mng rämibühl
Mathematisch-Naturwissenschaftliches Gymnasium

Crowd Simulations in Blender

Exploring Different Methods to Realistically Simulate Crowds Digitally

Author: Leo Martin
Supervisor: Pascal Forny
MNG Rämibühl
6 January 2025

Abstract

This paper explores various methods for simulating crowds in digital environments, focusing on their practical implementation within the 3D software Blender. The primary goal of this paper was to develop a prototype of a crowd simulation and introduce an improved method for animation synthesis, aimed at enhancing the realism of character animations. The core of this paper lies in the practical application of crowd simulation techniques in Blender in the form of a Python script. Using a hybrid approach for collision avoidance, the script makes use of the A pathfinding algorithm and a self-developed method of local collision avoidance. The final software prototype successfully handled various forms of crowds, ranging from small groups to larger crowds, with the ability to move through a space with an acceptable degree of error. The findings and developments presented in this paper pave the way for further improvements in Blender-based crowd simulation.*

Contents

1 Introduction

When Peter Jackson sought to bring J.R.R. Tolkien's epic battles from The Lord of the Rings to the screen, he faced a significant challenge: how to realistically depict large-scale crowds in the digital realm. With the computer hardware of the early 2000s, this required innovative solutions, leading to the development of a custom crowd simulation software called MASSIVE [1]. This use case is probably one of the most famous applications of crowd simulation. While the application of crowd simulators in film is one of their most famous use cases, their utility extends far beyond entertainment. From evacuation planning in urban centers to modeling pedestrian flow in public spaces, crowd simulations provide valuable insights across a wide range of fields. Despite the prominence achieved with The Lord of the Rings movies, crowd simulations for film remain a more niche application with relatively few software solutions available. Beyond MASSIVE, there are plugins for industry-standard 3D animation software such as Miarmy [2] and Golaem [3], both designed for use with Autodesk Maya[1]. Additionally, Houdini, a 3D simulation software, includes a built-in crowd dynamics system suitable for creating large crowds with soft body and ragdoll simulations.

In this context, I set out to create a crowd simulator for Blender, a free and open-source 3D animation software. Blender does not natively include a crowd simulation system like the proprietary software. This project aims to provide insight into the algorithms and methods used for crowd simulation and introduces a novel animation synthesis approach.

this paper first covers a overview of how crowd simulators work and will then later delve into the practical part where I coded my own crowd simulator as a blender addon.

Figure 1: *A scene from the Battle of Helm's Deep in The Lord of the Rings: The Two Towers. Source: New Line Productions, Inc.*

[1] Maya is the leading software for 3D animation in the industry

2 General Overview of Crowd Simulators

In this chapter, I will provide a broad overview of how crowd simulators function and discuss the respective limitations of the methods employed.

2.1 How do Crowd Simulators Work

A crowd simulator models the movement and dynamics of a large number of individuals within a defined space. These simulators are used in a wide range of applications, with the focus of each simulation tailored to specific needs and use cases. For example, while some crowd simulators aim to analyze crowd behavior to improve evacuation plans or enhance safety in dense gatherings, others focus on replicating human behavior for films or video games. Each crowd simulation operates on a defined time scale, which determines the speed at which the simulation runs. The time scale also dictates how frequently the crowd's motion and locations are updated. For instance, a larger time scale might be used to analyze phenomena such as the spread of a virus over time [4], where running the simulation at a visually smooth pace would be unnecessary. During each iteration of the simulation loop, the motion and location of individuals in the crowd are updated. Each individual is represented as a point in space with a radius that serves as a proxy for the agent's behavior. The method used to compute the motion and location of these individuals defines the crowd dynamics.

2.1.1 Methods for Crowd Dynamics

Crowd dynamics form the foundation of most crowd simulators, governing how individuals move and interact with their simulated environment. Effective crowd dynamics aim to replicate human movement to create realistic simulations. The core aspects of crowd dynamics typically include:

- **Collision Avoidance:** Ensuring individuals avoid bumping into other agents or obstacles.
- **Pathfinding:** Generating human-like paths to reach desired targets.
- **Agent Behavior:** Display agent specific behaviour such as attraction towards targets and repulsion from larger crowds.

To simulate these behaviors, different crowd modeling approaches can be implemented. Depending on the application and time scale, a specific modeling approach may be chosen. Broadly, crowd modeling approaches can be categorized into entity-based, agent-based, and flow-based methods [5]. These methods dictate how individuals in the simulation are represented and interact with their environment. Within each approach, various techniques can be employed to achieve the desired outcomes.

Entity-based Approach: In this approach, individuals in the simulation are defined as a set of uniform entities. Each entity adheres to the same rules and has the same attributes as all the others. The simplest way to simulate a large number of entities is through particle simulations. Particle simulations are primarily used for scenarios involving a large number of particles, such as in gases, liquids, or more general Brownian motion. Crowds can also be simulated using particle simulations, with each individual represented by a particle. Due to the computational efficiency of particle simulations [6], thousands of particles can be effortlessly simulated, interacting with each other and the environment. All particles share the same behaviors that define their movement and interactions in space. These behaviors

can be physics-based, such as gravity, collisions, friction, etc. Additionally, particle speeds can be randomized to create a more dynamic simulation. Particle simulations were widely used in the past when computer hardware was less powerful. However, one drawback of this method is that it can be difficult to control the simulation, which is undesirable for films and games. Moreover, complex avoidance algorithms are challenging to implement. While it is computationally simple, achieving more human-like behavior is not feasible with a homogeneous group of entities.

Agent-based Approach: Unlike the previous method, the agent-based approach defines its individuals not as uniform entities but as autonomous agents. Each agent can think and act independently without adhering to a globally dictated behavior. While agent-based systems can generate more complex motion than entity-based systems, they come with a higher computational cost. An improvement over the particle system is the boids simulation, first introduced by Craig Reynolds in 1987 [7]. Reynolds used the simulation of bird flocks to study the intricate dynamics and movements that emerge in such groupings. He coined the term "boid," short for "bird-oid object," to describe the agents in the system. Each boid operates autonomously by following a set of simple rules [8]:

- **Separation** Keep a safe distance from nearby agents to avoid crowding the area.
- **Alignment** Move in the same direction as nearby agents.
- **Cohesion** Move towards the average position of nearby agents to stay together.

Despite their simplicity, these rules result in the emergence of complex and coordinated motion.

What distinguishes boids from particle simulations is that boids move in a direction determined by their orientation, which represents where the boid is "looking". To create flock-like movement, boids must closely interact with one another. While particles move aimlessly and react only to collisions, boids make decisions based on their predefined rules. This highlights how simple rules can lead to emergent complexity. However, while boids exhibit more realistic behavior than particles, they still fall short of replicating the intricate dynamics of real-world bird flocks. To achieve more complex and human-like motion, additional layers of complexity must be introduced.

Many agent-based algorithms aim to find the optimal path for an agent to travel from point A to point B by minimizing a problem function. This function can take various forms, depending on the criteria for determining the best outcome. One of the most well-known pathfinding algorithms is A*, which navigates from cell to cell in a grid while calculating the energy required for traveling through through the current one. Ultimately, the algorithm selects the path that minimizes the energy needed to traverse the grid [9]. Optimal paths can also be formed using principles from biomechanics, which state that humans naturally seek paths of least resistance [10]. Additionally, advanced methods use artificial intelligence to influence an agent's decision on where to go next, based on various attributes.

Flow-Based Approach: Flow-based algorithms analyze the flow of a crowd as a whole rather than the motion of individual agents. This approach is particularly useful for simulating very dense crowds [11, 12] because it reduces computational costs by avoiding the need to simulate individual behaviors. Additionally, it naturally captures the emergent behavior of dense crowds. Due to the compressive nature of human crowds, a strictly collision-avoidance model would fail to produce realistic movement and could lead to issues such as agents exhibiting jittering [2].

[2] Jittering is a phenomenon that often occurs in rigid body simulations, where simulated objects exhibit jittering motion [13].

2.1.2 Methods of Animation Synthesis

Animation synthesis refers to the process of creating new animations based on a set of pre-existing animations. To create visually realistic simulations, it is essential to assign animations to individual agents that accurately reflect their movements in space. The need for animation synthesis arises because the specific movements of agents are often too dynamic and varied to have pre-designed animations for every scenario. Additionally, it is impractical to have animators manually create animations for every individual in a crowd simulation.

Animation synthesis has a wide range of applications beyond crowd simulators. For example, it is extensively used in video games, where animations must be generated in real-time. In such cases, memory usage and processing power have to be kept at a minimum to ensure a satisfactory user experience. By storing a limited number of base animations and synthesizing new ones, an array of different animations can be created efficiently. There are various approaches to animation synthesis, each with its own advantages and limitations depending on the specific demands of the animation. These approaches include:

Data-driven Animation: Data-driven motion synthesis uses animation data to generate new animations. This process can involve animation blending, where two animations are smoothly blended to create continuous motion. Additionally, new motion can be created by interpolating multiple animations to generate new states [14].

Motion Matching: Motion matching is predominantly used in AAA video games. It involves a set of animations stored in memory and a motion-matching algorithm that determines which animation best fits the current state of the agent [15].

Physics Based Animation: These animations are generated using rules that obey the laws of physics. They can produce locomotion gaits that result in quite believable motion [16]. Some methods even simulate the human body, including its muscles [17]. However, physics-based animation is mostly used to create simulations where the body dynamically reacts to its environment, as seen in ragdoll physics for death animations. Houdini is famous for its crowd simulations, where agents can maintain aspects of their animation while being integrated into soft body or ragdoll simulations, such as in this video [18].

Machine Learning: Since human motion is so diverse, it is advantageous to implement neural networks for animation. This approach allows for a greater variety of motions that agents can perform [19]. However, this method requires a substantial dataset of training animation data, sometimes up to four million frames, as demonstrated in [20].

Humans have a keen eye for detecting unnatural behavior and motion. In computer animation, this phenomenon is known as the uncanny valley [3]. While low-quality animations of humans may not disturb us as much as a CGI-generated face, they still break immersion. For this reason, and because animating manually is highly time-consuming, we utilize motion capture data. Motion capture works by tracking specific parts of an actor's body and transferring the data into the computer [21]. This animation method provides highly realistic results. However, it is important to note that the raw animation data from motion capture often contains imperfections and jittering, necessitating manual cleanup

2.2 Blender-Based Crowd Simulators

Blender is a free and open source software with an integrated Python API [22] (aplication programing interface) which alows the user to acces data from the scene or so. But as men-

[3] The uncanny valley is a concept that refers to the feeling of discomfort humans experience when they encounter a humanoid figure that does not quite move like a real person.

tioned before the usecase for crowd simulations is very limited so there are not many crowd simulation adons on the market. From the available ones two types can be differentiated; the procedural ones and the 'traditional' ones. You can differentiate two types of blender crowd simulators:

Blender is a free and open-source software with an integrated Python API [22] (Application Programming Interface), which allows the user to access data from the scene and more. However, as mentioned earlier, the use case for crowd simulations is quite limited, so there are not many crowd simulation add-ons available for Blender. Of the available options, two types of crowd simulators can be differentiated: procedural simulators and script-based simulators.

Procedural crowd simulation: Procedural simulators take advantage of Blender's inbuilt geometry nodes. As of version 2.92, Blender includes geometry nodes [23], which introduced the ability to create larger, more complicated, and non-destructive workflows [4]. Such non-destructive workflows are vital when creating complex models or animations, as they allow for easier changes down the line. Simulation software like Houdini has non-destructive workflows ingrained in its structure, but Blender is still catching up. The highly anticipated addition of simulation nodes, which would enable procedural ragdoll physics similar to Houdini, has not yet been implemented. Procedural simulations function like more complex particle simulations, where a character with animation can be projected to walk over a surface. The problem is that the agent logic in Blender's current system is very limited and cannot display human-like crowd dynamics. It is a good option for more static crowds, such as fans in a football stadium [24].

Script based crowd simulation: These add-ons make use of Blender's built-in API to change and control various aspects of the simulation. They are add-ons written in Python that can be installed internally in Blender. Traditional simulators, which do not utilize Blender's Geometry Nodes, include add-ons like CrowdSim3D [25] (which was presented at the Blender Conference). These add-ons replicate the structure of simulators commonly used in the industry. However, development on these tools is slow, and the software is far from being as powerful as professional add-ons for Maya and Houdini.

Overall, Blender remains far behind industry giants such as Autodesk and Houdini. Although Blender has improved remarkably over the years, especially since I started using it eight years ago, but it still has a long way to go. The likelihood of a professional crowd simulation tool being developed for Blender in the next few years is slim, as VFX houses are unlikely to start using Blender in their crowd simulation workflow.

2.3 Limitations in Existing Simulators

Creating crowd simulations that look visually realistic for film or games presents a multitude of challenges, as highlighted in [26]. Unlike simulations made for scientific applications, those designed for visual consumption have an additional level of difficulty. The agents must move realistically within the environment and exhibit corresponding animations. Due to the complexity of the task, many systems struggle to achieve the desired level of realism In the following section, I will describe the shortcomings of the previously noted methods.

[4] A workflow in 3D refers to the process of working towards a desired outcome. The opposite of non-destructive workflows are destructive workflows, where changes to the mesh are permanent (e.g., directly modifying the geometry).

2.3.1 Challenges of Crowd Dynamic

The challenges of realistic crowd dynamics, particularly obstacle avoidance, arise from the fact that many different algorithms exist, each designed for specific applications. Many of these algorithms are based on pathfinding for robotics, which does not always produce realistic human motion. Even with a good algorithm, they are typically tailored for specific types of crowds. Sparse crowds are the easiest to simulate, while dense crowds require entirely different approaches to accurately represent their movements. This becomes more complicated when scenes involve multiple types of crowds with different behaviors. Additionally, edge cases, such as panic situations, can further complicate the simulation and lead to unrealistic renderings.

It ultimately comes down to there being two primary use cases for crowd simulations. The scientific community has developed many different algorithms that appear believable, even with relatively low computational costs. However, their main problem is that they are difficult to control. This is the key difference between commercial crowd simulations used for films and scientific crowd simulations. The former is made for artistic reasons, where the artist must have control over the outcome of the simulation, while the latter tries to leave control over the movement to the algorithms. There are also societal norms that are not represented in many algorithms, such as everyone walking on the right side of a corridor. This would require an additional set of rules that are not intuitive, and the user would first have to research all the possible behaviors. On top of that, more complex group behaviors are difficult to create, such as a march before a battle. Crowd simulators might create a marching behavior by treating the army as a grid of people walking towards one goal, while a battle behavior would be agent-based.

Having an algorithm that can balance all of these different factors is quite challenging. Due to this, crowd simulators for films take advantage of the fact that scenes are often cut from shot to shot. This allows for different types of simulations to be run for differing crowd types and required behaviors. This comes down to how the agent logic is structured. Most commercial crowd simulators use a complicated and convoluted node tree of actions, situations, and reactions. In these systems, complex fighting behavior can be created, where a soldier may fight like a human would. However, for other crowds, such as those populating a city, more complicated node trees are still needed to simulate a range of daily situations.

Ultimately, it comes down to the fact that algorithms that account for all of these attributes often have subpar computational efficiency, which is why we are often stuck using more specific algorithms tailored to particular types of crowds.

2.3.2 Challenges of Animation Synthesis

The methods for animation synthesis outlined in Section 2.1.2 each have their own limitations, which affect their level of performance.

Just as with the crowd dynamics, the computation time must be low for the simulation to run in real-time or to avoid long baking [5] times.

This is particularly challenging with physics-based animation and machine learning. In the case of physics-based animation, the computer would need to calculate individual muscle fibers and their interactions with external forces like gravity for each character, which is not feasible for larger crowds.

On top of that, methods like motion matching or machine learning algorithms require large databases of animations, which are difficult to obtain. These datasets can be as large

[5] Baking refers to the process of computing and storing simulations (e.g., saving a rigid body simulation to keyframes).

as 4 million frames, making them cumbersome to store and requiring significant hardware. Additionally, machine learning can be computationally expensive at runtime, making it less feasible for simulating large numbers of agents. Furthermore, to create more specialized animations, such as for orcs or soldiers in heavy armor, the machine learning algorithm would need to be run again and would primarily require a massive amount of animation data. Due to the potentially non-human-like nature of these animations, it would be difficult to obtain them through motion capture.

Another common issue in animation synthesis is foot sliding. This occurs in situations like crowd simulators, where the movement of an agent is controlled as a point in 3D space, and animation is added after the agent's proxy point computation. If the agent moves faster than the animation was designed for, it will appear as though the agent is floating through space.

In conclusion, achieving realistic crowd simulation and animation synthesis remains a complex challenge, primarily due to the computational demands and the limitations of current algorithms.

3 Problem Description

The goal of this project was to delve into the inner workings of crowd simulators and improve certain aspects of my own crowd simulator. For my crowd simulator, I set forth three goals that it should achieve. They are as follows:

- **Flexibility:** The flexibility of the crowd simulator refers to its ability to adapt to different situations.
- **Convincingness:** The crowd should display realistic human movements. Both the crowd dynamics and individual rig animations of the agents should reflect real-life motion data. Issues such as foot sliding or unrealistic behavior should be avoided.
- **Execution Efficiency:** Since I will be using Python within Blender, achieving real-time performance will be challenging. Additionally, Blender runs Python scripts on the same CPU core used to display the UI, further reducing available computing power. However, the goal is to design a computationally efficient algorithm that, if rewritten in a different language and better integrated into Blender using parallel threads, would allow the simulation to run in real time.

4 Overview of the Software

My Blender addon consists of an animation synthesis part and a crowd dynamic simulation, which together form the crowd simulator.

Because I wanted to create a base program upon which more complexity could be added, I separated individual parts into functions that return data. Given the data-rich nature of the program, where an array of informations are passed through the individual functions, I decided to define agents using a class. It should be noted that it does not adhere to the principles of OOP programming

This following chapter is dedicated to the documentation of the software. I organized the code into sections where different functions are grouped together. In the following chapter, I will first discuss the crowd dynamics and then, afterward, cover the animation synthesis. This paper includes software to be implemented in Blender version 4.2 and above. The code is available in the following GitHub repository: https://github.com/leomartinch/blender-crowd-simulator.

4.1 Crowd Dynamic

Designing a crowd dynamics system with agents that can circumnavigate small to very large obstacles while staying on the most natural path is a complicated task. To achieve human-like motion, on top of that, skirts the border of the scope of this project, due to the random and sometimes irrational movement humans display. A compromise can be made by creating an algorithm that mimics simple human behaviors in a crowd, and then we can increase the realism by using the randomized attributes of each agent.

We can split up this section into a few different parts:

4.1.1 Global Path Planning

Each agent's main goal is to reach their target destination while avoiding collisions with obstructions or other agents. While many algorithms can guide an agent to its goal, not all display realistic human motion. A famous example is a U-shaped obstruction between the agent and its goal. A human would notice the obstruction and walk around it, while a simple path planning algorithm might walk into the U's convex and then search for a way out. For this reason, many crowd simulators avoid such complex tasks and allow agents to detect only local obstructions.

I tried several algorithms for path global path avoidance but encountered problems, primarily with computational complexity for integrating them into a crowd simulator. After many trials, I realized it was too complex and outside the scope of this paper to create both global and local object avoidance algorithms. I decided to use a hybrid approach, combining a global path planner and a local one. For distant targets, a global path planner is used, while local avoidance suffices for shorter distances. For the global path planner, I chose the A* algorithm due to its popularity and extensive documentation. A* works by traversing a grid representing the world, where individual cells are either free or blocked. There are different direction models available, but I opted for Manhattan motion, which uses lateral and diagonal movements. Given the low-poly nature of my grid, this limited choice does not affect the algorithm's accuracy. Since the A* algorithm has an average efficiency of $O(n)$, it is not calculated at runtime. Therefore, a rough path is pre-calculated for each agent before runtime. These points guide the agent along the optimal path to its target. The grid is defined in Blender space, with an origin point (0,0), grid width, and cell

width. A list of obstruction objects is provided, and the algorithm loops over them. This approach allows more user control and reduces unnecessary complexity, especially in a scene with houndres of meshes used for artistic reasons.

However, the algorithm was still missing a few aspects. Many crowd dynamic methods do not account for societal norms, which reduce realism. To address this, I added danger zones, which are obstructions that agents can still pass through. Danger zones in real life, like roads or railways, represent areas where you would avoid crossing diagonally if there's traffic. Each danger zone has a danger level ranging from 1 to 0.01.

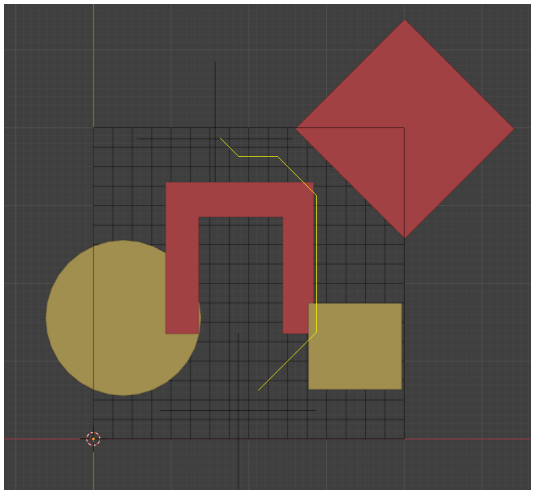

Figure 2: *The blender scene with obstruction meshes. Red are non pass through and orange are dangerzones.*

Figure 3: *The grid with the path rendered in the terminal to see if everything works.*

The algorithm uses a dictionary containing the names of obstruction meshes with their respective values. To convert the Blender scene into the grid, I use the Blender ray-cast function [27]. The binary grid, where 0 is false and 1 is true, determines whether a cell can be passed through. For this to work, obstructions must be meshes with filled faces. For each row and each column in the grid, the center of the cell is calculated. To check if the cell is blocked, I use the Blender ray-cast function. A ray is cast straight down in the direction of Vector((0, 0, -1)) onto the scene. There are two types of ray casts: scene and object. Scene casts check all objects in the scene, while object casts check if a specific object is intersected. The latter is more efficient.

A function using Blender's ray-cast sends a ray from a predefined height down to the scene, iterating over each obstruction. If an obstruction has a value of 1, the cell is automatically set as blocked. If two danger zones overlap, their values are added. The final cell value is returned after looping through all obstructions. For obstructions without a definitive value of 1, the cell value is incremented, but it cannot surpass 0.01 to maintain the walkthrough nature. The grid is stored in a list, with each row set in a sublist for columns. Since ray casting starts at the bottom-left corner, the grid is stored upside down in the list.

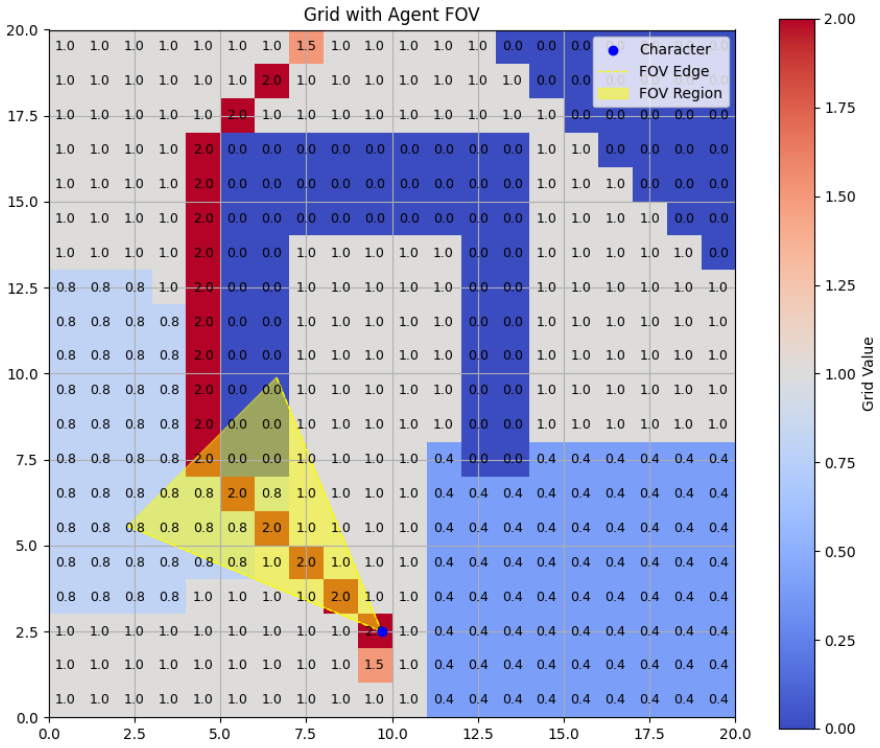

Figure 4: *The visual representation of the grid: the blue dot represents the agent, and the yellow triangle represents its field of view.*

Due to the nature of the low-poly grid, it can happen that thinner obstructions are defined with only one cell thickness. This causes problems if the obstruction is set diagonally, as the algorithm would think it could pass right through the object. To prevent this, I implemented a dictionary with the vertical and horizontal directions, where if one is blocked, its value is changed to True. For a diagonal movement of (x, y), if (x, 0) and (0, y) are blocked, then passing is forbidden.

4.1.2 Agent Vision

We could say that the agents are very goal-oriented and will stop at nothing to reach it—literally. For simplicity's sake, I simplified some attributes of the simulation that work for the demos but could cause problems in more sophisticated simulations. These are:

- The vision works in two dimensions. This means that the agent's sight is not limited by the z-axis or height.
- Agents cannot hear what happens outside of their field of vision. Even though hearing is not a function of eyesight, it still belongs to agent vision because it encompasses everything that informs the agent about what is going on around it.

Humans have a limited field of view compared to animals like the chameleon. Our field of view spans around 180 degrees [28], but I decided to use 90 degrees (see 4) to account for

peripheral vision loss [6].

The function checks if an agent is in the field of view by checking the vector to the target. If the angle between the direction to the neighboring agent and the direction of the agent is smaller than half of the field of view, and the distance to the neighbor is smaller than the radius of the field of view, it returns a true value and adds the agent object to a local list of neighbors.

4.1.3 Local Collision Avoidance

While the agent walks along the path to the global target, it must navigate through the local space. This involves avoiding collisions with other agents or generally avoiding obstructions. Humans display future path planning, where a neighboring agent's position can be estimated to predict where it will be in the future, helping to avoid collisions. To achieve this, I had to code a function that can detect such future collisions. Methods for local avoidance, such as the one proposed in [29], were considered for this. Initially, I intended to use potential fields to guide the agents. There are two types of collisions: local and global collisions. **Local collisions** are those between agents or other objects that are close to the agent. **Global collisions**, on the other hand, are those that an agent avoids by planning its path toward the target. These obstructions are usually larger, such as buildings, but they don't always have to be. For example, the path of two agents walking toward each other while avoiding a collision also exhibits global avoidance because the two agents don't wait until they are only a few meters apart to change course.

Given this vague definition of local and global collisions, it would be ineffective to strictly differentiate between the two. Instead, I found it more effective to categorize obstructions into static and dynamic obstructions. My first few attempts at collision detection did not work as expected. The function to detect collisions is as follows: Additionally, I implemented an effort function that calculates the effort exerted during a specific path or action. This was inspired by the idea from [10]. I modified the effort function to factor in the likelihood of impact, which is determined by another function. Then, I evaluate each rotation over a defined step count and select the one with the lowest value.

4.1.4 Target System

Each agent in the simulation can have multiple targets, which proves useful in situations where an agent might encounter an obstacle on the path to their main target. For example, a soldier might encounter an enemy soldier while on the way to the castle. This flexibility allows agents to react dynamically to changing environments. Two types of targets can be differentiated:

Static targets remain fixed throughout the simulation. These targets are stored as points in 2D space and do not change position. An example of a static target would be a fixed object, like a building or a designated point in the environment.

Dynamic targets are in motion during the simulation. These can include other agents, or moving obstacles like cars on a road. For dynamic targets, the simulation tracks the respective Blender object that is moving.

When a dynamic target is far away, continuously optimizing the path toward it for every frame can slow down the simulation. To prevent this performance issue, a static radius has been implemented. As soon as the target moves outside of this radius—meaning the distance to the target becomes greater than the radius—the path is only re-optimized at the point of escape. The radius is calculated by dividing the length to the target by a specific

[6] Peripheral vision is...

factor, ensuring that the simulation remains efficient even when targets are far away. Since humans do not exhibit the level of precision required to target a specific point exactly, each target has a radius of influence. For intermediary targets, such as a tight doorway, the radius of influence can be scaled down. This helps guide the agent to the correct spot while allowing for a margin of error in the target selection.

As the agent moves along a path toward a faraway target, the simulation uses the agent's current position to select an intermediate target point along a low-poly path. Because this target is not too far away, local avoidance is used to prevent collisions with other agents or obstacles in the environment. The agent decides which point along the path to choose at each time step based on the distance to the target. If the length of the agent's path to the target at the bottom of the stack is within a specified threshold, the agent selects that point as its next target. This dynamic target selection allows for more flexible and efficient navigation within the simulation.

4.2 Adaptive Animation Synthesis

I set out to address some of these problems by creating a novel approach to animation synthesis (see 5). Due to the complex nature of the program, I was unable to get my method working. Unfortunately, I had to revert to my earlier ideas on how to synthesize animation.

My initial plan was to use procedural animations, as proposed in [16]. However, this idea was scrapped because it did not allow for more complex walking gaits beyond the default one. If the animation system were to incorporate injured soldiers or other non-standard types of walking, it would quickly become overly complex.

For my crowd simulator, I ultimately decided to integrate a simple yet effective method. I settled on a hybrid approach based on motion matching.

4.2.1 Integration into Blender

As stated before, Blender has an integrated Python API. This API allows me to access location, rotation, scale, and other information from the 3D objects and subsequently allows me to change these values directly from my script. The Blender Foundation has been working for a while on a new animation system, and they have recently changed the default rotation for bones from Euler (XYZ) to quaternion (WXYZ) rotation. Quaternion rotation is preferred over Euler rotation because it avoids gimbal lock [30], which can make it harder to animate certain motions. On top of that, quaternion rotation is also computationally more efficient to calculate, so it is generally a better choice for programs that have to calculate thousands of vector rotations per second. I will be using quite a lot of quaternions to create an animation for the agents in the simulation.

Character animations are created by having a rig that consists of bones that influence the location and rotation of vertices on a mesh. These armatures have to be fitted to the character model, and then for each bone, the weighted area that the bone will influence is set during the rigging phase. There are many different standard rigs in the industry that make it possible to seamlessly move one animation to another rig. I went through many iterations of which rig to use, and I decided to use a standard rig that resembles the one used in Adobe's Mixamo software for ease of use. Mixamo has a library of animations that are available for free download [31]. This makes the process of retargeting the motion quite easy because they have very similar proportions.

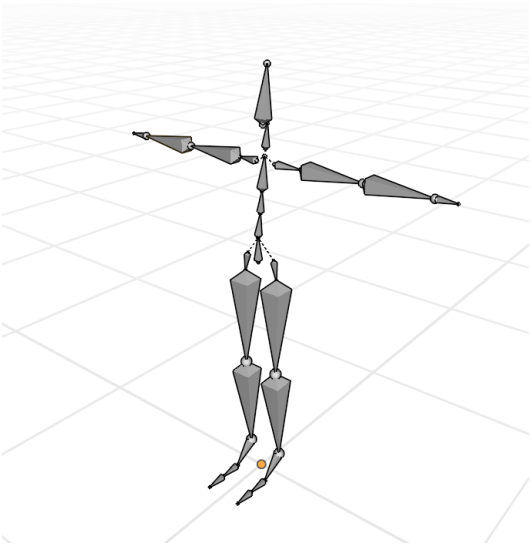

Figure 5: *The Rig in T-Pose from the side.*

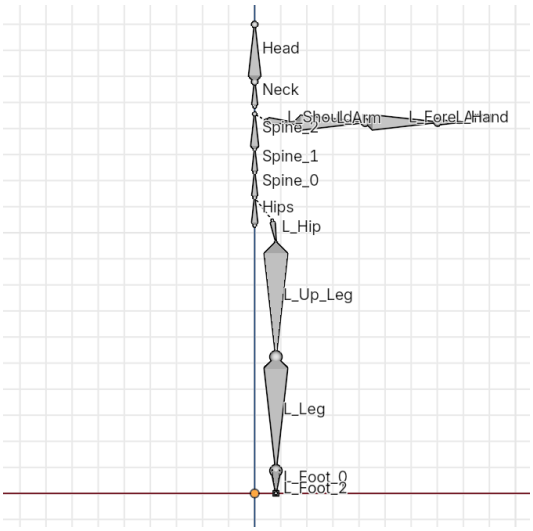

Figure 6: *Naming system of the bones. The names are the same for the right side.*

My rig is based on a human skeleton, keeping only the parts I deemed important for my animation synthesis. I removed the fingers because they would increase storage space without providing any benefits to the overall quality of the animation. For future reference, I would propose a hand animation system that could interact with objects with a higher quality than keyframe animation.

4.2.2 Base Animation Storing

The animation synthesis bases itself on existing animations. Usually these base animations are looping animations of around 30 frames which hold one cycle, for example one walking gait. To increase animation diversity and decrease the likelyhood of spotting identical animations in a crowd I allow the input of longer cycles. My method can handle base animations that hold multiple cycles. This increases the memory required to run the animation synthesis but this can be chosen by the end user if they want to use longer animation bases or not based on their computer specks.

The animation data is exported into a JSON file. JSON files are flexible when it comes to data types, which is desirable because I will be storing coordinates and quaternion rotations. While the standard file format for storing animation data is FBX, this would also include 3D geometry data, which increases memory usage unnecessarily. Additionally, I had to create a hybrid file in which the non-adaptive bones would have their quaternion rotations stored, while the adaptive bones would have their direction and target positions stored. I also did not want to store animation data for all the bones, such as the hands and feet, which significantly decreased the file size. JSON is a human-readable file format, so it contains many spaces and tabs to improve readability. To save space, I removed all unnecessary spaces and indentations. The average animation file, with 25 frames, was around 25 KB in size, which is an acceptable size. Blender starts animations at frame 1, which can make numbering messy when imported into a list. To avoid this, the animations are defined to start at frame 0. The animation cycle's length is defined by the end frame, with frame 0 and the final frame representing the same pose.

5 Excerpt: Experimental Method for Animation Synthesis

This whole project has been at the center of my mind for more than half a year. During that time, I explored a variety of methods for animation synthesis because my goal was to create better and more realistic animations for crowd simulators. I came up with a novel idea for animation synthesis, which I believe adds value to this paper, so I decided to include it here. For the upper body, I required a few more animations to cover the range of actions the characters can perform. These animations are tightly connected to the crowd dynamics component. The animation data is exported into a JSON file. JSON files are flexible when it comes to data types, which is desired because I will be storing coordinates and quaternion rotations. The standard file format for animation data is FBX, but this would also include the 3D geometry data, which increases memory usage unnecessarily. Additionally, I needed a hybrid file format where the non-adaptive bones store their quaternion rotation, while the adaptive bones store the direction and target position. I also chose not to store animation data for all the bones, such as the hands and feet, which significantly reduced the file size.

JSON is a human-readable file type, and because of this, it contains many spaces and tabs to improve readability. To save space, I removed all unnecessary spaces and indentations. The average animation file, which is 25 frames long, is around 25 KB in size, which is an acceptable size. Blender starts animations on frame 1, but when imported into a list, this makes the numbering complicated and messy. For this reason, the animations are defined to start on frame 0. The animation cycle length is defined by the end frame. Frame 0 and frame end share the same pose.

The most problematic part of synthesized animation is the legs during locomotion. A major issue in animation is foot sliding. This occurs in situations like crowd simulators, where the movement of an agent is controlled as a dot in 3D space, and animation is added afterwards. If the armature moves at a speed greater than the baked-in animation, it will appear as if the agent is floating through space. Research on human walking shows that a walking gait can be defined by the transition from support mode to swing mode. There is a moment when both legs are in support mode, and during running, this transition is highly noticeable. This is because, in moments when the leg is supporting the body, it still moves, which is mechanically impossible. My initial plan was to use procedural animations as proposed in [16]. However, this idea was discarded because it did not generate deterministic variable animations. For a single input, it would only create one animation, and it did not allow for more complex walking gaits. If the animation system had to incorporate injured soldiers or other non-standard walking styles, the complexity would grow too quickly. Another approach I considered was animation blending, which is the main method used by most crowd simulators. However, the issue with animation blending is that it doesn't take into account the momentum of swings, which negatively affects the realism of swinging animations. Finally, I decided to implement a hybrid approach based on motion matching. The most common way to synthesize animations in crowd simulators is through animation blending. This method requires a large dataset of animations for various scenarios, which are blended to create the desired output. These datasets contain thousands of animation frames, often obtained through motion capture. There are several blending methods, each with its advantages and disadvantages, but they all share the common requirement of large datasets.

5.0.1 Define next Step position

A locomotion animation can be defined as a cyclic series of right and left leg motions. The legs can either be in swing mode or support mode. Unlike animation synthesizers that use blending, I steer the character in the animation phase.

At each point where both legs are in support mode, the crowd dynamics functions are called, and the rotation and speed at which the agent will continue on are returned.

Based on the information of the agent, we know which leg will be the swinging leg at that moment. To calculate the position where the swinging foot will land, a straight line is simple to calculate.

5.0.2 Retarget Animation Path

The path the foot follows during animation is stored as coordinates in 3D space. This path has a home point and a target point, starting at the home point and ending at the target point.

Once the animation is stored in memory and the next step position is determined, I must retarget the path to the new target. The retargeting occurs on a straight line that spans from the home to the target point. The retargeting function takes all the points along the path and adjusts them accordingly.

5.0.3 Inverse Kinematics

The human body must be simplified into basic components that can be calculated mathematically. The process of rigging a character already provides us with this structure, but it only shows how the surface or model itself changes. What we need is a mathematical rig upon which we can calculate the motion paths and more.

Legs and arms, in their simplest form, are two connected sticks with movable endpoints. The endpoints are either the foot or hand, and the other end of the chain is connected to the main body (the shoulder or hip). To make the calculations as simple as possible, we use inverse kinematics to calculate the position of the knee or elbow.

In 2D space, this is a straightforward trigonometric equation, but once you move to 3D space, you encounter an infinite number of possibilities, making the calculations more complex.

6 Evaluation

In this section, I describe the results of the tests conducted on my crowd simulator. As a prototype, the program is not fully optimized, and the use of Python, known for being relatively slow, impacts performance. I ran the tests on an AMD Ryzen 9 5900X 12-Core Processor. Since Blender's Python scripts share cores with graphics rendering, performance may be affected.

The tests, commonly used in crowd simulations, assess whether human-like behaviors emerge in the system. Several scenarios were created to evaluate the crowd system.

6.1 Agent Circle

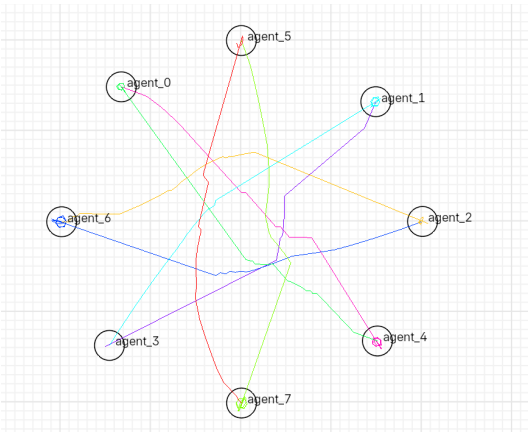

Figure 7: *Simulation with width set to 0.5 for a bit more collision avoidance.*

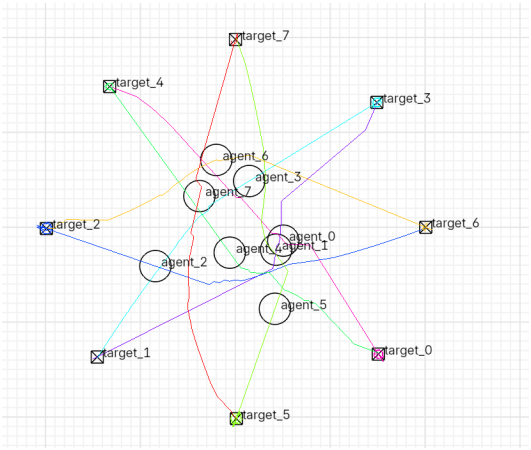

Figure 8: *Screenshot at frame 40 of the simulation showing a collision where Agent_0 and Agent_1 overlap.*

In other crowd simulators this scenario results in a spiraling motion at the center. My crowd simulator also had such a spiraling motion.

6.2 Frontal Collision

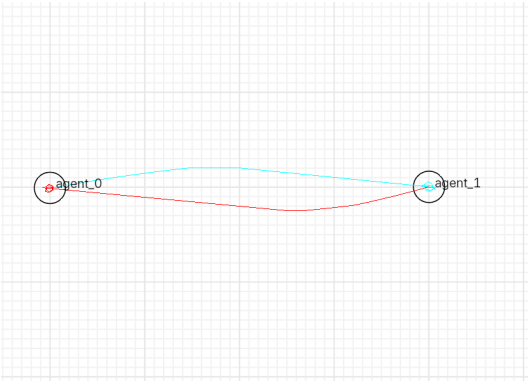

Figure 9: *Frontal Simulation with width set to 0.4.Agent_0 and Agent_1 switch position.*

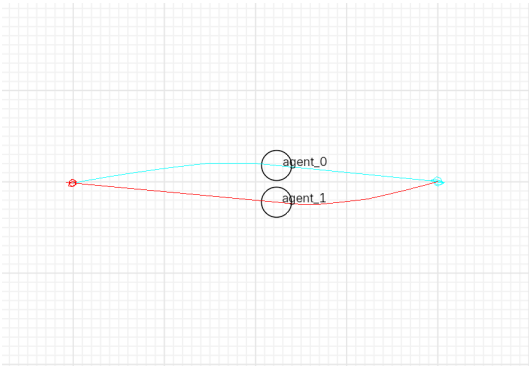

Figure 10: *Screenshot at frame 20 of the same simulation. Both agents do not collide with each other.*

Frontal collisions are used to test if the agent can predict future collisions. In my scenario this was the case

6.3 Group Frontal Collision

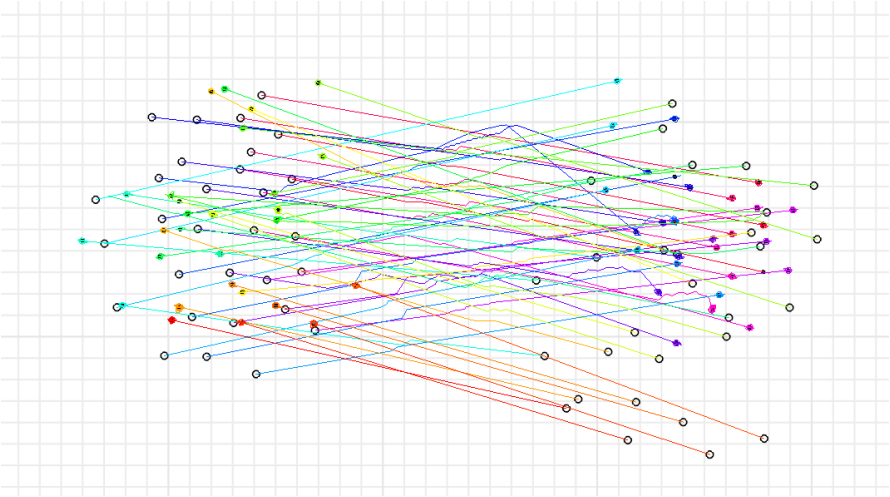

Figure 11: *Simulation of 60 Agents with simmilar speeds spanning from 0.4 to 0.6.*

To test out how the crowd simulator works in crowds with higher density I set off two groups of each 30 agents towards each other. The whole simulation for 200 frames took an average of 2.6 seconds to run on my machine.

7 Conclusion and Future Improvements

Based on the results of the evaluation, I can confirm that my crowd simulator does exhibit crowd behaviors. Unfortunately, due to time limitations, I was unable to implement my animation synthesis method into the code. Additionally, the finished crowd simulator lacks functionality, as it does not include a user interface for easy access.

There are many areas I would improve in this prototype, primarily based on scripts I've already written but was unable to integrate.

8 Reflection

Working on this project has provided me with valuable insights. I started with limited knowledge of programming and the inner workings of 3D software, and I've come a long way. I've learned how to effectively search for solutions when the code doesn't work and improved my time management skills, which was once a weakness.

More importantly, this experience has opened my eyes to the multitude of possibilities in this field. One of my biggest challenges was revisiting my old code and understanding it. Looking back, I can see how much my coding abilities have developed, particularly in writing code that is easier to understand later on.

Another significant challenge was knowing when to stop pursuing an idea. I spent weeks or even months working on approaches that ultimately weren't successful. These setbacks, combined with high expectations, led to an outcome that didn't meet my initial goals. Unfortunately, I didn't have enough time to test my experimental animation synthesis ideas. Despite this, my morale remains high, and I'm eager to continue developing this project—though the code will need to be completely rewritten.

I'm already focused on a new goal: creating a short film featuring medieval battles, which will be exciting to code. While this project was stressful at times, especially towards the end when I had to abandon one method to ensure the crowd simulator worked, I'm proud of the progress I've made as a programmer. Though I wasted time on unused scripts, I've learned valuable lessons along the way. The individual scripts I created each represent unique ideas and functionalities that could contribute to a larger, more complex project.

References

[1] Massive Software. *Massive*. URL: https://www.massivesoftware.com/ (visited on 10/14/2024).

[2] Basefount Technology HK Ltd. *Miarmy*. URL: https://www.miarmy.com/ (visited on 10/14/2024).

[3] Golaem Software. *Golaem*. URL: https://golaem.com/ (visited on 10/14/2024).

[4] Rafael Blanco, Gustavo Patow, and Nuria Pelechano. "Simulating real-life scenarios to better understand the spread of diseases under different contexts". In: *Scientific reports* (2024).

[5] Suiping Zhou. "Crowd Modeling and Simulation Technologies". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* (1987).

[6] Simon Green. "Particle Simulation using Cuda". In: *NVIDIA whitepaper* (2010).

[7] Craig W. Reynolds. "Flocks, herds and schools: A distributed behavioral model". In: *Association for Computing Machinery* (2010).

[8] Craig W. Reynolds. *Boids*. URL: https://www.red3d.com/cwr/boids/ (visited on 08/23/2024).

[9] GeeksForGeeks. *A* Search Algorithm*. URL: https://www.geeksforgeeks.org/a-search-algorithm/ (visited on 12/05/2024).

[10] Stephen J. Guy. "PLEdestrians: A Least-Effort Approach to Crowd Simulation". In: *Eurographics/ACM SIGGRAPH* (2010).

[11] Rahul Narain. "Aggregate Dynamics for Dense Crowd Simulation". In: *SIGGRAPH Asia* (2009).

[12] N. Pelechano. "Controlling Individual Agents in High-Density Crowd". In: *Eurographics/ACM SIGGRAPH* (2007).

[13] Richard Tonge, Feodor Benevolenski, and Andrey Voroshilov. "Mass Splitting for Jitter-Free Parallel Rigid Body Simulation". In: *ACM Transactions on Graphics (TOG)* (2012).

[14] Daniel Holden. *Code vs Data Driven Displacement*. URL: https://theorangeduck.com/page/code-vs-data-driven-displacement (visited on 12/09/2024).

[15] Kristjan Zadziuk. *GDC 2016 - Motion Matching, The Future of Games Animation... Today*. YouTube. URL: https://www.youtube.com/watch?v=KSTn3ePDt50 (visited on 08/15/2024).

[16] Thomas W. Calvert Armin Bruderlin. "Goal-Directed, Dynamic Animation of Human Walking". In: *Computer Graphics* (1989).

[17] Thomas Geijtenbeek, Michiel van de Panne, and A. Frank van der Stappen. "Flexible Muscle-Based Locomotion for Bipedal Creatures". In: *ACM Transactions on Graphics* (2013).

[18] Dave Fothergill vfx. *I've fallen, and I can't get up!* Vimeo. URL: https://vimeo.com/109169719 (visited on 11/13/2024).

[19] Daniel Holden. "A Deep Learning Framework for Character Motion Synthesis and Editing". In: *siggraph.*. (2017).

[20] Yi Shi et al. "Interactive Character Control with Auto-Regressive Motion Diffusion Models". In: *ACM Trans. Graph.* (2024).

[21] Rokoko. *What is Motion Capture, and How Does it Work in 2022?* URL: https://www.rokoko.com/insights/what-is-motion-capture-and-how-does-it-work-in-2022 (visited on 12/09/2024).

[22] Blender Foundation. *Blender 3.4 Python API*. 2024. URL: https://docs.blender.org/api/current/index.html (visited on 12/02/2024).

[23] Blender Foundation. *Blender 2.92: New Features Overview*. 2021. URL: https://www.blender.org/download/releases/2-92/ (visited on 12/02/2024).

[24] Blender Market. *Procedural Crowds*. URL: https://blendermarket.com/products/procedural-crowds?ref=247 (visited on 12/09/2024).

[25] Crowd Sim 3D. *Crowd Simulation*. URL: https://www.crowdsim3d.com/ (visited on 10/23/2024).

[26] Daniel Thalmann. "Challenges in Crowd Simulation". In: (2004).

[27] Blender. *Object Ray Cast*. URL: https://docs.blender.org/api/current/bpy.types.Object.html#bpy.types.Object.ray_cast (visited on 12/09/2024).

[28] Riad I Hammoud. *Passive eye monitoring: Algorithms, applications and experiments*. Springer Science & Business Media, 2008.

[29] Abhinav Golas, Rahul Narain, and Ming Lin. "Hybrid long-range collision avoidance for crowd simulation". In: *Proceedings of the ACM SIGGRAPH symposium on interactive 3D graphics and games*. 2013, pp. 29–36.

[30] Wikipedia. *Gimbal lock*. URL: https://en.wikipedia.org/wiki/Gimbal_lock#Gimbal_lock_in_three_dimensions (visited on 12/13/2024).

[31] Mixamo. *Mixamo Home Page*. URL: https://www.mixamo.com/ (visited on 12/09/2024).

Ich habe in meiner Arbeit ChatGPT als Hilfsmittel verwendet um mir bei dem Schreibprozess zu helfen. Auch um Bugs im Code wurde ChatGPT verwendet, jedoch nicht oft.

Der/die Unterzeichnete bestätigt mit Unterschrift, dass die Arbeit selbständig verfasst und in schriftliche Form gebracht worden ist, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und dass alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind