DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

3D Visualization of Scientific Results in Pedestrian Dynamics

Nikola Tomic

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

3D Visualization of Scientific Results in Pedestrian Dynamics

3D Visualisierung wissenschaftlicher Erkenntnisse am Beispiel von Fußgängersimulationen

Author:	Nikola Tomic
Supervisor:	Prof. Dr. Hans-Joachim Bungartz
Advisor:	Prof. Dr. Gerta Köster, Dr. Felix Dietrich, Marion Gödel
Submission Date:	15.04.2021

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.04.2021 Nikola Tomic

Acknowledgments

I want to acknowledge

- ...my advisor, Felix Dietrich, for his encouragement and availability,
- ...my family and friends for helping me stay positive and supporting me.

Abstract

Pedestrian dynamics are an effective way of studying the movement and behavior of pedestrians in different scenarios. A problem is that these simulations are mostly done in 2D which is not easily understandable for people without a scientific background. To tackle this problem this thesis will build on an existing 3D visualization in Unity to simplify the transporting of scientific findings and data to the untrained eye.

Fußgängerdynamik ist eine effektive Möglichkeit, die Bewegung und das Verhalten von Fußgängern in verschiedenen Szenarien zu untersuchen. Ein Problem ist, dass diese Simulationen meist in 2D durchgeführt werden, was für Menschen ohne wissenschaftlichen Hintergrund nicht leicht verständlich ist. Um dieses Problem anzugehen, wird diese Arbeit auf eine bestehende 3D-Visualisierung in Unity aufbauen, um das übertragen von wissenschaftlichen Erkenntnissen und Daten für das ungeschulte Auge zu vereinfachen.

Contents

1 Introduction

With the current COVID-19 pandemic massively restricting public life for everyone it is essential to reduce physical contact, especially with unfamiliar people, to decrease the risk of potential infections and get the pandemic under control. This is where pedestrian dynamics can make a big difference. Pedestrian dynamics is a broad research area where a lot of different scientific areas meet so that scientists, ranging from psychology to computer science, have to work together [10]. The Vadere pedestrian simulation tool provides a perfect opportunity to identify hotspots in public spaces where a lot of unfamiliar persons get together and where there is an increased risk of getting infected [10].

Because COVID can be transmitted via droplet infection, among others, identification of hotspots is essential when trying to control the pandemic. Based on the different findings like hotspot formation and the pedestrian flow in public places, a pedestrian simulation can support the government at reacting with less latency when trying to optimize possible problem zones instead of having to wait for corona case numbers to rise and therefore increasing the risk of more people getting infected. Furthermore, different rules to control the pandemic like reduction of public life and social distancing can be enforced more efficiently instead of establishing restrictions and limitations without actually knowing their benefits and risks. Different ideas can be tested inside the simulation first and then be evaluated. This does not only go for optimizing COVID-specific rules but can also be utilized in everyday life in soccer stadiums, subway stations and public places for evacuation plans or general optimizations.

I will present different possibilities of how to visualize scientific results from the Vadere simulation software with the existing Unity software, and propose further improvements that can be made to easier visualize information for people without great knowledge about pedestrian dynamics. This would lead to a better understanding of the COVID-19 spreading risks and therefore a greater support in the fight against it. It is very important that the public can understand policy decisions in the pandemic, because the rules only work if everybody follows them regardless if they are strictly enforced or not. Therefore, good visualization of scientific results is crucial. However, at the moment, the Vadere software is focused on presenting scientific results efficiently to scientists and does not have many possibilities of giving the researchers a better idea of the spreading risks. This thesis is structured as follows: In Chapter 2 the current state of art for pedestrian dynamics is shown and explained to give a broad overview of how they are structured and how they work. Examples like the MomenTUMv2 and the Vadere simulation are given, the software will be explained in depth and its different use-cases and possibilities will be presented. Additionally, the existing 3D animation software will also be proposed to show the current state of art which will be developed further. Chapter 3 will focus on the improvements and implementations made to the animation software in Unity while also discussing the improvements made using the individual optimizations. Chapter 4 concludes all the changes made to the software and their usability in the context of the pandemic.

2 State of the Art

In this chapter I include all the necessary prerequisites about the Vadere pedestrian simulation and animation to be able to understand and follow the further chapters. Additionally, this proposes a short introduction to the state of the art of pedestrian simulations and addresses some possible use cases of such.

2.1 Pedestrian Dynamics

Pedestrian simulations describe the flow of pedestrians during a specific timeframe and place. The most frequent use cases of pedestrian simulations are analyzing different types of data, for example evacuation time, flow or density, in a specific scenario. As the dynamic of the pedestrians is mostly unpredictable and the motion is 2-dimensional, its simulation presents itself as a complex task. Additionally, the factors counterflow and clogging increase the complexity of simulating the movement, as the pedestrians can hinder each other when trying to reach their desired destinations [13].

When developing pedestrian dynamics, different modeling approaches also have to be taken into account as they can significantly influence the outcome of the simulation and can therefore also falsify the resulting information that is obtained from the simulation. The different classifications are: "*description*, *dynamics*, *variables*, *interactions* and *fidelity*" [13]. For example if we go into deeper detail the description classification can either be microscopic or macroscopic. The microscopic simulation software treats every agent as an individual with its own attributes like speed. In this model factors like psychology and different preferences for each pedestrian can be taken into consideration while on the other hand the macroscopic model aggregates the pedestrians together so that it is similar to a fluid simulation where the overall flow for the pedestrian group can be analyzed [13, 19].

Microscopic models have the advantage that they are much more precise and realistic, but it also has to be taken into account that these simulations are much more performance intensive [19].

As can be seen later on the description for the Vadere simulation software is microscopic as it treats every pedestrian as an individual.

A soccer stadium is a good example where pedestrian simulations can be used. Here the software can be used to analyze the flow and the behavior of different pedestrians. Furthermore, in the case of a fire the simulation can offer helpful ideas when trying to evacuate the building or even just when aiming to optimize the flow of the pedestrians at the end of a game.

A real world implementation is the Hermes project. This is a cooperation between the University of Cologne, the Esprit Arena in Düsseldorf and the police and fire department. This project aims to create an evacuation assistant that uses cameras and different sensors to get an overview of the building and the pedestrians, computes the information and gives advice to police, firefighters and the security service to optimize the evacuation time in high risk situations like a fire [13].

TUM Pedestrian Dynamic Software

The *chair of computational modeling and simulation* at the Technical University of Munich created a new pedestrian simulator under the lead of *Dr. Peter M. Kielar* [1]. The newly developed software tries to implement as much useful features as possible. To achieve the best solution in terms of performance and number of features, many different existing frameworks are looked at and evaluated to try and develop a highly useful software. Further, the framework is extensible to make it easier for other researchers to use the software and implement own functionalities [11].

Similar to the Vadere simulation the *MomenTUMv2* software is an agent-based system where each agent does not have the possibility of executing own code but is solely altered by the higher level framework. The pedestrian movement is also controlled by using models that modify all the needed attributes [11].

The TUM pedestrian software is not used in this thesis as the Vadere team provides a starting point for the 3D visualization that can be developed further. The software already has all of the important features implemented.

2.2 Vadere Software

In the following I will go into deeper detail about the Vadere simulation and 3D visualization. The Vadere simulation offers the base data that is needed for the 3D visualization and plays a big role in the agent movement. In contrast to the 2D Vadere simulation, the 3D visualization does not compute any important data for the pedestrian dynamics but is only used for the visualization of the pedestrians.

2.2.1 Simulation Software

The Vadere pedestrian simulation software is developed by the research group of *Prof. Dr. Gerta Köster* at the Department of Computer Science and Mathematics at the Munich University of Applied Sciences. It is an open source software to build and test different pedestrian scenarios and it can be used for a wide variety of problems including optimization of pedestrian flow. Its focus is to scientifically analyze and test different scenarios and to evaluate them based on the output data. The software is specifically developed for the simulation of microscopic pedestrian dynamics, meaning that every agent is treated as an individual [10].

Although it is developed for 2D use it offers a straight-forward opportunity to test pedestrian scenarios on a small scale. The software is built in a way such that it allows setting simulation values e.g. the finishing time of the simulation and the step length for the agents [10]. There are also different models that can be chosen so that the agents move in different ways. The different models that the Vadere 2D simulation offers are the *OptimalStepsModel*, *SocialForceModel* and the *OptimalVelocityModel* among others. "The different models describe how pedestrians *walk, overtake, accelerate, evade, and finally approach a visible point in space*" [11]. In the 2D simulation there are many different attributes that are set in the different models. Additionally to the existing model users can create their own models to use for their specific use case. This thesis mainly focuses on the *OptimalStepsModel*.

The *OptimalStepsModel* works by modeling the stepping behaviour of the agent using a disc at the current agent-position. The disc has the radius of the agents step-length and tries to optimize the agents next move [6]. In order to compute this a, utility function is used that takes three goals into consideration: "*reaching the agent's destination based on the distance between the current position of the agent and its destination, avoiding obstacles and keeping a distance from other pedestrians*" [6].

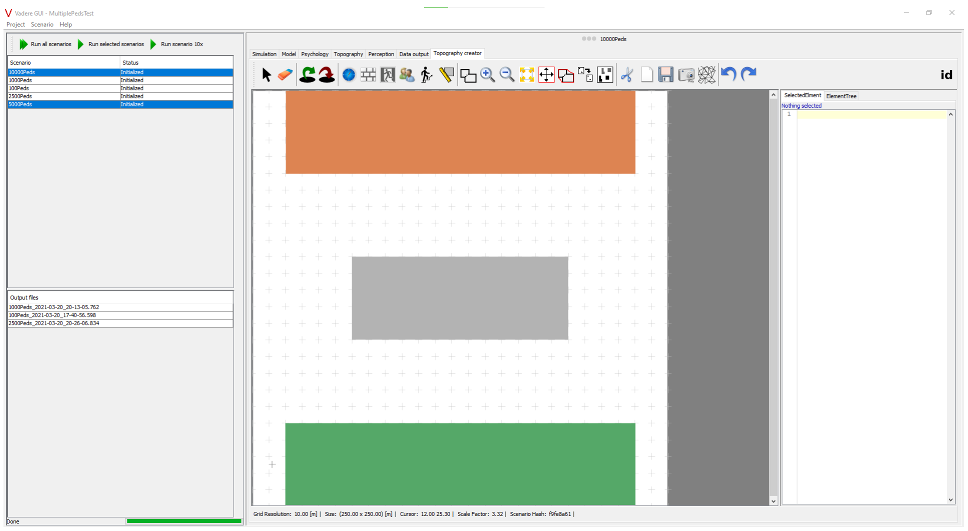
A further important feature of the Vadere simulation software is the topography creator, in which the user can create a topography based on different scenarios shown in 2.1. This yields the possibility of rebuilding desired places from the real world inside of the software to be able to experiment with pedestrian dynamics in different scenarios. In the topography creator the user can build walls, create spaces where the agents spawn and the desired locations where they should move to, as shown in 2.1 [10].

Users can set different processors that define what they should process for the output data. It is possible to link individual processors, thereby performing various computations so the needed data is generated.

An example is the *EvacuationTimeProcessorMinMaxAvg* processor. The processor is used to compute the minimum, maximum and average time the agents need to reach the desired location. But the processor needs information from the *PedestrianEvacuationTimeProcessor* and *PedestrianStartTimeProcessor*. The processors work together as the *PedestrianStartTimeProcessor* returns the spawn time step for every agent and the *PedestrianEvacuationTimeProcessor* then uses the result to compute the evacuation time for the agent as soon as it reaches its goal. The processor can then use that information to compute the needed output.

2.2.2 Animation Software

Because the Vadere simulation only allows for models in 2D, *Maxim Dudin* from the Vadere group developed an animation tool in Unity to be able to showcase all of the findings from the simulation in an even easier way to understand. The animation gives the possibility of displaying the walking direction of every pedestrian, the density of every pedestrian and the contacts between them which can be a huge benefit when presenting the information to the untrained eye. This leads to the broader goal of being


Vadere GUI - MultiplePedsTest
Project Scenario Help
Run all scenarios
Run selected scenarios
Run scenario 10x
Simulation Model Psychology Topography Perception Data output Topography creator
10000Peds
SelectedElement ElementTree
Nothing selected
id
Done

Scenario	Status
10000Peds	Initialized
1000Peds	Initialized
100Peds	Initialized
2500Peds	Initialized
5000Peds	Initialized

Output files
1000Peds_2021-03-20_20-13-05.762
100Peds_2021-03-20_17-40-56.598
2500Peds_2021-03-20_20-26-06.834

Grid Resolution: 10.00 [m] | Size: (250.00 x 250.00) [m] | Cursor: 12.00 25.30 | Scale Factor: 3.32 | Scenario Hash: f9fe8a61 |

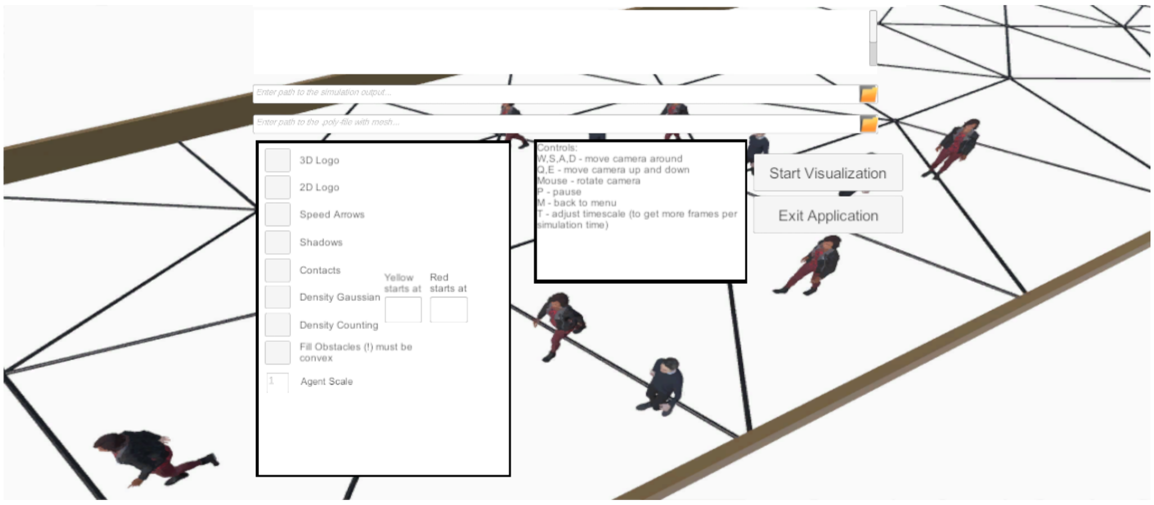
Figure 2.1: Vadere simulation software topography creator. The white plane is the floor where the pedestrians move on, the green plane is the section where the pedestrians are spawned and then move to the orange plane where they disappear. The grey plane indicates an obstacle that the pedestrians have to move around.

able to interpret scientific findings and present them to an audience without greater knowledge requirements, only by using easy to understand visual presentation ques. The visual presentation is still scientifically valid as all of the data stems from the Vadere simulation which is proven to be scientifically correct. This would then lead to an easier identification of hotspots with a higher density of pedestrians, which could then lead to an improvement in public space. The COVID-19 pandemic is a great example indicating how improvements can take a big impact, as hotspots can lead to the virus spreading faster and further. As already stated this can also lead to a greater acceptance of the political decisions in the population [2].

The 3D visualization is the most important factor in this context. Even though all of this would certainly also be possible in 2D, humans have the tendency to immerse in the visualization and understand information better if they can relate to it, so the presentation with human looking agents plays a big role in the understanding of the information.

The software provides a GUI from where everything can be controlled. The in this thesis newly introduced features are implemented in a way that everything that is added can be controlled from the same GUI. The overall structure should remain the same 2.2.

The current 3D visualization handles a small amount of pedestrians in a good way. However when trying to visualize more than 1000 pedestrians at a time the performance drops drastically. To be able to visualize more than 1000 pedestrians in an efficient way the agent controller and instantiating is altered to gain a significant amount of performance.



```
Controls:
W,S,A,D - move camera around
Q,E - move camera up and down
Mouse - rotate camera
P - pause
M - back to menu
T - adjust timescale (to get more frames per
simulation time)
```

Figure 2.2: Vadere animation software main menu

Now I will go into deeper detail about the agent controller and the current state of the density depiction as they play a big role in the further course of this thesis.

Agent Controller

In order to compute the movement of the agents, an agent controller script is used. The script reads the trajectory output file from the simulation where the locations and time steps are stored for every agent. The script does that by reading through the lines and saving the steps in an array for inside of the respective agent gameobject.

During the runtime of the simulation the agents compute their trajectory to the next point on the map and move there over time [10].

Density Depiction

The current way that density is shown in the Vadere pedestrian simulation is by coloring the pedestrians according to the density of other pedestrians around them. This is achieved by using different density files that saves all the densities from every pedestrian for every time step so that the density can be shown. There are 2 different methods used to compute the density: the Gaussian and the Counting method. In the unity software the density file parsed and then used to color every pedestrian. For that the user can define the range in which the colors yellow and red represent the different densities such that simulations with different numbers of people can be adapted [10].

3 3D Visualization of Scientific Results in Pedestrian Dynamics

Here I want to present the ideas of this thesis and implementation details in order to further improve the Vadere animation software to be able to showcase scientific findings even better and make the overall animation smoother.

The goal is to be able to present the animation to people without greater knowledge about pedestrian simulations and present in an intuitive way how different scenarios can change the outcome and the animation process.

3.1 Visualizing Scientific Results in 3D

The Vadere simulation software proposes a great opportunity to monitor and analyze different pedestrian scenarios. It is proven to be scientifically correct in terms of pedestrian motion. This makes the 2D simulation a powerful tool for researchers when analyzing the different scientific findings as it proposes a simple way of simulating real world places with the topography creator and psychology features amongst other.

To further build onto this the 3D visualization can greatly improve transporting information to people without much knowledge about pedestrian dynamics and physical social interaction in everyday life on a scientific level. 3D visualization lets users observe the scenario more carefully and relate to it more direct. This creates a better immersion such that the user understands the visualized data better as one can project oneself into the same situation. Additionally, the visual representation of human-like agents offers the user a more intuitive way of understanding what is going on in the visualization.

A key component that makes a big difference from the 2D to the 3D visualization is that the interactivity is more adapted to a real scenario, where people would also see others in 3D, with perspective distortions and occlusions in place [16]. This results in the user being able to interact with the pedestrians using the camera to fly around in the virtual space. Further, it lets the user choose where to look much more freely e.g. when looking for high-density spots. The 2D simulation only allows the user to focus at a plane where the dots representing the pedestrians move.

Problems that can also arise with the 3D visualization are that because users can move around and look at specific sections of the floor, not all of the results from the whole scenario are looked at which is not optimal from a scientific standpoint as all of the data has to be taken into account. Furthermore the occlusion and other perspective camera effects can hide important visual cues which cannot happen in 2D.

Even with all of these advantages the 3D visualization has the flaw of being more computationally intensive in terms of rendering than the 2D simulation. This problem is not of substantial severity as the advantages of the 3D visualization outweigh the disadvantages as discussed in the further course of this thesis. Additionally, the rendering can be implemented very efficiently using the GPU. GPUs are much more efficient when running different parallel tasks as they consist of many more cores than the average CPU. Although the speed comes at the price of less image quality the advantages of the GPU outweigh the disadvantages and have a high use benefit for many different use cases [3].

In the course of the thesis there are a few optimizations to the 3D visualization that directly correlate with the COVID-19 pandemic. The social distancing representation, density floor shader and voronoi shader aim at helping to evaluate different scenarios specifically aimed at the pandemic. The social distancing representation takes a general rule that exists in most countries to evaluate how good these rules can be enforced and what benefit can come from it.

The shaders do not only aim to be used in the context of the pandemic as they are a good way of identifying hotspots which can also be utilized in other real world use cases. The density floor shader is inspired by overall density representations that can easily be transferred onto the pedestrians. Higher density spots can then be indicated using different coloring which should be easy to understand. The voronoi shader works similarly to that but the higher density spots are not shown as clear as with the density shader, although it has further benefits as the space around every agent can be identified more easily.

Other optimizations are aimed at making the 3D visualization more efficient utilizing a new way of rendering the agent's. Using batch processing a huge performance increase can be noted, although this comes at the expense of the agents being rendered as capsules.

A further optimization is used to present critical data about the pedestrians and the simulation at the end of the visualization run.

3.2 Social Distancing Representation

One of the biggest changes to everyday life that COVID-19 brought is social distancing. In most places all over the world it is mandatory to stay away at least 1.5 meters from other people in public places.

As COVID-19 is a relatively new disease and the Vadere Animation software was developed by *Maxim Dudin* who is not with the Vadere team anymore there are no implementations that have the same or similar effects as this social distancing representation.

I have already stated the goal of the visualization is to be able to showcase it to an audience that does not have the scientific understanding of pedestrian simulations. With this COVID specific case I aim to be able to visualize if the pedestrians in the simulation respect these different rules and represent different scenarios where social distancing may be easier or harder to achieve. For example given a big public place where the pedestrians can spread out and use different routes to get to the same location, it is easier to comply with social distancing rules. As opposed to if someone takes a subway tunnel on the route it can be seen that the pedestrians all have to walk a similar path towards a specific location. In this case the social distancing cannot be maintained as easy as in the first example which would be identifiable with the proposed social distancing visualization.

In order to achieve a good social distancing visualization is is needed to find an easy to understand way of visualizing the distance that should represent the needed social distancing between the pedestrians. As different countries also have different rules regarding social distancing I also need a way of depicting the different radii to comply with these different rules. This can easily be done by using a circular representation of the needed social distancing radius below the agent, as the circle shows the social distancing needed for all agents. This should be easy to understand as the circle can be colored and if a pedestrian is inside of the social distancing circle of another pedestrian the social distancing does not hold anymore. To further improve on this the circle can be colored depending around if the social distancing is maintained and thus further simplify the identification of such problems. Additionally, the radius of the circle should be configurable in the GUI, to comply with the desired and varying social distancing rule set.

In contrast to other solutions how this can be implemented, the chosen way yields the advantage that problem zones are easily identifiable. Without the circle implementation, there is no inherently clear way of indicating social distancing violations to the user. With the circle it can even identified if pedestrians are coming close to each other even without violating the social distancing.

For the implementation inside of the Vadere animation I added a circle to the agent prefabs, as seen in 3.1, that is clamped below the agents and has the standard size of the prescribed social distancing rules although the radius can be modified from the GUI to support the various differences. The circle has a material that can be altered from the script and should change color between red and green if agents are too close to each other or if the distance is adequate. It also adjusts the opacity of the color to represent how close they are and make it easier for persons to determine how critical the contact between the agents is.

In order to compute the distance of agents I implemented a loop that checks if any of the other agents are inside of the radius to the current agent inside of the agent controller script using the *OvelapSphere* function in Unity. The *OverlapSphere* method checks for any colliders that are inside of a given radius. But to be able to use this method the agents need a collider in order for it to return an array of colliders that interfere with the given sphere, so I added a box collider to the agent's prefabs. The method now returns every collider that collides with the sphere but the colliders from the agent gameobjects are the only ones that should get taken into consideration. To

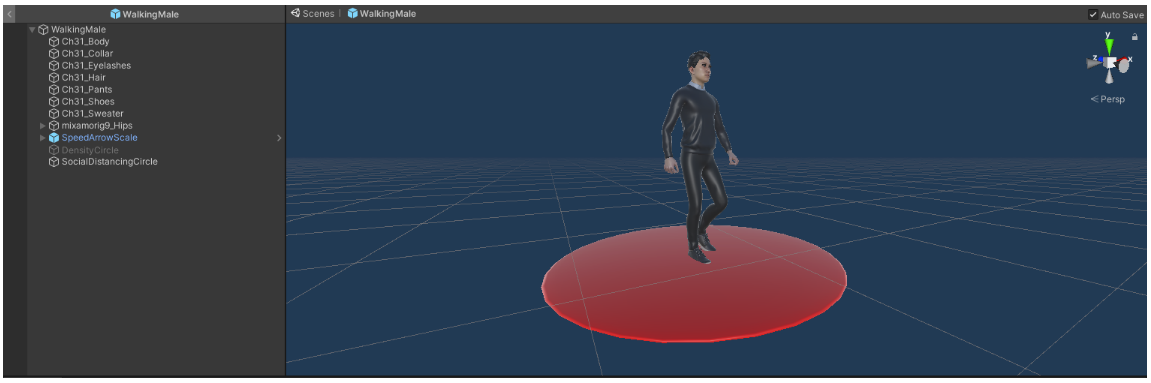


Figure 3.1: Agent prefab with the circle attached to it for the social distancing representation

counteract this, I set the agent prefabs so that they are on their own layer which can also be considered in the *OverlapSphere* function so that it only takes gameobjects into account that are on the desired layer [14].

The color of the circle is then computed by using the distance from the current agent to the colliding agent and then scales it to a value between 0 and 1. To achieve this, a new vector is created from the current agents position to the other agents position. The length of that vector can then be divided by the social distancing radius to convert the value between 0 and 1. As this value is used for the alpha of the color the opacity of the circle is higher when an agent is further away than closer which is not very intuitive. The value must be inverted to get the opposite effect. Furthermore, as the agents will never be on the same spot the alpha value will never reach a value of 1. Therefore 0.2 is added to the opacity value to make the visual representation of the closeness between agents more clear.

This is then used to set the opacity of the material. If no agents are inside of the social distancing circle of the current agent, the color of the circle is changed to a static green. This must be calculated for every agent, so a function was implemented inside of the agent controller that gets called during the *Update* function of Unity, as the access to the circle is easier. There the color and the radius can be accessed and set from the script.

Additionally an input box which takes a float in meters is added to the GUI to be able to set the social distancing radius for the animation.

The social distancing can now be represented by using the circle which should be an easy to understand way of doing this, as shown in 3.2. A further improvement that could be made in the future is implementing a selection of standard social distancing rules from different countries instead of having to insert the desired distance every time. To add to that the Vadere team is working on implementing the computation of possible infection cases from one agent to the other. That could also be taken into account in the future by coloring the circle in a different color if an agent is infected and also if an infection was carried over.

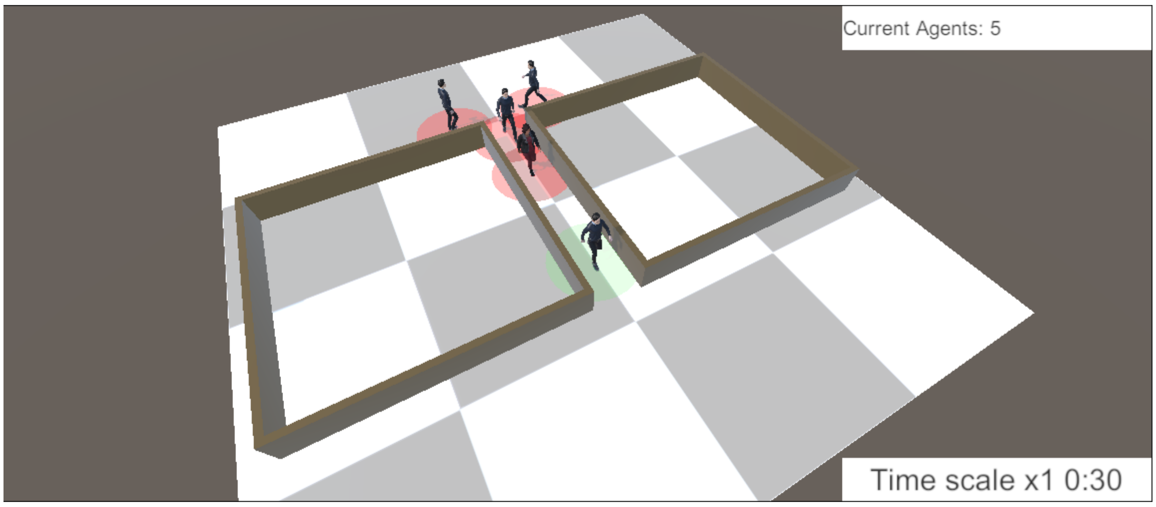


Figure 3.2: Social distancing represented by circles around the pedestrians

3.3 Hotspot Identification

Hotspot identification is a great area that can be represented in the 3D visualization. Identification of hotspots can help with evacuation of buildings in real-time [13]. Furthermore hotspots lead to crowding which means that the flow of pedestrians is disrupted. This can lead to different problems like overcrowding of public places because of inefficient designs.

To achieve good hotspot identification I will use two different approaches. A standard density representation where high density and low density places should be colored differently and a voronoi diagram. The voronoi diagram also the additional property of showing the space that an agent has around it.

3.3.1 Density Floor Shader

The current implementation of density visualization is done by coloring the pedestrians according to the current density at their position. To further improve this, the density can be computed for every point on the floor that the pedestrians move on and then colored correspondingly. This significantly simplifies the identification of hotspots as the density is shown for every point on the map and areas with lower density could be considered when trying to improve on existing design of coloring the agents in their respective density color.

Again this implementation to the Vadere animation software targets the identification of hotspots especially in the context of COVID-19. On the one hand this offers the possibility to demonstrate how different strategies in the fight against COVID perform and thus could be used for testing before these strategies would be applied in the real world. On the other hand this also assists if someone is not convinced about how useful the strategies are, as the animation gives the possibility of showing the utility in real world scenarios.

To achieve this, shaders can be utilized in Unity. As shaders are executed on a Graphics Processing Unit (GPU), real-time visualization using them does significantly outperform fully CPU driven solutions.

Particularly *Shader Graph* is a simple tool that enables everyone to make a shader without greater knowledge of how to write shader code. It works by connecting nodes, where each node accepts various inputs and uses these to compute an output [18].

The first idea was to use *Shader Graph* to compute the color of the floor. To be able to compute the density, the position of every person is needed. This brings up a problem that I encountered when using shadergraph as initializing the positional data from the agents to use it inside of shadergraph is complex and not very feasible.

A more straightforward idea was to implement the visualization using a *HLSL* shader. Here I can create buffers which can be filled with agent positions using a C# script and read inside the shader, which is simpler and has a higher configurability than what can be done with shadergraph. The buffer must be attached to the material of the floor, so that the shader can use the data.

The idea for the density computation is similar to the approach I will use in the voronoi computation and comes from [8].

In order to implement this concept I created a script where all of the positional data from the agents is processed and attached to the material. This is done by accessing the agents' gameobjects during runtime and copying their positions into the buffer for the shader. Additionally I bind the number of agents as an integer to the material. The is required because the shader loops over all agent positions in the given buffer, however the itself buffer does not provide any information about the total number of positions stored in it.

In the first version of the density shader I compute the distance of each agent to the world space position of the current pixel and use these to compute the color of this pixel based on the fraction of agents which are within a radius of 10 units from the current position. More specifically a weighted sum is computed over all agent positions, where the weight of a position increases the closer it is to the current one, as this indicates a higher local density. Finally the weighted sum is normalized and used to linearly interpolate between blue and red, where blue indicates a low agent density and red indicates a high agent density. Additionally, the interpolation factor is fed into the *smoothstep* function, which increases the contrast due to a steeper midsection and also creates smoothed transitions at local density extrema 3.3.

In the second edition of the shader I take nearly the same approach as for the voronoi shader as the minimum distance to the agent closest to the vertex has to be calculated and use that to linearly interpolate the color again between blue and red. A problem with this approach is that the spots where one agent is located are already shown as high density spots. The previous approach handles this much better as the weight that is included in the computation counteracts this 3.4.

In the third and also last implementation idea for the density shader I use a new

```
fixed4 col = fixed4(0,0,1,1);
float sumDist = 0;
float sumWeight = 0;
for (int y = 0; y < agentCount; y++)
{
       float3 pos = agentBuffer[y].xyz;
       float dist = distance(pos, i.worldPos);
       float weight = 1 - smoothstep(1, 10, dist);
       sumDist += dist * weight;
       sumWeight += weight;
}
col.rgb = lerp(float3(0,0,1), float3(1,0,0),
       smoothstep(0,1,sumWeight/sumDist));
return col;
```

Figure 3.3: Density shader code Version 1

```
fixed4 col = fixed4(1,1,1,1);
float minDist = distance(agentBuffer[0], i.worldPos);
for (int y = 0; y < agentCount; y++)
{
       float3 pos = agentBuffer[y].xyz;
       float dist = distance(pos, i.worldPos) * 0.1;
       minDist = min(minDist, dist);
}
col.rgb = lerp(float3(0, 0, 1), float3(1, 0, 0),
       smoothstep(0, 1, 1-sqrt(minDist)));
return col;
```

Figure 3.4: Density shader code Version 2

```
fixed4 col = fixed4(1,1,1,1);
int nearbyAgent = 0;
float minDist = distance(agentBuffer[0], i.worldPos);
for (int y = 0; y < agentCount; y++)
{
        float3 pos = agentBuffer[y].xyz;
        float dist = distance(pos, i.worldPos)*0.5;
        if (dist < 1.0) {
                nearbyAgent++;
        }
}
col.rgb = lerp(float3(0, 0, 1), float3(1, 0, 0),
        smoothstep(0, 10, nearbyAgent));
return col;
```

Figure 3.5: Density shader code Version 3

approach. Here it is saved how many agents are closer than a set distance and use that to again linearly interpolate the color for the vertex. The difference to the approach beforehand is that the I take the perspective from the vertex for which the color is computed instead of the agents perspective 3.5. This approach also handles the densities at the exact spots of the pedestrians a bit better as it does not identify them as the highest density spots as it did in the previous version.

As expected, the density visualization can be really useful when trying to identify hotspots, the color is distinguishable very efficiently even when many pedestrians are close to each other and could possibly obstruct the view. Of course the main idea behind implementing this feature lies in hotspot identification with COVID in mind especially when used in combination with the social distancing representation, but there are more use cases as already proposed like stadiums or public places similar to subway stations, as shown in 3.6.

All of the different approaches have advantages and disadvantages, as can be seen the first edition already had smaller problems with the density projection over the whole floor but in contrast to the other approaches the first edition takes the distance to every agent into account to be able to compute a weight so the density is distributed more realistically.

3.3.2 Voronoi Floor Shader

A voronoi diagram shows polygons below the agents to display all of the points that are closest to the agent in the center and no other agent. The voronoi diagram is useful as it offers another approach to identifying high density spots as smaller polygons in the voronoi diagram indicate that the agent has many agents that are close and therefore

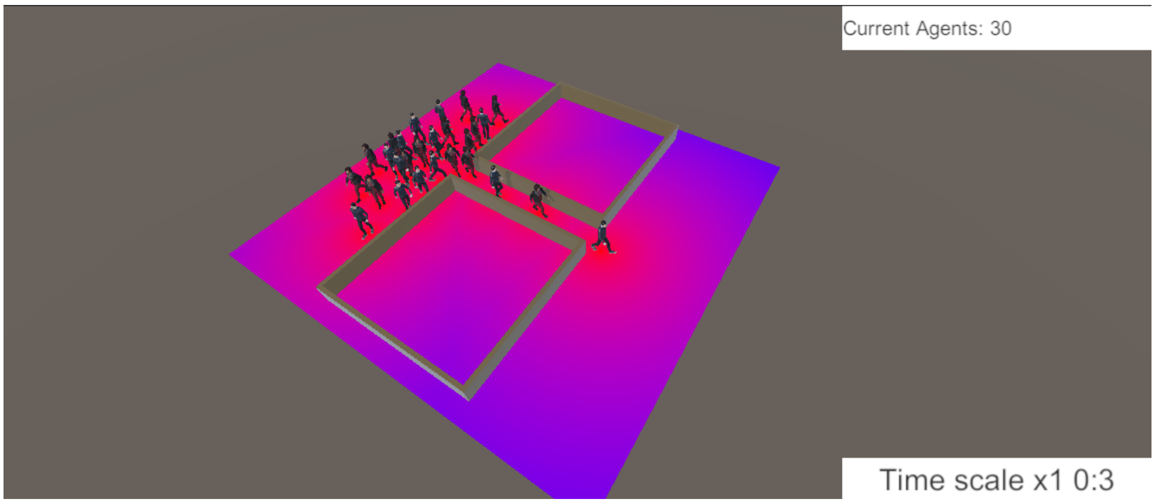


Figure 3.6: Density shader example. Regions with high density are colored red, low density regions are colored purple

create a hotspot [9].

To be able to implement this a shader can be used again. For this a similar approach as for the density computation is used, taken from [8]. A buffer is used again to store all of the data from the different agents. Specifically the positional data of every agent is needed. Similar to the density shader I use the same script with small changes again, to process all of the information for the buffers and attach them to the floor material.

In the vertex shader all of the vertex positions are first transformed to get their respective world positions. This is necessary to be able to compute the distance between the point on the floor and the positions of the agents in the fragment shader. Next the distance to the nearest agent is stored. This is done by looping over all of the agents' positions and testing if the distance is smaller than the current minimum distance.

In the first version of the voronoi shader the edges between every agent were colored black, but this did not work for the whole floor. There was only a small radius around every agent where this would work so the voronoi diagram was only correct when all of the agents where close to each other. Making the desired radius bigger was not working as the edges would not be visible anymore 3.7.

As the voronoi diagram aims to create polygon for every agent, the idea came up to just color every one of these in a different color. This would firstly solve the problem that the edges are not visible, secondly the polygons would cover the entire floor and thirdly help when looking to identify hotspots as the polygons are easier to distinguish from each other even when agents cover parts of them.

In order to achieve this, a color has to be computed for every agent so that when the agent closest to the current point is processed the according color for the point can be chosen. This is done by using a random function that lets us input a float and returns a random color based on the input [15]. The first edition of this idea just used index

```
fixed4 col = fixed4(1,1,1,1);
float minDist = distance(agentBuffer[0], i.worldPos);
int minIndex = agentBuffer[0].w;
for (int y = 0; y < agentCount; y++)
{
        float3 pos = agentBuffer[y].xyz;
        float dist = distance(pos, i.worldPos);
        if (dist < minDist) {
                minDist = dist;
                minIndex = agentBuffer[y].w;
        }
}
col.rgb *= rand1dTo3d(minIndex);
return col;
```

Figure 3.7: Voronoi shader code

of the agent inside of the agent buffer. The problem I encountered here is that as the agents reached their destination point and became invisible the color of the polygons changed as the index for every agent changed inside the buffer. To counteract this a new variable is introduced to the buffer where the *instanceId* for every agent is stored as they are unique for every agent inside of Unity which is then used to compute the color for every agent [12].

This shader now returns a voronoi diagram on the floor based on the agents positions seen in 3.8. Similar to the density shader this can help with the identification of hotspot as this is one of the main use cases especially with COVID. Such software could be used in COVID test centers to see where the hotspots in the queues are and how the process of leading the pedestrians through the center can be optimized with the infection chains in mind. But the voronoi shader does not only help in the context of COVID it can also be used when optimizing the flow of pedestrians while taking personal space into consideration. The voronoi diagram shows all the points where the agent in the center is closest to so depending on how big the area is the pedestrian can be more or less comfortable.

3.4 Displaying a Summary of Important Simulation Results

An approach for being able to evaluate different simulation scenarios is to implement the showing of important values at the end of the animation. This would help for example when trying to improve the flow rate through a certain passage. The average evacuation time can be used as a score and different models can be simulated to see which one performs best. Another scenario where this could prove useful especially

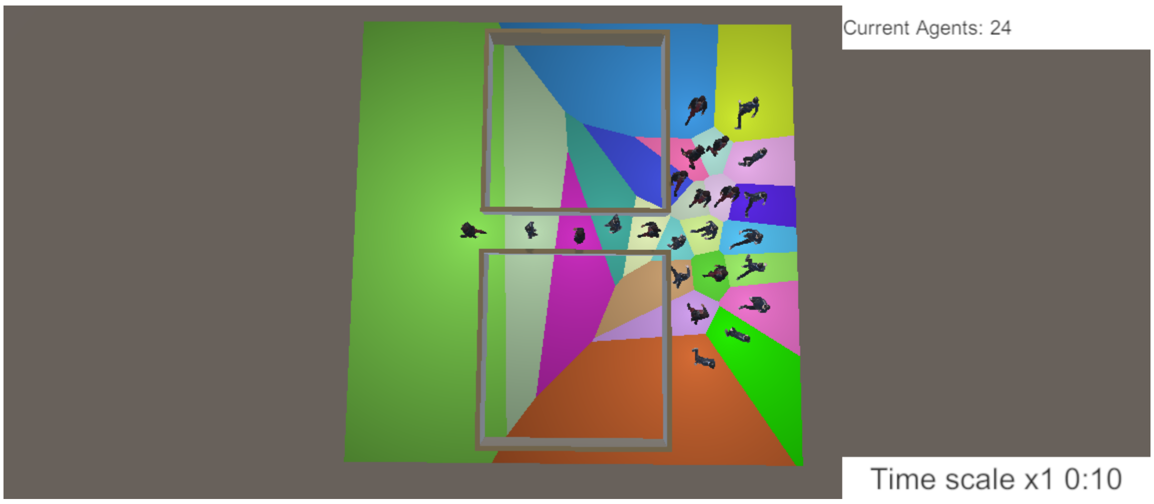


Figure 3.8: Voronoi diagram on floor below the pedestrians

with the COVID-19 pandemic, in most public places the density should be as low as possible so that fewer people infect each other. The average density value can be used as a scoring method for different simulation models.

Further values like average speed, density and evacuation time can be used to determine how changes to the starting point influence the simulation and if they are useful or not.

To be able to show information on screen a canvas with different text fields is used [4]. This will be presented using the example of the average evacuation time. In the Vadere simulation software I can define a new output file that holds the data gathered from the average evacuation time processor. The file can now be parsed at the start of the animation similar to the already existing parsing of files in the animation software [17]. To choose that these values should be shown a toggle was implemented also oriented at the already existing GUI to keep the software uniform.

The values are then stored in a newly created script to save them during the runtime of the animation. In the unity update function, it is tested how many agents are currently still active and as soon as no agents are active anymore the canvas is turned on to show the values 3.9.

This principle can be extended to show much more information but for now the evacuation time is the only information that is implemented to be shown in Unity as a proof of concept. There is far more information like the average flow rate or average density that can be shown and used for evaluation.

3.5 Performance Improvements with Batch Processing

Because the Vadere simulation uses a microscopic model it is very cost expensive, especially when simulating a lot of pedestrians at the same time. This also has a huge

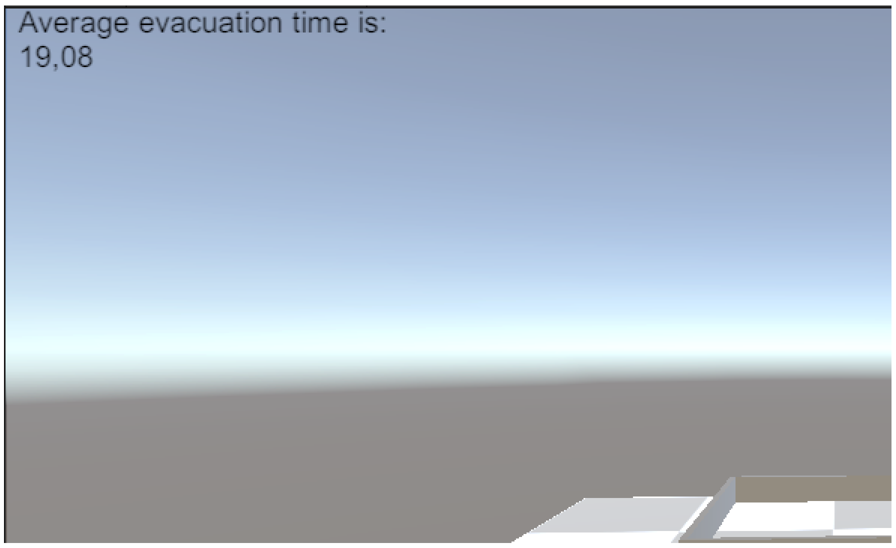


Figure 3.9: Example for displaying the summary

influence on the 3D animation in Unity as a lot of gameobjects have to be initialized and rendered. I will concentrate on trying to optimize the rendering time of the agent gameobjects. The motivation behind this is that with the current state of the 3D animation small scale scenarios like an office can be tested but scenarios like festivals or concerts would lead to problems. The aim is to be able to visualize around 10 000 agents at the same time. This already takes up a lot of time when done with the simulation software as every agent is treated as an individual.

In order to accomplish this, batch processing is a nice way of doing this in Unity. To understand it draw calls have to be explained. A draw call is a function that is called for every material that is in the scene to render them, thus show them in the game. This means the more gameobjects with materials are in the scene the more draw calls are produced by Unity. Batch processing is used to batch identical materials into a group which can then be drawn using just on draw call. Each group can hold up to 1023 objects so a batched draw call can only draw up to 1023 objects. This can immensely improve the performance as the GPU can render all objects faster [5].

Unity automatically uses batch processing for objects with identical materials if it is activated and performance gains can be expected. In this case the agent prefabs use skinned mesh renderers where dynamic batching is not supported so this will not work of the shelf. In order to be able to use dynamic batching the agent prefabs have to be adjusted as seen in 3.10. The new prefab should use one of the supported meshes from Unity. I will use a mesh renderer and replace the agent with a capsule. The prefab will remain the same otherwise so the agent controller is still attached to it. This will ensure that the movement still works with all of the agents. During testing it became apparent that Unity did not automatically use batch processing to draw the gameobjects in this case.

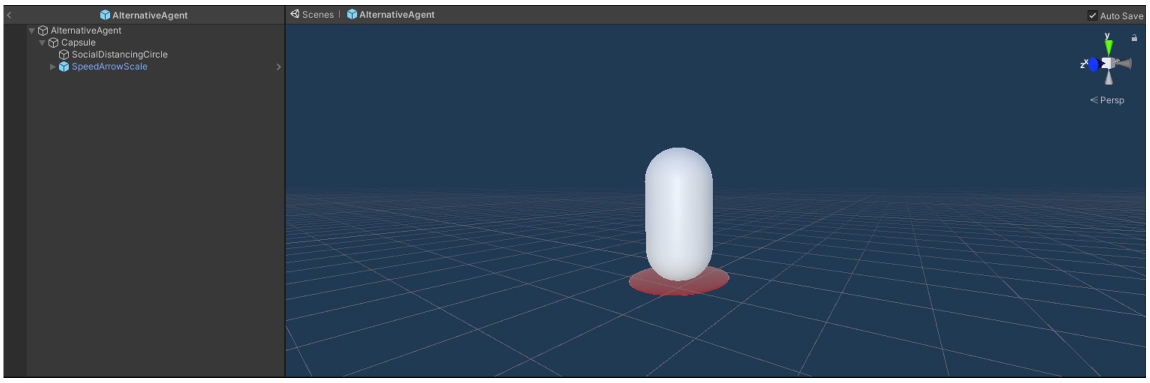


Figure 3.10: Prefab for the new Agents

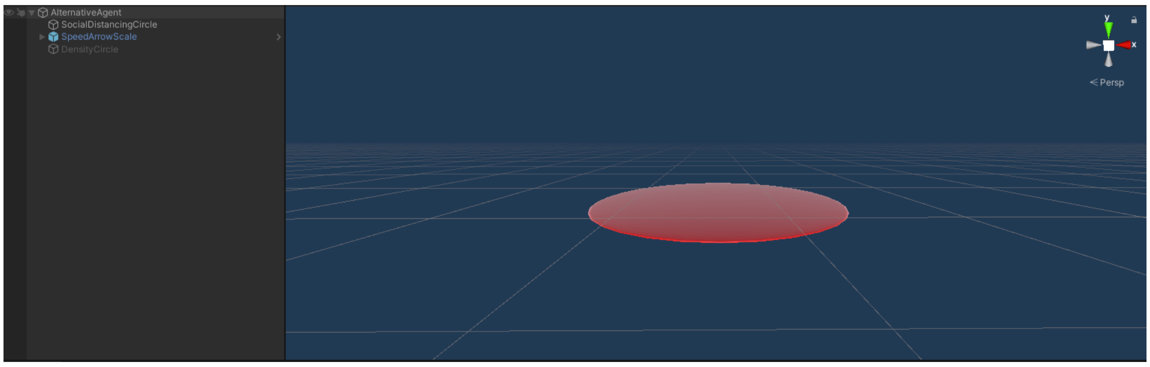


Figure 3.11: Prefab for the newer Agents without own renderers

Another possibility to implement batch processing is doing it from script using the *DrawMeshInstanced* function. This lets me call a batched draw call from the script by providing the desired mesh, material and the positions where they should be drawn [7]. To be able to do this I created a new prefab that does not have a capsule mesh and a renderer but the movement can still be processed as before but rendering would be handled from the script but the social distancing circle is still retained to be able to use it 3.11. This can be done by accessing the agents' gameobjects during runtime and saving their positions in an array to be able to draw the meshes and materials at these points 3.12. The mesh and the material are referenced using the editor.

In order to compute the positions and the different batch groups the number of active agents has to be saved. In this case all of the agent gameobjects are instantiated inside of the empty agent gameobject in the scene that can be accessed from the code. Therefore the childcount of the gameobject can be taken to get the number of agents and then saved to the $numOfShowingAgents$ variable as can be seen in 3.12 row 3.

After that the array for the matrices and the *runs* variable are initialized. The *runs* variable is needed to compute how many groups are needed to visualize all of the agents as they can only be as big as 1023 objects.

In the loop all of the agents' positions are saved to the matrices array including a set rotation and scaling. The rotation and scaling are always the same so they can be hardcoded.

After the loop there is a last run for the last group of agents to be drawn. The *lastShow* variable saves how many agents need to be drawn on the last run. This is then used to create another loop that iterates over the last agents and computes the positions the same way as the loop beforehand.

This had a small increase in performance but was not able to visualize more than 5000 pedestrians. A problem that could lead to this is that all of the gameobjects take a huge toll on the performance so the next idea is instead of instantiating the agents as gameobjects a list of agents as classes can be used that save the positions and needed data 3.13. This works in a similar way to the gameobjects but should be much more performance efficient as the agents only exist in the code as opposed to the gameobjects that require much more computational power.

For the agent class the *AgentController* can be used as reference as it will work in a similar fashion. The biggest difference here is that the class does not support the standard Unity functions like Update. The class needs to have the *setInitialValuesForAttributes* to set the needed attributes that are parsed from the output file of the simulation, the *changeCurrentDestinationIfItIsReached* function to parse the next destination from the string list and the *moveTowardsCurrentDestination* function to be able to compute where the agent should be for the next time step. The agent movements are computed using the *FixedUpdate* function from Unity where I iterate through the agent list and call all of the needed functions 3.14. These functions include the *moveTowardsCurrentDestination* and *changeCurrentDestinationIfItIsReached*. As the agents are saved in a list all of the agents that already reached their end destinations need to be removed. To do this the list has to be copied into a temporary list or else it is not possible to iterate over and alter it at the same time. After that the agent count can be taken from this list. This is needed as the render groups can only hold up to 1023 agents as stated earlier. To counteract this I divide the *agentcount* by 1023 to get the number of groups that hold exactly 1023 agents. This number is stored in the runs variable. The agents are drawn using a loop and as the positions of the agents change every iteration the *agentPosition* has to be computed inside of the list. After this loop is done the last draw call is computed using the runs variable and computing how many agents remain to be drawn.

In order to test the new implementation I created scenarios with 1000, 2500, 5000 and 10 000 pedestrians. At 1000 pedestrians it can be seen that the implementation already leads to a significant performance gain, which increases with a bigger agent count. It can also be seen that from the 1000 pedestrians count onward the original implementation does not have usable framerates as they drop below 20 fps.

After testing it becomes apparent that the new implementations do have great effects when compared to the original implementation. Even though the rendering was improved significantly the fps and the initialization of the agents takes a long time regardless of the implementation. This could come down to the way the agents get initialized and their movement gets computed. Although the implementation as classes

```
void drawAlterntiveAgents()
{
    int numOfShowingAgents = agents.transform.childCount;

    Matrix4x4[] matrices;
    int runs = (int)Mathf.Ceil(numOfShowingAgents / 1023);
    for (int i = 0; i < runs; i++)
    {
        matrices = new Matrix4x4[1023];
        for (int j = 0; j < 1023; j++)
        {
            int agentPosInList = i * 1023 + j;
            Vector3 position = agents.transform.GetChild(agentPosInList)
                            .transform.position;
            position.y += agentScale * 0.5f;
            Quaternion rotation = new Quaternion(0, 0, 0, 1);
            Vector3 scale = new Vector3(agentScale, agentScale, agentScale)
                        * 0.5f;
            matrices[j] = Matrix4x4.TRS(position, rotation, scale);
        }
        Graphics.DrawMeshInstanced(mesh, 0, mat, matrices, 1023);
    }

    int lastShow = numOfShowingAgents - (runs * 1023);
    matrices = new Matrix4x4[lastShow];
    for (int j = 0; j < lastShow; j++)
    {
        int agentPosInList = (runs) * 1023 + j;
        Vector3 position = agents.transform.GetChild(agentPosInList)
                        .transform.position;
        position.y += agentScale * 0.5f;
            Quaternion rotation = new Quaternion(0, 0, 0, 1);
            Vector3 scale = new Vector3(agentScale, agentScale, agentScale)
                        * 0.5f;
            matrices[j] = Matrix4x4.TRS(position, rotation, scale);
    }
    Graphics.DrawMeshInstanced(mesh, 0, mat, matrices, lastShow);
}
```

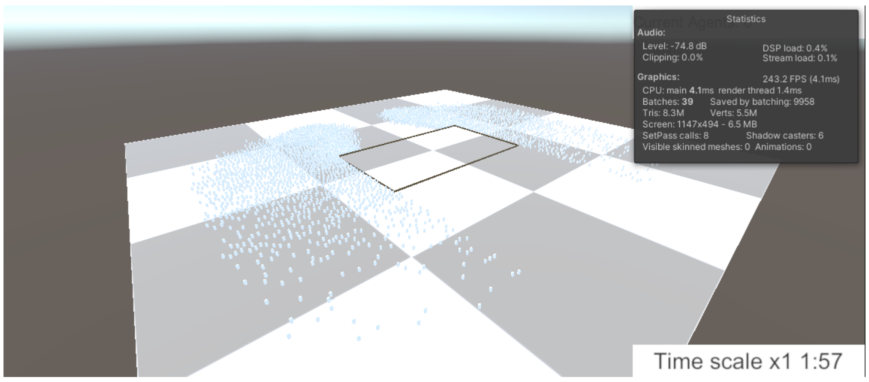
Figure 3.12: First implementation for batching

```
public List<string> trajectorySteps;
public Vector3 currentPosition;
public float timeDelayBeforeSpawn;

public float appearingTime;
public int currentStepNumber;
private float nextStepChangeTime;
public float currentSpeed;
public Vector3 currentDestination;
public bool show = true;

public Agent(Vector3 pos, List<String> trajectorySteps, float timeDelay)
{
    currentPosition = pos;
    this.trajectorySteps = trajectorySteps;
    timeDelayBeforeSpawn = timeDelay;
}

private void setInitialValuesForAttributes()
{
    appearingTime = Convert.ToSingle(double.Parse(trajectorySteps[0].
              Split('␣')[1],System.Globalization.
              CultureInfo.InvariantCulture));
    appearingTime += timeDelayBeforeSpawn;
    currentStepNumber = 0;
    nextStepChangeTime = 0f;
    currentSpeed = 0f;
}
```

Figure 3.13: Part of the new agent class

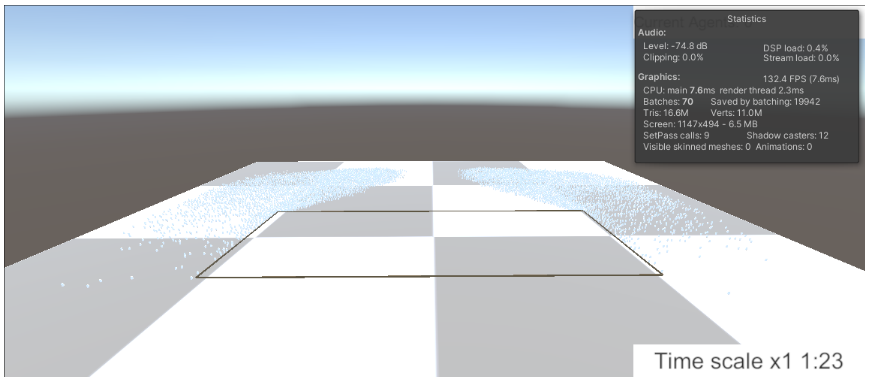
```
private void changeCurrentDestinationIfItIsReached()
{
    if (Time.time > nextStepChangeTime)
    {
        currentStepNumber++;
        if (currentStepNumber > trajectorySteps.Count - 2)
        {
            show = false;
            return;
        }
        nextStepChangeTime =
                    Convert.ToSingle(double.Parse(trajectorySteps
                    [currentStepNumber + 1].Split('␣')[1],
                    System.Globalization.CultureInfo.InvariantCulture));
        nextStepChangeTime += timeDelayBeforeSpawn;
        Vector3 oldPosition = currentPosition;
        currentDestination = new Vector3(Convert.ToSingle(double.Parse(
                    trajectorySteps[currentStepNumber + 1]
                    .Split('␣')[3],System.Globalization.CultureInfo.
                    InvariantCulture)), 0, Convert.ToSingle(
                    double.Parse(trajectorySteps[currentStepNumber + 1]
                    .Split('␣')[4],System.Globalization.
                    CultureInfo.InvariantCulture)));
        currentDestination.y = oldPosition.y;
        float dist = Vector3.Distance(oldPosition, currentDestination);
        float time = Convert.ToSingle(double.Parse(trajectorySteps
                    [currentStepNumber + 1].
                    Split('␣')[1],System.Globalization.CultureInfo.
                    InvariantCulture))- Time.time + timeDelayBeforeSpawn;
        currentSpeed = dist / time;
    }
}

private void moveTowardsCurrentDestination()
{
    // keep moving towards currentDestination with current speed
    currentPosition += (currentDestination - currentPosition) *
                    Time.fixedDeltaTime * currentSpeed;
}
```

Figure 3.14: Agent position update functions



```
Statistics
Audio:
  Level: -74.8 dB                 DSP load: 0.4%
  Clipping: 0.0%                  Stream load: 0.1%

Graphics:                         243.2 FPS (4.1ms)
  CPU: main 4.1ms  render thread 1.4ms
  Batches: 39     Saved by batching: 9958
  Tris: 8.3M      Verts: 5.5M
  Screen: 1147x494 - 6.5 MB
  SetPass calls: 8            Shadow casters: 6
  Visible skinned meshes: 0  Animations: 0
```

Figure 3.15: Example visualization of 5000 pedestrians with the second implementation



```
Statistics
Audio:
  Level: -74.8 dB                 DSP load: 0.4%
  Clipping: 0.0%                  Stream load: 0.0%

Graphics:                         132.4 FPS (7.6ms)
  CPU: main 7.6ms  render thread 2.3ms
  Batches: 70     Saved by batching: 19942
  Tris: 16.6M     Verts: 11.0M
  Screen: 1147x494 - 6.5 MB
  SetPass calls: 9            Shadow casters: 12
  Visible skinned meshes: 0  Animations: 0
```

Figure 3.16: Example visualization of 10 000 pedestrians with the second implementation

instead of gameobjects led to a further improvement, the parsing of the steps for the agents could be the bottleneck.

It can be seen that there is a 385% gain in fps from the original visualization to the first implementation and another 17% from the first to the second implementation at a pedestrian count of 100. This is a significant increase and it should be noted that the performance boost from the different implementations give the possibility of using the software much better as it is much smoother. The biggest factor why this happens is that the batch count was reduced from 1400 to just 17. That is a reduction of approximately 98%. When looking into the higher counts of pedestrians the software drops below a framerate of 30 from 1000 pedestrians onward. The new implementation can be used at framerates above 200 which is much more usable than the original implementation. The performance gain that was made here is significant as the human eye needs at least 30 fps to deem a visual representation as fluid which is given as seen in table 3.1.

When looking at 2500 pedestrians or more it can be seen that the performance using the first implementation starts to degrade drastically. The second implementation keeps up the significant performance gain so we have a 11900% increase from the original to the newest implementation at 2500 pedestrians. The newest implementation is also the

Pedestrian Count	100	1000	1050
Original Rendering	140 fps 1400 Batches	13 fps >10 000 Batches	12 fps >11 000 Batches
Gameobjects with batch processing	680 fps 17 Batches 198 saved	230 fps 19 Batches 1996 saved	220 fps 23 Batches 2092 saved
Agent list with batch processing	800 fps 17 Batches 198 saved	560 fps 18 Batches 1996 saved	520 fps 23 Batches 2092 saved

Table 3.1: Comparison between the fps and batches with the different implementations for batch processing with 100-1050 pedestrians. All of the data was gathered while all pedestrians were still visible. The hardware used is a desktop machine with an AMD FX 6300 six core CPU and an NVIDIA GTX 1060 GPU.

Pedestrian Count	2500	5000	10 000
Original Rendering	3 fps >21 000 Batches	1.3 fps >40 000Batches	nd
Gameobjects with batch processing	95 fps 27 Batches 4986 saved	20 fps 42 Batches 9972 saved	nd
Agent list with batch processing	360 fps 29 Batches 4986 saved	200 fps 43 Batches 9972 saved	100 fps 70 Batches 19 942 saved

Table 3.2: Comparison between the fps and batches with the different implementations for batch processing with 2500-10 000 pedestrians. All of the data was gathered while all pedestrians were still visible. The hardware used is a desktop machine with an AMD FX 6300 six core CPU and an NVIDIA GTX 1060 GPU. nd means that no data was gathered as either Unity crashed or did not initialize the agents in a timely manner

only one that is able to visualize 10 000 pedestrians while even having framerates of around 100 fps as seen in 3.15 and 3.16.

A problem that arises following the batch processing implementation is that the shaders and the social distancing representation do not work anymore. As the shaders rely on the empty agent prefab that hold the agent prefabs after instantiation is not used in the second version of the batch processing the positional data has to be transferred through the code. This can easily be done as the script that precomputes the data for the shaders can be altered slightly to get all arguments from the agent list in the *AgentController*. For the social distancing there is no simple way of implementing a new collision check as the *OverlapSphere* function uses the gameobjects and box colliders but as the implementation that still uses gameobjects is significantly more efficient than the original implementation which is already a big improvement in itself.

4 Conclusion and Future Work

Concluding all of the implementations, the social distancing representation offers a great tool in the context of the pandemic as nearly all countries have similar rules in place so the use case is not limited to a specific case. It offers a great opportunity for testing and researching including showcasing all of the findings to people without greater knowledge. Hotspot identification is also another useful tool in this context as it offers another way of presenting data to the untrained eye. The density and the voronoi shader create a good visualization as with the density the different coloration of the floor can be made clear. The voronoi diagram also demonstrates this in an easy way and additionally shows how the spaces for the agents change over time. The displaying of important simulation results can be really useful when showing certain important data at the end of the visualization as the user can then concentrate more on what can be concluded about it from the data. Lastly the performance improvement offers high quality visualization even with a significantly bigger amount of pedestrians.

These improvements to the existing Vadere animation software lets people with little to no knowledge about pedestrian simulations understand the depicted information more intuitively especially as the animation uses human looking objects that move and behave similarly to the real world.

While the density depiction and the social distancing representation aim to specifically work on visualizing specific scientific information about the simulation, improvements to the overall animation fluidity like the batching of agents aim to be able to visualize a great amount of agents while keeping a stable framerate.

The density and social distancing representation provide a useful tool set in the fight against COVID because they can be used to remodel public places and therefore can help when further trying to refine existing COVID rules to help to reduce infection cases even more. The 2D simulation does not give this possibility as it is designed to run the simulation and computes the output to fast for users to pick up and it is not as intuitive for inexperienced users.

Future works that can be made to these features include the combination of two versions of the density shader so that the floor is covered without having to multiply the distance with a smaller float. Especially the approach with the weight is very promising so this could be used as a base to further build on. The voronoi shader could be optimized to can be optimized such that the coloring of the polygons is differentiated better although this is not a big problem currently. Another improvement that can be made is only showing the borders of the polygons but this is probably easier to implement without a shader. This would also hold the advantage of being able to calculate the area for each pedestrian and could further be used for improvements different scenarios and in the density calculation using the surface area. The current displaying of important simulation results only works with the average evacuation time but this serves as a proof of concept and could be reused and upgraded to show more data. For overall visualization performance the batching of agents could be improved further by replacing the capsule meshes with static human meshes. This would help to improve the overall look and feel of the visualization as it would look more realistic as originally intended. Other than that the initialization can be improved further by optimizing the loading of the Vadere simulation output files as this could be the bottleneck because the rendering was already optimized to a significant extend. Also the social distancing should be implemented to work with the newest batching implementation because the *OverlapSphere* function cannot be used with it though good performance was already achieved with the second implementation of batch processing which can be used when social distancing is needed.

List of Figures

List of Tables

Bibliography

[1] Technical University Munich. URL: https://www.cms.bgu.tum.de/de/17-research-projects/99-momentum-v2-pedestrian-simulator-de (visited on 04/07/2021).

[2] University of applied sciences Munich. URL: https://gitlab.lrz.de/vadere (visited on 04/07/2021).

[3] URL: https://renderpool.net/blog/cpu-vs-gpu-rendering/ (visited on 04/11/2021).

[4] *Canvas*. Unity. URL: https://docs.unity3d.com/ScriptReference/Canvas.html (visited on 04/07/2021).

[5] *Draw call batching*. Unity. URL: https://docs.unity3d.com/Manual/DrawCallBatching.html (visited on 04/07/2021).

[6] M. Gödel, R. Fischer, and G. Köster. "Sensitivity Analysis for Microscopic Crowd Simulation." In: *Algorithms* 13.7 (2020). ISSN: 1999-4893. DOI: 10.3390/a13070162.

[7] *Graphics.DrawMeshInstanced*. Unity. URL: https://docs.unity3d.com/ScriptReference/Graphics.DrawMeshInstanced.html (visited on 04/07/2021).

[8] J. Kalathil. URL: https://gist.github.com/josephbk117/a0e06d34aadb43777a1e35ccde508551 (visited on 04/07/2021).

[9] J. M. Kang. "Voronoi Diagram." In: *Encyclopedia of GIS*. Ed. by S. Shekhar and H. Xiong. Boston, MA: Springer US, 2008, pp. 1232–1235. ISBN: 978-0-387-35973-1. DOI: 10.1007/978-0-387-35973-1_1461.

[10] B. Kleinmeier, B. Zönnchen, M. Gödel, and G. Köster. "Vadere: An open-source simulation framework to promote interdisciplinary understanding." In: *CoRR* abs/1907.09520 (2019). arXiv: 1907.09520.

[11] *MomenTUMv2: A Modular, Extensible, and Generic Agent-Based Pedestrian Behavior Simulation Framework*. Technical University Munich. 2016. URL: https://publications.cms.bgu.tum.de/reports/2016_Kielar_MomenTUMv2.pdf.

[12] *Object.GetInstanceID*. Unity. URL: https://docs.unity3d.com/ScriptReference/Object.GetInstanceID.html (visited on 04/07/2021).

[13] *Pedestrian Dynamics*. http://www.thp.uni-koeln.de/~as/Mypage/NonEq/pede.pdf. Universität zu Köln.

[14] *Physics.OverlapSphere*. Unity. URL: https://docs.unity3d.com/ScriptReference/Physics.OverlapSphere.html (visited on 04/07/2021).

[15] H. Plass. URL: `https://gist.github.com/h3r/3a92295517b2bee8a82c1de1456431dc` (visited on 04/07/2021).

[16] T. Selja. “Player immersion in video games Designing an immersive game project.” B.s. Thesis. 2018.

[17] *Separation of string to parse*. URL: `https://answers.unity.com/questions/688512/how-to-separate-parts-of-a-string.html` (visited on 04/07/2021).

[18] *Shader-Graph*. Unity. URL: `https://unity.com/de/shader-graph` (visited on 04/07/2021).

[19] N. Shiwakoti and T. Nakatsuji. “Pedestrian simulations using microscopic and macroscopic model.” In: (2005).