not.
- **FName BoneName:** Bone where to attach the grasped object, in this case *hand_r* or *hand_l*.
- **AActor* &OverlappedActor:** Current Overlapping Actor.
- **bool &isActorAttached:** Defines if the Overlapped Actor is attached.
- **bool &DisableGrab:** Determines if the current hand can grab an object.
- **AActor* &AttachedActor:** Holds a reference to the currently grasped Actor.
- **bool &isActorAttachedOtherHand:** This property will be true if an Actor is attached to the "other hand" and not to the one we are checking in this execution. This is done because we can have two different actors attached at the same time, one on each hand. The overlapped actor we receive can be different for each hand, so we only want isActorAttachedOtherHand and AttachedActorOtherHand in order to check if we are passing the same object from one hand to the other, as this case requires some special logic.
- **bool &DisableGrabOtherHand:** Determines if the opposite to the current hand can grab an object. It is useful to avoid attaching and detaching the object repeatedly (and quickly) between hands. After detaching from one hand, that hand won't be able to grab any other object until it is completely opened.
- **AActor* &AttachedActorOtherHand:** Holds a reference to the currently grasped Actor on the other hand.

As we saw, the Grasp function needs information about the currently overlapped actor, this information will be retrieved from the callbacks we defined in the colliders initialization. The Figure 4.38 shows these overlapping functions that will grant this information to the Grasping method.

```
bool AROXBasePawn::SetOverlap(TMap<EHandFinger, bool> &FingerOverlapping, TMap<EHandFinger, bool>
{
    FingerOverlapping.Emplace(Finger, true);
    AlreadyOverlappedActor = OverlappedActor;

    if (CurrentGrip >= GripDeadZone)
    {
        FingerBlocked.Emplace(Finger, true);
        PrintHUD("Finger overlap: " + EHandFingerToString(Finger));
    }

    return CurrentGrip >= GripDeadZone;
}

bool AROXBasePawn::SetOverlapEnd(TMap<EHandFinger, bool> &FingerOverlapping, EHandFinger Finger)
{
    FingerOverlapping.Emplace(Finger, false);
    PrintHUD("Finger end overlap: " + EHandFingerToString(Finger));
    return true;
}
```

FIGURE 4.38: Begin and End overlap callbacks for the hand colliders.

As we can see, the **SetOverlap** function sets the OverlappingActor (passed by reference) that we will use later on the Grasping method. **SetOverlapEnd** only updates the overlapping state of that finger by accessing the map.

Now that we know where the data comes from, we can disassemble the Grasp function better. But first, let's define it's purpose.

The Grasp function needs to move all the fingers of the hand following a grasping animation until they collide with something. If a finger of the hand collided with something, it will be blocked on the position it collided, meaning that it won't be able to go through the object. The same process is repeated for the rest of the fingers.

for *auto Entry : FingerBlocked.CreateIterator()* **do**

```
const EHandFinger& Finger = Entry.Key();
const bool& bFingerBlocked = Entry.Value();
float fFingerGrip = *FingerGrips.Find(Finger);

bool bFingerOverlapping = *FingerOverlapping.Find(Finger);
bool bSmoothGrasp = false;

...
```

end

Algorithm 5: Iterating through the fingers array.

First, we iterate through the fingers array caching out all the information we need for its post process as we can see in Algorithm 5.

Following next, if the finger in this map element is blocked, then we update the block state of this finger to false and set *bSmoothGrasp* to true to call it afterwards.

```
...
if bFingerBlocked then
    if CurrentGrip < fFingerGrip then
        FingerBlocked.Emplace(Finger, false);
        bSmoothGrasp = true;
    end
else
    bSmoothGrasp = true;
end
if bFingerBlocked then
    SmoothGrasp(FingerGrips, Finger, CurrentGrip, 0.6f);
end
```

Algorithm 6: Updating finger block and calling SmoothGrasp if conditions are met.

SmoothGrasp simply updates smoothly the position of the fingers by accessing the *FingerGrips* hash map having in consideration the delta time to achieve a slow and smooth grasp movement. Then, we retrieve overlapping information to know if an object can be grasped, as we can see in Figure 4.39.

```
// Detect grasp and attach object to hand
bool isThumbOverlapping = *FingerOverlapping.Find(EHandFinger::HF_Thumb_3);
bool isIndexOverlapping = *FingerOverlapping.Find(EHandFinger::HF_Index_3);
bool isMiddleOverlapping = *FingerOverlapping.Find(EHandFinger::HF_Middle_3);
```

FIGURE 4.39: The three important fingers to consider an object to be grasped are the thumb, the index and the middle finger.

Then based on that information, we will decide if we should grasp or not an object. Specifically the rules would go as it follows: If the thumb is overlapping along the index or the middle finger, then we can attach the overlapped actor to the proper hand socket, however in order to do this, the actor can't be already attached and the hand can't be marked as *Disabled*. If we grasp an object that is already hold by the other hand, we will just detach this object from the other hand and attach it to the hand we are grasping with. If the first condition wasn't met and we have an actor attached, we'll just detach it.

To attach and detach actors to the hands of the skeleton, we'll use the built in UE4 functions *AttachToComponent*[11] and *DetachFromComponent*[12]. The latter needs to re-enable the physics of the object as well as enable its collision.

[11] https://api.unrealengine.com/INT/API/Runtime/Engine/Components/USceneComponent/AttachToComponent/index.html

[12] https://api.unrealengine.com/INT/API/Runtime/Engine/Components/USceneComponent/DetachFromComponent/index.html

4.7 The Tracker

The Tracker Actor is one of the most important pieces of *UnrealROX*, as it is the class responsible for recording and reproducing the actions of the scene, including the Pawn.

First in Section 4.7.1 we will explain what is the Tracker and how to use it on *UnrealROX*. Then, in Section 4.7.3, we will explain how the scene actions are saved to a text file which we later interpret with the playback mechanism explained in Section 4.7.4. We will also see how the tracker gets initialized in Section 4.7.2.

4.7.1 Usage

As we explained in the introduction, the Tracker is the class in charge of recording and reproducing a scene to obtain the dataset. Without a Tracker Actor in a scene none of this can be achieved.

Now that we know what is the responsibility of this Actor, we can explain how to use it inside UnrealROX.

The Tracker can be found under the default folder *C++ Classes* in the Content Browser[13] (this directory only appears in the content browser if you are working with a C++ UE4 project) under the robotrix folder in the public category, we can use the searching function of the content browser to find it immediately as we can see in Figure 4.40.

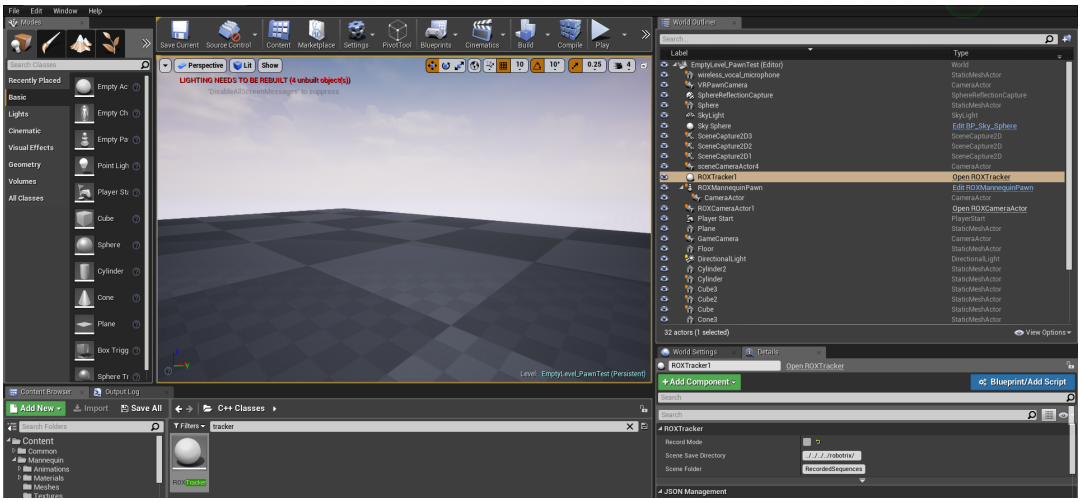


FIGURE 4.40: UE4 content browser.

As we explained previously, in order to record a scene, we need to add a Tracker to the scene we want to record. To do that, we simply have to click and drag the Tracker Actor from the content browser to the level viewport. This will add a tracker to the scene. We can verify that the Actor is placed in the scene by finding it in the World Outliner, which is located to the right of the level viewport, we can see the placed ROXTracker in the World Outliner in Figure 4.40.

Once this Actor is added to the scene, we can click on it on the World Outliner and see it's exposed properties. Figure 4.41 shows the first section of the exposed properties of the Tracker.

[13] https://docs.unrealengine.com/en-us/Engine/Content/Browser

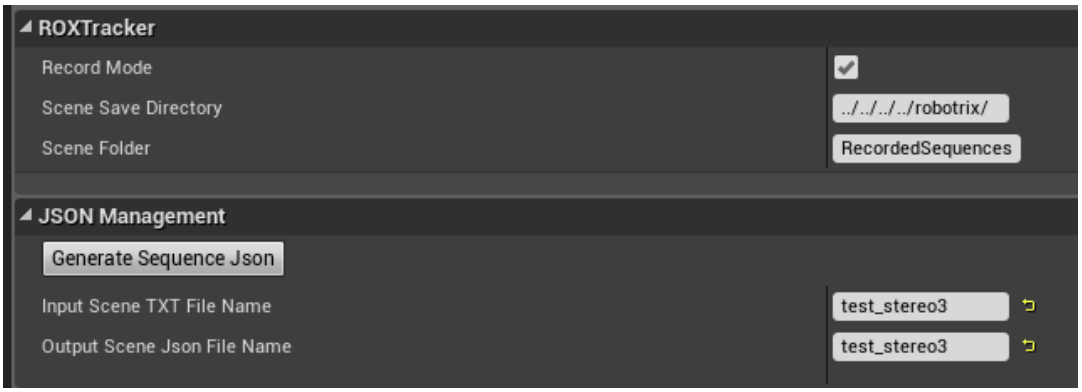


FIGURE 4.41: Tracker Actor generic settings.

These are the settings we can see in the previous Figure and their description:

- **Record Mode:** If this value is true, we will be on recording mode, meaning that if we trigger the recording action, the Tracker will start populating a TXT, that will be translated on a further step into a JavaScript Object Notation (JSON) file. If it's set to false we will be interpreting the data of that JSON and reproducing it.
- **Scene Save Directory:** Absolute path in which we will save the JSON data.
- **Scene Folder:** Folder inside the *Scene Save Directory* where we will save the JSONs for every sequence.
- **Generate Sequence JSON:** It transforms the txt data into a JSON. It can be pressed in the editor.

The next properties we can find on the Tracker are under the *Recording* category, which will remain active only if the *Record Mode* boolean is set to true. We can see these settings in Figure 4.42.

FIGURE 4.42: Tracker Actor recording settings.

This section has 3 arrays. The first one called *Pawns*, contains the Pawn references that we will be recording, which we will see in detail in Section 4.7.3. The second array stores all the camera references that will be tracked. An finally, the third array defines the stereo distance (if any) for these cameras following the same order as the previous array. *Scene File Name Prefix* is a simple naming convention setting.

The final exposed properties belong to the Playback category on the Tracker as we can see in Figure 4.43.

FIGURE 4.43: Tracker Actor playback settings.

- **Json File Names:** This array stores the input JSONs we generated in the recording phase that the playback system will process.
- **Start Frames:** This array correlates with the Json File Names array, meaning that it allows us to set a certain start frame per JSON. The first array entry will set the first frame to start for the first JSON on the *Json File Names* array.
- **Playback Only:** It *plays* the JSON without generating any data.
- **Playback Speed Rate:** It's only available when we set the Playback Only option to true and it allows us to set the speed at which the playback will play.
- **Generate RGB:** It generates a RGB image per camera per frame, we can adjust the format to JPG-80, JPG-95 (the number is the percentage of quality set to the JPG compression algorithm) or PNG if this boolean is set to true.
- **Generate Depth:** It generates a Depth image per camera per frame.
- **Generate Object Mask:** It generates an Object Mask image per camera per frame. This image gets generated by setting the viewmode to unlit and changing every single instance material to a unique color per object instance.
- **Generate Normal:** It generates a Normals image per camera per frame. We can obtain the normal information of the objects inside UE4.
- **Generate Depth Txt Cm:** Generates a text file where each number correlates to the depth value in centimeters of the k pixel of the depth image.
- **Screenshot Folder/Save directory:** It determines where the screenshots will be saved.
- **Generated Images width/Height:** Resolution of the image.

4.7.2 Initialization

Now that we covered all the properties we can start with the logic. In this first section we will study how the Tracker initializes and prepares other classes for the recording and the playback.

If we look at the Tracker *Begin Play*, we can see that we store the original *EngineShowFlags* variable in the *GameShowFlags* variable defined on the Tracker class. ShowFlags are a set of bits that are stored in the ViewFamily that can alter the way a scene renders in UE4. We store the original reference of the engine *ShowFlags* to reset its values after we made a change to the *EngineShowFlags* variable in other part of the code as we can see in Listing 4.5.

LISTING 4.5: GameShowFlags variable usage example.

```
GameShowFlags = new FShowFlags(Viewport()->EngineShowFlags);
// Altering the ShowFlags variable
AlterEngineShowFlags(Viewport()->EngineShowFlags);
// Setting back default value
Viewport()->EngineShowFlags = *GameShowFlags;
```

Following next, *GScreenshotResolutionX* and *GScreenshotResolutionY* are set, these variables are part of the engine and set the resolution of the images that *TakeHighResScreenShot*[14] outputs. In UnrealROX there is a functionality that allows us to take screenshots in the runtime by using this function, so the resolution set by the user in the Tracker Actor will be the one this function will be processing.

Next, Pawns get initialized by looping through the Pawn array that we saw in Figure 4.42. That loop calls the function *InitFromTracker* (explained in Section 4.6.2) in every pawn of this array. The initialization also caches the ControllerPawn variable, which represents the pawn user-controlled pawn.

The camera actors set in the editor get added to an Actor view target array, which we use to swap between different view targets, as we explained in Section 4.6.2. To recapitulate, these view targets are the points of view we set in our scene, to do that we place camera Actors in the editor in the locations from where we desire to have a point of view.

```
EROXViewModeList.Empty();
if (generate_rgb) EROXViewModeList.Add(EROXViewMode::RVM_Lit);
if (generate_depth || generate_depth_txt_cm) EROXViewModeList.Add(EROXViewMode::RVM_Depth);
if (generate_object_mask) EROXViewModeList.Add(EROXViewMode::RVM_ObjectMask);
if (generate_normal) EROXViewModeList.Add(EROXViewMode::RVM_Normal);

if (EROXViewModeList.Num() == 0) EROXViewModeList.Add(EROXViewMode::RVM_Lit);
```

FIGURE 4.44: View mode settings.

The last step we do in the generic initialization of the Tracker is filling the *EROXViewModeList* array. As we can observe in Figure 4.44, this array is populated based on the settings the user introduces in the editor, as we saw previously in the introduction. This array will be used in the main Tracker algorithm to determine which view modes should be computed.

Finally if we are in playback mode, we call *RebuildModeBegin()*, which will be in charge of *playing* the JSON file.

[14] https://api.unrealengine.com/INT/API/Runtime/Engine/FViewport/TakeHighResScreenShot/index.html

PrepareMaterials is a function that will prepare the object mask materials. This will be done by spawning flat materials and caching the original materials of every single mesh in the scene, it will also assign a color per mesh component. Masked materials are flat colored materials that rendered in Unlit mode, let us segment *Actors* by colors.

4.7.3 Recording the scene

When we set the Tracker in recording mode, the system is prepared to record the scene we are on. For that, we have the *StartRecording* handler on the Controller, as we explained in Section 4.5.2. When that handler is used, the function *ToggleRecording* on the Tracker is called. This function works like a flip flop, meaning that if the recording mode is activated and the function gets called, it will be deactivated and vice-versa. *ToggleRecording* resets the frame counter to zero, so every time we want to record, it will be properly initialized, it also writes the header in the text file we will be writing on this recording phase.

The header is the initial information the playback system (explained in Section 4.7.4) needs to be aware of. The *WriteHeader* function writes this initial part of the TXT. In order to do it, first, prints the number of the cameras the Tracker has on the *CameraActors* array. Then, iterates through this array printing the name of each camera accompanied by certain settings, such as the stereo distance and the field of view. Next, prints the number of static mesh actors tagged as movable contained in the scene and iterates through them printing their names and bounding box information. The following step is printing the number of Pawns that we will be tracking accompanied by their names and socket number (number of bones in the skeleton used). Lastly, the non movable objects information gets printed following the same format as the movable objects. Figure 4.45 shows a sample header.

```
Cameras 3
VRPawnCamera 0.0 90.0
CameraActor3_5 0.0 90.0
CameraActor_2 0.0 90.0
Objects 5
Cube3_11 OBB:X=30.000 Y=30.000 Z=-30.000 X=30.000 Y=30.000 Z=30.000 X=-30.000 Y=30.000 Z=30.000 X=-30.000 Y=30.000 Z=-30.000
X=30.000 Y=-30.000 Z=-30.000 X=30.000 Y=-30.000 Z=30.000 X=-30.000 Y=-30.000 Z=30.000 X=-30.000 Y=-30.000 Z=-30.000
Sphere_14 OBB:X=40.000 Y=40.000 Z=-40.000 X=40.000 Y=40.000 Z=40.000 X=-40.000 Y=40.000 Z=40.000 X=-40.000 Y=40.000 Z=-40.000
X=40.000 Y=-40.000 Z=-40.000 X=40.000 Y=-40.000 Z=40.000 X=-40.000 Y=-40.000 Z=40.000 X=-40.000 Y=-40.000 Z=-40.000
Cone2_17 OBB:X=100.000 Y=100.000 Z=-100.000 X=100.000 Y=100.000 Z=100.000 X=-100.000 Y=100.000 Z=100.000 X=-100.000 Y=100.000
Z=-100.000 X=100.000 Y=-100.000 Z=-100.000 X=100.000 Y=-100.000 Z=100.000 X=-100.000 Y=-100.000 Z=100.000 X=-100.000 Y=-100.000
Z=-100.000
Cylinder_20 OBB:X=40.000 Y=40.000 Z=-100.000 X=40.000 Y=40.000 Z=100.000 X=-40.000 Y=40.000 Z=100.000 X=-40.000 Y=40.000 Z=-100.000
X=40.000 Y=-40.000 Z=-100.000 X=40.000 Y=-40.000 Z=100.000 X=-40.000 Y=-40.000 Z=100.000 X=-40.000 Y=-40.000 Z=-100.000
wireless_vocal_microphone_2 OBB:X=3.052 Y=3.052 Z=-0.172 X=3.052 Y=3.052 Z=21.653 X=-3.052 Y=3.052 Z=21.653 X=-3.052 Y=3.052
Z=-0.172 X=3.052 Y=-3.052 Z=-0.172 X=3.052 Y=-3.052 Z=21.653 X=-3.052 Y=-3.052 Z=21.653 X=-3.052 Y=-3.052 Z=-0.172
Skeletons 1
ROXMannequinPawn 69
NonMovableObjects 2
Floor X=0.000 Y=0.000 Z=20.000 P=0.000000 Y=0.000000 R=0.000000 MIN:X=-5000.003 Y=-5000.000 Z=-30.000 MAX:X=5000.001 Y=5000.000
Z=20.000 OBB:X=5000.001 Y=5000.000 Z=-50.000 X=5000.001 Y=5000.000 Z=0.000 X=-5000.003 Y=5000.000 Z=0.000 X=-5000.003 Y=5000.000
Z=-50.000 X=5000.001 Y=-5000.000 Z=-50.000 X=5000.001 Y=-5000.000 Z=0.000 X=-5000.003 Y=-5000.000 Z=0.000 X=-5000.003 Y=-5000.000
Z=-50.000
Plane_2 X=-1000.000 Y=300.000 Z=160.000 P=-0.000014 Y=180.000000 R=89.999977 MIN:X=-1400.000 Y=300.000 Z=-90.000 MAX:X=-600.000
Y=300.000 Z=410.000 OBB:X=400.000 Y=250.000 Z=-0.000 X=400.000 Y=250.000 Z=0.000 X=-400.000 Y=250.000 Z=0.000 X=-400.000 Y=250.000
Z=-0.000 X=400.000 Y=-250.000 Z=-0.000 X=400.000 Y=-250.000 Z=0.000 X=-400.000 Y=-250.000 Z=0.000 X=-400.000 Y=-250.000 Z=-0.000
```

FIGURE 4.45: Sample header from the recording TXT file.

Once the header is written and the recording mode is set to true, the Tracker starts calling every frame *WriteScene*, which is the function in charge of writing every tick the information of the scene. To do that, it prints the word frame along the number of the frame processed accompanied by the the time, the frame counter gets increased every time this function gets called. This variable is convenient since it allows us to know the frame related to a specific setup, like an ordered array structure.

The execution flow of this function goes as it follows: First, we iterate through the camera Actor array, obtaining their names followed by their transform information to then print them in order. Next we iterate through the pawns getting their name and transform information, in addition, we iterate through the sockets of the skeleton of each pawn to obtain the desired socket information. This socket information is what will make the skeleton of our pawns move, since thanks to that we can know the location and the rotation of every socket in the skeleton. The last thing we do is printing the information of every mesh in the scene following the format explained previously. As we said before, this information is retrieved in every tick, and we can see the format in Figure 4.46.

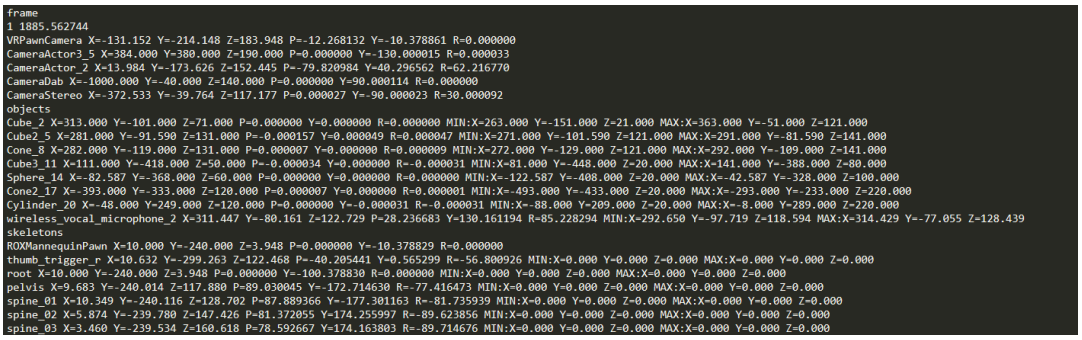


FIGURE 4.46: TXT recording file.

In order to write the information to a text file and not lock the main game thread, we have created an Async task[15] that we use to write strings on a file as we can see on Figure 4.47. This task uses a built in function of UE4 for writing a string in a file.

```
class FWriteStringTask : public FNonAbandonableTask
{
    friend class FAutoDeleteAsyncTask<FWriteStringTask>;
public:
    FWriteStringTask(FString str, FString absoluteFilePath) :
        m_str(str),
        m_absolute_file_path(absoluteFilePath)
    {}
protected:
    FString m_str;
    FString m_absolute_file_path;
    void DoWork()
    {
        FFileHelper::SaveStringToFile(m_str, *m_absolute_file_path, FFileHelper::EEncodingOptions::ForceUTF8WithoutBOM, &IFileManager::Get(), EFileWrite::FILEWRITE_Append);
    }
    FORCEINLINE TStatId GetStatId() const
    {
        RETURN_QUICK_DECLARE_CYCLE_STAT(FWriteStringTask, STATGROUP_ThreadPoolAsyncTasks);
    }
};
```

FIGURE 4.47: Async task for writing a string on a file.

This task has as arguments the absolute path of the file, including the name, besides the string that we want to add to the file.

We mentioned earlier that the playback mode uses a JSON file, and so far we have commented that the data is written in a TXT file. UnrealROX has a feature, already commented in Section 4.7.1, that allows us to transform this generated TXT file into a JSON file. We do this separately because the data in a JSON is more bulky, that means that you have to write more in every frame, hence the performance of the project could be compromised. That's why we generate first a TXT, which is lighter to create. File which we will transform to JSON from the editor, out of runtime.

Once the JSON file is generated, we are ready to use the playback mode.

[15] https://wiki.unrealengine.com/Using_AsyncTasks

4.7.4 Reproducing the scene

Playing a scene from a JSON file in UnrealROX requires a previous setup, which consists of several phases. As we have already explained, the objective of this stage is to generate a dataset, which is composed of images taken in UE4 from different points of view in a specific scene.

The first step to generate this dataset is to place as many *Camera Actors* as we need. These actors, as described in Section 4.2, will be the points of view that will be taken into account when generating the data, in addition to the default camera of the pawns. These cameras need to be set up on the Tracker in the recording phase, so we can track their position.

Once the JSON is ready, we can set the Tracker Actor on playback mode, to do this, we set the *bRecordMode* variable of the Tracker to false. If it is set to false, *BeginPlay* will call *RebuildModeBegin*, which is in charge of the playback loop.

The design of the playback mode of the Tracker allows us to define several JSON files to reproduce in an array. This option enables the generation of data from different recordings in a single playback session. By this statement we are defining our first outer loop.

First, we load the JSON file we want to play onto our *JSONParser* class. This class will be in charge of parsing the JSON data into a more UE4 friendly format. It's also essential to disable the gravity in all the meshes of the scene, since we don't want any physic simulation to happen while we are setting the transform of the objects recorded previously. Once everything is initialized we can start with the inner loop which takes place on the *RebuildModeMain* function. The following Algorithm 7 represents a simplistic version of the playback loop.

```
for current_json_file in json_file_names do
    for frame in current_json_numframes do
        for actor in Scene do
            FROXJsonParser* ActData := frame.Find(actor->GetName());
            if actor not null then
                actor->SetActorTransform(ActData->Transform);
            end
        end
        if !playback_only then
            RTs := CreateRenderTargets(Cameras->Transform);
            for viewmode in viewmodes do
                for rt in rts do
                    rt->TakeScreenshot(viewmode);
                end
            end
        end
    end
end
```

Algorithm 7: Very simplistic version of the playback loop execution flow.

Following next, we iterate through all the frames contained in a single JSON file doing the following: All the objects in the scene retrieve and set their transforms for the current frame from the JSON data we explained above. Once all the objects are placed for a specific frame, we can start taking screenshots for that frame (only if the user desires so).

To take the screenshots, we use the *Render Target*[16] class, since it allows us to define specific render modes to retrieve precise data from the render target instance. To do this, we will spawn one render target per viewmode in the spot where the camera is located; which means that the cameras are only used as a helper class to spawn the necessary render targets in the appropriate locations.

This means that we will have four render targets per camera: Lit, Depth, Object Mask and Normal. To make this approach more seamless and performant, we have four arrays containing the cached render targets (or scene capturers), one for each mode, SceneCapture_Depth, SceneCapture_Lit and so on. These arrays are encoded in a way that the *n* index of the lit array corresponds to the *n* index of the depth array and the remaining arrays. This means that *SceneCapture_x[n]*, being x the viewmode, represents one single viewpoint.

If we describe the problem at a lower level, we can elaborate that each viewmode consists on a series of techniques applied over each render target. We will see a more complex explanation of these techniques below.

- **Lit:** Normal RGB lit picture. Gets generated by setting the *Scene capture source*[17] in SCS_FinalColorLDR mode, the *Render Target Format*[18] in RTF_RGBA8 mode, and the Gamma correction to 2.2, so the final result is accurtate with what we see in the screen (without this gamma corrections the pictures would be darker).

- **Depth:** This mode represents what a depth camera would do if it would take a picture of the scene. Each pixel represents the distance between the viewer and the surface where this pixel resides, the darker the pixel is, the nearer this surface is from the viewer. It gets generated by setting the *Scene capture source* in SCS_SceneDepth mode, the *Render Target Format* in RTF_RGBA16f mode, and the Gamma correction to 0. There is also an option to write the pixel data on a txt, we do that by iterating through the pixel data and printing the content to a TXT file.

- **Object Mask:** This viewmode consists on painting in different colors every object instance in the scene. We do this to differentiate every instance in the screen by its pixels color in a very precise way. In order to do this, we use the BaseColor RenderTarget in combination with plain-colored materials. This material has a color parameter which permits us to set a unique color to every object instance. By doing that we can have a total control of the color that we set to every actor. It gets generated by setting the *Scene capture source* in SCS_BaseColor mode, the *Render Target Format* in RTF_RGBA8 mode, and the Gamma needs to be set to 1, which is the only notable difference we can find with the Lit mode setup.

[16] https://docs.unrealengine.com/en-us/Engine/Rendering/RenderTargets

[17] https://api.unrealengine.com/INT/API/Runtime/Engine/Engine/ESceneCaptureSource/index.html

[18] https://api.unrealengine.com/INT/API/Runtime/Engine/Engine/UTextureRenderTarget2D/RenderTargetFormat/index.html

- **Normal:** The normal viewmode simply displays the normals[19] of the world. This is done by applying a post process material to the render target. The material can be seen in Figure 4.48. It gets generated by setting the *Scene capture source* in SCS_FinalColorLDR mode, the *Render Target Format* in RTF_RGBA8 mode, and the Gamma correction remains unmodified.

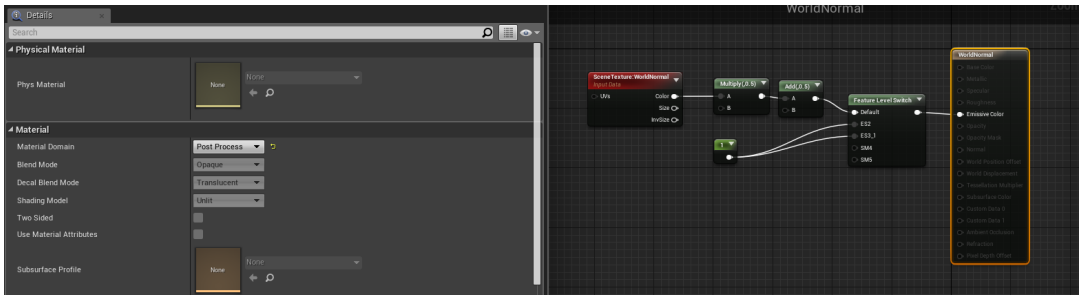
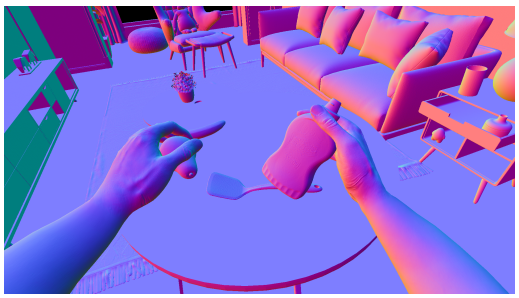


FIGURE 4.48: Normal viewmode material.

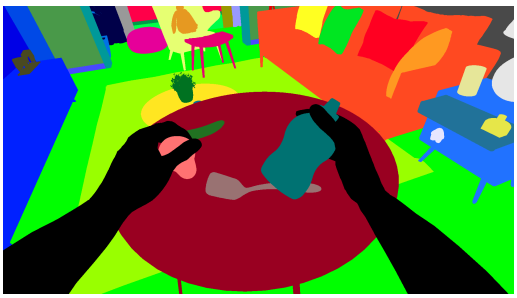
The following Figure 4.49 shows some samples extracted directly from the dataset in which the viewmodes are demonstrated. One of the most important points when working on this project is to get correct data, that's why we emphasize on the formats for each viewmode, as they are essential to produce a quality dataset. Continuous studies have been made of the generated data with software such as IrfanView, explained in Section 3.2, to corroborate the produced results.

(A) RGB viewmode

(B) Depth viewmode



(C) Normal viewmode



(D) Object Mask viewmode

FIGURE 4.49: UnrealROX viewmodes raw results.

[19] http://wiki.polycount.com/wiki/Normal_map

Chapter 5

Conclusion

This chapter makes an overview of the conclusions of this work. This chapter is divided into three sections: Section 5.1, summarizes the work done in this Thesis. Section 5.2, highlights the main points of this project. And finally, Section 5.3 delineates future research lines that could derive from this project.

5.1 Conclusions

In this work, we have reviewed a large part of the methods that have been beneficial to reduce the gap between the synthetic and the real-world domains. In that regard, we proposed UnrealROX, a simulator that improves the drawbacks of the different simulators proposed until now.

First, we analyzed the two main avenues to minimize this gap: photorealism and domain randomization. Then, we discussed the main advantages and disadvantages about some of the most relevant simulators and generators belonging to the Sim-To-Real area[1].

As for photorealism, we have concluded that it is one of the most important points for the transfer of synthetic learned models to the real world. For that, we have determined that using a game engine that conforms to the state of the art in terms of rendering techniques leads to great results, as we discussed in Section 2.1.3. We have also commented on the current limitations of real time ray-tracing to improve the overall quality of the final image; at this date, there is not enough computing power on average devices to carry out a perfect ray-tracing setup.

On the other hand, we found that models used in simulators do not reflect the semantic complexity of real-world, and this makes them prone to poor generalizations when it comes to new data. This issue can be improved with domain randomization, which proposes to generate random variations of the synthetic world so that after the model has seen enough variations it can identify the real world as just another variation. However, this type of randomization can lead to meaningless data, as we discussed in Section 2.1.2. Various models, such as ADR and SDR propose an improved paradigm where this randomization gets controlled following different criteria.

As we have commented before, we have reviewed some of the most relevant simulators and generators to the present day. This task has been critical when defining UnrealROX since one of the main objectives of our simulator is to compensate for the shortcomings of other environments.

[1] https://sim2realai.github.io/

UnrealROX is a simulator that implements grasping interaction using an interchangeable skeletal mesh mannequin that reproduces movements recorded previously with a VR device. Our simulator is capable of translating seamlessly the operator's movements into the robot's degrees of freedom. In UnrealROX the user can select the type of data to generate, avoiding unnecessary long calculation times. This data will be generated from the multiple points of view placed and configured by the operator, thus making the system more flexible, as well as improving the dataset. All of this accompanied by the base features of UE4, which allow the user to import new scenarios and models. In addition, any developer can download and extend UnrealROX, thanks to the fact that the simulator is open sourced and available at Github[2]. UnrealROX also counts with a polished documentation open to the public. All of this opens up new research avenues for future work, which we will cover in Section 5.3.

5.2 Highlights

The highlights of this work are the following:

- In depth study of the Sim-To-Real field analyzing it's two main avenues: photorealism and domain randomization.
- Review and analysis of the most relevant simulators and synthetic data generators to date.
- Proposal and implementation of UnrealROX, a simulator that covers the limitations of other environments.
- In depth documentation for UnrealROX and its components[3].
- Various video presentations for RobotriX[4], UnrealROX[5] and UnrealGrasp[6].
- A dataset featured at **IROS 2018**[7]: The RobotriX [3], a photorealistic indoor dataset for deep learning.
- UnrealGrasp [13], a realistic grasping system designed for UnrealROX.
- An incoming UnrealHands, a hand pose detection algorithm using synthetic data.
- Generation of synthetic data for segmentation of actions in video sequences. UnrealROX has been used as a tool to generate the dataset from previously labelled sequences of actions.
- Automation of UnrealROX agents and generation of synthetic actions for semantic segmentation.

[2] https://github.com/3dperceptionlab/unrealrox
[3] https://unrealrox.readthedocs.io
[4] https://www.youtube.com/watch?v=CiRc5tCtCak
[5] https://www.youtube.com/watch?v=YOiVr2A2TZo
[6] https://www.youtube.com/watch?v=4sPhLbHpywM
[7] https://www.iros2018.org/

5.3 Future Work

We have been mentioning throughout this work some of the possible future research lines that could be opened to complement this work. Some of them were initially proposed as part of the planning of this project, however, due to time constraints it has been impossible to address some of them. In this section we summarize them to conclude this Thesis:

- **Interaction mask:** One of the proposed objectives of this project was to create an interaction mask. This method would consist of creating a binary categorization for the objects with which the user interacts in a scene, drawing in white those pixels which correspond to the objects the operator is interacting with, and in black those which don't. Once the interaction mask is created using synthetic data, we can generate an interaction dataset. Finally we can transfer the learned model to the real world to see how it performs.

- **Domain Randomization:** In Section 2.1.5, we discussed the future possibility of including domain randomization in UnrealROX. This would consist of having a total control over the assets in the scene, trying get feasible scenarios as SDR proposes. A possible approach might be to create different arrays of meshes and assign predefined random points in the scene where they would appear. We could also control the materials these assets spawn with and define random lighting options for the scene.

- **Ray-tracing:** We have analyzed in depth the current state of the art of ray-tracing at the beginning of this work, and we concluded that to use ray-tracing in a feasible manner there needs to be a selective way to decide where to trace more or less rays. As of today, UE4 implements a ray-tracer in its 4.22 version; however, the technology is still too primitive to be used directly in real time without a major performance hit and framerate drops in VR.

- **Non-rigid objects manipulation:** This is the most complex research option out of the proposed ones, since the support for non-rigid objects in UE4 is very limited. One of the few presences of non rigid objects in the engine is the clothing tool[8], but the integration is more focused on visuals rather than interaction. Integrating non-rigid interactive objects in game engines is one of the future state of the art challenges when it comes to physics in real time. Deforming a shader is not enough to bring collision data to the CPU, which is the main research topic of this matter.

[8] https://docs.unrealengine.com/en-us/Engine/Physics/Cloth/Overview

Bibliography

[1] Angel Chang et al. "Matterport3D: Learning from RGB-D Data in Indoor Environments". In: *International Conference on 3D Vision (3DV)* (2017).

[2] Adrien Gaidon et al. "VirtualWorlds as Proxy for Multi-object Tracking Analysis". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016). DOI: 10.1109/cvpr.2016.470. URL: http://dx.doi.org/10.1109/CVPR.2016.470.

[3] Alberto Garcia-Garcia et al. "The RobotriX: An Extremely Photorealistic and Very-Large-Scale Indoor Dataset of Sequences with Robot Trajectories and Interactions". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2018). DOI: 10.1109/iros.2018.8594495. URL: http://dx.doi.org/10.1109/IROS.2018.8594495.

[4] A. Geiger, P. Lenz, and R. Urtasun. "Are we ready for autonomous driving? The KITTI vision benchmark suite". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2012. DOI: 10.1109/cvpr.2012.6248074. URL: https://doi.org/10.1109%2Fcvpr.2012.6248074.

[5] Ian Goodfellow et al. "Generative Adversarial Nets". In: (June 2014). URL: https://arxiv.org/pdf/1406.2661.pdf.

[6] Matthew Johnson-Roberson et al. "Driving in the Matrix: Can virtual worlds replace human-generated annotations for real world tasks?" In: *2017 IEEE International Conference on Robotics and Automation (ICRA)* (2017). DOI: 10.1109/icra.2017.7989092. URL: http://dx.doi.org/10.1109/ICRA.2017.7989092.

[7] Eric Kolve et al. *AI2-THOR: An Interactive 3D Environment for Visual AI*. 2017. arXiv: 1712.05474 [cs.CV].

[8] Christian Ledig et al. "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network". In: (May 2017). URL: https://arxiv.org/pdf/1609.04802.pdf.

[9] Yang Liu et al. *Stein Variational Policy Gradient*. 2017. arXiv: 1704.02399 [cs.LG].

[10] Bhairav Mehta et al. *Active Domain Randomization*. 2019. arXiv: 1904.04762 [cs.LG].

[11] Paul W. Munro and Antony P. Gerdelan. "Stereo Vision Computer Depth Perception". In: 2006.

[12] NVIDIA. "NVIDIA TURING GPU ARCHITECTURE". In: (Sept. 2018). URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[13] Sergiu Oprea et al. *A Visually Plausible Grasping System for Object Manipulation and Interaction in Virtual Reality Environments*. 2019. arXiv: 1903.05238 [cs.GR].

[14] Aayush Prakash et al. *Structured Domain Randomization: Bridging the Reality Gap by Context-Aware Synthetic Data*. 2018. arXiv: 1810.10093 [cs.CV].

[15] Weichao Qiu and Alan Yuille. *UnrealCV: Connecting Computer Vision to Unreal Engine*. 2016. arXiv: 1609.01326 [cs.CV].

[16] Weichao Qiu et al. "UnrealCV: Virtual Worlds for Computer Vision". In: Oct. 2017, pp. 1221–1224. DOI: 10.1145/3123266.3129396.

[17] Tim Salimans et al. "Improved Techniques for Training GANs". In: (June 2016). URL: https://arxiv.org/pdf/1606.03498.pdf.

[18] Manolis Savva et al. *MINOS: Multimodal Indoor Simulator for Navigation in Complex Environments*. 2017. arXiv: 1712.03931 [cs.LG].

[19] Shuran Song et al. "Semantic Scene Completion from a Single Depth Image". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017). DOI: 10.1109/cvpr.2017.28. URL: http://dx.doi.org/10.1109/CVPR.2017.28.

[20] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[21] Thang To et al. *NDDS: NVIDIA Deep Learning Dataset Synthesizer*. https://github.com/NVIDIA/Dataset_Synthesizer. 2018.

[22] Josh Tobin et al. "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)* (2017). DOI: 10.1109/iros.2017.8202133. URL: http://dx.doi.org/10.1109/IROS.2017.8202133.

[23] Jonathan Tremblay, Thang To, and Stan Birchfield. "Falling Things: A Synthetic Dataset for 3D Object Detection and Pose Estimation". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (2018). DOI: 10.1109/cvprw.2018.00275. URL: http://dx.doi.org/10.1109/CVPRW.2018.00275.

[24] Jonathan Tremblay et al. *Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects*. 2018. arXiv: 1809.10790 [cs.RO].

[25] Michael F. Worboys. "GIS: A Computer Science Perspective". In: (Oct. 1995), p. 232. URL: https://books.google.es/books?id=duT2fcnQeJMC&pg=PA232&redir_esc=y#v=onepage&q&f=false.

[26] Yi Wu et al. *Building Generalizable Agents with a Realistic and Rich 3D Environment*. 2018. arXiv: 1801.02209 [cs.LG].

[27] Fei Xia et al. "Gibson env: real-world perception for embodied agents". In: *Computer Vision and Pattern Recognition (CVPR), 2018 IEEE Conference on*. IEEE. 2018.