



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

*Distributed  
Computing*



# Crowd Simulation

Michael Weigelt

`weigeltm.student@ethz.ch`

Distributed Computing Group  
Computer Engineering and Networks Laboratory  
ETH Zürich

## **Supervisors:**

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

January 18, 2016

# Acknowledgements

I would like to thank my supervisors Klaus and Michael for the valuable feedback and the regular discussions, and Prof. Wattenhofer for the inspiration to work with Miarmy. I am also very grateful to Yeah Yang, one of the lead developers of Miarmy, who kindly answered many questions and resolved issues with the software.

# Abstract

Crowd simulations are becoming increasingly important for public authorities and researchers. The film and video game industry have been developing and using tools for crowd simulations for decades. The Maya plugin Miarmy is such a tool, but it was not designed specifically for scientific purposes. We perform experiments in Miarmy to test its functionality. Simple scenes can be set up, simulated and rendered quickly and efficiently with Miarmy. However, with increasingly complex scenes, the Miarmy interface becomes less useful, and performs suboptimal on tasks it was not explicitly designed for. We overcome some of Miarmy's limitations by exploiting its Python script interface. Although much flexibility can be gained this way and the tool is still being developed, it is unlikely that Miarmy will be useful for exact sciences, where dedicated custom simulations can be used instead. However, Miarmy is very useful to create and visualize diverse crowds, as it offers interfaces to various renderers and makes randomizing agent properties easy. Finally, we simulate and render two scenes using the methods we developed in the experiments: A huge ball crashing into a train station hall and a simple imitation of a soccer game.

# Contents

|  |           |
|--|-----------|
| <b>Acknowledgements</b>                  | <b>i</b>  |
| <b>Abstract</b>                          | <b>ii</b> |
| <b>1 Introduction</b>                    | <b>1</b>  |
| 1.1 Motivation . . . . .                 | 1         |
| 1.2 Related Work . . . . .               | 1         |
| <b>2 Miarmy and Maya</b>                 | <b>3</b>  |
| 2.1 Agents . . . . .                     | 4         |
| 2.2 Perception Objects . . . . .         | 4         |
| 2.3 The Decision System . . . . .        | 5         |
| <b>3 Experiments</b>                     | <b>8</b>  |
| 3.1 Basic Tests . . . . .                | 8         |
| 3.1.1 Agent Avoidance . . . . .          | 8         |
| 3.1.2 Road Following . . . . .           | 10        |
| 3.1.3 Dynamics and Densities . . . . .   | 11        |
| 3.2 Reproductions . . . . .              | 14        |
| 3.2.1 Flowfield . . . . .                | 14        |
| 3.2.2 Rotating Bar . . . . .             | 15        |
| 3.3 Further Scenes . . . . .             | 16        |
| 3.3.1 Target Spots . . . . .             | 16        |
| 3.3.2 Avoid Ball . . . . .               | 17        |
| 3.3.3 Train Station Hall Video . . . . . | 19        |
| 3.3.4 Soccer Video . . . . .             | 19        |
| <b>4 Discussion</b>                      | <b>22</b> |

CONTENTS

iv

**Bibliography**

**25**

# Introduction

---

## 1.1 Motivation

In many cities of the world the human population grows quickly. Urban environments are relatively slow to adapt to more people, so population densities increase. This leads to a host of problems, for instance for public and private transport. With ever more people on the streets it has become a matter of security to understand their collective behaviour. Crowd simulations model the movement of large numbers of agents as realistically as possible. Applications and goals of crowd simulations include the prediction of mass-hysterias, the planning of high pedestrian traffic areas such as train stations or airports, and the simulation of huge armies and crowds in films and video games. One piece of software that was designed for the latter is Miarmy. The primary question of this thesis is: Is Miarmy also suitable for answering scientific questions about crowds?

## 1.2 Related Work

Many problems related to crowd simulation and path-finding need to be solved in various video games. The methods these games implement are rarely perfect because in real-time games, resources are scarce, so quick and cheap solutions are required. But cheap solutions can fail when many agents are involved, for example when many agents need to cross a bridge and a group of agents already on the bridge blocks oncoming agents. In some cases, even increasing computing power does not help: Akker et al.[1] show that finding optimal solutions to multi-agent path-finding is NP-hard, and present a heuristic which does not suffer from that problem.

Another approach to realistic crowd movement and pathing is to use dynamic potential fields and treat agents and obstacles as particles obeying physical laws. Treuille et al.[2] describe such a model which is efficient enough to run in real-time. Even though individual agent movement and planning is not controlled,

some phenomena emerge automatically, so that for example collision avoidance does not have to be explicitly handled.

One typical problem that arises with large numbers of agents is when two groups approach each other and need to find a way around or through the opposite group. The game Supreme Commander 2 has an interesting approach to this problem [3]. It involves a so called Flowfield, which splits and aligns agent groups in an efficient way so that they can pass through each other with very little resistance. This solution is replicated in section [3.2.1](#).

# Miarmy and Maya

---

Maya is a piece of software by Autodesk used for the modeling, animation, simulation and rendering of 3D scenes. Miarmy is a plug-in for Maya, which is still being developed actively by Basefount. Miarmy makes it possible to simulate thousands of agents at a time, each interacting with its environment and behaving according to rules given by the user. Because Miarmy is a plug-in, agents can be created, animated, simulated and rendered all in the same environment.

Concerning agents, Miarmy offers some finished solutions, such as walking human agents, horse riders, ancient fighters and a type of agent used to fill up stadia. These solutions include geometric models, textures, animations and decisions, which are Miarmy's way to control agent behaviour.

The creation of a Miarmy simulation roughly follows these steps:

1. Import or create of one or several agent types
2. Set the scene. This includes non-interactive environment as well as objects which can be perceived by the Miarmy agents, such as bounds or paths
3. Place the agents in the scene
4. Run test simulations and refine the scene and the agents
5. Cache the final simulation
6. Render the scene

Since the realistic modelling and animation of agents require a large amount of time and expertise, Miarmy's *Walker Agents* are used throughout this project.



## 2.1 Agents

Figure 2.1 shows the agent node (Agent\_woman). It consists of several child nodes, most notably the original agent, the geometry and the decisions. The original agent node is where the bone structure and the bounding box of the agent model are located. They are needed for the Miarmy simulation. The geometry node contains the body and cloth geometry of the agent. These are needed when the scene is rendered. Under the decision node all decisions for this agent type are listed. Decisions with lower priority are grouped together under an empty parent decision node. All decisions are checked in each frame of the simulation. They determine the behaviour of each agent of this type. More about this in section 2.3.

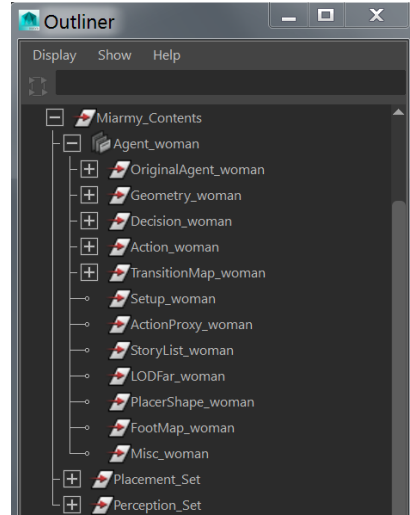


Figure 2.1: Agent nodes

The agent node has several parameters, e.g. *TranslateX*, *RotateY*, *Sphere Range*, *Scale Min/Max*, *HP Min/Max*, *MP Min/Max*. With these parameters, some general properties of the agent type can be set. Sphere Range is the maximum distance an agent “feels” or “hears” other agents. It is used in decision sentences, for example to determine whether another agent is within the sensory range of an agent. If Scale Min and Max are not identical, the individual agents’ sizes will be randomly distributed within the given range. HP and MP stand for Hit Points and Mana Points. Having no predefined purpose, these are the only parameters that can be used freely: They can be checked and changed during the simulation. If the agents should start out with randomly distributed HP or MP, the maximum and minimum of each can be set in the agent parameters.

## 2.2 Perception Objects

Miarmy provides objects which agents can detect: spots, bounds, roads and paths and also solid objects called *kinematic shapes*. All these objects can be assigned a unique ID, in case several of the same kind are needed in one scene. They are interacted with by logic sentences. For example, if an agent needs to act based on its distance to a spot with ID 2, a sentence like `my distance from spot(2) > 50` is used.

**Bounds** are cuboid shapes which simply allow to check whether an agent is inside or outside their borders.

**Roads** are two dimensional paths which are created from smooth curves. Agents can check whether they are on the road as well as the angle between their own direction and that of the road. The Miarmy road has some important parameters: Road width, ID, *flow edge* and *flow orientation*. The first two parameters are self explanatory, the other two determine the behaviour of agents in the edge regions of the road. The parameter *flow edge* is the percentage of the road that is counted as edge region. When an agent walks into the edge region, it will automatically steer inwards, with an angle determined by the *flow orientation* parameter.

**Paths** are extensions of Roads into the third dimension. An example of a sentence referring to a path is `I am in path(0) and it points right`. A typical decision based on this sentence would make the agent rotate to the right, so that it stays on the path.

**Kinematic Shapes** are objects agents can collide with. Usually, a collided agent will have `enable dynamics` as an output sentence, which stops its current animation and posture and enables rag-doll physics.

Perception objects by themselves do nothing. They need to be used in conjunction with the appropriate decisions, which are explained in the next section.

## 2.3 The Decision System

The behaviour of each agent type is controlled by decisions. A decision consists of at least one input sentence and at least one output sentence. In each step of the simulation, the input sentences of every decision are checked. If all inputs of a given decision are true, then all outputs are activated. For example, the following decision checks if there is another agent within the agent's sphere range, and if so, the agent slows down whatever animation it is currently playing.

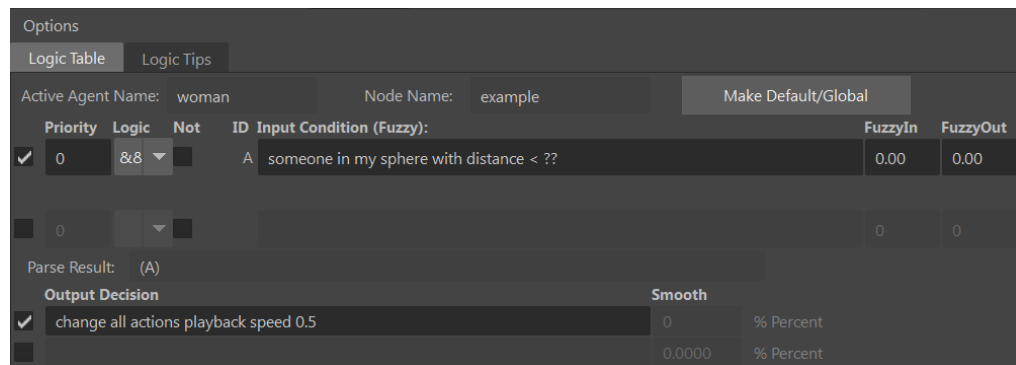


Figure 2.2: Example Decision

Miarmy provides a reasonably large set of predefined input and output sentences, which are formulated in Miarmy’s “human language form” (i.e. questionable English). Input sentences can be concatenated with the logic operators *and*, *or*, *not*, but bracket terms are not possible.

Some frequently used sentences are listed here:

- `someone in my sphere with distance < ??`
- `someone in my sphere with angle from ?? to ??`
- `someone in my sphere with relative speedX < ??`
- `I’m on road[??] and it point to LEFT`
- `I’m in path[??] and it point to UP`
- `my velocity of translation in X > ??`
- `I’m in bound[??]`
- `spot[??] is on RIGHT`
- `my master is on LEFT`
- `maya python:exampleScript() return value > ??`
- `marked bone collideBy: KINEPRIM_`
- `my HP < ??`

Three of these sentences are not self-explanatory: The `master` sentence can only be used after a master-slave relationship between agents has been established. To this end, a Python script has to be run after agent placement, which links agents to each other. The script can be run from the Maya console, and it looks something like this:

```
cmds.connectAttr("McdAgent23.masterOf", "McdAgent22.masterOf",
force=True)
```

This makes agent number 23 the master of agent number 22<sup>1</sup>. A master can have several slaves, but a slave only has one master. It is possible for an agent to be master and slave, as long as the relationship graph has no cycles. If it does, Miarmy crashes.

For the `python:exampleScript()` sentence, the user can write a custom Python script which must return a numeric value. The return value can be compared to a fixed reference value (for which `??` is a placeholder).

---

<sup>1</sup>There is no copy-paste error in this command: Both agents’ “masterOf” attributes are connected.

`_KINEPRIM_` is a kinematic shape as described in section 2.2. The above sentence is true when an agent's bone which has been marked for this purpose collides with a kinematic shape.

Some output sentences are:

- `move FORWARD as speed ??`
- `rotate to LEFT as speed ??`
- `play action:??`
- `change all actions playback speed ??`
- `set hp value ??`
- `change hp as speed ??`
- `enable dynamics`
- `follow my master`

The `play action` sentence enables one of the agent's animations, like a walk, run or sit animation.

Whenever `change x as speed y` is in a sentence, `x` is changed by `y` per second, not per frame. Only fixed values can be used, so it is impossible to make the rate of change `y` dependent on the value `x`, or on some other value.

The `enable dynamics` sentence stops the agent's current animation and enables rag-doll physics. This means that the agent loses all control over its posture and falls to the ground. Its body becomes subject to the physics engine. In the current version of Miarmy, this is not reversible.

If two decisions with conflicting outputs are active, the output intensities are added. For example, if one decision leads to a clockwise rotation with speed 50 and the second decision to a anti-clockwise rotation with seed 20, the agent will rotate clockwise with speed 30. To lessen this kind of cancellation, there is a fuzzy-logic system built into the decisions. If it is used, the intensity of a decision output will depend on the intensity of the input, i.e. the rotation speed depends on "how true" the input is

# Experiments

---

In order to test Miarmy’s engine and interface, we conduct a series of experiments. We describe the general idea of each scene, go into the details of the implementation where it is interesting, summarize the results and compare them with our expectations. The scenes are very simple at the beginning and get progressively more complex.

## 3.1 Basic Tests

### 3.1.1 Agent Avoidance

This first experiment tests the Miarmy *Walker Agents* decision set which is supposed to prevent agents from colliding or overlapping. Three decisions are involved:

- `avoidSphereLeftTurnRight`
- `avoidSphereRightTurnLeft`
- `avoidSphereFrontSlowDown`

Like the names suggest, two of these decisions are symmetric. The first and third decisions look like in figures [3.1](#) and [3.2](#).

As a very simple first test, 30 agents are placed in a loose crowd, each agent facing in a random direction. Their only objective is to walk forward and avoid other agents. Figure [3.3](#) depicts the situation after a few frames.

Between any two agents, one of three things can happen: Either they never meet and neither is affected by the other. Or they can walk towards each other and intrude into the other’s sphere range. In that case, they both slow down to half their speed and rotate away from each other. If they are facing each other directly, i.e. within 35 degrees to the left or right, they slow down to 20

The screenshot shows the Logic Table configuration for the decision `avoidSphereRtTurnLf`. The active agent is `fatWoman`. The table has columns for Priority, Logic, Not, ID, Input Condition (Fuzzy), FuzzyIn, and FuzzyOut. Two conditions are defined: A (someone in my sphere with angle from -90 to 0) and B (someone in my sphere with distance < 20). The parse result is (A & B). The output decision is 'rotate to LEFT as speed 120' with a smooth value of 0. A second output sentence 'change all actions playback speed 0.5' is also present.

| Priority | Logic | Not | ID | Input Condition (Fuzzy):                      | FuzzyIn | FuzzyOut |
|----------|-------|-----|----|---|---------|----------|
| 0        | &&    |     | A  | someone in my sphere with angle from -90 to 0 | 30.00   | 30.00    |
| 0        | &&    |     | B  | someone in my sphere with distance < 20       |         | 5.00     |
| 0        |       |     |    |   | 0       | 0        |

Parse Result: (A & B)

| Output Decision                       | Smooth      |
|---------------------------------------|-------------|
| rotate to LEFT as speed 120           | 0 % Percent |
| change all actions playback speed 0.5 | 0 % Percent |

Figure 3.1: Decision `avoidSphereLeftTurnRight`

The screenshot shows the Logic Table configuration for the decision `avoidSphereFrtSlowA`. The active agent is `fatWoman`. The table has columns for Priority, Logic, Not, ID, Input Condition (Fuzzy), FuzzyIn, and FuzzyOut. Two conditions are defined: A (someone in my sphere with angle from -35 to 35) and B (someone in my sphere with distance < 20). The parse result is (A & B). The output decision is 'change all actions playback speed 0.2' with a smooth value of 0.

| Priority | Logic | Not | ID | Input Condition (Fuzzy):                       | FuzzyIn | FuzzyOut |
|----------|-------|-----|----|--|---------|----------|
| 0        | &&    |     | A  | someone in my sphere with angle from -35 to 35 | 10.00   | 10.00    |
| 0        | &&    |     | B  | someone in my sphere with distance < 20        |         | 5.00     |
| 0        |       |     |    |  | 0       | 0        |

Parse Result: (A & B)

| Output Decision                       | Smooth      |
|---------------------------------------|-------------|
| change all actions playback speed 0.2 | 0 % Percent |

Figure 3.2: Decision `avoidSphereFrontSlowDown`

percent of their original speed. There is also a very small chance that the agents' directions are exactly anti-parallel, in which case the rotation speeds for the left and right side are equal. Because placement nodes can and may be set up exactly anti-parallel (and thus agents might be aligned perfectly), that is a problem.

The first results of this simulation showed some agents passing through each other, therefore the `FrontSlowDown`-decision was slightly adapted. A second output sentence was added to it, similarly to the left/right decisions: `rotate to left as speed 60`. This prevents ties and improves agent avoidance.

It is noteworthy that in the Miarmy GUI, there is a preset decision set available which is called "Avoid Collide by Sphere" consisting of three decisions with the same names as the ones described above. However, these preset decisions do not slow down the agents upon close contact with another agent, they only rotate them. Tests with both variants showed that unless agents are running,

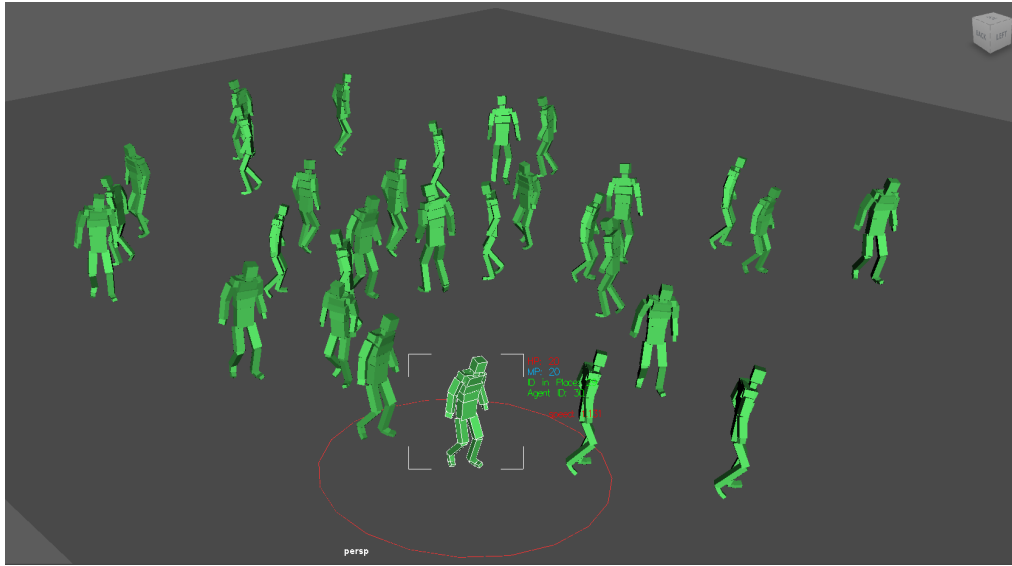


Figure 3.3: Randomly placed agents

the preset “Avoid Collide by Sphere” is sufficient to avoid collisions, at least in this simple scene.

### 3.1.2 Road Following

In the previous experiment, agents do not have a destination. They walk straight ahead and if their direction is changed due to an obstacle, they do not correct it after passing the obstacle. Because this is not realistic behaviour, we now give the agents a constraint. They are following a road in a given direction and they may not leave the road. Miarmy offers a symmetric decision preset for this scenario, shown in figure 3.5.

The agent avoidance decisions from the last experiment are included as well, preventing agents from walking through each other. This leads to a problem: If an agent is at the left border of a road, its road-following decisions tell him to rotate right. If there is another agent very close to its right, its agent avoidance decision tells him to rotate left. If the rotation speed of the latter decision is greater, the agent leaves the road and all road following decisions become inactive, making it impossible for him to find the road again.

To try to prevent this, three methods are tested: Firstly, the *road edge flow* parameter is set so that the edge flow points inwards. The edge flow decision preset was imported, with input sentence *I’m on flow and it points to left*. Agents should steer inwards as soon as they enter the edge zones. Unfortunately, these decisions offered by Miarmy do not seem to work properly. Even though the agents are clearly inside the edge zone of the road and the decisions are for-

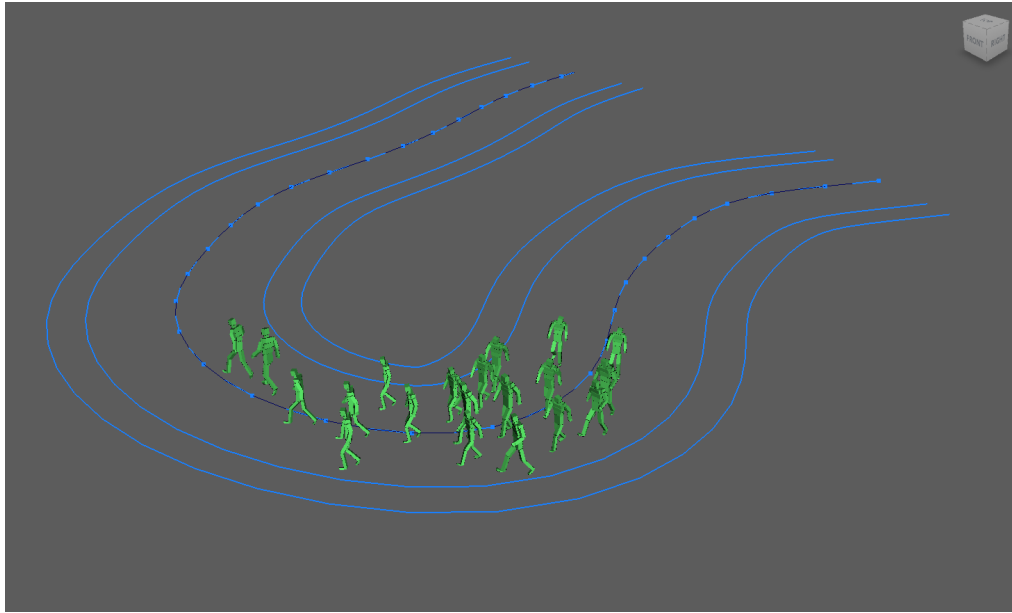


Figure 3.4: A Miarmy Road

mulated by the Miarmy developers, they do not activate. This might be a bug or incomplete feature, considering that Miarmy is being developed and frequent bug fixes are published.

Secondly, a lesser priority is assigned to the agent avoidance decisions. This should correctly resolve conflicts, but the simulation shows that this measure does not work properly either. Prioritization has worked in other experiments though, so it is unclear why this measure is insufficient here.

The last option is to make sure that the rotation speed of the higher priority decision is always greater. Tests from several simulations with all three methods considered, this is the most reliable option. It is also the least favourable option, because it limits the user's possibilities. A situation might arise where the user needs the rotation speeds to be in the reverse order.

### 3.1.3 Dynamics and Densities

In this experiment, the agents' ability to avoid each other is tested, if the available space becomes more and more sparse. To achieve this, 50 agents walk along a path, equipped with the decision set from the previous experiment: Agent Avoidance and Road Following. The Miarmy Road is replaced with a 3D Path, allowing for a varying path width. The start of the path is wide enough for the group of agents to walk in a loose cluster, but it narrows down to the width of only a few agents, before it opens up again.



Options

Logic Table Logic Tips

Active Agent Name: woman Node Name: followRoadRtTurnRt

| Priority                              | Logic | Not                      | ID | Input Condition (Fuzzy):          |
|---------------------------------------|-------|--------------------------|----|-----------------------------------|
| <input checked="" type="checkbox"/> 0 | &8    | <input type="checkbox"/> | A  | I'm on road and it point to RIGHT |
| <input type="checkbox"/> 0            |       | <input type="checkbox"/> |    |                                   |

Parse Result: (A)

Output Decision Smooth

☒ rotate to RIGHT as speed 120 0

Figure 3.5: Road Follow decision preset

In order to have a failure state, a new decision output is introduced: **enable dynamics**. The corresponding decision becomes active if an agent trespasses the critical sphere of another agent. The critical sphere's radius is chosen as half of an agent's bounding box. As soon as an agent's dynamics are activated, it falls to the ground like a rag-doll and stays there.

If the experiment is run like described, almost all agents fall down shortly after the path narrows. The decisions don't allow the agents to react quickly enough to the changing environment. If an agent is surrounded by other agents who threaten to come too close, it has no appropriate reaction: Neither rotating left nor right will give it more space. The only reaction that helps is slowing down. However, this is only useful if the agents behind can slow down quickly enough to avoid a collision. In the described experiment, this is evidently not the case.

To improve the results of this simulation, the Miarmy Global setting "Auto Collision Avoidance" is activated, which moves agents apart automatically if their bounding boxes overlap. Although this makes a big difference (collisions now occur only in the narrowest region), it looks unrealistic if the crowd is very dense. Due to this setting, bounding boxes cannot overlap anymore, thus the agents act almost like particles, repelling each other perfectly elastically. The speed at which agents repel each other can be changed, and simulations show that the slower this speed is, the better it looks. Unfortunately, the minimal speed still is too high for a realistically looking simulation. But it is preferable to dozens of agents colliding just because they walk through a narrow path.

It would be desirable to make agents avoid fallen bodies on the ground because walking over a fallen body activates the collision decision of the walking agent, so it falls, too. Unfortunately, an agent whose dynamics have been en-

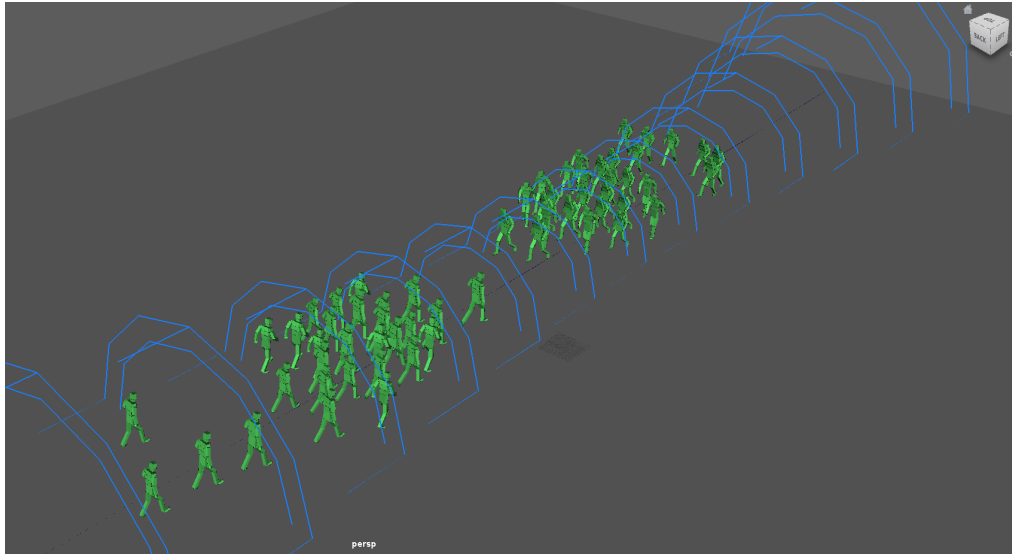


Figure 3.6: A 3D Miarmy Path

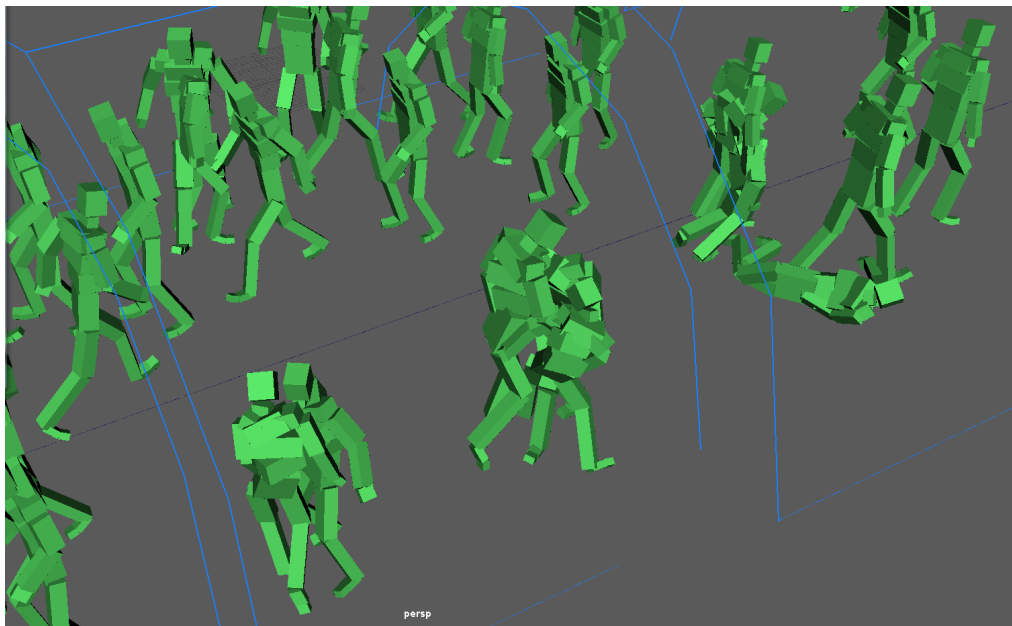


Figure 3.7: Collided agents with active dynamics. Note: In this picture nearby agents were rendered invisible to better display the colliding agents. If the agent density were as low as in the picture, these agents would probably not have collided.

abled is no longer considered an agent by the Miarmy engine. The developers have confirmed that there is no way around this in the current version of Mi-



Figure 3.8: Original Flowfield

army. It seems therefore impossible to prevent a pileup of agents once an agent has fallen in the path of many others.

## 3.2 Reproductions

These scenes or motives we encountered while researching crowd simulation. The first example is a feature of a video game, the second is a reproduction of a video showing off Miarmy’s capabilities.

### 3.2.1 Flowfield

Two groups of agents walk towards each other and have to pass each other in a limited space. Every agent from one group should end up where the other group starts and vice versa. In Supreme Commander 2 [3], this problem is solved with “Flowfields” The goal of this experiment is to replicate this behaviour.

As a failure state, dynamics are used, i.e. agents fall to the ground if they come into too close contact with another agent. The Agent Avoidance decision set is also used, together with the Miarmy global setting Auto Collision Avoidance. To confine the agents in space, both groups follow a Miarmy Path in opposite directions.

When the simulation is performed with walking agents, the results are similar to the ones in the original. The front row of agents of the first group meet the front row of the second group and try to avoid them. The front widens, because the agents at the edges can rotate freely away from the crowd - as long as they are within the confining path. If the path is too narrow to allow the groups to spread, some collisions do occur. The first row of agents finds a path through the oncoming crowd, and with each subsequent agent this path becomes more clear and straight. The result looks like expected, shown in figure 3.9.

When the agents are set to run instead of walk at the beginning of the simulation, Agent Avoidance fails. Several agents collide and fall to the ground, obstructing the path of the subsequent agents, who have no way to detect the fallen agents and fall themselves. Thus, if the experiment fails, it fails catastrophically.

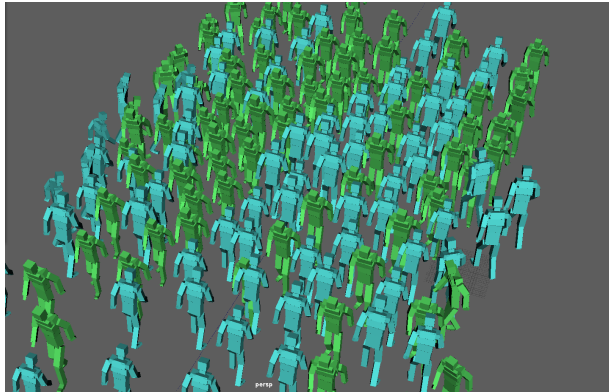


Figure 3.9: Flowfield Replication

Why can the agents not evade oncoming agents at running speed? One reason is that for higher movement speeds the rotation speed in the avoidance decisions needs to be higher. But even tests with increased rotation speeds failed. It is likely that the time-resolution, i.e. the time difference between two ticks of the Miarmy simulation, is not high enough for this scenario. The agents can not react quickly enough to a changed environment. They come too close to each other before they even have a chance to avoid each other. Although the user has a little control over the time resolution in Miarmy's global options, even the highest setting does not change the above results.

### 3.2.2 Rotating Bar

In a popular video [4] created with Miarmy, some agents approach a big rotating bar. When an agent is hit by the bar, it is pushed around, eventually falls over the bar and cannot get up again. Here, this scene is reproduced.

Apart from the usual agents, the setup involves a rotating bar. Miarmy provides basic objects that can interact with the agents, called kinematic shapes. In the current version of Miarmy, they can either be rectangular boxes or spherical. In order to make them rotate during the simulation, Maya's *keyframe* system is used. With this system, only the desired position, rotation and size of the bar at different points in time have to be fixed. These points are called keyframes, and the Maya engine generates the movement between the keyframes automatically. Here, only the rotation of the bar around the y-axis has to be keyframed. The duration of the simulation is about 300 frames, and the bar rotates 90 degrees every 32 frames. This makes for a reasonable rotation speed relative to the agents' walking speed.

In the original video clip, the agents play a custom animation upon impact with the bar. To make such an animation look good and to provide it with enough variation for a realistic crowd simulation would exceed the scope of this

experiment. Therefore, the output sentence `enable dynamics` is used to make the agents fall down after being hit by the bar.

The result differs from the original only in the lack of movement of fallen agents. As an extension of the original scene, it would be interesting to let agents stand up again after being hit, and continue walking or start fleeing from the threatening bar. Unfortunately, agents lose their status as agents once their dynamics are activated. All decisions are disabled and the agent's body remains only as a physical object, similar to the kinematic shapes. Since decisions are the only interface to the agents' behaviour, the Miarmy user loses all control over the agent, once its dynamics are activated.

There is a decision output `disable dynamics`, which at the moment seems unusable because once the dynamics are activated, no decision can become true anymore. Perhaps this means that the developers are planning to change this.

### 3.3 Further Scenes

#### 3.3.1 Target Spots

In a public transport area like an airport or a train station hall, there are many agents with many different targets. This experiment simulates such a scene. A number of agents and several spots are placed randomly within a given radius. Each agent is assigned a spot as an initial target at random. The agents seek their target and are assigned a new one once they have reached it.

The key feature of this experiment is that agents need a state that determines which spot is their current target. To this end, the parameter HP is used. The agents are initialized with HP values ranging from 0 to 100. Agents with HP between 0 and 10 seek spot 1, those with HP from 11 to 20 seek spot 2 and so on. The current HP value can be read and set by a decision sentence. HP and MP are the only free variables available for such a purpose.

The first decision consists of these sentences:

**Input 1:** `spot(0) is on LEFT`

**Input 2:** `my HP from 0 to 10`

**Output:** `rotate to LEFT as speed 180`

The second decision is a mirrored version of the first one. The third corresponds to a state change:

**Input 1:** `spot(0) to me distance < 8`

**Input 2:** my HP from 0 to 10

**Output:** set hp value 11

Once this agent's HP values is set to 11, these above decisions are no longer active, but an almost identical set of decisions becomes active, with adapted spot ID and HP values. Consequently, three decisions have to be created for each spot in the scene.

Normally, decisions are created through the Miarmy GUI, requiring a lot of mouse clicks and context menus. Because this is too much work if a scene with 20 or 50 spots is required, we reverse-engineered some of the Miarmy GUI, which is accessible as a collection of Python scripts in the Miarmy directory. With a combination of Maya and Miarmy commands, we put together a script which creates an arbitrary number of spots and distributes them within an adjustable perimeter. For each spot, the three decisions above are automatically generated, with almost no GUI call (footnote: for some reason, a GUI update function needs to be called after each created decision, but the user is freed from having to give any input).

As a result, it is no longer necessary to create dozens of decisions by hand. Instead, the script can be run, the agents placed and the scene is ready to be simulated.

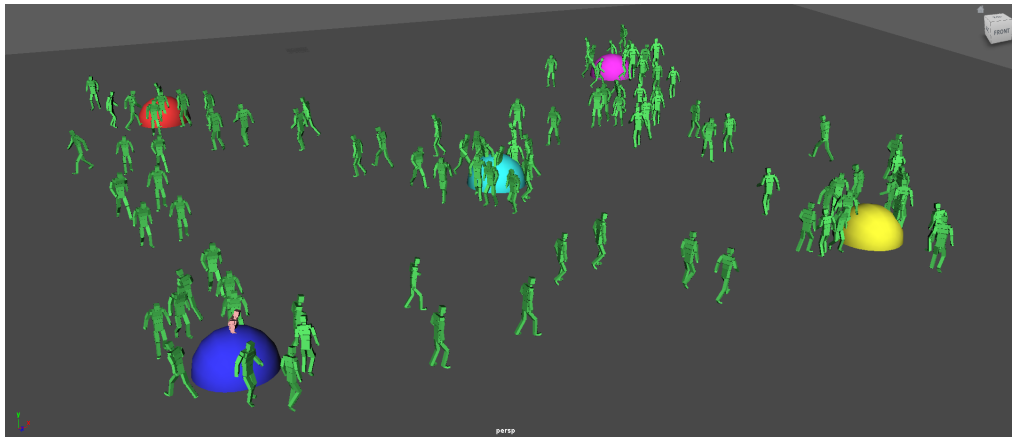


Figure 3.10: Agents seeking spots

### 3.3.2 Avoid Ball

In this experiment, agents are confined to a rectangular space. They walk around in random directions, avoiding each other and trying to dodge a kinematic ball which is jumping around. If the ball hits an agent, its dynamics are activated and the agent falls.

The kinematic shapes cannot be sensed by the agents directly. There is no decision input sentence that checks for a nearby kinematic shape. Because agents need to speed up and run away when the ball approaches them, a spot is placed at the center of the ball and moved together with it at all times. This way, agents can just respond to proximity of the spot and the results should be the same.

There is no out of the box way to move a kinematic shape or a spot in Miarmy. Shapes are not agents, so decisions cannot be used, and the simulation process is inaccessible, so it's not possible to simply update the shape's position in each frame. So in order to move a kinematic shape, it needs to be keyframed. To keyframe means to pick points in time and fix the shape's desired position at that time. The Maya engine will automatically generate the position for all the frames in between. It is almost impossible to get the ball to behave in a physically realistic manner this way, but an imitation of "jumping around" was created by keyframing the ball not only on different positions on the plane, but by varying its vertical coordinate as well.

Keyframing a ball and a spot at random for a simulation that should run for a minute or more is another tedious job that is better done by a script. The commands used are from the `Maya.cmds` package, i.e. `cmds.setAttr()` and `cmds.setKeyframe()`.

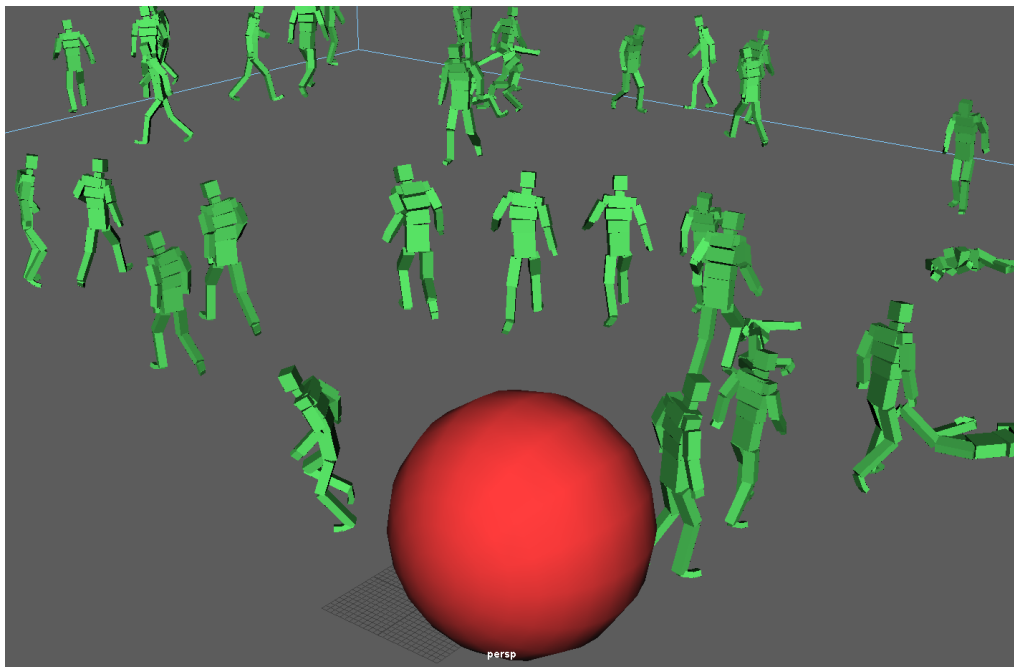


Figure 3.11: Agents dodging ball

Except for the unrealistic movement of the ball, the results of this experiment look good. The agents start running as soon as the ball comes too close, and their

flight pattern looks realistic. The only problem with the scene is that the agents can apparently sense the ball behind their heads. This is due to the nature of the decision that determines the distance between agents and ball. The decision can tell the distance, but not the direction. Miarmy offers no way to find out the angle to a given spot in the human language system, so the agents either see nothing or all around 360 degrees.

### 3.3.3 Train Station Hall Video

This scene combines the target spot seeking scenario with the ball dodging. The setting is a model of a train station hall where lots of agents seek many different targets. Suddenly, a ball rolls through the masses, mimicking a catastrophic event like a derailed train, an amok driver or something of that nature.

The setup is similar to that of Target Spots 3.3.1, but with walls, columns and a kinematic ball rolling through the scene after a couple of seconds. The agents dodge the ball as well as they can, but some are too slow and are being crushed. This scene was rendered and cut into a video.

### 3.3.4 Soccer Video

This scene is a simulation of a simple, soccer-like game. A custom ball-shaped agent is chased around by two groups of human agents on a rectangular field. If the ball enters one of two goal-areas at the far ends of the field, it is reset to the middle of the field, and the two groups return to their half of the field.

The ball agent is a very simple custom agent, consisting of a single-bone skeleton, a bounding box and a ball-shaped geometry. The decisions governing the ball's behaviour are the following:

- An initial decision to make it move forward with speed 60.
- Two decisions to ensure it stays on the field.
- Two decisions to slightly randomize its movement. Their input sentences call a custom Python script which returns a random value. Based on that value, the ball rotates left or right a tiny bit. This makes the ball's movement look better, and can be regarded as modeling spin, uneven ground and tidal forces from the moon.
- A `scoreGoal` decision checks if the ball is in one of two bounds representing goals and runs a custom Python script `reset()` if that is the case.
- A decision labelled `flee` checks for agents in the ball's sphere range. If an agent is very close, the ball speeds up to about three times its speed and



calls a Python script `flee()` which orients the ball away from the closest agent. As a result, it looks like the closest agent kicks the ball a few meters.

After the agents are placed, a Python script is run to establish a master-slave relationship between ball and agents.

The agents are controlled by these decisions:

- Initial decision making them run straight ahead.
- Agent avoidance
- Speed adjustments based on randomized HP for variation in agent movement
- Decisions to ensure agents stay on the field
- Two decisions to follow the ball: If the master is closer than 100 and on the left, rotate left; the same for the right side.
- One decision to make the agents charge for the ball by increasing the agent's playback speed if the ball is close enough

The custom Python scripts which the ball's decisions call are necessary, because the perception system of Miarmy cannot achieve the desired effects by itself. For example, the `reset()` script simply sets the ball's coordinates to 0/0, a feature which is entirely lacking from the offered decision output sentences. Fortunately, the decision input sentence relies on something Miarmy does offer. But if that were not the case, there would be no way to trigger the execution of the script.

When an agent from group one is close enough to the ball, it should dribble it towards the goal of group two. In a decision input sentence, it is possible to detect the ball within a given angle. However, it is impossible to store that angle and use this value in the corresponding output sentence (cf. the last paragraph of Avoid Moving Ball 3.3.2). But this is necessary to kick the ball in the correct direction. To achieve this, the Python script `flee()` retrieves the coordinates of the ball and all agents. It calculates the distance between each agent and the ball, finds the closest agent and calculates the angle between that agent and the ball. Then, it rotates the ball in the direction facing 180 degrees away from the closest agent. At this stage, the simulation looks entertaining but a little erratic, because the agents just chase the ball without a goal in mind. Since we want the agent to eventually score a goal, the calculated angle is tilted slightly towards the goal. As a consequence, an agent in possession of the ball curves slowly towards the goal, as long as it is undisturbed by others. This scene was rendered and cut into a video.



Figure 3.12: A mimicry of soccer



Figure 3.13: The budget did not allow for coloured shirts

# Discussion

---

Miarmy delivers on what it promises. Once the user understands how to use it, Miarmy makes it easy to create simple scenes with many agents. Tasks like placing agents in a variety of ways or randomizing cloth geometries and textures require one click and work well, which makes crowds look much more realistic and scenes more interesting.

However, Miarmy is poorly documented. The online documentation is translated from Chinese into a hardly understandable English. Some pages of the documentation are unfinished or entirely empty. Many of the existing topics are not properly introduced, others consist of mostly examples. Although examples are good and appreciated, a general approach to solve a problem is preferable to only examples. Some topics merely refer to the YouTube tutorials. Although the language is difficult to understand at times and it is not always clear what hotkeys the teacher is using, the videos are helpful. There is a playlist with 118 videos, made by the developers, each discussing a topic or feature of Miarmy. This learning resource is valuable, but problematic: The YouTube tutorials are necessary to learn how to use Miarmy. Most of the know-how in these videos is not written down anywhere, which makes it hard to find a specific topic. While we recognize that some of these problems are due to Miarmy being actively developed, their extent is rather too large.

The decision system is inflexible and therefore limits Miarmy's possibilities. The system is designed for human convenience: A few clicks are enough to generate a decision with mostly functioning preset input and output decisions. There are many available sensory inputs and outputs, but not enough. At first it seems useful that so many sentences are pre-formulated. But when one tries to adjust them to new needs, there is no way to debug them. There is no way to tell if a sentence is well-formed or not. There is no documentation about the general syntax of the sentences, there are only examples. Consequently, whenever a user comes up with a scenario the developers have not yet thought of, they have to perform an unjustifiable amount of guesswork and trial and error. As an example, it would be desirable to detect another agent within an agent's sphere range (possible), store the distance (impossible) and write an output decision

based on the stored value (impossible). More explicitly: Measure how close another agent is and move away from him at a speed proportional to the inverse (or an arbitrary function) of the distance. Generally, arithmetic expressions in the human language system are non-existent, except for (in-)equalities. Granted, there are ways and hacks to circumvent this problem, for example by creating ten decisions, the input sentence of each monitoring an interval of distance from the agent and each enabling a stronger output decision, the closer the interval is to the agent. But this creates overhead and is very inelegant.

Some of the needs of our experiments clearly exceeded Miarmy’s offered decision sentences. So we looked at the code which is called by the Miarmy GUI, did a little bit of reverse-engineering (due to lack of documentation) and managed to overcome some of the boundaries of the human language system. The scripts in the last experiment are an example: If an agent is close enough to the ball, the execution of our custom script is triggered. But what if the condition of something that needs to happen is too complex to form in an input sentence? Then there is no “injection point” for our custom scripts. Or what if we need a function to be called in every frame? Although there is an option for that, the scope of the script called in each frame is local and there is no obvious way to store data between frames or to communicate with the agents’ decisions. To write to a file and read from it in *every frame* of the simulation in order to keep track of global states is possible, but bulky and slow.

In the experiment Target Spots we used HP as a custom state and duplicated a set of decisions to act as a state machine. Again, duplicating decisions that are basically the same creates overhead both in simulation and setup (if it were not for the script we wrote). What if we need to track more than one kind of state? We can use MP for a second kind of state, but that is all (and what if the user needs MP and HP for something else?). We attempted to use Maya attributes, which can be added with Maya commands before simulation and then read with input sentences. But they could not be changed during the simulation. After contacting the developers, they published an update to Miarmy to enable this, but we did not have time to test this new feature. Although it is great that the developers understood and acted so quickly, this measure does not solve the problem that the agents lack easily usable custom states with an interface to the decision system.

Another example: If an agent must overtake another agent in front of it, it can detect their relative speed, it can rotate to the left or right when it is close enough, but how can it rotate back and assume its initial direction after overtaking the other agent? It cannot, unless there is a way to either use timing sentences (i.e. to perform an action for an arbitrary time span) or store measurements (like how much it needed to rotate, how much it has deviated from its original path, how long it has to be rotated in the opposite direction to find its own path and direction again). All these things are either not possible, non-obvious or have to

be implemented with a hack similar to what we used.

We simulated and rendered two videos which incorporate the methods we developed. The first video represents a catastrophic event in a high-traffic area like a train station. We wrote a script to automatically generate dozens of decisions which are necessary for the desired agent behaviour: The agents seek one of several destinations, approach it and then choose a new destination. This would lead to streams of agents with the same target who have to avoid other groups crossing their own path — if it were not for a huge ball crashing through the scene, hitting agents who try to dodge it and run away.

The second video shows a dynamic simulation of a soccer-like game. Two groups of human agents chase a ball agent and try to steer it into the opposing group’s goal. The human agent decisions are set up with a master-slave script. The ball’s behaviour is controlled by several scripts that are called based on decision inputs. Most notably, the ball’s direction is set at an angle facing away from the closest agent, if any agent comes too close. The angle is biased towards the opposing team’s goal. Another script resets the positions of the ball and the agents when a goal is scored, and the game starts anew. These scripts perform actions which are not achievable with Miarmy decision sentences alone.

Although the *need* for such scripts is a point of criticism, the *possibility* to use them is a great advantage of Miarmy. Another redeeming quality of Miarmy is the developers’ eagerness to help and answer questions. The fact that they basically published an update to their software just to satisfy our needs is amazing. Miarmy will continue to be developed and although there are many things to improve, it is already a good and usable tool for non-scientific applications like the animation of film and video game scenes.

# Bibliography

- [1] van den Akker, M., Geraerts, R., Hoogeveen, H., Prins, C.: Path planning for groups using column generation. (2010)
- [2] Treuille, A., Cooper, S., Popovic, Z.: Path planning for groups using column generation. (2006)
- [3] GamerSpawn: Supreme commander 2 flowfield. <https://www.youtube.com/watch?v=jA2epda-RkM> (2010)
- [4] Dave, F.: I've fallen, and i can't get up! <https://www.youtube.com/watch?v=daysCqmqd2Y> (2014)