Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Crowd Simulation

A Python Framework for the Simulation of Human Actors in Miarmy

Bachelor Thesis

Christoph Maurhofer

cmaurhof@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Michael König, Klaus-Tycho Förster
Prof. Dr. Roger Wattenhofer

August 17, 2016

Acknowledgements

I would like to thank my advisors Michael and Klaus for the inspiring conversations, the valuable feedback, and their helpful advice. I also want to express my gratitude to Prof. Wattenhofer for the possibility of working on this project. And finally, I want to thank my family and my friends, who not only supported me in every aspect but were also a source of humorous ideas and important feedback.

Abstract

The widespread application of crowd simulation in movies and games has led to numerous tools that offer a simplified approach to this topic and have a strong focus on visual quality. The objective of this thesis is to evaluate if one such tool is suitable for the implementation of more complex, intelligent behavior. The chosen Autodesk Maya plug-in Miarmy has been extended with multiple Python scripts to provide the agents with the ability of realistic pathfinding with obstacle avoidance. Additional features are the individualization of the agents, flight behavior near dangerous objects, as well as the ability to chase other agents. The decision logic has been tested in numerous scenarios to receive a visual feedback and to confirm the realistic behavior of the agents. While the model has limitations and there are problems with the used tool, the result is a working example, positively answering the central question.

Contents

Chapter 1

Introduction

1.1 Motivation

Large numbers of characters or other entities behave in ways that can be difficult to predict because of the complex interaction and navigation. The simulation of such a crowd can provide valuable information for the planning of buildings, traffic ways or events, where an efficient guidance of people is critical for their safety or is of economic interest. A lot of research has been focused on the movement of pedestrians. Such as in the case of Kneidl et al. [1] with the evacuation of a soccer stadium or Curtis et al. [2], simulating huge numbers of pilgrims circling the Kaaba during the tawaf to compute the density, speed and the number of people completing it per hour.

Crowd animation has found its way into the entertainment industry. Not only do many movies feature impressive images of thousands of individuals on screen, video games also need efficient algorithms to simulate many characters in real-time. In recent years, a large variety of software has been developed to offer a simpler approach to creating these animations. The best known may well be Massive [3], developed for the *Lord of the Rings* trilogy [4]. Another tool is Miarmy [5], a plugin for simulations in the 3d software Maya [6].

Most of these programs have been designed for artists and therefore mainly feature more intuitive but less complex tools for configuration of the simulation and rather limited extension possibilities. They allow the fast creation of inspiring videos like *I've fallen, and I can't get up!* [7], in which the characters do not act in a realistic way, but by simpler decision patterns. Building on the work of Michael Weigelt [8], we analyze the suitability of these tools for scientific work by developing more a complex decision and movement logic with Miarmy and Python. The result is tested in multiple different scenarios.

1.2 Related Work

There are many different approaches one can take to simulate large numbers of individual entities. Zhou et al. [9], and more recently Ijaz et al. [10], propose the classification of those methods by *crowd size* and *time scale*. Even though the borders have gradually been blurred with respect to the size, one can often distinguish macroscopic from microscopic approaches. Macroscopic models describe abstract behavior and mostly ignoring unique traits and interactions between the characters. Microscopic models, on the contrary, focus on those individual entities with possibly individual features. Therefore, they need more computational power to simulate the behavior of an individual entity and the approach is more suited for a smaller scope. But within the last years, many models have combined benefits and aspects of those two extremes, as seen in the survey of Ijaz et al. [10].

Another important aspect of the model is the representation of each entity or rather the crowd itself. Naturally, this has a big influence on the computational cost and is therefore directly influenced by the number of entities in the scene. One possibility is to describe each individual as a particle in a physics system, which is influenced by global and local laws that represent physical and social rules. The homogeneity of such a crowd allows for a more efficient and less complex computation. Braun et al. [11] used such a global particle model from Helbling et al. [12], which describes panicking pedestrians, and expanded it with additional parameters for different personalities.

In an agent based approach, each entity in the crowd is an autonomous character with its own decision rules and properties. Individual characteristics, complex pathfinding, awareness of the environment and social interaction can be implemented. This high degree of freedom results in a very natural behavior of the crowd but comes at a high cost of computational power. Even path planning with multiple groups having different origins and destinations is already NP-hard [13]. Thus, this approach is only reasonable for lower numbers of agents.

In this thesis, we extend the agent based system of Miarmy to incorporate complex logic for collision avoidance, pathfinding and fleeing with the influence of individual attributes in a microscopic approach. The term *agent* is subsequently used to refer to an individual entity in the simulation, which is described by Miarmy's own representation (see Chapter 2) and the corresponding state that is stored and updated separately in Python (see Section 3.1).

Chapter 2

Simulation System

2.1 Overview

In Miarmy, the individuals of a crowd in a scene are instances of a specific agent type. Such an agent is a description of a certain appearance and behavior that is shared among all characters of his type. Important components of this group are:

Original Agent: A collection which includes the skeleton (commonly called character rig) and a simple approximation of the geometry used for physical simulation and the representation of this type of agent prior to rendering.

Actions: These nodes describe animations by the transformation of the agent's bones per frame.

Decisions: The logic described by decisions is the driving force of the agent. In each frame, every agent checks its decisions sequentially. If the input conditions of a node evaluate to true, the output sentences are executed. With the *human language logic* Miarmy offers a variety of phrases for such inputs and outputs to interact with Maya and the plugin itself. For example "change action:... playback speed ..." to adjust the playback speed of a specific animation.

Geometry: The visual representation of the agent is defined by a collection of textured geometry, bound to the character rig. To vary the appearance of an agent type, Miarmy offers the possibility of randomizing the assignment of geometry groups to each character instance.

The agents are instantiated by defining placement nodes in the scene. The crowd's animation is then computed by simulating each agent's decisions and interactions with additional objects in each frame, with a second of the final simulation being 24 frames. The movement is cached by saving the transformation of the character's rig per time step. For rendering, every agent is assigned

some or all of the geometry defined in the agent type, which is then deformed according to the cached animation.

While this approach seems tedious for a simple crowd simulation, the organization of the data and the separation of these steps allow for much customization in the visual aspect. The user has many possibilities to adjust the appearance of the crowd and the environment by having all the tools of Maya available. By caching the simulation itself before the visualization, Miarmy is able to offer the preparation of the geometry for other rendering solutions like Arnold [14].

A problem with such crowd animation tools with a focus on the visual aspect is that the simulations created for movies are usually rather short and less complex. Therefore, the possibilities offered by the default logic sentences are limited. Complicated processes like pathfinding can be imitated with Miarmy's perception objects as helper. An example would be a road defined by a curve: agents can be guided around obstacles without the need of intelligent decisions. While this is a benefit for a specific animation, it is clear, that the independent behavior needed in more complex simulations, can't be achieved this way.

Luckily in Miarmy, there is the possibility of calling Python scripts from input conditions and from resulting commands of decisions. This enables large extensions to the logic system, as all the standard Maya commands and even additional Python modules can be used. Unfortunately, the interaction between Miarmy with its agents and such a script is limited and there is little documentation available.

The Software used for this thesis is Miarmy Express 4.7 in Autodesk Maya 2016, along with the included Python 2.7 interpreter. In addition to the official maya.cmds and the standard modules in Python, Numpy has been used to simplify some calculations.

2.2 Python Interface

The visual part of the scene, including geometry and animations, the physics, as well as driving and caching the simulation is done with the standard tools of Miarmy. As these features work well and have no influence on the agent's behavior, there is no need to redo this work. For the realistic performance of the characters, the complex computations are done in Python scripts. They are called by decision nodes, that act as links between the custom logic and Miarmy but also execute specific commands if necessary. Each frame, the scripts are called when Miarmy updates the agents. They compute the current state of each individual and return values to steer the animations. With this approach, the standard system is only expanded by some calls to custom python scripts.

While it would be convenient to directly code and define custom functions in the input and output fields of the decision nodes, or being able to expand the

human language logic of Miarmy, such features do not seem to be planned. And even though there are hints that the development on these parts is encouraged, the documentation lacks fundamental information to do so. Still, the approach mentioned above works quite well for this purpose.

2.3 Agents and Decisions

For this thesis, we designed one system for the decision logic of all human characters. Multiple different agents were created in Miarmy because the character rig and the animation differ between certain types (e.g. between men and women). Each agent has a set of three actions: *idle, walking, running.* As they practically only have to solve navigational problems, this simple set of animations suffices. The actions can be varied in speed to adjust the velocity. To diversify the appearance of the characters, the male and female agent type both have three different geometries that are chosen randomly. The 3d character models have been created with Adobe Fuse [15], with the automated rigging and the animations from Mixamo [16].

As previously mentioned, the decision nodes are the link between the customized logic in the Python scripts and the representation of each character in Miarmy. To decouple the logic system as much as possible, most decision nodes mainly act as getter and setter for the values in the scripts and of the agents. The following nodes have been implemented:

updateAgent: This node only calls the method *AS_updateAgentState(agent_id)* which updates the state of the agent, namely of the class *AgentState*, stored in memory. This object holds values such as the current action to be executed, its playback speed, a description of the target and many others, which then are retrieved by other decision nodes. Please see Chapter 3 for additional details on the implementation in Python.

getID: Because it is not possible to directly pass the agent's ID to a Python function, a separate decision node is needed to write it into to an attribute attached to the agent at runtime. The value of the attribute can then be referenced as argument to the function call.

updateAlive/checkAlive: To keep the information between Miarmy and the agent's state in memory coherent, these nodes repeatedly call Python functions to update and read the value indicating if the agent still is alive. This is necessary, as the agent may either collide physically or may be deactivated for some other reason by a script. One reason, that the *updateAlive* node is needed, is that calling a Python function from a decision node's output sentences is bugged and highly unreliable. Arguments are not passed

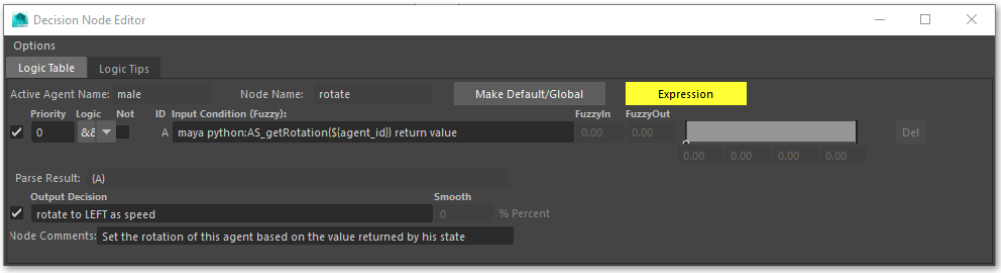

Figure 2.1: An example of the usage of *expression mode* in Miarmy. The rotation value is determined in multiple Python scripts, written to the agent's state and finally retrieved and directly set by this decision.

correctly and sometimes the function is not called at all. Therefore, some of the calls had to be moved to the input sentence of additional decisions.

stand/walking/running: The agent can either stand still or move forward. There is no motion such as strafing left or right, jumping, crouching, or other. Such actions would be needed rarely, as with varied speed forward and mostly free rotation to left and right, practically every target can be reached. Each of these three nodes gets the value *actionId*, which is set in the agent's state. It states, which animation needs to be executed. Multiple different nodes are required because Miarmy doesn't allow case distinctions on a value to decide between different outputs.

speed: Similar to the nodes getting the current action, these nodes retrieve the value that defines how fast the animations should be played. As walking and running are animations that move the character's position, the playback speed directly influences the movement speed of the actor. There are four different levels of speed: *0.5, 0.75, 1*, and *1.5* (*1* is the default speed). This discrete approximation is necessary because setting the values via the direct expression mode inexplicably crashes Maya.

rotate: To set the agent's rotation to either left or right, a feature introduced in Miarmy 3.5 is used. A decision node can be marked to use *direct expression control*. In this mode, the input sentence of the decision may be altered to not return *True* or *False* but the value generated by this new sentence. So in the agent's state, one can explicitly define, how fast left or right (negative left) should rotate. This value is then passed on to Miarmy to adjust the character's direction (Figure 2.1).

collision: In some scenes, kinetic primitives are used. These are objects that are considered in the physics simulation run by Miarmy. It is possible to check for a collision in the input conditions of a decision node. If one has occurred, the agent can react properly and adjust values, influencing the

behavior of other agents. In some scenes, such a primitive is marked as dangerous by the agents, if it did collide with one of them.

Chapter 3

Logic System in Python

With Miarmy's own decision nodes mostly reduced to simple value passing functionality, the system is missing the part where each agent's behavior is controlled. As Maya and Miarmy support Python, more complex computations are easily possible. Miarmy also allows calling *MEL* commands (*Maya Embedded Language*: Maya's internal scripting language), but the convenience and versatility of Python, let alone the huge number of external libraries, was preferable. Especially because Miarmy offers a Python wrapper with support for most commands.

Miarmy, of course, can handle movement in all three dimensions. But to simplify the system needed in this thesis, navigation has been reduced to the X/Z axes. In most situations, this is enough and does not limit the possibilities significantly.

3.1 Structure

The main functionality for the decision logic has been packed into two different modules: *decision_pathfinding* and *obstacle_detection* but there are multiple additional scripts. Most importantly *scene_initialization*, which is executed to set up the states of the agents and search for objects in the scene that are to be recognized by the obstacle detection.

Each agent has its state in a global list, addressable by his identification number, which is assigned by Miarmy on creation. As an instance of the class *AgentState*, it contains basic information for the interaction with Miarmy (such as the previously mentioned *actionId*) and additional values needed for more complex computations (e.g. a collection of recently seen obstacles, acting as a memory of the character). The latter are explained and discussed in the following sections.

Nearly all changes in the behavior of an agent are initiated by the method *AS_updateAgentState(agentId)* in the module *decision_pathfinding*. In multiple stages, the current target is located and it is determined if the character is

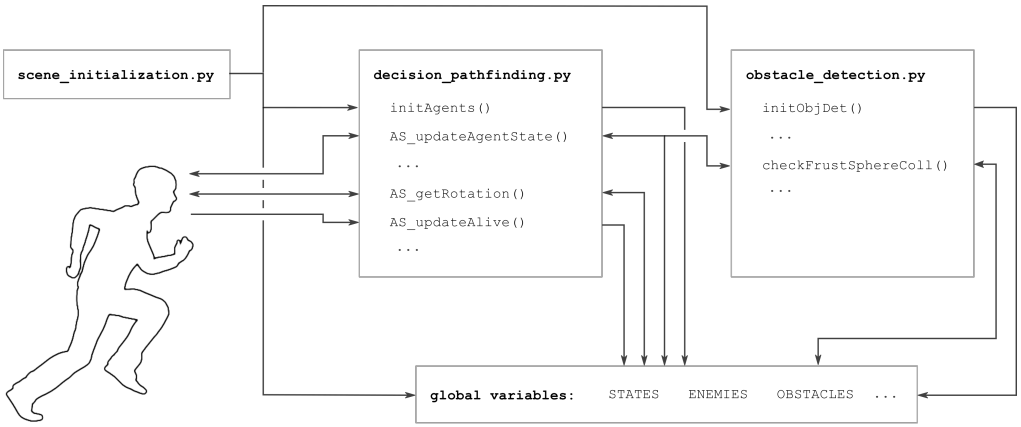

Figure 3.1: An illustration of the interactions between the Miarmy agent on the bottom left and the Python scripts used in this project. Function calls are depicted by arrows. Please note that some functions are omitted to keep the figure comprehensible.

escaping an enemy, if there are any obstacles, how to evade them, and in which direction to go at which speed. According to the given situation, the agent's state is adjusted. As mentioned in Section 2.2, this function is invoked each frame, when Miarmy updates the agent's decision nodes. All the changes then are again queried by the agent via other nodes.

In the following sections, important parts of the decision system organized in *AS_updateAgentState(agentId)* are explained further.

3.2 Movement

In the *AgentState* class, there are four values directly affecting the movement of an agent in the scene: *actionId, actionSpeed, rotLeft*, and *evadeRotLeft*. While the first two define, if and how fast the agent is walking or running forward, the other two are responsible for adjusting the direction of the movement. When querying the rotation, *evadeRotLeft*, which is only responsible for redirecting the agent around an obstacle, always has priority over *rotLeft*. This has the effect, that the agent's primary goal always is to avoid obstacles. All these values are influenced by most of the other parts of the decision system.

3.3 Targets

An essential aspect of human navigation is to direct one's movement towards a specific destination. In our solution, every agent can aim for one target, which is

either described by the name of an object or a tuple of X/Z coordinates. Usually, this position is defined when setting up the scene and creating the agent. Of course, it can be changed by a script during runtime (e.g. see Section 3.6) by adjusting the *target* variable in the agent's state.

At the same time, each agent has a list of *checkpoints* that can be used to define a path towards the final target. They carry additional information of when they were created and which, if any, obstacle was the reason for a checkpoint. The checkpoint list is used as a stack: the most recently appended point is visited first and deleted when reached. Checkpoints are dynamically added to avoid obstacles but also removed in certain situations (e.g. when one seems unreachable or not beneficial anymore). After a certain amount of time, they are forgotten and removed.

If there is a valid target or checkpoint at the beginning of the update of the agent's state, the difference to the agent's current direction is computed. Depending on whether it lies to left or the right, *rotLeft* is set a positive or negative number to indicate the rotation speed to the left.

3.4 Obstacle Avoidance

Designing a system that is able to navigate around objects collision-free and in an intelligent manner is an intricate task. Obviously, to achieve human-like performance is beyond the scope of this thesis. As with the targeting mechanics, a compromise is made regarding the complexity of the system and the quality of its results. As a result, agents react to obstacles they see. There is no abstract mapping of the environment as humans usually perform. Navigating with such an overview would help the agent to find very efficient paths around objects and towards targets. But grid-based algorithms have not been used because of the increased complexity of such a model and the memory needed for storing a representation of the environment accurate enough for each agent separately.

Objects that are to be recognized as obstacles must be listed in the global container *OBSTACLES*. During the initialization of the scene, they are automatically detected by the prefix "geom_" for objects and the standard naming of the Miarmy agents "McdAgent". By default, an agent doesn't treat itself and its target as an obstacle.

3.4.1 Detection

The field of vision of the agents is composed of six circular sectors (depicted in Figure 3.2). The frustum is defined by two of those, one on each side and adjacent to the z-axis (in front of the agent). Each side's sector is separately defined by a radius and an angle. Two other sectors are symmetric to the z-axis

and defined by only one radius and one angle (*sphAng*), again oriented to the front and intended much smaller than the frustum. With the focus on this closer area, these two are mostly used to react to obstacles that appear directly in front of the agent and require an immediate reaction. In contrast, the large frustum with the first two sectors offers the possibility of recognizing obstacles in the distance, which is especially useful for the prediction of their movement (Section 3.4.2). Representing the backside of the perceptive area, the last two sectors are also symmetric and described by yet another radius and the supplementary angle to *sphAng*.

When *checkFrustSphereColl()* in *obstacle_detection.py* is called, every object marked as an obstacle and not excluded by the caller is checked for collisions with each of these sectors. Other agents are tested by their position with a specified radius around them. Objects marked with the "geom_" prefix are checked by observing each of the vertices defining their geometry. For performance reasons, it is recommended to use separate, more simple meshes for collision detection of high-poly 3d models.

In each sector, only the obstacle closest to the agent is remembered. In addition to the binary value that indicates, if a sector is obstructed, further information is gathered: the distance to the obstacle, its name, if it is a predicted obstacle, and the angular values of how far it extends to the left and right (in (-180°,180°]) from the perspective of the agent. Also, every object colliding with one or more of the sectors is remembered with the name, position, as well as the rotation around the vertical axis, and the current frame number and is returned as part of a dictionary.

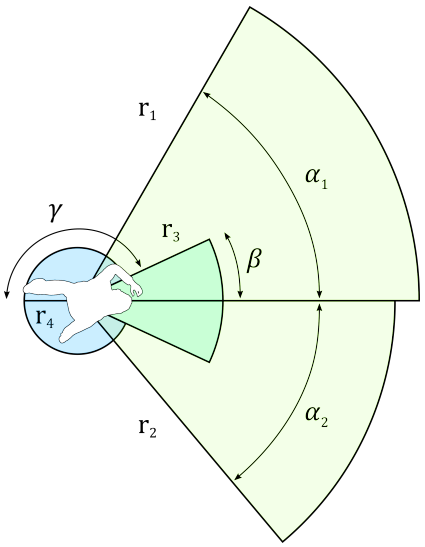

Figure 3.2: The six sectors used for obstacle detection. Specified by the variables *frustDis* (r_1, r_2), *frustAng* (α_1, α_2), *sphDis* (r_3, r_4), and *sphAng* (β). Please note that $\gamma = 180° - \beta$.

3.4.2 Object Memory and Prediction

In many cases, when objects move, reacting only to the current situation can guide an agent to a spot, which is blocked after the next time step. An example is the scene with the rotating bar (such as in [7]). To avoid such situations, agents need to predict the position of an obstacle for a certain number of future

frames.

To achieve this effect, all the obstacles that have been seen in one of the six circular sectors (see Section 3.4.1) are memorized in the agent's state. When checking all potential obstacles for collision and one of these objects is remembered from a previous frame, its future position and its rotation around vertical axis are predicted linearly in each aspect with the difference per frame of the remembered values and the current ones. It is possible to configure the amount of predictions calculated per object or agent and to set a factor that influences the number of time steps between each prediction. If an object at the current frame has not moved from its position in the past, no prediction is computed. By default, the current position of a moving object is neglected and at least the first prediction is checked on collision. This simulates the expectation that object in motion keep moving.

If a prediction obstructs one of the six areas is indicated by another list that is returned with the other information gathered from the obstacle detection (see Section 3.4.1). This can be useful in situations, where an agent seems to be completely enclosed by obstacles. If one of those is just a prediction, the agent can still try to evade the others by choosing this direction.

Keeping the information about encountered obstacles indefinitely long, is often not necessary and can result in wrong predictions. For example, when an object is last seen on the right side of an agent's visual range and later appears from the left side again. From these two positions, the agent would predict a possibly incorrect movement from the right to the left. To prevent such mistakes and for performance reasons, memories of obstacles are deleted after a certain number of frames after their creation.

3.4.3 Reaction

How well the obstacle avoidance works clearly does not only depend on the perception of objects, but also on how the agents react to them. While the goal is to create a collision-free system, reality is different. If the density of people in some area is too high, collisions occur. To simulate the effect of harmless collisions and pushing, we use the *auto collision avoidance* feature of Miarmy. Other characters are simply pushed out of a certain radius around an agent.

Three values of the agent's state are adjusted to actively avoid obstacles: *evadeRotLeft*, *actionId*, and *actionSpeed*. Having the first value affect the direction and the remaining two adjust the speed, maneuvering around an object is easily possible. As mentioned in Section 3.3, checkpoints can be used to improve the path taken by the agent.

The forward speed is set separately from the rotation. By default, if the front up close is not obstructed, the agents are playing action *running*. If the front is

blocked, the *walking* speed is linearly decreased until a minimum distance to the object is reached and the agent halts. This creates time to rotate to the correct direction.

To determine how to evade a certain obstacle, the circular sectors to the front are checked from the inner frustum to the outer one. If either the left or right side from the agent's perspective is free, the character can be rotated accordingly to steer towards this free space. But if both sides are obstructed, the situation is examined in more detail. To compensate for the uncertainty of which parts of the sectors are blocked, the angular extent of the obstacles computed before (in Section 3.4.1) is used to determine the parts of the front, that are not blocked. If there is a gap between the objects blocking the left and the right side, the agent can be guided towards it. If the obstacles are rather small, a slight rotation may be enough to evade them. Yet if the whole front is blocked, it can be smart to create a checkpoint, to lead the agent around the obstruction.

As agents act mostly stateless, it would be easy to create a situation, where they would enter a loop (e.g. by trying to evade to the left, after some correction to the right, then to the left again and so on). A checkpoint smooths the path and is used as short-term memory to remember the chosen direction. Along with the position, the reason (the names of the obstacles) for this checkpoint and the current frame number is stored. No other checkpoint can be created for the same objects, while one is already active. Otherwise, multiple identical ones would be created during the evasion in the following frames, as the situation usually is practically the same. If there is a valid checkpoint for the currently obstructing objects, *evadeRotLeft* is set to 0, so that the agent may freely move towards it.

3.5 Enemies and Flight Behavior

To achieve a high density of people in a crowd or a realistic path around objects, obstacle avoidance can't be too careful when evading. But in cases where a human is actually blocked by a dangerous object, just barely avoiding it would not be realistic. Threatening obstacles need to be evaded more actively.

To this end, there is a global list *ENEMIES*, with the names of objects that are dangerous. In large scale scenes, it would make more sense, that each agent has his own set, but as most of the scenarios are reduced to a small area and at most 100 agents, this simplification works well. While such objects do not have an influence on obstacle avoidance, they alter the direction towards an agent's target. If an agent is within a certain distance to one or multiple of the *ENEMIES*, he will adjust *rotLeft* to move away from the closest one. The chosen direction still depends on the current target. But the closer he is to a dangerous object, the more he deviates from his path to evade it. When an agent is evading an enemy, its state is adjusted, which has an influence on the character's speed.

The list of the dangerous objects can be changed dynamically by adding names or removing entries. In some scenarios, this is used to simulate a collective learning effect regarding hunting agents or kinetic primitives.

3.6 Hunting Behavior

Another addition made to the system is the hunting behavior. A hunting agent will target some other agent closest to him, that is alive and not hunting itself. Once the hunting agent reached his target, the variable *bIsAlive* of its victim is set to *False*. As a result, the agent will search for another character close to it. When the hunter adjusts *bIsAlive* of a victim, its name will also be added to *ENEMIES*, marking this agent as dangerous.

3.7 Individualization

Having a great number of agents with exactly the same behavior is not authentic. In reality, there are multiple differences in character and physical ability between people. To imitate this diversity, male and female agents feature different animations and a small number of character-specific attributes are introduced, which influence aspects of the obstacle avoidance and the movement in general. They are drawn randomly from a normal distribution with the default value 1 as mean when the scene is initiated. The following attributes have been added:

riskFactor: Agents with a higher *riskFactor* are more careful and keep a bigger distance to obstacles and enemies.

speedFactor: Some people are faster than others. The *speedFactor* directly influences the playback speed of the character and therefore the movement speed itself.

stressFactor: Characters that are in a hurry, have a lower *stressFactor*. They take a higher risk by running up closer to other agents and obstacles. The value also has an influence on how fast they are able to change direction.

fearFactor: Set to a higher value, the *fearFactor* increases the distance an agent tries to keep from an enemy. Additionally, the movement and rotation speed is increased when escaping.

memoryFactor: Agents forget checkpoints and obstacles faster if the *memoryFactor* is low. A higher value indicates a better memory.

In addition to the characteristics mentioned above, the values defining the perceptive area for recognizing obstacles are varied similarly. This has the result, that some agents have better eyesight than others.

Chapter 4

Evaluation

To test the system and receive visual feedback, a large number of scenarios were created. Some of them are presented in this chapter to give an impression of the results.

4.1 Obstacle Course

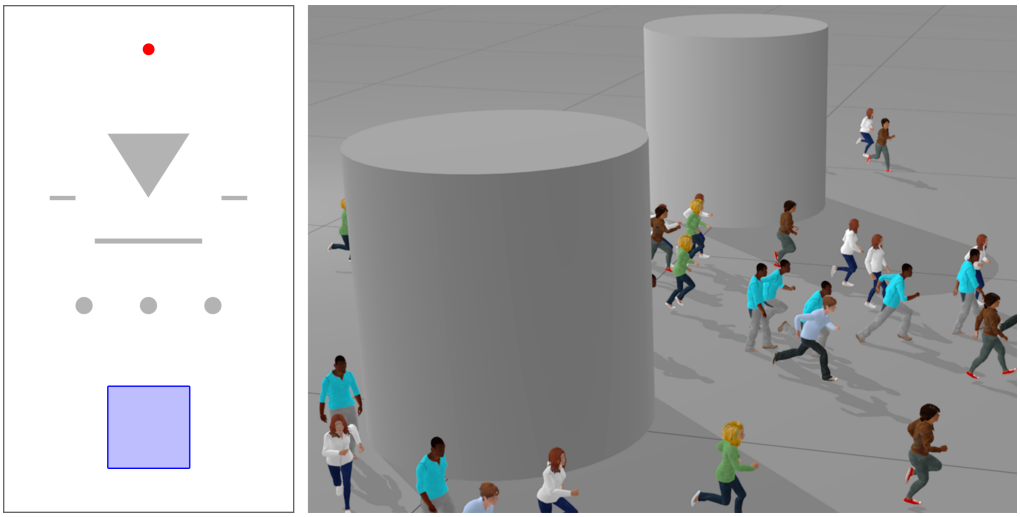

Figure 4.1: On the left: An overview of the obstacles (in grey) between the agents (spawned in the blue area) at the bottom and their target (marked red). A rendered image of the resulting simulation is shown on the right.

One of the most straight-forward scenarios to test the abilities of the mechanics is to let the agents find a target in an environment with multiple obstacles. The goal, of course, is that as many as possible reach their destination. In this scene, 100 agents are placed on one side of multiple obstacles with their target on the other side (Figure 4.1).

In the simulation, the agents first spread to bring some distance between each other. Some of the more risk-averse people then try to walk around the obstacles while others search a path through them. This works reasonably well, even if real humans probably would be more efficient. Only a few situations arise, where people get too close to an obstacle and believe to be blocked. After a short while, most have managed to reach the target. A problem which can arise is that because the agents check the obstacles by their vertices, the models might not be detailed enough. People may not recognize them properly and pass through them. This can be solved by adding more vertices to the obstacles' geometry or by checking the objects by their edges. Such a test would be computationally more demanding, but the meshes could be kept simpler.

4.2 Rotating Bar

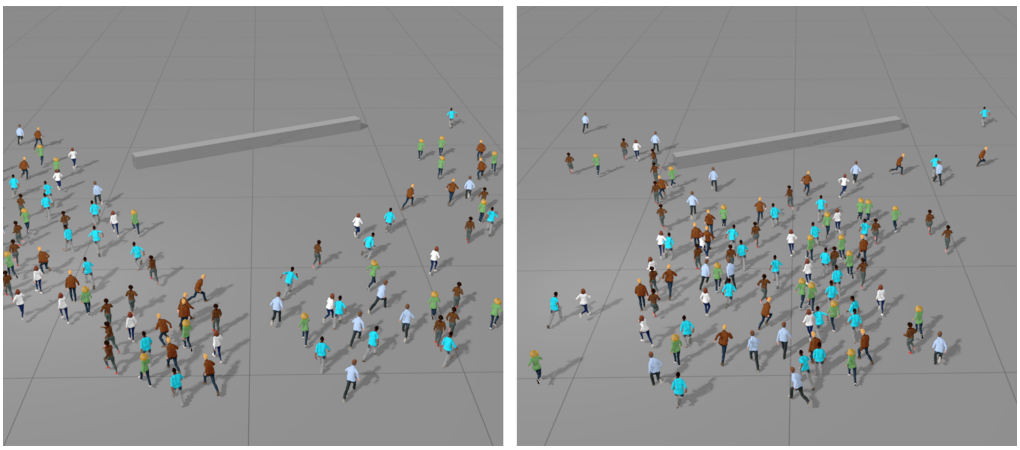

Figure 4.2: Ignoring other agents in the large frustum decreases the number of collisions (shown in the left image), as the rotating bar is not hidden by other agents (image on the right) and can be avoided. Marking the bar as dangerous has a similarly positive effect.

As this thesis is partially inspired by the video *I've fallen, and I can't get up!*, a scene had to include a rotating bar between the agents and their target. In contrast to the original video, the agents now can benefit from the obstacle avoidance and the movement prediction. Theoretically, there should be significantly fewer collisions. In the scene, a kinetic primitive was attached to the bar. On collision with an agent, the characters' body dynamics is enabled.

We tested multiple configurations of this situation. One, where the bar was not assessed as dangerous, even after it had hit someone. Another one, where it was added to *ENEMIES* after the first collision. And finally, with the rotating bar declared dangerous at the start of the simulation.

As expected, the number of people reaching the target increased from the first to the third approach. Even though the rotating motion is fairly difficult to handle (probably even for real humans), a majority of the 100 agents survived the experiment. In the second configuration, there usually were around one or two victims until the characters realized, that the bar was indeed dangerous. When the obstacle was marked as an enemy at the start of the simulation, usually there was no victim at all. Only when one of the agents' *fearFactor* and *riskFactor* were more extreme, the path was sometimes chosen too boldly.

While a single agent evades the rotating bar without any difficulty, detecting only the nearest obstacle per sector is a significant disadvantage for people in a crowd. Usually there are always agent close-by and thus limit the vision. As a result, agents may ignore bigger obstacles such as the rotating bar, until they stand right in front of them. By having them ignore other agents in the frustum and only react to people in close proximity (when detected by the two smaller sectors in front), the rotating bar is detected early enough and all agents can easily evade it.

4.3 Obtrusive Clowns

Figure 4.3: An overview of the scene with the clowns (in yellow) hunting the normal people (in blue) that try to reach the target (marked red). An image from the simulation is shown on the right.

To test the hunting mechanics implemented in the decision logic, we constructed a scene, where a group of the normal male and female agents is chased by a smaller number of clowns, armed with bad jokes. At the beginning, no clown is treated as dangerous, but each clown is added to *ENEMIES*, as soon

as he has reached his first victim. To make the scene more interesting, obstacles were added to the environment and the clowns were given a small speed bonus.

The simulation rapidly becomes chaotic. Depending on the individual attributes of the characters, some of the agents can get away for a long time, while others are caught easily.

In some situations, the hunting clowns are an obstacle to themselves. When two or more close in on a victim, they occasionally slow down because of each other, allowing the victim to escape repeatedly. To address this, some special logic would be needed for obstacle avoidance as a hunter.

4.4 Crossing the Road

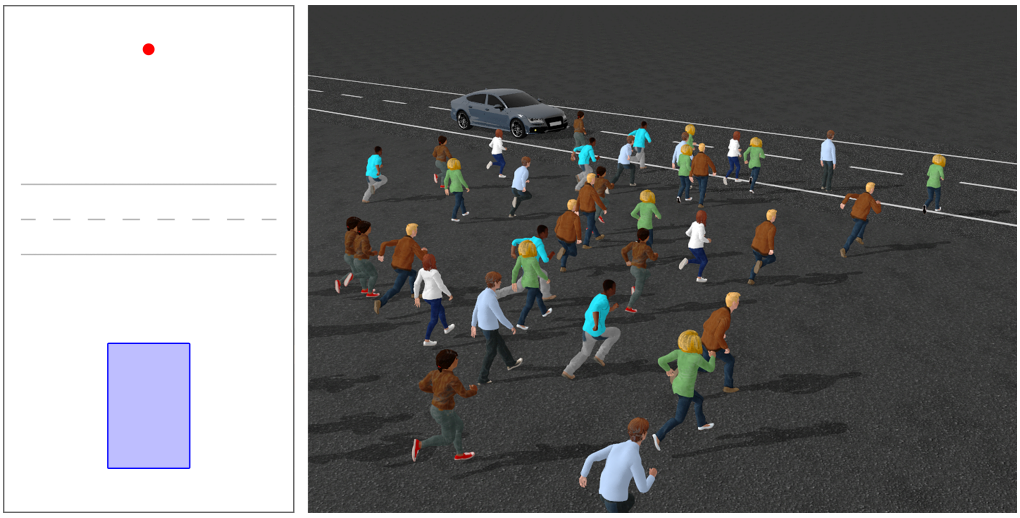

Figure 4.4: As illustrated by the overview on the left, the agents can only reach their target by crossing the road in between and trying to evade the cars on it.

Evading fast moving objects like cars is a difficult task for agents and is therefore an interesting scenario to observe their ability of predicting the movement of obstacles. The agents and their target are set on the opposite sides of a road. Some cars, which are a combination of a high-poly 3d model (from TurboSquid [17]), a low-poly approximation for collision detection, and another simple model for physics simulation are animated to occasionally cross the path of the crowd at rather high speeds.

As expected, most, but not all, agents are able to reach the target unharmed. Many of the accidents are results of the compromises in the decision model. For achieving an efficient evasion around stationary objects and other agents, the distance held to an obstacle can't be too large. But this means that they risk to

collide with a fast car when evading it.

The movement prediction itself works well. In most cases, the agents correctly anticipate the car in front of them and wait. To prevent more accidents, the object prediction could be configured to predict more steps or adjusted by scaling the temporal difference between each step. This can provide a better coverage of the area obstructed later on. Additionally, the agents' frustum could be widened to detect the cars earlier and to give more time to react.

Chapter 5

Conclusion

In this thesis, we have presented decision logic for Miarmy agents that simulates human behavior and features complex aspects, such as pathfinding with obstacle avoidance. The standard decision nodes of Miarmy have only been used as a link between the logic programmed in Python and the agent's visual representation. As all the information from the Python scripts has to be gathered individually by those nodes, there is some impact on the performance. But by keeping their number small, the effect can be reduced. This approach works surprisingly well and allows extending Miarmy with practically all possibilities Python has to offer. With regard to the visual aspect, as well as the physical simulation, this combination is a convenient and powerful tool.

Unfortunately, some of the bugs in Miarmy are problematic. In some cases, the outputs of decision nodes were not executed, calling a Python script from them usually didn't work and the *direct expression mode* has lead to crashes in some cases. Even though there is a workaround for most of the problems, the solutions are not always ideal. But in the end, the system works.

The decision logic itself is a working approximation of realistic human behavior. There are limitations from the choice of some procedures and the simplification of some aspects. The restriction on six circular sectors for perception and only reacting on the closest obstacle in each of them does not always allow for an ideal decision. How an agent reacts to obstacles could also be improved by including a more detailed assessment of the situation the character is in. Additionally, the recognition of obstacles by checking their vertices is a compromise with the limitation, that at some point close to the object, the resolution never is high enough.

But as demonstrated in multiple videos with different scenarios, the result is convincing. It is a working example, that intelligent behavior with a more complex logic can be implemented in Miarmy. With enough time at hand and maybe with a better documentation regarding the expandability of the software, much better results would surely be possible. An intriguing idea for further extending the system is to use neural networks as decision logic on the values received by the obstacles detection and with the direction and speed of the

agent as output. One would need to create a large amount of training data describing the ideal reaction in numerous situations. But this could probably be generated automatically from an ideal path placed in a 3d scene by using the agent's perception functions. There are even multiple Python libraries offering a simple interface for creating and using neural networks. Unfortunately, due to the limited time, this idea could not be explored.

Bibliography

[1] A. Kneidl, M. Thiemann, D. Hartmann, and A. Borrmann. *Combining pedestrian simulation with a network flow optimization to support security staff in handling an evacuation of a soccer stadium.* In: Proceedings of 23rd European Conference Forum Bauinformatik. 2011.

[2] S. Curtis, S. J. Guy, B. Zafar, and D. Manocha. *Virtual Tawaf: A case study in simulating the behavior of dense, heterogeneous crowds.* In: 2011 IEEE International Conference on Computer Vision Workshops. 2011.

[3] Massive Software. *Massive.* 2016. URL: http://www.massivesoftware.com/.

[4] The Lord of the Rings: The Fellowship of the Ring, Dir: Peter Jackson, NZ/US, 2001, New Line Cinema.

[5] Basefount Technology HK Ltd. *Miarmy.* Version 4.7. URL: http://www.basefount.com/miarmy.html.

[6] Autodesk Inc. *Maya.* 2016. URL: http://www.autodesk.com/products/maya/overview.

[7] Dave Fothergill vfx. *I've fallen, and I can't get up!* Vimeo. URL: https://vimeo.com/109169719.

[8] Michael Weigelt, Bachelor Thesis, ETH Zürich, 2016.

[9] Suiping Zhou, Dan Chen, Wentong Cai, Linbo Luo, Malcolm Yoke Hean Low, Feng Tian, Victor Su-Han Tay, Darren Wee Sze Ong, and Benjamin D. Hamilton. *Crowd Modeling and Simulation Technologies.* In: ACM Transactions on Modeling and Computer Simulation. 2010.

[10] Kiran Ijaz, Shaleeza Sohail, and Sonia Hashish. *A Survey of Latest Approaches for Crowd Simulation and Modeling Using Hybrid Techniques.* In: Proceedings of the 2015 17th UKSIM-AMSS International Conference on Modelling and Simulation. 2015.

[11] Adriana Braun, Bardo E. J. Bodmann, and Soraia R. Musse. *Simulating Virtual Crowds in Emergency Situations.* In: Proceedings of the ACM Symposium on Virtual Reality Software and Technology. 2005.

[12] Dirk Helbing, Illés Farkas, and Tamás Vicsek. *Simulating dynamical features of escape panic.* In: Nature. 2000.

[13] Marjan van den Akker, Roland Geraerts, Han Hoogeveen, and Corien Prins. *Path Planning for Groups Using Column Generation*. In: Motion in Games: Third International Conference, MIG 2010, Utrecht. Proceedings. 2010.

[14] Solid Angle S. L. *Arnold*. 2016. URL: https://www.solidangle.com/arnold/.

[15] Adobe Systems Incorporated. *Fuse CC (Preview)*. 2016. URL: https://www.adobe.com/products/fuse.html.

[16] Adobe Systems Incorporated. *Mixamo*. 2016. URL: https://www.mixamo.com/.

[17] quads_tris_and_ngons. *Audi S7 Sportback*. TurboSquid. URL: http://www.turbosquid.com/3d-models/audi-s7-max-free/857221.