h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

– Fachbereich Informatik–

Enhancement of a Pedestrian Simulation Data Visualization

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von

Ebru Demir

Matrikelnummer: 744672

Referent	:	Prof. Dr. Elke Hergenröther
Korreferent	:	Björn Frömmer

DECLARATION

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, Juni 2021

Ebru Demir

ABSTRACT

Increasing urbanization is prompting an increasing number of pedestrians in cities. This affects areas with high pedestrian traffic, such as public facilities, public transport infrastructure and public events. In cases of temporary high crowds, it is particularly important that environments are designed to ensure pedestrian safety. To understand how large crowds move in environments surrounded by walls and stairs, pedestrian simulation programs can be used to calculate where pedestrians move along to get to their destination. Different tools exist to analyze the visualization of pedestrian simulation data, allowing the user's attention to be drawn to points of interest.
In this paper, we present different visualization concepts for pedestrian simulation data. Visualizations can be conducted for pedestrian movement speed, display of flow at a given location, flow visualizations to reveal potential movement patterns and visualization of dense locations.
From the list of all presented visualization concepts, the visualization of the density using a heat map is concretely presented and realized in this work. The heat map has a grid-like structure. As soon as pedestrians are located on cells of the heat map, their color signals whether there is a high or low density of pedestrians. A red coloration signals a high density of pedestrians, while lower density values are signaled with colors ranging from orange over yellow to green.
The main goal of the heat map implementation is to provide the user with a quicker insight into where points of interest are located in the geometry. It was possible to achieve this goal with the implementation.

ZUSAMMENFASSUNG

Die Urbanisierung geht mit einer zunehmend erhöhten Anzahl an Fußgängern in Städten einher. Dies betrifft beispielsweise öffentliche Einrichtungen, die Infrastruktur des öffentlichen Personennahverkehrs und öffentliche Veranstaltungen. In Fällen von zeitweise hohem Aufkommen an Personen ist es besonders wichtig, dass die Umgebungen so gestaltet sind, dass die Sicherheit von Fußgängern gewährleistet werden kann. Um nachzuvollziehen, wie sich große Menschenmengen in Umgebungen bewegen, die von Wänden und Treppen umgeben sind, können Fußgänger-Simulationsprogramme herangezogen werden. Programme, die zur Fußgängersimulation genutzt werden, simulieren, wo sich die Fußgänger entlang bewegen, um an ihr Ziel zu gelangen. Um eine Visualisierung von einer Simulation genauer zu untersuchen, können Analysewerkzeuge genutzt werden. Diese ermöglichen es die Aufmerksamkeit des Nutzers auf interessante Stellen zu lenken.
In dieser Arbeit werden verschiedene Konzepte für die Visualisierung von Fußgänger-Simulationsdaten vorgestellt. Visualisiert werden kann beispielsweise die Bewegungsgeschwindigkeit von Fußgängern, die Anzeige des Durchflusses an einer bestimmten Stelle, Strömungsvisualisierungen um potentielle Bewegungsmuster zu offenbaren und die Visualisierung von dichten Stellen.
Im Rahmen dieser Arbeit wird von den genannten Visualisierungskonzepten die Visualisierung der Dichte mittels einer Heatmap konkret vorgestellt und realisiert. Die Heatmap weist eine gitterartige Struktur auf. Sobald sich Fußgänger auf Zellen der Heatmap befinden, signalisieren sie durch ihre Farbe, ob eine hohe oder niedrige Dichte an Fußgängern vorliegt. Eine rote Einfärbung signalisiert eine hohe Dichte an Fußgängern. Niedrigerere Dichtewerte werden mit Farben, die von organge über gelb bis zu grün reichen signalisiert.
Das Hauptziel der Heatmap-Implementierung ist es, dem Benutzer einen schnelleren Einblick darüber zu ermöglichen, wo sich Interessante Stellen in der Geometrie befinden. Dieses Ziel konnte mit der Realisierung erreicht werden.

CONTENTS

LIST OF FIGURES

Part I

THESIS

1 INTRODUCTION

Increasing urbanization in several countries such as Germany[20], the UK[21], France[19], Russia[22] and China[18] causes a rise of the number of pedestrians in cities. This affects areas with high pedestrian traffic, such as public facilities, public transport infrastructure and public events. In case of particularly high pedestrian density occurring in those areas, it is very important to design the environment in a way that allows pedestrians to walk and remain safely in those areas. In order to determine how different numbers of pedestrians move surrounded by walls and stairs, pedestrian simulations come into play. Simulation programs take geometry files as input and let the user adapt several settings such as how many pedestrians are inside the simulation, their positions, movement speed and destination. The simulation then calculates where pedestrians move and considers that walls are not passable and the possibility of other pedestrians blocking the way, so that it may come to dense areas where pedestrians are unable to move for some time or only slowly. In order to examine the visualization of simulation results, analysis tools can be used to help shift the viewer's attention to interesting or dangerous areas of a geometry.

In this work, we propose the concept in relation to several visualization techniques on pedestrian simulation data. We provide methods to visualize the current movement speed of pedestrians by either attaching arrows to pedestrians that visualize their current movement speed through the length or color of the arrow. We also describe a trajectory based solution, in which the trajectories of pedestrians are colored section-wise to resemble their movement speed in different sections, showing the gradient of the movement speed. We also provide a concept to realize the display of the pedestrian flow in certain areas. To reveal movement patterns of pedestrians, it is also possible to conduct a pedestrian flow visualization, which can be carried out on a flat plane on the floor, showing patterns by summarizing movements with arrows to represent streams or a flow patterns. Finally, we present several possible approaches to visualize the density of pedestrians, one of which is a heat map. It is also possible to visualize density by using Voronoi diagrams.

From the list of all presented visualization techniques, we concretely present and implement the visualization of density using a heat map in this work. We used a grid-based approach, whereby a grid of regularly-distributed quadratic heat map cells will be located on top of the floor and stair meshes of the geometry at hand. A heat map cell will light up if a pedestrian is located on it. Additionally, the color of the cell indicates whether the cell

represents a high-density area. A red color indicates that the number of pedestrians is high, while a green color indicates a low pedestrian count. The main goal of implementing the heat map is to make it possible for the user of the visualization to identify interesting and dense areas of the geometry more quickly than without a heat map. This should also be possible when zooming out of the visualization.

The results show that the heat map enables the user to identify dense areas more easily than without it. The user can set the desired cell size according to his/her needs. The number of pedestrians considered as high can also be set by the user. If the pedestrian count is in between one and the maximum value of pedestrians per cell, the amount will be mapped to a color in the range of green over yellow and orange to red accordingly. The user can fit the settings to adapt the heat map for it to meet his/her analysis requirements.

The remaining part of the thesis is structured as follows´. The subsequent and second section provides an introduction to relevant subjects in relation to this work, such as the environment in which the application runs, an introduction to implementations prior to this work on which this work is based, and finally an introduction to the subject of data visualization. The third section presents related work. Section four presents the concepts in relation to several visualization techniques to enhance the visualization of pedestrian simulation data. Section five introduces all concepts related to the visualization technique heat map. In the sixth section, we provide implementation details regarding the realization of the heat map. In section seven, we present the results of the heat map visualization, while we discuss these results in section eight. Finally, in section nine we conclude the thesis with a summary and an outlook of possible future work.

2 PREREQUISITES

In this section, we explain various topics that are relevant for this work. In the first section, we provide an introduction to the environment in which this work is implemented, namely Unity. We also offer an introduction to various concepts that are specifically Unity-related or related to computer graphics. In the second section, we provide an introduction to JuPedSim, a simulation software that calculates the movement of pedestrians. In the final section, we introduce the general concept of data visualization techniques.

2.1 UNITY

Unity is a cross-platform game engine that has been developed by Unity Technologies[35]. It allows the users to implement 3D games for desktop, mobile, console and virtual reality platforms. It is also possible to develop 2D games. However, developing with Unity is not limited to games, as Unity applications can also be implemented in the fields of simulations, film, automotive and construction. Unity Technologies have launched several different Unity versions. The latest version is Unity 2021.1.0, which was released on 23rd March 2021.

Unity comprises a comprehensive software package that gives the users a large environment to realize their software. Unity aids their users with concepts such as physics calculations and a framework for mesh creation. Unity also provides a comprehensive website with documentations that show how the framework's functions can be used.

COORDINATE SYSTEM

Unity uses a left-handed coordinate system. Consequently, when positioned inside the origin of the coordinate system while looking towards the positive z-axis, the positive x-axis will be on the right side and the positive y-axis will points upwards. Figure 2.1 shows an image taken from the Unity manual in [34]. The image shows the positioning of the coordinate axes inside of the Unity environment.

MESH CLASS

Unity provides a mesh class that takes care of assembling and rendering meshes if the necessary data is assigned correctly. The lowest amount of data that needs to be specified for Unity to render the data is a list of vertices, triangles and UV coordinates.

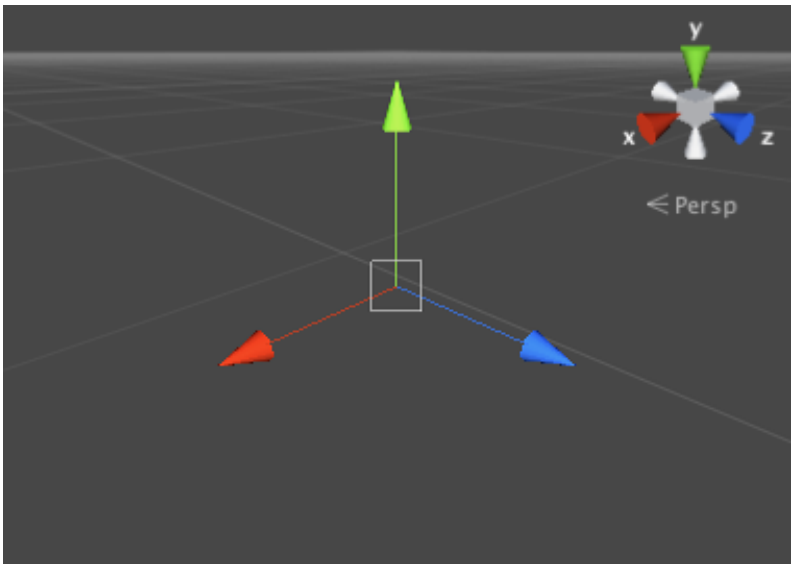


Figure 2.1: Excerpt of Unity's coordinate system, taken from [34].

Vertices represent the point data of a mesh. These points need to be connected via edges, whereby a surface will be spanned over the points. This surface can then be rendered by the game engine.

However, the connection of the points needs to meet one requirement, namely three of them have to be connected to a triangle. Triangles are the smallest primitives that are used in most 3D graphic programs, given that the three vertices that form a triangle always form a plane. With these triangles, it is now possible to create any desired shape by rotating and scaling the triangle and readjusting the position of the vertices. It is also possible to mathematically calculate a point intersection with a triangle very quickly. The graphics processing units (GPUs) of computers are optimized to undertake calculations with triangles.

In order for Unity to correctly assemble the list of vertices to triangles, the triangles list needs to be filled with according data. The triangle list comprises integers, which resemble a reference to the indices of the vertices in the vertices list. Three vertices together form a triangle. This means that three entries inside the triangles list always belong together and the number of indices inside the list has to be a multiple of three. When assembling triangles in Unity, it is important to consider that the triangle's vertices need to be connected in a clockwise order, otherwise the triangle will be facing towards the other direction and will only be visible from the back, but not from the front[27].

Finally, to determine in what way a mesh will be rendered by the game engine, we also need to determine which material will be applied to a mesh. The material includes a texture that will be applied to the mesh. A list of UV

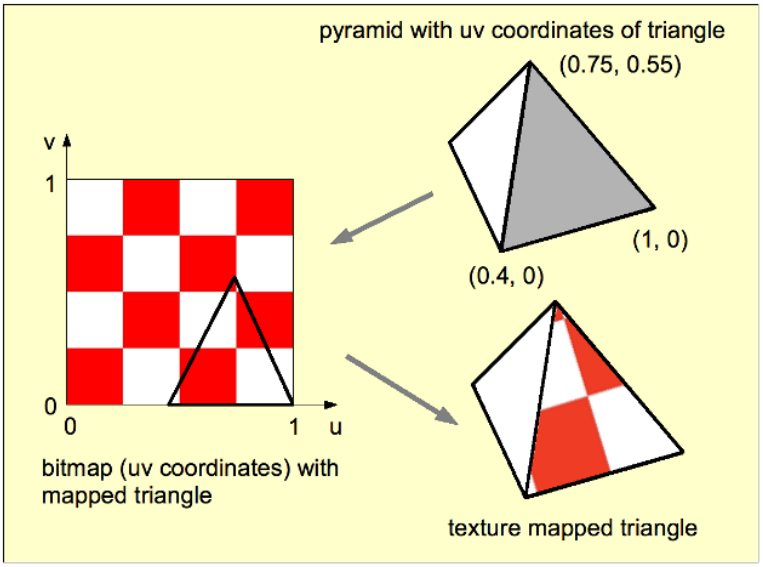


Figure 2.2: Texture mapping, taken from [26].

coordinates needs to be assigned to the mesh object.

In most cases, UV coordinates are two-dimensional coordinates. A texture is an image that will be put onto the mesh. It is a map that comprises pixels along the width and height, resulting in a two-dimensional image. To refer to a specific pixel inside of the texture (also called "texel"), UV coordinates come into play. The coordinates of the texels along the width and height of the texture are normalized into a range from zero to one. When addressing the texel that is positioned exactly in the middle of the image, the necessary UV coordinates are (0.5, 0.5). U describes the horizontal texel coordinate, while V is used to address the vertical location of a texel.

Figure 2.2 shows how a certain part of a texture is mapped onto a triangle. The example is taken from [26], showing how texture mapping is undertaken in OpenGL, although texture mapping is done similarly in Unity. In the image, we can see that the triangle in the upper right corner defines UV coordinates for every single corner of the triangle. These coordinates are used to look up which part of the texture is spanned over the triangle. With the UV coordinates used in this example, a triangular shape is taken from the texture and put onto the triangle of the mesh.

2.2 JUPEDSIM

Jülich Pedestrian Simulator (JuPedSim) is an extensible framework developed in Forschungszentrum Jülich. JuPedSim is introduced in [7]. It mainly comprises four loosely-coupled modules, which can be used independently.

The JPSeditor module provides a graphical user interface and allows the user to create a geometry file, which can later be used for simulating pedestrian movement. JPScore is the core module. After configuring some settings, such as desired destinations of the pedestrians, movement speeds and preferred route choices and providing it with a geometry file, it simulates the movement of pedestrians. After finishing the simulation, the module produces a trajectory file containing information about which pedestrian is located in what position at which time. JPSvis is used to visualize the geometry and trajectory file. Finally, JPSreport is a module that analyses the results from the simulation.

In this work, JPSvis is not used as the visualization framework. Instead, the geometry and trajectory files are both used to visualize the simulation data inside SumoVizUnity, a visualization software programmed in Unity. Further information on SumoVizUnity will be provided below in section 2.3.

2.3 SUMOVIZUNITY

In his master's thesis, Daniel Büchele considered various environments that are suitable for the post-visualization of pedestrian simulation data[3]. He decided to realize the post-visualization of pedestrian simulation data in the 3D game engine Unity and named the project "SumoVizUnity"[4]. The software is open-source software that may be extended as desired.

Mohcine Chraibi and Fabian Plum later adapted the project to JuPedSim simulation data[24]. SumoVizUnity can now take the trajectory and geometry files that have been created by JuPedSim and visualize the geometries and pedestrian movement.

In his bachelor's thesis, Marvin Weisbrod adapted the SumoVizUnity project, that had been customized by Chraibi and Plum[36]. With his implementation, it is now possible to use SumoVizUnity in a virtual reality (VR) version on the VR device Oculus Rift CV1[12]. He has also completed the update of the Unity version from version 5.5.0f3 to 2019.3, as well as adding optimizations to the project that allow the application to run more smoothly. This SumoVizUnity version will be used as the base for this work, after a removal of the VR-specific components is completed.

2.4 DATA VISUALIZATION

Data visualization is a field that represents abstract data graphically by using computer graphics-based methods. One of the most widespread examples of visual representation of data are world maps. A world map may visualize various kinds of data, e.g. borders of different countries. Figure 2.3 shows a world map that visualizes the overall composition of the earth. Through the image, we can see which areas are covered with water, the locations of

different continents and how high or low certain areas are. Alternatively, the data could also be presented in a list of coordinates and corresponding information. For humans, processing the data in this way would be less easy than in visual form.

Figure 2.3: Composition of the earth, taken from [5].

If we were more interested in finding out the average height of males by country, the same map can be taken and colored differently. Figure 2.4 shows a map of the earth from which we can extract the information about the average height of men for different countries. Humans are faster when extracting information from an image rather than looking through a list of countries with this information. Moreover, the map allows us to compare the data of several different countries very quickly without having to look through a long list, extracting the information from several countries and comparing them.

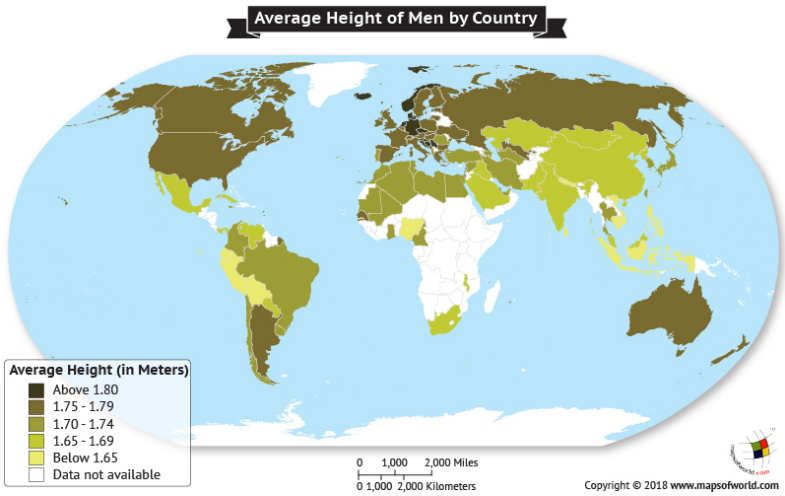


Figure 2.4: Average height of men by country, taken from [11].

With this example, we not only understand that data can be presented visually, but it also shows us that the same form of visualization can prepare different kinds of information by using different color-coding.

Data visualization is also very common in the fields of statistics. Different kinds of diagram to visualize the composition of data can also be considered as data visualization techniques. When it comes to data visualization, there is no limit to the subject area in which it can be used. It can help humans to show, explore, monitor and validate data. Data visualization can also be used to help to draw new conclusions from the data by disclosing details, that would not have been evident without visualization.

3

RELATED WORK

In this section, we present work from other authors that are related to the subject of pedestrian data visualization.

Matt Daniels released "Human Terrain: Visualizing the World's Population, in 3D" on "The Pudding" website in [13]. In this project, he visualizes data acquired from the Global Human Settlement Layer[6] with population density maps. Figure 3.1 shows a screenshot taken from the website in the European area. The user of the website can move freely to look at certain locations. The visualization makes it possible for the user to compare the world's population in 1990 to 2015.

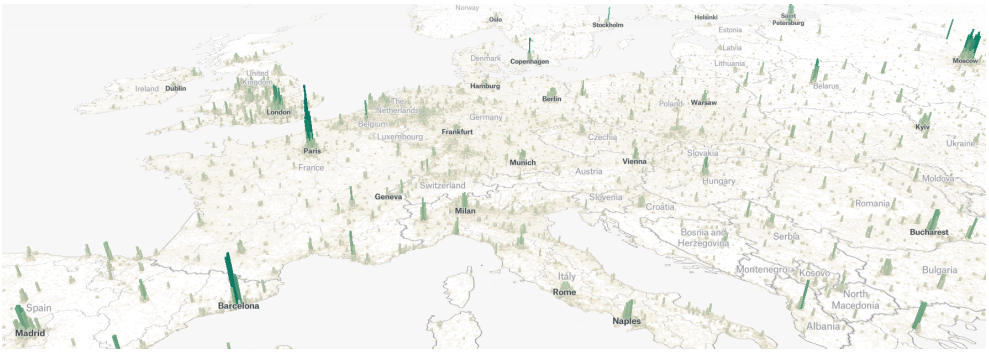


Figure 3.1: The Pudding - Human Terrain, taken from [13].

Morphocode published an article on visualizing pedestrian activity in the city of Melbourne[10]. In the article, they describe that in the future they will release a "Horizon Graph" that can reveal movement patterns of pedestrians. The goal of their work is to measure walkability and make it possible to identify measures to encourage people to walk outdoors. On their website, they mention that they are working on developing an interactive version of the data visualization that allows users to explore pedestrian activity by location and time. Unfortunately, we were unable to find any further information on this project.

The company acc:urate[38] extended the SumoVizUnity framework of Daniel Büchele[4] with functionalities to create virtual reality apps for Google Cardboard or Gear VR[25]. The viewer is immersed into the scene and can choose to either walk around freely in the scene or follow one pedestrian and see through his/her eyes. It is also possible to create a video or 360° video of flying along a specific tour through the scenery.

4

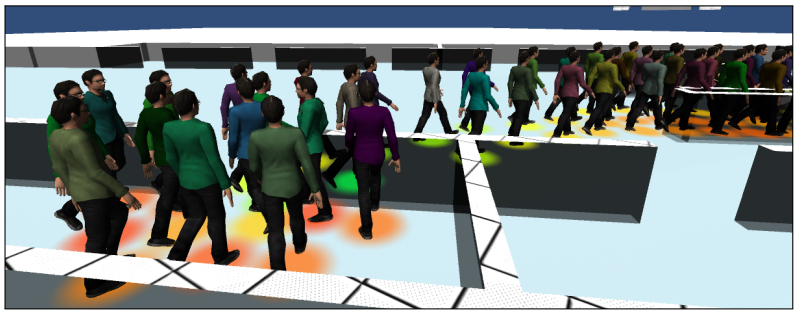
ENHANCEMENT OF A PEDESTRIAN SIMULATION DATA VISUALIZATION

In the scope of pedestrian data visualization, several measurements can be visualized. The following sections describe the concepts related to possible visualization techniques.

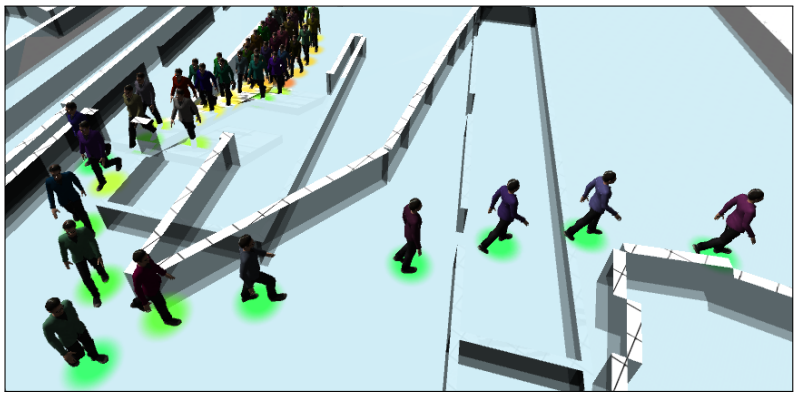
4.1 MOVEMENT SPEED/ VELOCITY

Movement speed - also called velocity - indicates how quickly a pedestrian is moving.

A visualization of movement speed has already been achieved in SumoVizUnity within the scope of previous work. A round area under the pedestrians indicate their high movement speed through a green color, while a red color indicates very low movement speed. Examples of this can be seen in figures 4.1a and 4.1b.



(a) Dense area with mostly low pedestrian movement speed.



(b) Pedestrians in the front are moving freely and fast.

Figure 4.1: Visualization of pedestrian movement speed through circles underneath the pedestrian.

Additional ideas to visualize movement speed include a three-dimensional arrow that can be attached to pedestrians and colored according to the current movement speed of the pedestrian, similar to the circles on the floor. Additionally, the arrow could be implemented in an interactive way in which it changes its length to suit the current movement speed. A higher movement speed would result in a longer arrow, while a lower movement speed would result in a shorter arrow. The arrow should be attached to the front of the pedestrian, showing the direction into which the pedestrian is moving. Figure 4.2 shows what this could look like inside SumoVizUnity in a prototypical way.

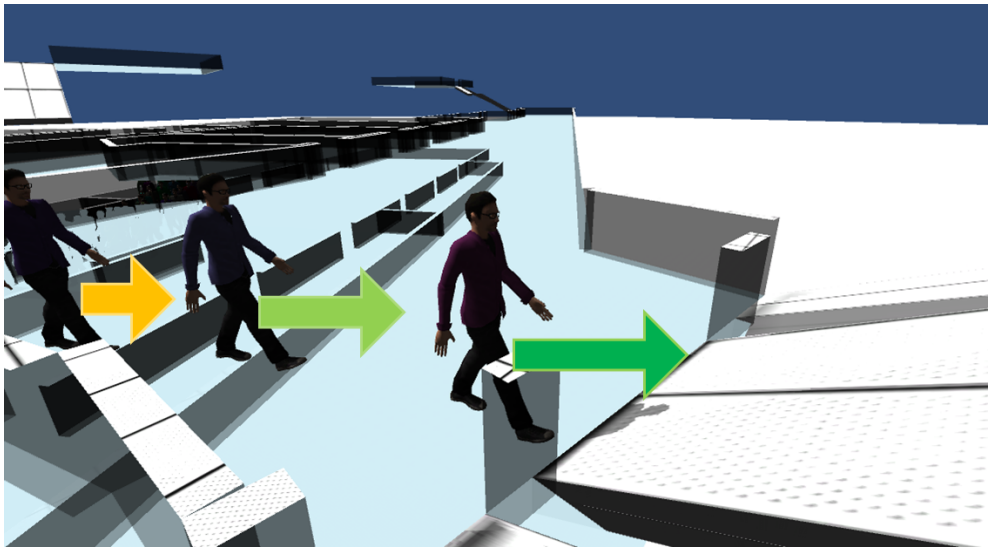


Figure 4.2: Possible visualization of arrows that display the movement speed of a pedestrian through its size and color.

However, in an area where many pedestrians stand close to each other, this method could make the visualization seem very overloaded. Therefore, when implementing this kind of feature, the user could have the option to toggle between whether single pedestrians have an arrow display or not. The user could then decide whether all pedestrians will display their arrow or only some, which he/she could select clicking on them.

A different approach to visualize the movement speed of a pedestrian could be a trajectory-based solution. Previous work has already implemented the option for the user to show the trajectories of all pedestrians. This option shows the complete course of a trajectory and uses different colors for different pedestrians to display which pedestrian a trajectory belongs to. Figure 4.3 shows what this looks like.

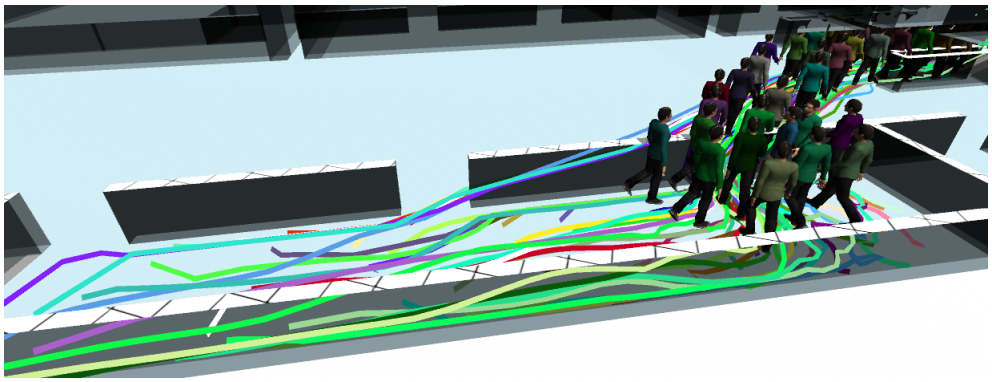


Figure 4.3: Trajectories of all pedestrians.

To visualize movement speed through the trajectories, there are two possible solutions. One possiblity is to color the trajectory section by section, depending on the speed of the pedestrian in that section. Figure 4.4 shows what this approach could potentially look like when implemented. Red areas resemble areas in which the pedestrian has a very low movement speed, while green areas display high movement speed areas.

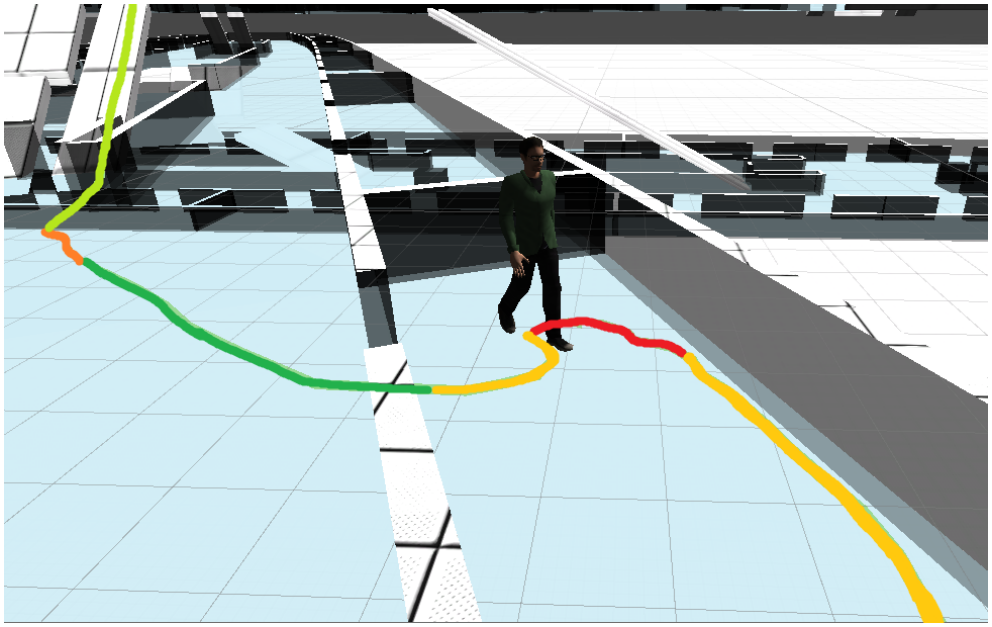


Figure 4.4: Trajectory of a pedestrian, colored section-wise to resemble the movement speed within different sections.

A different trajectory-based approach is to simply make the whole trajectory of a pedestrian change its color according to his/her current movement speed, which could seem like a more interactive option. When the pedestrian is moving quickly, the whole trajectory could be colored green, whereas the trajectory could be colored red when the pedestrian is moving slowly.

With both options, showing the trajectory of only one pedestrian at a time would be the preferable option over showing all trajectories of all pedestrians

at the same time. If the number of pedestrians within the simulation is high, trajectories start to overlap so that some trajectories will not be visible to the user. This is especially then the case when the trajectories start to become very similar, such as when all pedestrians have the same destination. An example of this can be seen in figure 4.5.

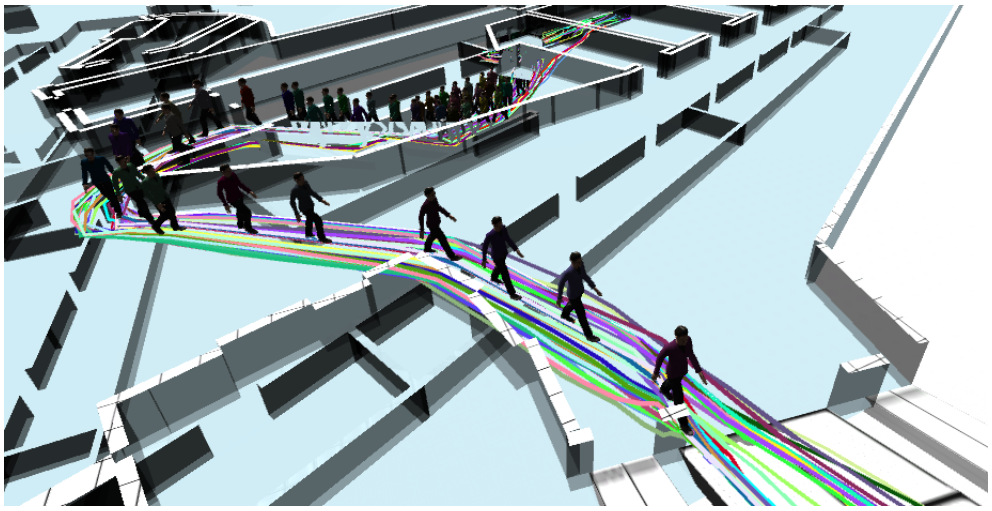


Figure 4.5: Showing all trajectories at once can cause several trajectories to be covered by others.

4.2 FLOW

The pedestrian flow rate is the number of pedestrians who pass a certain point per time unit. To implement this, the user should be able to draw a line into the scene, whereby the application can count how many pedestrians surpass this line per time unit. The value of the pedestrian flow rate can then be displayed close to the line drawn by the user. An additional visual feature would be possible by coloring the line differently depending on whether the flow rate is high or low. It could be colored red when no pedestrians have been passing the line for a certain time and green when many pedestrians pass the line per time unit. Which values will be considered as high or low will depend on the length of the line and its position. A decision on these details will have to be made when implementing this feature.

4.3 PEDESTRIAN FLOW VISUALIZATION

In contrast to the pedestrian flow - which was explained above in section 4.2 - the pedestrian flow visualization covers the visualization of the movement of pedestrians in a flow and stream manner. This visualization technique would try to find a pattern of the movements of pedestrians and visualize them with arrows on a flat plane on top of the floor areas. Figure 4.6 shows an example of a stream/ flow visualization. This technique could be used to reveal interesting movement patterns of pedestrians in high-density areas.

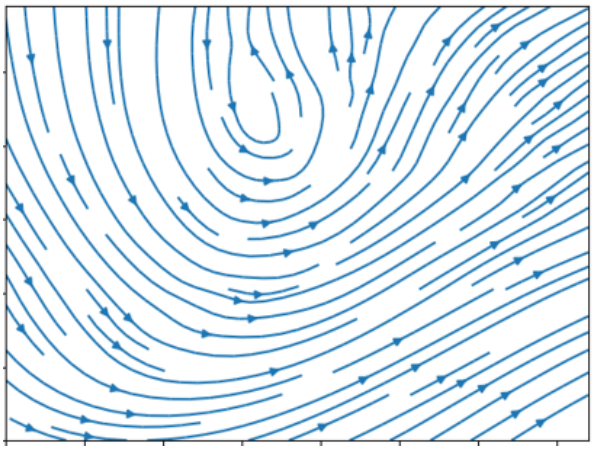


Figure 4.6: Flow visualization with flow arrows, taken from [8].

4.4 DENSITY

Density is a measure to indicate how many pedestrians are in a certain area. A high density value can indicate that certain areas are dangerous for pedestrians. To visualize density, different approaches can be considered.

Similar to the pedestrian flow described in 4.2, it is possible to offer the user the option to draw a square into the scene. The application could calculate the coordinates of the square and then determine how many pedestrians are inside the square and color it accordingly. Coloring the square red could indicate a high density, while coloring it green could resemble a low density inside the area.

A different approach to visualize density throughout the whole simulation is to use a heat map. Similar to the rectangle, which has to be drawn by the user, the whole geometry of the simulation could be covered by a grid of many squares. Through their coloring, each square could indicate how dense the area is at a certain time. By using a heat map instead of a single square, the user would not have to find interesting areas by him-/herself. Instead, the heat map would rather show the user where to find interesting areas. Due to the regular structure of a grid-based heat map, it would be possible to gain a useful impression of the more and less dense areas, even when zooming out.

A similar approach to heat maps that would also allow covering the whole area of the geometry to show dense areas is the Voronoi diagram. According to Steffen and Seyfried in [23], it is indeed suitable to visualize pedestrian density. Figure 4.7 shows an example of a Voronoi diagram.

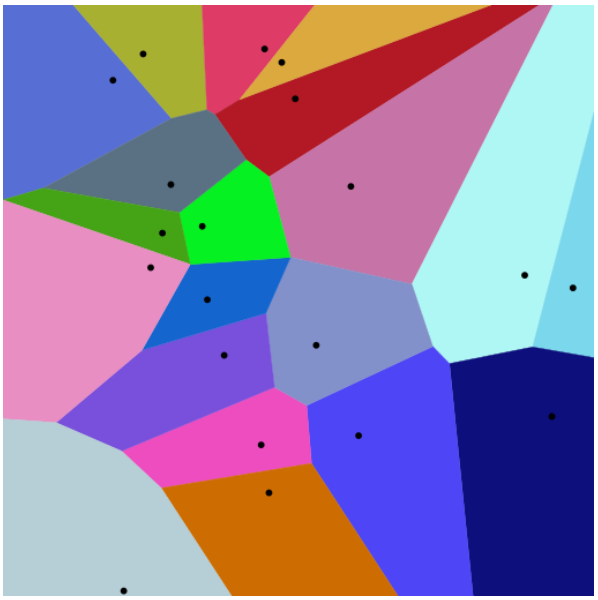


Figure 4.7: Example of a Voronoi diagram, taken from [2].

In this example, the Voronoi cells are each tinted in a different color, which makes it easy to visually separate the points. In a Voronoi diagram, areas are assigned to the point that is closest to the area, thus creating these cell shapes[16]. In the context of visualizing pedestrian density, each pedestrian would have his/her own cell. The denser an area, the smaller the resulting cell sizes would be. In contrast to a grid-based heat map, a Voronoi diagram would not allow zooming out of the geometry and still having a good impression of how dense areas are, since the cells might be too small to properly see them from far away.

Not all visualization techniques for pedestrian flow characteristics can be implemented within the scope of this work. The fact that heat maps are not restricted to only show the density locally in once area but instead work globally for the entire geometry at the same time makes the heat map a promising analysis tool to start the implementation with. The mesh-based structure of SumoVizUnity is also very suitable for a grid-based heat map, which could be built on top of the current mesh structure. Therefore, we decided on implementing the heat map as the first visualization concept from the list. Heat map-related concepts are presented in the following sections.

5

VISUALIZATION TECHNOLOGY: HEAT MAP

In the following sections, we will first provide an insight into different heat map types. Subsequently, we will describe the structure of the heat map that will be implemented for SumoVizUnity and then describe the influence of the visualization technique on the data structure. Finally, we will describe the creation process of the heat map.

5.1 HEAT MAPS AS A VISUALIZATION TECHNIQUE

According to Wilkinson and Friendly in [37], the usage of heat map dates back to at least 1873, when Loua summarized several social statistics about Paris in a hand-drawn matrix in [9]. The color of the cells range from white - indicating a low value - through yellow and blue to finally red, which finally indicates the highest value. Figure 5.1 shows the two pages of the heat map.

(a) Left page. (b) Right page.

Figure 5.1: Heat Map taken from Loua in [9] from 1873. It shows a summary of 20 districts of Paris and their corresponding characteristics.

A heat map is a tool that graphically represents data by converting it into color. It is supposed to provide a quick overview of various situations. Of course, it can also be used to gain a deeper meaning into topics. Different areas have different colors depending on the underlying meaning. In most

cases, the lowest values are resembled by a white color, while high values are represented by red. The colors in between go from blue, over green and yellow to orange. The color scheme is associated with temperatures, hence the naming "heat map". A heat map assists in directing the viewer to the most interesting areas through the coding with color.

There are different types of heat maps to represent various kinds of data.

DENSITY FUNCTION

Probably one of the most commonly-known kinds of heat maps is the density function visualization, which visualizes the density of point sets. As an example, figure 5.2 shows an image taken from Perrot et al. in [15], where they implement a density function to show very large scales of point sets. We can see in the images how the very densely-grouped points form a red cluster, while less dense points have a color ranging from blue to yellow. The density function is especially useful when handling large sets of points that are placed densely in certain locations.

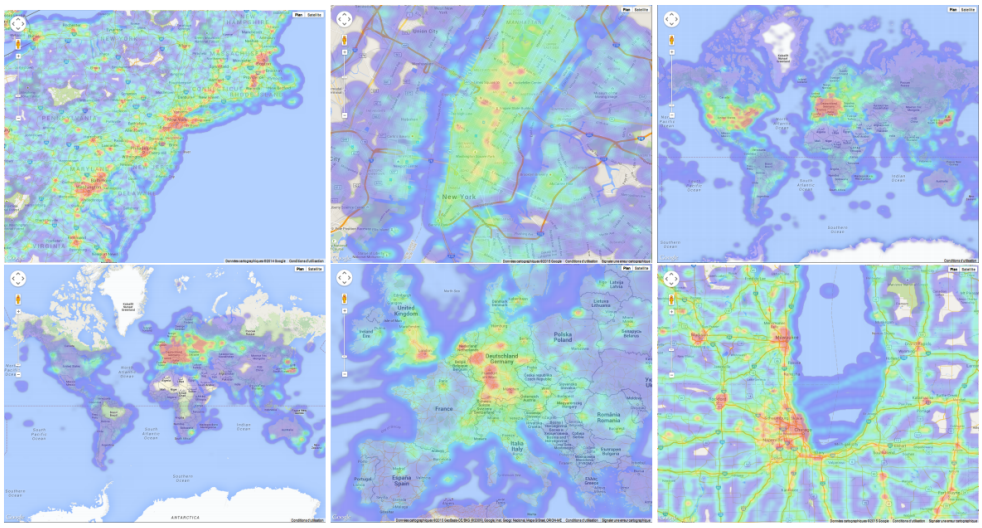


Figure 5.2: Density function to display very large point sets, taken from Perrot et al. in [15].

GRID-BASED HEAT MAP

A heat map can also be realized by dividing an area into a grid, creating equally large grid cells. Figure 5.3 shows an extracted image from a sports article from BR Wissen in [1], where this kind of heat map is used. It portrays the movement profile of a single player in a soccer game and can be used to conduct post-game analysis. In three-dimensional images, varying cell heights make it possible for the viewer to pay even more attention to interesting areas.

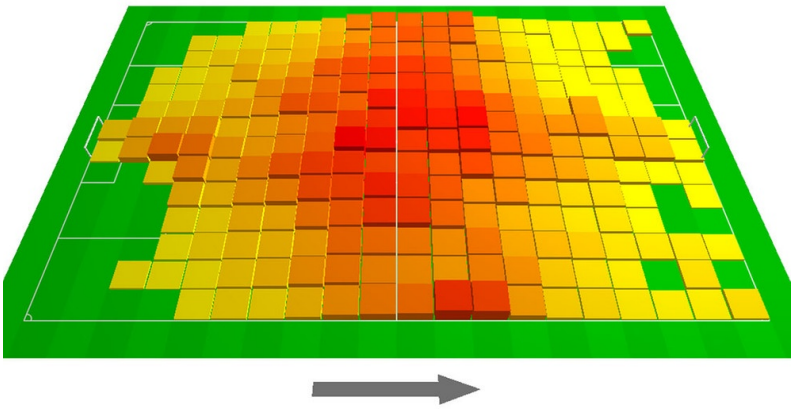


Figure 5.3: Grid-based heat map for a soccer field, taken from [1].

CLUSTER HEAT MAP

It is possible to compare data by inserting it into a heat map matrix and coloring the cells according to their values. In this case, the partitioning of the data is not locally defined like in the above example to analyse a sports game; instead, the order in which data appears can either be arbitrary or even forcefully picked to show an underlying pattern within the data. These heat maps are called cluster heat maps. Figure 5.4 shows a heat map that has been created in the fields of biology and genes.

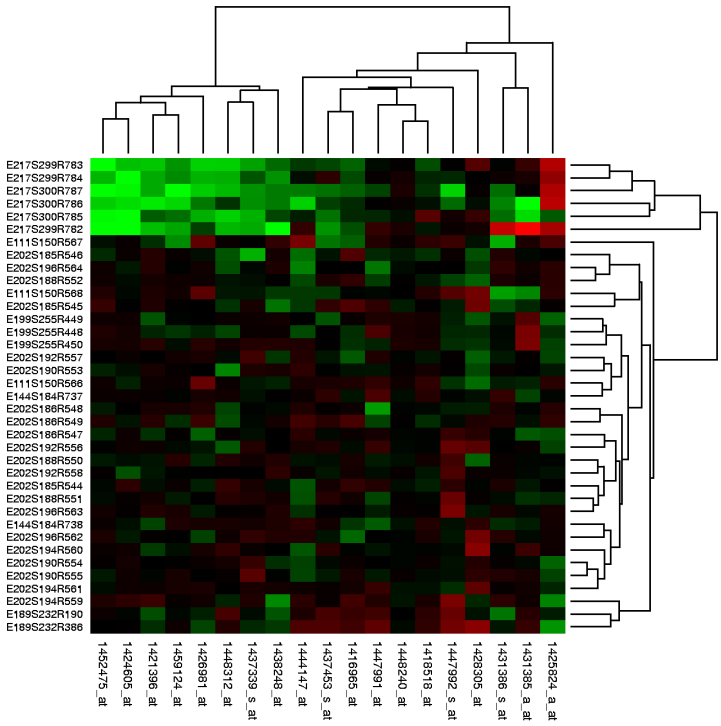


Figure 5.4: Cluster heat map to compare genes with samples, taken from [14].

ISOPLETH/CONTOUR LINE/ISOLINE HEAT MAP

Finally, another commonly-used type of heat maps is isopleth maps. These heat maps connect the points of a map that have the same value. The connection of the points is called isopleth, contour line or also isoline. Figure 5.5 shows an image taken from the website PrimeLocation in [17]. Red areas equal high property values of properties in the UK, while blue areas show the properties with the lowest value. Instead of having only dots of data like the density function or strictly dividing the area into a rectangular grid, the straight lines connect areas with the same values, whereby the shape drawn is then colored according to the value of the points.

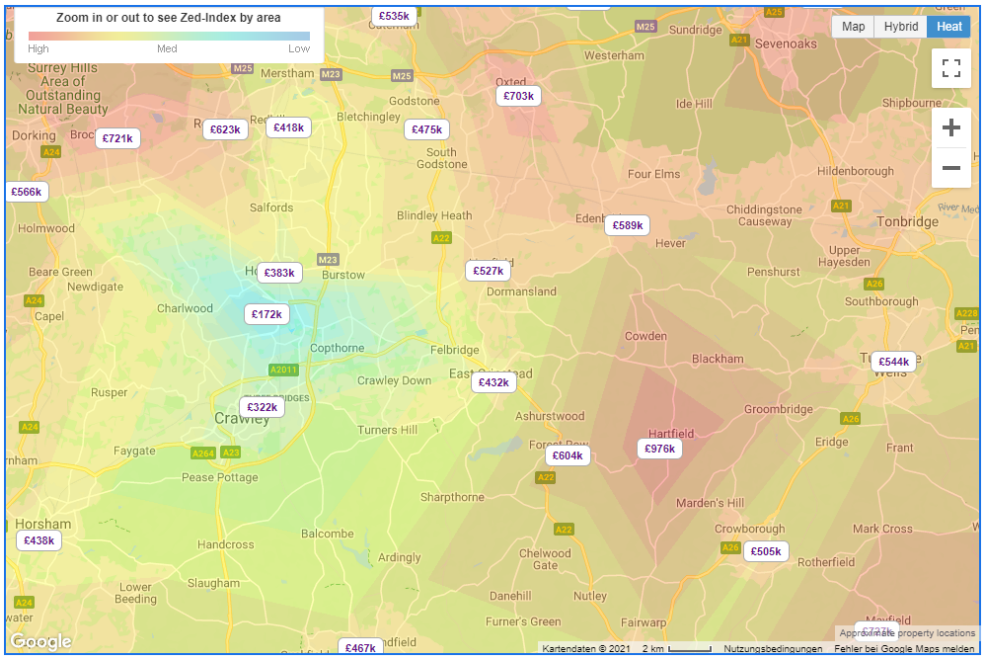


Figure 5.5: Isopleth heatmap of UK property values, taken from [17].

5.2 USING A HEAT MAP FOR SUMOVIZUNITY

In this work, a heat map will be implemented for the pre-existing SumoVizUnity project. The heat map is used as a visualization technique to visualize the density in certain areas.

Several of the aforementioned heat map implementations can be considered for SumoVizUnity. However, the heat map must meet the requirement, that it represents a visualization of density. Taking the density function to create a heat map is more commonly used in the scope of having many points that are very close together and may also be on top of each other. It is especially suitable for applications, where the viewer of the heat map wants to zoom quite far out of the heat map and still be able to see densely-clustered points. In SumoVizUnity, pedestrians cannot be clustered so densely in one space or on top of each other in one point, and therefore this approach is not ap-

plicable for our requirements.

On the other hand, when using a grid-based heat map, it resembles what the standard definition of an equation to determine the density also does taking a clearly-defined area and counting how many pedestrians are within that area. When using the density function, pedestrians would have to be handled as a set of independent points without the possibility to visualize the number of pedestrians in a certain area. Therefore, implementing a grid-based heat map is the most plausible approach in this work to correctly visualize the area-based density. Using this approach especially makes sense because previous work on the Unity project already created separate meshes that form the ground of the imported geometry. The pre-existing ground meshes can be used as a base to create a heat map mesh grid on top of it.

It is theoretically also possible to implement an isopleth heat map from the grid-based heat map, whereby areas with the same number of pedestrians in them could be connected with an isoline. Consequently, the heat map could appear smoother to the viewer. A major drawback would be that the heat map would then represent the density calculation in a wrong way, as smoothing the edges would incorrectly represent the considered areas. However, for the calculation of the density, the area needs to be strictly determined. Therefore, implementing an isopleth heat map was not considered as an option in this work, whereas a grid-based heat map would clearly show where the area of a grid cell starts and ends.

In this work, the heat map will comprise a grid that will be filled with smaller rectangular cells, which will be colored differently depending on how many people are on a cell. The heat map will be regularly tiled. Through the position coordinates of pedestrians, we can later determine which heat map cell is populated by how many pedestrians. The color of grid cells will give the user of the program a color-coded feedback of where many pedestrians are accumulated on a dense space. The cells will have a transparent color if not a single pedestrian is currently on it, which means that the heat map cell will not be visible to the user. The user can set which number is the maximum number of pedestrians that should be located on a single heat map cell. If this number is reached, the cell will be colored red. A single pedestrian on a cell will make the cell turn green. Any number of pedestrians in between one and the set maximum will be assigned a color from light green, over yellow to orange, depending on the concrete number of pedestrians. The color red is chosen for a more populated quad to indicate the possible danger of a high-density area, while green is used to indicate that the cell is not as dense and thus most probably not dangerous.

It is not possible to specify a fixed size for cells that will be useful for every simulation in advance. Moreover, the user him-/herself may want to vary the size of the cells. Therefore, the application has an option to set the size

of the grid cells.

As an additional feature, we do not want to color only the exact cells where pedestrians are located. In situations with a high density of pedestrians, the most dangerous areas are those where pedestrians are surrounded by further dense crowds of pedestrians, as the pedestrians who are located in the center cannot escape the core in which they are trapped. Moreover, the forces pushing from the outside onto these pedestrians "spread" and can be added up to the forces that are already affecting a neighboring area. We want to simulate the additional forces through neighbors by not only coloring the affected cells but also their direct neighbors. This feature will be implemented prototypically.

5.3 INFLUENCE OF VISUALIZATION TECHNIQUES ON THE DATA STRUCTURE

In previous work, a project has already been created that generates meshes for the floors, walls and stairs from the geometry data of the simulation. For visualizations techniques on a flat ground plane, such as the heat map or the pedestrian flow visualization, we need to extract flat plane meshes on which the visualizations can be conducted. Consequently, the present data structure needs to be adjusted accordingly.

In the following sections, the influence of the visualization techniques on the data structure will be explained. The floor meshes are discussed first in section 5.3.1, followed by the stair meshes in section 5.3.2.

5.3.1 *Floor Meshes*

Floor meshes are currently created by reading the geometry files per sub-room and extracting the location of the walls inside the geometry. Per sub-room, the bottom vertices of the walls are connected to form a floor.

Due to the architecture - which creates floor meshes per sub-room - it can occur that a floor mesh is created in a suboptimal way. One example of a perfectly rectangular floor is shown in figure 5.6. Since the floor is created based on the positioning of walls, it does not comprise one rectangle with four vertices in each corner; instead it comprises three distinct rectangles, one for each sub-room, with each four vertices. Therefore, instead of having four vertices for once rectangle, we have twelve vertices for one rectangle that comprises three smaller rectangles.

After extracting all sub-room floor meshes from the file, the floor meshes are merged into a single mesh. Figures 5.7 and 5.8 show an example of all floor-meshes being merged. In figure 5.7, we can see the mesh from above.

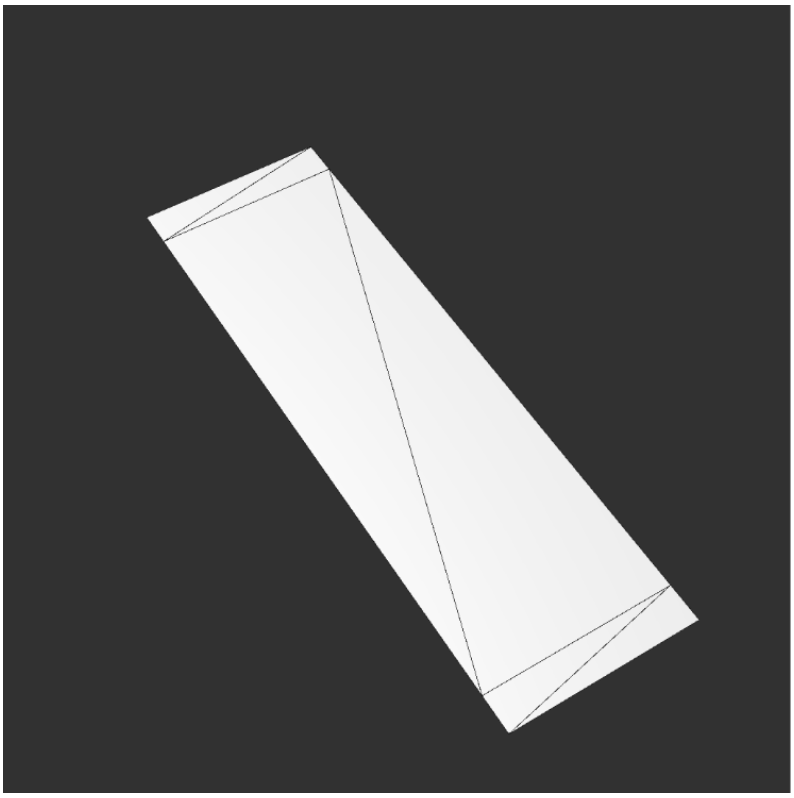


Figure 5.6: Three rectangular floor meshes that have been imported from a geometry file.

By merging the meshes, it is possible to derive a mesh with a highly complex shape.

Figure 5.8 shows the merged mesh from the side view. Here, we can see that not all vertices of the mesh are connected. The side view shows how different parts of the whole mesh are on different levels. Throughout one level, the y-values of the vertices do not change.

Keeping the current structure of the meshes - where all meshes are combined into a single mesh with not all vertices being connected - makes it impossible for us to extract a flat plane. We need to extract a flat surface so we can use it for various visualizations later. For example, when visualizing the movement of pedestrians through flow visualization, a flat surface on which the flow can be drawn on is necessary. Moreover, for the heat map we need a flat surface and rearrange the vertices to form a grid of cells. Therefore, we need meshes that have a defined starting point and end point. Furthermore, the vertices need to be continuously connected. A gap between the vertices or a cavity inside the mesh would not only make it impossible to create a regular grid of cells with the exact same offset between every cell for the heat map, but it would also make the tracking of cells that are populated by pedestrians very difficult and costly, which can cause performance issues.

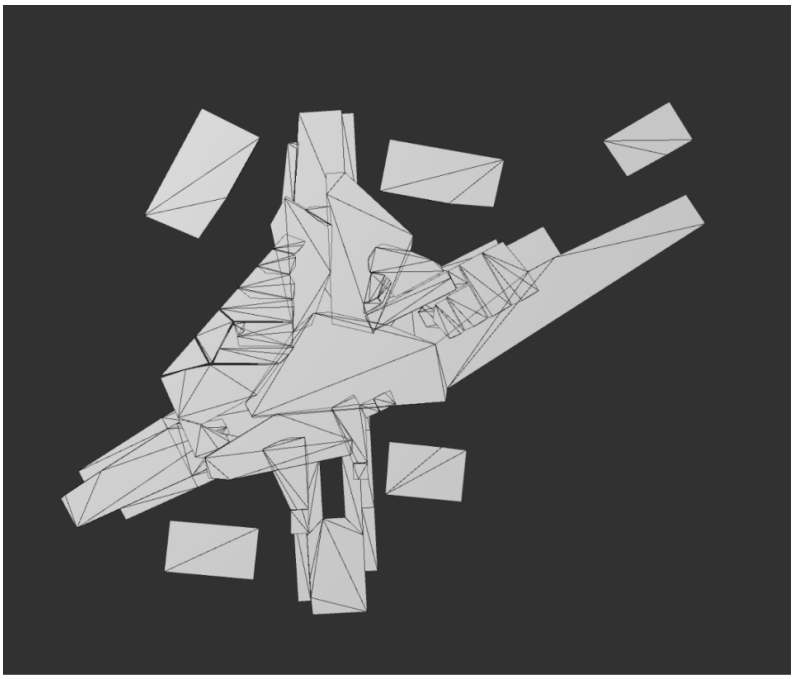


Figure 5.7: Example of a merged mesh from above.

Figure 5.8: Example of a merged mesh from the side view.

Therefore, we need to adapt the current mesh data structure to our needs.

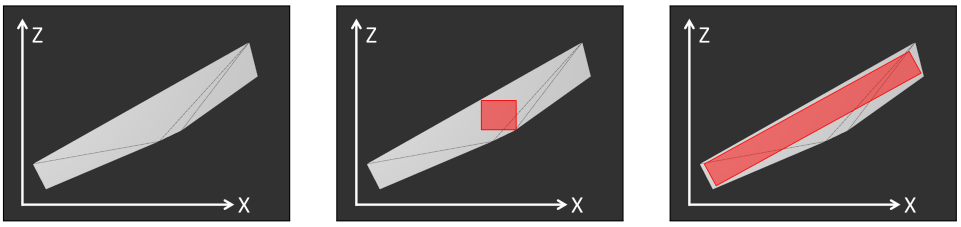
There are multiple possible approaches to proceed from the existing mesh data structure to a mesh structure that allows us to extract flat planes. We can roughly differentiate between using many smaller meshes - i.e. one plane for each sub-room mesh that is read from the geometry file - or grouping and combining meshes to several larger meshes and thus creating a plane for larger mesh groups. This combined mesh needs to satisfy aforementioned conditions, such as having a defined starting and end point and being continuously connected through the vertices. Both approaches will be discussed in the following sections.

5.3.1.1 *Extracting A Flat Plane From Every Sub-Room Mesh*

When creating many smaller flat planes for each sub-room, one possible approach is to fill the shape of the mesh as much as possible with quads without exceeding the boundaries of the mesh, i.e. "filling it from within". We look for the maximum possible plane area without exceeding the boundaries of the mesh. Consequently, part of the mesh may not be properly covered by the extracted plane. A different approach is to find an outer boundary that has a minimum size but covers the entire mesh safely. Both approaches are covered in the following paragraphs.

FINDING A MAXIMUM AREA WITHOUT EXCEEDING THE MESH BOUNDARIES

Filling the mesh shape as much as possible without exceeding the bounds can be achieved by looking for the largest possible rectangle that fits into the sub-room shape. Sub-rooms can be arbitrarily rotated. Not taking the orientation into account would lead to the creation of plane, that may not even come close to covering the floor of the sub-rooms. Figure 5.9 shows why taking the orientation into consideration can be very important. The first image in figure 5.9a, shows the original sub-room floor mesh from the above view. Figure 5.9b shows how the rectangular bounds of the plane may be positioned when trying to find the largest axis-aligned rectangle for the heat map. We can see that the plane would not cover the sub-room at all. Figure 5.9c shows how the rectangular bound of the plane could nearly fill the sub-room floor when considering the orientation.



(a) Original shape of the mesh.

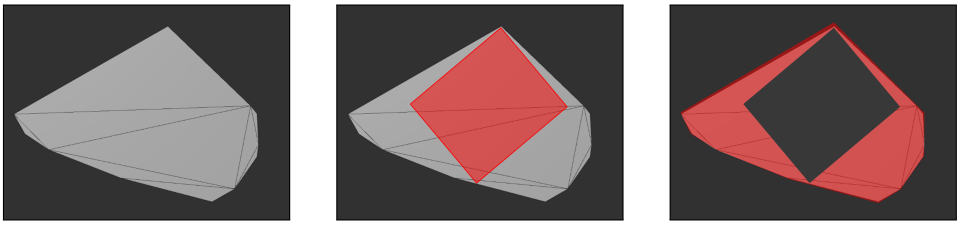
(b) Fitting the largest possible axis-aligned rectangle.

(c) Fitting the largest possible rectangle when considering orientation.

Figure 5.9: Fitting the biggest possible quad into a mesh shape with and without considering orientation.

Finding the best orientation of the rectangle is an optimization problem that is complex to implement and might be too time consuming when trying to extract the flat plane, leading to a slow performance of the application.
The example sub-room floor mesh in figure 5.9 can be considered near to rectangular, and therefore most of the mesh is covered with not much space being left out around the rectangle. However, generally sub-rooms have no limits to how complex their shape can be.

Figure 5.14 shows an example of a sub-room that is not rectangular. The second image shows how the largest possible rectangle might be fit into the mesh. Due to the shape of the mesh, there is a lot of space that would not be considered as part of the plane. The third image shows the amount of space that would be left unconsidered for the plane.



(a) Original shape of the mesh.

(b) Largest fitting rectangle inside the mesh.

(c) Remaining unconsidered space around the rectangle.

Figure 5.10: Fitting the largest possible rectangle inside a non-rectangular mesh.

A possible solution to fix the large amount of unused spaced would be, to fill the left space either with triangles or to continue using quads. Using triangles would make it possible to fill the whole space without any gaps. However, the use of different shapes would later make the heat map very inhomogeneous, which would defeat its initial purpose of providing a quick impression of high-density areas. Moreover, suddenly using different cell shapes would resemble the density of the area in an incorrect way. When showing density with a heat map, every single cell of the heat map should

cover the same area size for each cell's color to resemble the same density value. A smaller cell could have a smaller chance of having a certain number of pedestrians on it than a larger cell. Using quads instead to try and fill the remaining space as well as possible can also cause an inhomogeneous heat map if the quads are not placed in a way to continue the column or row of already-existing heat map cells.

Using either triangles or quads to fill the remaining space around the largest fitting quad is again an optimization problem, and both options are not trivial to implement. Moreover, especially when using triangles to fill the remaining space to its fullest, calculating the mesh's vertices can be very costly and time consuming, depending on how complex the sub-room shape is. However, not only the initial calculation of the plane can be very costly. Tracking the heat map cells during run time to correctly color them if a pedestrian is located on a cell can become a very complex problem. It leads to the heat map grid not having a clear start and end point with the same offset in between each column and row cell. Additionally, when finding a solution that makes it possible to correctly track pedestrian with this inhomogeneous structure, tracking could then still be very costly and might influence the performance of the application, making it unable to run smoothly by having a low frame rate.

FINDING MINIMUM AREA WHILE COVERING THE ENTIRE MESH

A less expensive and less complex way to obtain the flat planes for each sub-room floor while covering the entire sub-room floor mesh securely can be achieved by taking the bounding box of each sub-room floor mesh. Especially when taking the axis-aligned bounding box, this can be achieved very easily, since Unity provides us with corresponding functions in [29]. An axis-aligned bounding box around the sub-room floor of figure 5.14 can be seen in figure 5.11, where the mesh is surrounded by the red axis-aligned bounding box.

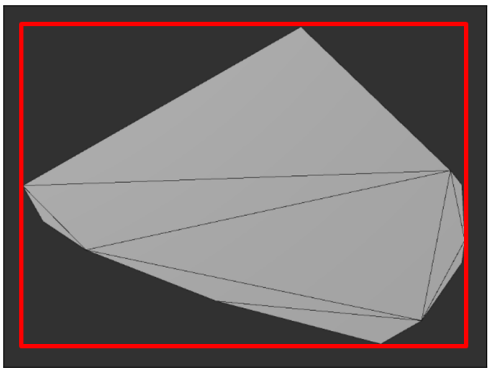


Figure 5.11: Axis-aligned bounding box around the mesh.

The axis-aligned bounding box does not create a bounding box of minimum size, since it does not take the orientation of the shape into account. It only creates the smallest possible rectangle with its sides being parallel to the co-ordinate axes. Using oriented bounding boxes instead to obtain a rectangular boundary of minimum size would again be an optimization problem that is

more complex to implement and more costly for the run-time performance. Moreover, a different orientation of the bounding boxes can later make the heat map seem inhomogeneous if the heat map cells are then oriented differently. This will be especially evident when differently oriented grid cells are immediate neighbors. In contrast to filling the mesh as well as possible from within and possibly not using all of the mesh's area, the bounding box will cover the entire mesh area. The disadvantage here is that it will most likely cover more space than the area of the mesh, depending on the shape of the mesh and the bounding box that can be created around it, due to creating a rectangular box to surround the whole mesh.

The disadvantage with creating a flat plane for each sub-room mesh is that depending on the geometry file at hand it is possible that several planes might overlap. In figure 5.12, we can see an example of sub-room floor meshes that are all on the same level. Areas where the meshes overlap are colored red.

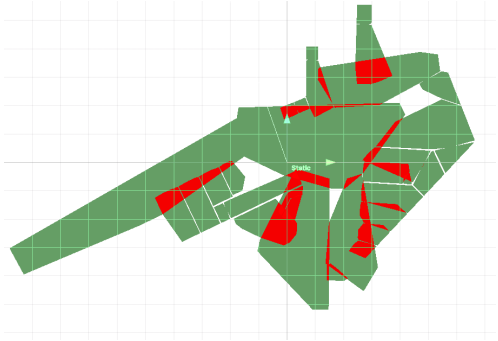
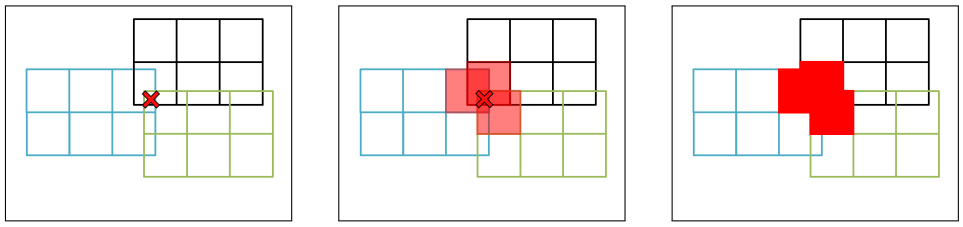


Figure 5.12: Combination all meshes on the same level. Red areas are those where meshes overlap.

Overlapping meshes leads to the problem of heat map cells also overlapping. Due to the individual shape of each sub-room, overlapping cells are most likely not overlapping all over, but instead only have certain intersection areas. When pedestrians enter multiple cells at once, all affected cells would change color, most likely leading to a non-quadratic shape appearing. Figure 5.13 offers an example, whereby in the first figure (5.13a) we can see a sketch of three heat map meshes colored blue, black and green. The red cross represents the position of the pedestrian. In figure 5.13b, the affected heat map grid cells are colored transparently red, whereby the darker red shades depict the areas of intersection. Finally, figure 5.13c shows the resulting shape when all grid cells are colored after a pedestrian steps on all three cells at once. We can see that the shape is not quadratic.



(a) Three overlapping meshes. The red cross depicts the pedestrians position.

(b) All affected cells are highlighted.

(c) Coloring all affected heat map cells results in odd shape.

Figure 5.13: Overlapping of grid cells results in non-rectangular combination of grid cells.

In order to obtain a quadratic shape, we could only color one cell by either taking the first cell that matches the pedestrian position or implementing a prioritization, where e.g. only the cell where the pedestrian is closest to its center is colored. Nevertheless, to the user of the application it could seem like the cells are "jumping around". Depending on the implementation, this would happen as soon as the pedestrian moves and a different cell is more prioritized. This could again be counter-intuitive to how a heat map is expected to behave.

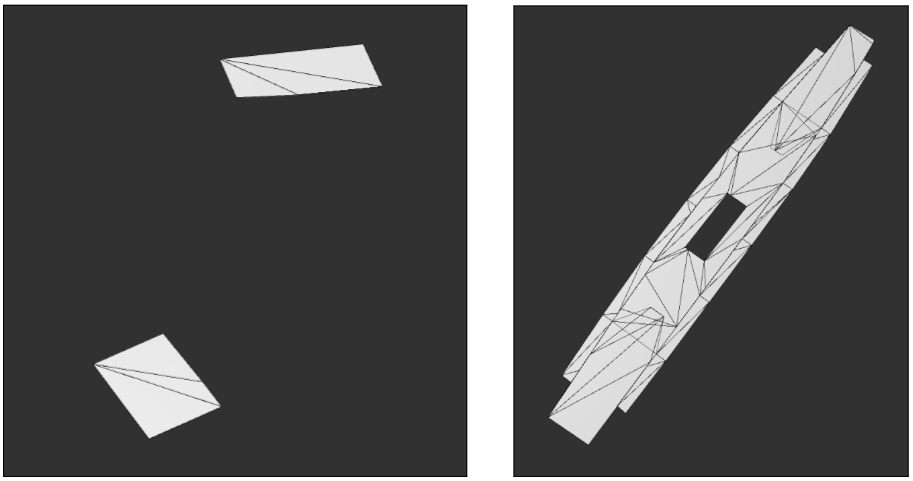
Since this overlapping of heat map cells can happen at multiple locations at once, the heat map would appear inhomogeneous. The degree to which the overlapping happens depends on the geometry file at hand. This varying behavior of the heat map - with heat map cells sometimes being placed with an overlap, without the user being able to identify any regularity - could lead to it behaving counter-intuitive for the user of the application. To solve this problem, the following section describes a different approach.

5.3.1.2 *Extracting A Flat Plane From Combined Groups of Meshes*

A different approach that prevents overlapping of meshes is to combine the meshes. However, as mentioned above, combining *all* meshes together would make it impossible to extract flat planes where all vertices of one mesh are connected to the mesh without any gaps. Thus, we need to strategically combine certain meshes.

When retrieving the sub-room floor meshes from the geometry file, all vertices within one mesh are always on the same level, i.e. they have the same y-value throughout the whole mesh. Connecting all meshes that have the same y-value throughout all vertices leads to a combined mesh, where all vertices have the same y-value and thus all vertices are on the same flat plane. Consequently, we do not have the problem of the meshes not being connected due to the difference in height. Nonetheless, not all vertices might be connected within this resulting mesh. In figure 5.14a, both meshes are on the same level, but they are located on different ends of the geometry. On

the other hand, the vertices of the mesh in figure 5.14b are not located in a way that they are complete disconnected. Instead, the mesh has a gap in the middle, in which no triangles have been defined.



(a) Gap between different mesh parts. (b) Gap inside of the mesh.

Figure 5.14: Combinations of meshes that lead to gaps.

To define our flat plane, we can again consider to fill the shape of the combined mesh either from "within" by finding the maximum possible area without exceeding the mesh's boundaries, or we can create a minimum possible bound around the mesh that will securely cover the whole mesh.

Filling the mesh from within does not work with the combined mesh. When again looking at figure 5.14a, we can see that we cannot create one flat plane from the two meshes that have been combined without exceeding the mesh's boundaries. Moreover, in figure 5.14b we would have a gap within the mesh, which would make it impossible to create continuous defined plane without gaps.

Therefore, when using the combined mesh, the usage of a bounding box around the mesh is the only approach that works. As mentioned above, taking the axis-aligned bounding box is less costly and complex than taking an oriented bounding box, while it also leads to a more homogeneous heat map later.

The main disadvantage of taking a bounding box - especially when not taking an oriented bounding box - is the amount of space, that will be considered as part of the flat plane, but does not intersect with the original shape of the mesh. However, depending on what visualizations the flat plane is used for, this will not be noticeable for the user of the application. For the visualization of the pedestrian flow, the flat plane will be used as an anchor to correctly place the flow of the pedestrians. For the heat map, cells in which no pedestrian is currently located will have the texture coordinate of the

transparent part of the texture. Consequently, the redundant grid cells will not be visible for the user.

On the other hand, when using axis-aligned bounding box, one advantage is that we can securely cover the whole mesh. Moreover, Unity provides a function that delivers the axis-aligned bounding box without us having to calculate it[29]. The same orientation of the axis-aligned bounding box of every group of meshes ensures that every single heat map cell has the same orientation. A possibly different orientation of heat map cells per level of combined meshes may make the heat map inhomogeneous and thus counter-intuitive for the user. Finally, one major advantage of using an axis-aligned bounding box per combined mesh is that due to the guaranteed rectangular shape of the bounding box, it will be especially easy and not expensive to implement the heat map. With the axis-aligned bounding box, we have a definitive starting point of the rectangle and a definitive end point. We can easily take the bounding box and re-arrange its vertices to form a regular grid of cells for the heat map.

5.3.2 *Stair Meshes*

This section covers the extraction of flat planes for stair meshes. Given that stair meshes have a different structure than floor meshes, this needs to be handled separately.

In the geometry files, stairs comprise four vertices. These four vertices represent a rectangular plane, which resembles stairs by having a lower positioned edge and a higher positioned edge. In the Unity project, the four vertices are extended by additional vertices to make the meshes of stairs into a parallelepiped, i.e. a three-dimensional parallelogram. An example of a stair mesh can be seen in figure 5.15. The upper and lower surface have the same x- and z-coordinates, while the y-coordinate's value differs by 0.3.

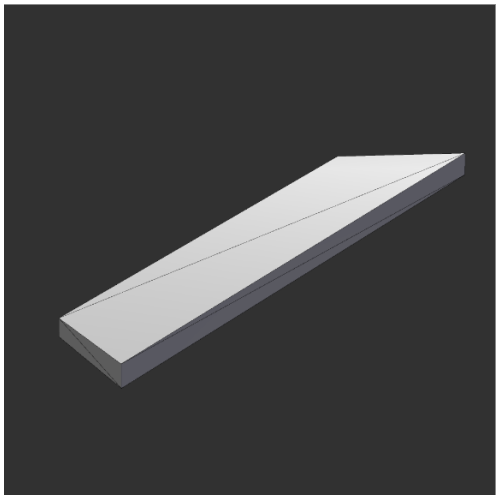


Figure 5.15: Example of a stair mesh.

After creating all stair meshes, they are combined into a single mesh, which then comprises all stair meshes. Figure 5.16 provides an example of a combined stair mesh.

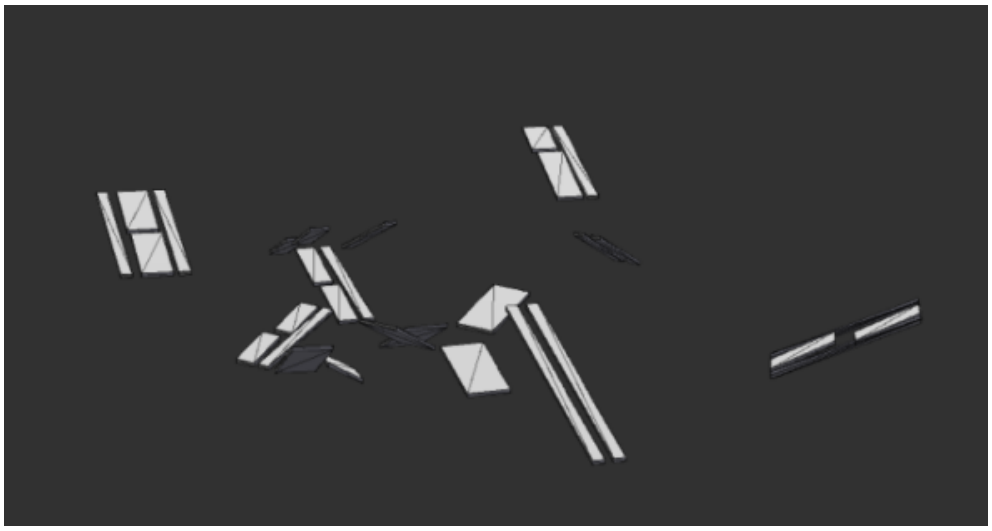


Figure 5.16: Example of a stair mesh where all stair meshes have been merged.

Again, like with the floor meshes, it is not possible to extract a flat plane with all stair meshes being combined, since their vertices are not connected either. Possible solutions would be to either find a rule according to which a grouping of the meshes can be made or extract flat planes from individual stair meshes.

In figure 5.16, we can see that some stair meshes seem to be grouped in a way, e.g. the left-most stairs. A zoom into these four meshes is depicted in figure 5.17. Especially the two stair meshes in the middle seem to belong to the same stair. Grouping this type of stair meshes to then create a flat plane through both meshes seems like a possibility upon first glance. However, a look at figure 5.18 shows that this is not the case.

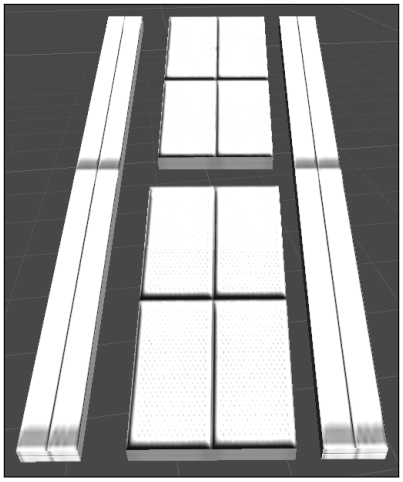
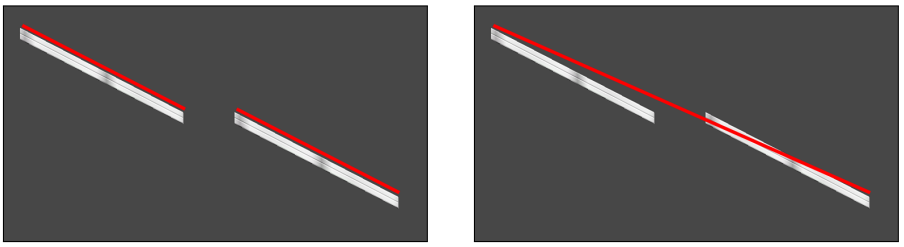


Figure 5.17: Group of four stairs.



(a) Extracting flat planes separately per stair mesh.

(b) Extracting one flat plane for two consecutively placed stair meshes.

Figure 5.18: Extracting flat planes from stair meshes.

Figure 5.18a shows how the flat planes would be positioned if they were created per individual stair mesh. Figure 5.18b on the right side shows how the flat plane would be positioned when trying to span the plane in a way to start at the bottom of the lower stair mesh and end at the top of the upper stair mesh. Since both stair meshes cannot be connected via a virtual straight line from the side view, it is also not possible to create a flat plane that connects both stair meshes continuously.

Moreover, when again taking a look at figure 5.17, an addition drawback of connecting all four stair meshes is that heat map cells would be placed in the gaps, where pedestrians are unable to walk.

Since these problems can be solved easily by creating flat planes per individual stair mesh, this is the approach that we went chose.

Although stair meshes always have a perfectly rectangular shape, taking an axis-aligned bounding box does not work for the stair meshes as opposed to floor meshes, for several reasons. First, axis-aligned bounding boxes do not take the orientation of the element into consideration. However, stair meshes can be arbitrarily rotated around the y-axis. In figure 5.19, the left rectangle represents a rectangle whose sides are parallel to the x- and z-axis. The rectangle in the middle has been rotated.

The axis-aligned bounding box outlines the rectangle on the left side perfectly, while the axis-aligned bounding box on the middle creates a that is larger than the rectangle it is outlining. On the right, the rectangle is rotated back into its original position. We can now see that the bounding box became larger after rotating the rectangle.

The surface of the stairs represents a perfectly rectangular area, so a perfectly rectangular bounding box could in theory be generated around it if the bounding box was oriented. However, since stairs can be arbitrarily rotated around the y-axis, an axis-aligned bounding box generates a non-perfect bounding box around a stair if its sides are not parallel to the x- and z-axis.

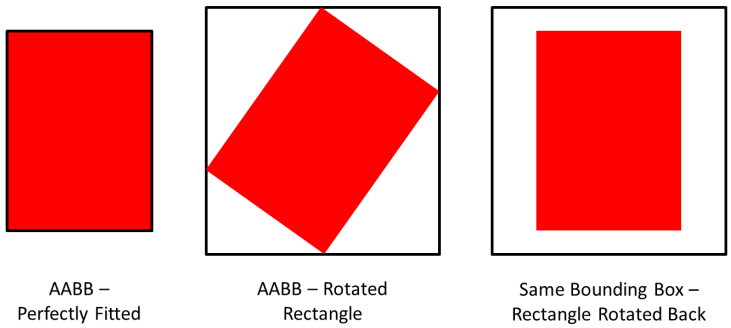


Figure 5.19: Axis-aligned bounding box around a non-rotated and rotated rectangle.

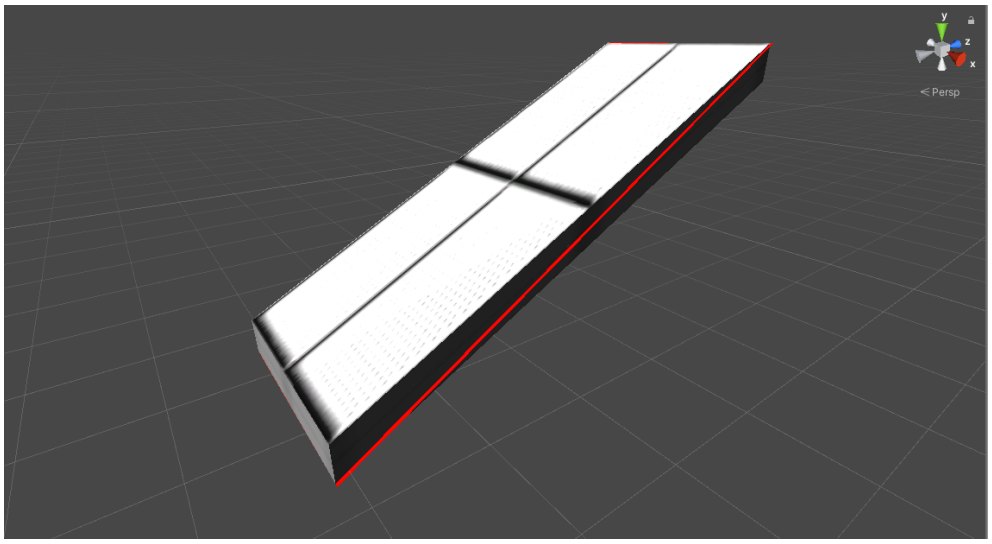


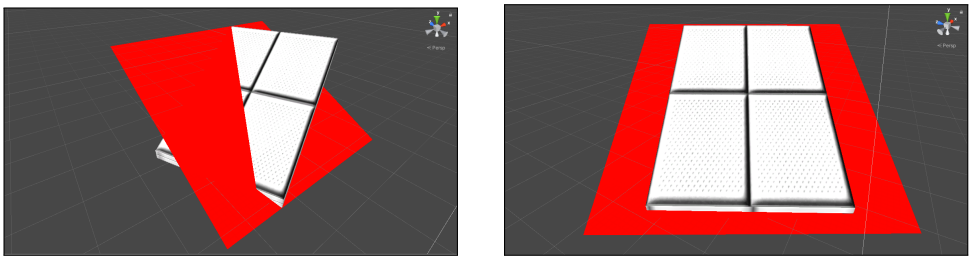
Figure 5.20: Bounding box of stair taking the "wrong" min value, leading to it being placed inside the stair mesh.

Hence, it cannot be used to extract the needed flat plane.

Second, taking the axis-aligned bounding box considers the whole three-dimensional stair mesh when creating the bounding box, i.e. not only the top rectangular surface of the parallelepiped, which we actually want as the flat plane. The red line in figure 5.20 shows the resulting plane if an axis-aligned bounding box was taken. The plane's bottom edge would start at the lowest part of the stair instead of starting at the lowest part of the top surface of the parallelepiped. This causes the plane to be trapped inside of the stair mesh instead of on top of it.

Both mentioned problems combined with a real stair mesh example can be seen in figure 5.21. Figure 5.21a on the left shows how the plane would be generated with the axis-aligned bounding box. In figure 5.21b, we rotated the exact same plane by hand to match its orientation to the stair mesh. We can see that the plane does not match the surface of the parallelepiped at all. It

lies crosswise inside the stair mesh due to its incorrect orientation. Through the "back-rotation", we can see that the bounding box has a different height and width than the top part of the parallelepiped. Finally, we can also see how the plane is not floating above the surface but instead cutting through the mesh. As mentioned above, this is due to the fact that the minimum y-value of the bounding box takes the whole three-dimensional parallelepiped into account and not only its top surface.



(a) Axis-aligned bounding box around a rotated stair - original positioning.

(b) Axis-aligned bounding box around a rotated stair - after aligning the bounding box to the stair mesh through rotation.

Figure 5.21: Axis-aligned bounding box of a rotated stair mesh.

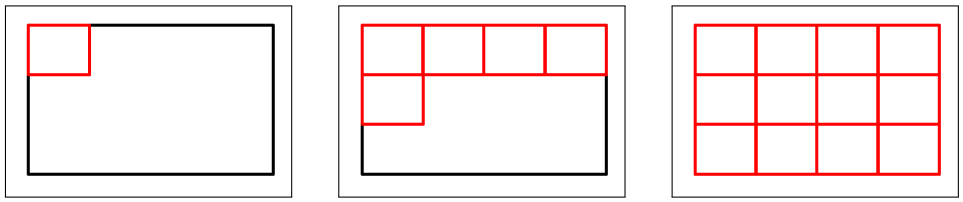
Consequently, instead of being able to use an axis-aligned bounding box, we have to take the concrete vertex coordinates of the stair meshes into consideration to obtain the oriented bounds of the stair meshes. By comparing the vertex coordinates, we can identify the corner vertices of the top surface and subsequently extract the flat plane.

Having extracted the flat planes from both the floor meshes and stair meshes, in the next section will explain how the flat plane will be used to realize the heat map.

5.4 CREATION OF THE HEAT MAP

For the heat map creation process, the previously-extracted flat planes serve as the basis. All planes have a rectangular shape, which makes it possible to create a grid of rectangular cells.

In order to create heat map meshes, we take the rectangular bound of the plane and chose a point from which onwards on we start to define vertices. In this work, we chose the top left corner of the rectangle as our starting point. The first cell will be created here, as can be seen in figure 5.22a. By moving towards the top right corner of the rectangle, we distribute the vertices regularly to finish our first row of rectangular grid cells and then continue with the next row, as shown in figure 5.22b. Consequently, we finish the heat map mesh by filling the whole plane with grid cells, as portrayed in figure 5.22c.



(a) First cell will be on the top left corner.

(b) After completion of the first row, the second row starts below the first cell.

(c) Completion of the heat map grid.

Figure 5.22: Heat map creation process.

Creating heat map cells for the floor meshes can be achieved very easily. Given that the flat plane of the floor meshes are axis-aligned, we can simply move along the x- and z-axis to calculate the vertex coordinates. We only need to add the size of one cell to the x- and/or z-value of the top left vertex of the currently-created cell, depending on which corner vertex is created next. For the top right vertex, we only need to add the cell size to the x-value of the top left vertex. For the bottom right vertex, we need to add the cell size to both the x- and z-values of the top left vertex. Finally, for the bottom left vertex, we only need to add the cell size to the z-value of the top left vertex. When creating floor heat map meshes, the y-value of every single vertex is the exact same given that floor meshes always reside on the same level without changing their height.

However, this method of calculating the positions of the vertices does not apply to stair meshes. The sides of the stair mesh's planes are not always aligned with the x- and z-axis. Moreover, the y-values of the vertices vary throughout the plane, starting at a lower y-value and gradually increasing towards the top edge of the stair. Therefore, the vertex positions cannot be calculated by moving along the axes. Instead, we have to calculate the position of the corner vertices. The corner vertices of the flat plane provide us with all necessary information to calculate all positions of the vertices that together form the grid of heat map cells. This is possible by undertaking vector multiplication and addition. The exact calculations will be discussed later in section 6.2.

The calculation of the heat map vertices is slightly more complex for stair meshes than for floor meshes implementation-wise, but it is not too complex to affect the run-time performance. Given that the calculation of the vertices that needs to be used for stair meshes also works for the floor meshes, it also makes sense to use this approach for the floor meshes. This avoids unnecessary, redundant code, since both approaches serve the same goal.

When creating the heat map vertices, it is possible to align the vertices so that the heat map grid cells are either rectangular or square. Making the grid cells rectangular has the advantage that is it possible to fill the heat map perfectly without the need for edge cells to be smaller, since the height

and width could be adjusted as needed. However, given that a geometry has more than one level of floor meshes, it is highly improbable that the chosen side lengths will fit for each individual floor mesh. The option of separately setting the grid cell side lengths for each heat map on a different level would also not be suitable as an option. Different grid cell sizes would then consider different area sizes when calculating the density and later coloring the heat map. This would make it impossible for the user to quickly and correctly identify areas with high density when different cell sizes are used. Therefore, filling the heat map with square grid cells is the approach that we decided to implement.

6 REALIZATION OF THE HEAT MAP

In the subsequent paragraphs, the implementations related to the heat map are presented.

When creating the heat maps for both the floors and stairs, a rectangular shape is regularly filled with quads. The extraction of the rectangular bound and the necessary pre-processing will be described in section 6.1. In section 6.2, the creation of the heat map meshes that yields in the rectangular bound being filled with a grid of regular quadratic cells will be explained. Section 6.3 describes how the heat map meshes will be stored and managed, in order them to react to pedestrians stepping on heat map cells. Afterwards, in section 6.4, we explain how heat map cells are colored according to the number of pedestrians who are currently located on them. Finally, in section 6.5 we explain how we optimize the run time by changing the routine of how pedestrian movement is tracked.

6.1 EXTRACTION OF THE BOUNDARY OF THE FLAT PLANE

This section explains how the corner vertices of the flat plane are retrieve, specifically the vectors describing the rectangle's top left, top right and bottom left corner, which are needed for the following steps. The process of finding these three vectors differs depending on whether floor or stair meshes are at hand. The first paragraph covers floor meshes, while the second paragraph covers stair meshes.

FLOOR MESHES

As mentioned in 5.3.1, floor meshes that are on the same height should be grouped. The floor meshes that are on the same height share the same y-value, which depicts the height value in the coordinate system at hand.

To group these meshes, an empty C#-dictionary is created. A dictionary is a collection that stores key/value pairs. The dictionary uses floats as keys and stores a list of the type `Mesh` per key. The y-value of the meshes are used as the key to group the meshes by the same height. Moreover, at this point we have access to the list that contains all floor meshes that have been created after reading the geometry file. When iterating through the list, the y-value of the first vertex of the current mesh is read. Since the y-value does not differ throughout the whole floor mesh, it does not matter which vertex's y-value of the mesh is consulted. If the dictionary does not already contain a key with the same value as the current y-value, a new entry is created and the current mesh is the first to be added. The list being empty means that so

far there is no other mesh on the same level. If the dictionary already has an entry, the current mesh is simply added to the existing list. This leads to all meshes that share the same y-value and therefore residing on the same level being grouped in the same list entry of the dictionary. By iterating through the dictionary, every list of meshes on the same level can now be combined to a single mesh.

To get the rectangular bounding box around the combined mesh, Unity provides us with the function `Mesh.bounds`, which returns the axis-aligned bounding box of the combined mesh[29]. The vertices describing the top left, top right and bottom left corner of the bounding box - which are needed for the following step - can now be extracted from the Mesh.bounds-object.

Now that we have the correct bounding box vertices for the floor meshes, we can create the actual heat map meshes for all floor meshes.

STAIR MESHES

As stated above in section 5.3.2, we cannot use the Mesh.bounds to get the rectangular bounding box of the stairs. Instead, we have to take the concrete vertices of the stair meshes into account to obtain the oriented bounds of the stair meshes. Each stair mesh comprises four vertices per surface of the three-dimensional parallelogram. This results in a total of 24 vertices given that there are six sides. For the heat map, only the top surface's four vertices are needed because the heat map will be a two-dimensional plane, which will be sitting on top of the stairs, similar to the floor meshes.

Only eight vertices of the total 24 stair vertices represent distinct coordinates. because the surfaces of the stair mesh share the same corner vertices. Therefore, only keeping the distinct vertices' values results in eight remaining vertices. Within these eight remaining vertices, there are four pairs of vertices, which share the same x- and z-coordinate. The y-coordinate differs by 0.3. Taking the vertex with the larger y-value of each pair results in the vertices that form the upper surface of the stair mesh. These vertices represent the correct bounds for the rectangle of the heat map. The next step is to ascertain which of these remaining four vertices represent the top left, top right and bottom left vertices of the rectangular bound.

Stairs always comprise a bottom edge and a top edge, which are connected with side edges to form a rectangle. Figure 6.1 portrays what is considered the left, right, bottom and top edge of the stairs. The top edge is always the edge whose vertices have a larger y-value, while the bottom edge comprises two vertices whose y-values are smaller than the top edge's vertices. Moreover, the surface of the rectangle whose normal is pointing upwards with a positive y-value is always considered as the front side of the stairs.

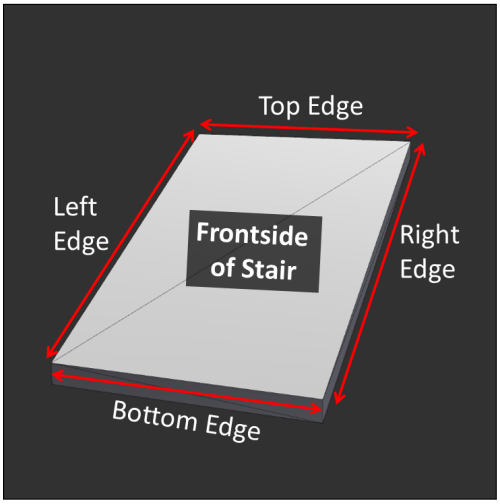


Figure 6.1: Orientation of a stair with a description of its sides.

Next, we want to ascertain which two vertices of the remaining four vertices represent the top edge of the stairs and which represent the bottom edge of the stairs. For this purpose, we can simply sort the four remaining vertices by their y-value, sorting from smaller to larger values. The first two values will be the vertices that describe the bottom edge of the stairs and the last two will represent the top edge of the stairs.

Next, we need to find the find the correct top left vertex, from which point we will later start creating the heat map's quads. Choosing the correct vertex is crucial. In Unity, the triangles need to be defined clockwise for them to be visible from the front [27]. When creating the triangles counterclockwise, triangles will be facing into the opposite direction and they will only be visible from below.

If the top right corner is passed as the starting point, the bounds within which the quads are created are still correct. However, starting on the top right, the quads will be created row-wise towards the left. The top part of figure 6.5 illustrates how the quads will be created if the top left is correctly chosen as the starting point. The triangles will be created clockwise and from left to right. Therefore, the heat map will be visible from the front.

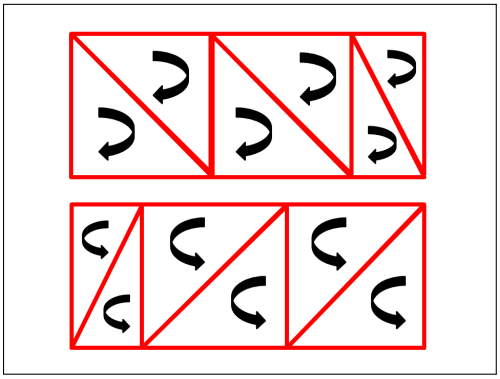


Figure 6.2: In the upper illustration the top left corner is chosen correctly as the starting point, while in the lower image the top right corner was chosen as the starting point, leading to the triangles being defined counter clockwise.

In the bottom part of the image, we can see what happens if the top right is chosen as the starting point. The triangles will be created counter-clockwise and from right to left, which means that they will only be visible from the back of the stair. Figure 6.3 shows what it looks like when the stair heat map is created on the backside of the stair from within the application.

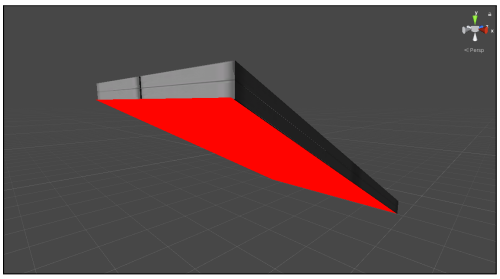


Figure 6.3: Example of the heat map mesh being created on the backside of a stair.

Given that stairs can be arbitrarily rotated around the y-axis, there are two cases that we need to consider when trying to find the top left corner of the stairs. In the first case, the top edge and bottom edge may each be parallel to the z-axis. The x-value will then not change along the top and bottom edge. To correctly identify the top left vertex, we need to find out whether the bottom edge of the stairs is facing towards the right or left side from the world space's perspective. If the x-value of the bottom edge's vertices is smaller than that of the upper edge's vertices, this means that the bottom edge of the stair is starting on the left side and the top edge is facing towards the right side, as can be seen in figure 6.4. The top left vertex will then be the vertex of the upper edge that has the larger z-value.

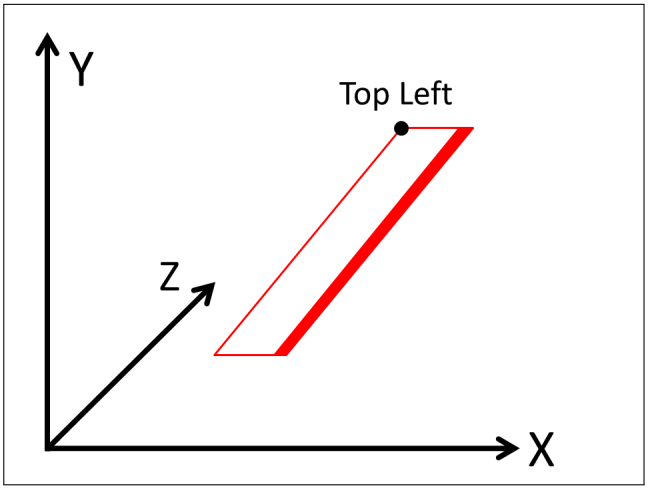


Figure 6.4: Staircase rises to the right. The lower edge has a smaller x-value than the upper edge.

On the other hand, if the x-value of the bottom edge's vertices are larger than the x-value of the top edge's vertices, the stairs will be facing towards the left. Figure 6.5 shows that the top left vertex will be the vertex of the top edge that has the smaller z-value.

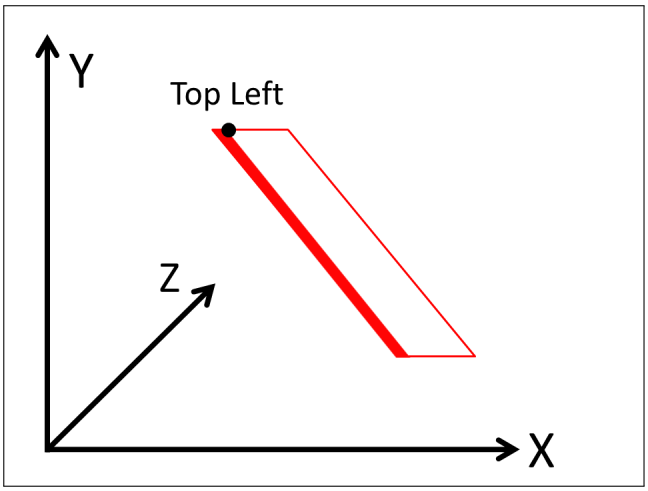


Figure 6.5: Staircase rises to the left. The lower edge has a larger x-value than the upper edge.

In the second edge case, the top and bottom edge are each parallel to the x-axis, which means that along each edge the z-value does not change. In this case, we cannot identify the left vertex of the top edge by comparing the z-values along the edges, as we did before. Figure 6.6 visualizes the case of the stair rising to the back. Based on the top edge having a larger z-value than the bottom edge, we conclude that the top left vertex of the top edge will be the vertex with the smaller x-value.

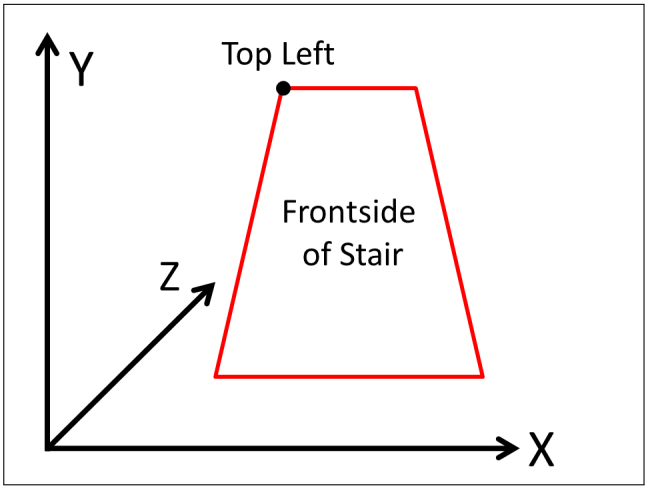


Figure 6.6: Staircase rises to the back. The lower edge has a smaller z-value than the upper edge.

We can also have a stair that rises to the front, as portrayed in figure 6.7.

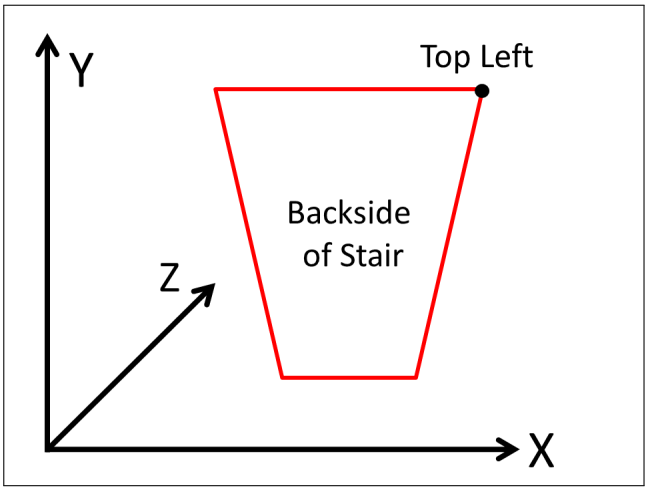


Figure 6.7: Staircase rises to the front. The lower edge has a larger z-value than the upper edge.

In this case, the top edge vertices have a smaller z-value than the bottom edge's vertices. The top left corner of the rectangle will then be the vertex of the top edge that has the larger x-value.

If the stairs are neither parallel to the x-axis nor to the z-axis, either one of the two processes can be used to determine the bounding vertices.

After locating the top left vertex, we also need to identify the top right and bottom left vertex. Depending on which vertex of the top edge was identified as the top left vertex, the other remaining top edge vertex is the top right vertex. For the bottom left vertex, the same rule applies as for the top left vertex of the top edge. If the top left vertex was the vertex with the smaller or larger x- or z-value, the same rule applies to the bottom left edge when

compared with the remaining bottom edge vertex.

Now that we have the correct bounding box vertices - like with the floor meshes - we can create the actual heat map meshes in the following sections.

6.2 CREATION OF HEAT MAP MESHES

We now have the three vectors describing the bounds of the rectangle that will be used for the heat map. The bounds are represented by the top left, top right and bottom left corner vectors of the rectangle. The three gray points in figure 6.8 represent these three bound vertices. With these, we can start to create the heat map meshes.

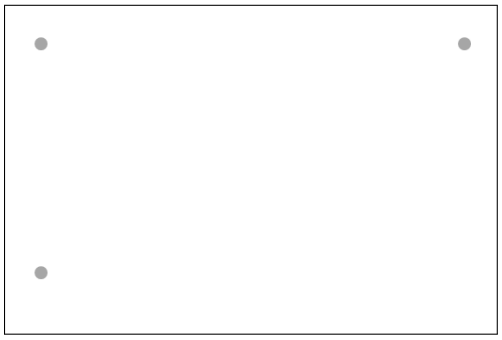
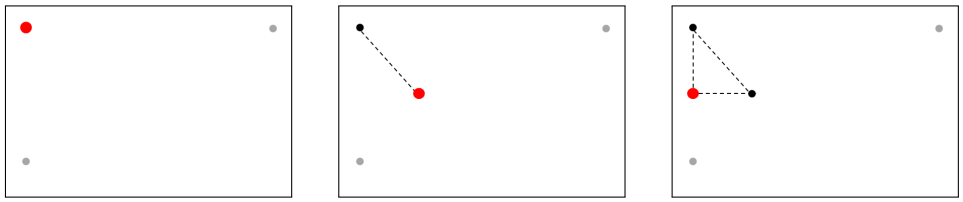


Figure 6.8: The three gray points representing the top left, top right and bottom left bound of the heat map rectangle.

As explained in 2.1, when creating a new mesh object, the lists of vertices, triangles and UV coordinates are the minimum elements that need to be defined.

VERTICES

The basic idea behind the creation of vertices for the heat map is to start at the top left corner of the rectangular bound and calculate where the vertices lie when the quadratic cells are created row-wise, ending with the last quad filling the bottom right corner of the rectangle.



(a) The first vertex is the starting point in the top left corner of the rectangle.

(b) The second vertex is located on the right side and below the starting point.

(c) The third vertex is below the starting point.

Figure 6.9: Draft of the three vertices for the first triangle. Red points represent the most recently-added vertex.

In detail, this means that the very first vertex is always the top left corner of the bounding box. This point will be called the "starting point", and in figure 6.9a it is represented by the red point. In the following figures, the red point displays the vertex that was most recently added to the mesh's list of vertices.

In Unity, a triangle's vertices need to be defined clockwise for them to be visible. Therefore, the second vertex is several units to the right and below the first vertex. This can be seen in figure 6.9b. The third and final vertex of the triangle is several units to the right below the first vertex, which is displayed in figure 6.9c. The distance between the vertices is defined by the size of the quads. It can be set by the user of the program. These three vertices conclude the vertices for the first triangle. The dashed lines within the figures only display the respective belongings of the vertices to the triangles. The actual connection of the vertices to triangles follows below in section 6.2.

The concrete calculation of the vertices is undertaken by using the starting point, top right and bottom left vectors. Subtracting the top right vector from the starting point vector results in a directional vector, which will be called the "width vector". Through subtracting the bottom left vector from the starting point, we obtain the directional vector "length vector". Both are portrayed in figure 6.10.

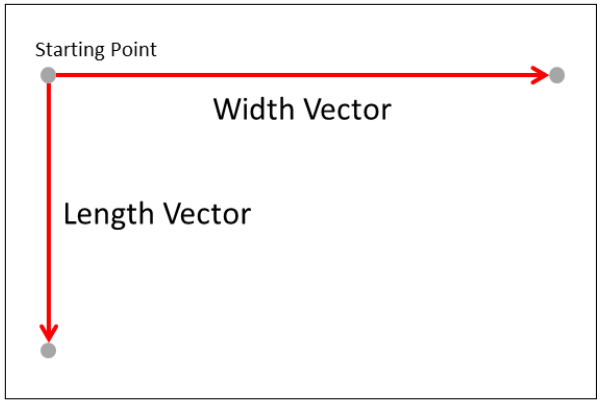


Figure 6.10: Width and length vector.

The width and length of the rectangle can be accessed by using Unity's `magnitude()` function on the width and length vector[33]. Moreover, to normalize the width and length vector, Unity provides us with a `Normalize()` function[31]. In the following equations, the normalized width vector will be called $\overrightarrow{NWV}$ and the normalized length vector will be called $\overrightarrow{NLV}$. The position of the vertices can now be easily calculated.

For the first vertex $\overrightarrow{V1}$, we simply take the starting point, which will be $\overrightarrow{SP}$. Therefore, no further calculations are needed here:

$$\overrightarrow{V1} = \overrightarrow{SP} \tag{6.1}$$

For the second vertex $\overrightarrow{V2}$, we need to move along the width *and* length vector. With the normalized vectors, this is achieved by multiplying $\overrightarrow{NWV}$ and $\overrightarrow{NLV}$ with the quad size and adding it to $\overrightarrow{SP}$.

$$\overrightarrow{V2} = \overrightarrow{SP} + \overrightarrow{NWV} * QuadSize + \overrightarrow{NLV} * QuadSize \tag{6.2}$$

Since the third vertex $\overrightarrow{V3}$ is right below $\overrightarrow{SP}$, we only need to move along the length vector from the starting point by the same unit as the quad size:

$$\overrightarrow{V3} = \overrightarrow{SP} + \overrightarrow{NLV} * QuadSize \tag{6.3}$$

The second triangle will share its first and third vertex with the first triangle, since both triangles will have the same color at all times and do not need separate UV coordinates. Thus, only one new vertex is needed. The remaining second vertex of the triangle is located on the right-hand side of the starting point. An outline of the vertex's positioning can be seen in figure 6.11.
These four vertices result in two triangles, which combined yield in a quad.

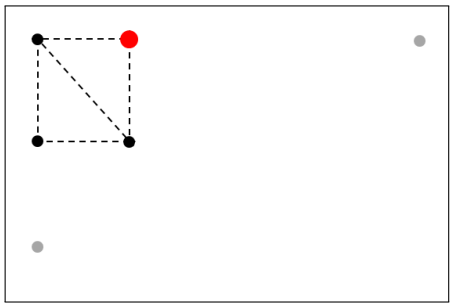


Figure 6.11: For the second triangle only one new vertex is needed, which is located on the right-hand side of the starting point.

To calculate the fourth vertex $\overrightarrow{V4}$ of the quad, the equation is very similar to equation 6.3. The difference is that we are moving along the width vector instead of the length vector:

$$\overrightarrow{V4} = \overrightarrow{SP} + \overrightarrow{NWV} * QuadSize \tag{6.4}$$

After completing adding all vertices of the first quad, the starting point will be manipulated by moving it along the width vector. The calculation is equal to the equation in 6.4. This new starting point is the starting point for the second quad and it is added to the list of vertices. As displayed in figure 6.12, the red dot represents the new starting point, which also has the same coordinates as the previously-created quad's last vertex. This means that we now have duplicated vertices in the mesh's list of vertices. This is necessary since these vertices are not shared among one quad and need to be able to have different colors, i.e. distinct UV coordinates. This is the case if the number of pedestrians on the quads differ. In the following figures, duplicated vertices are colored blue.

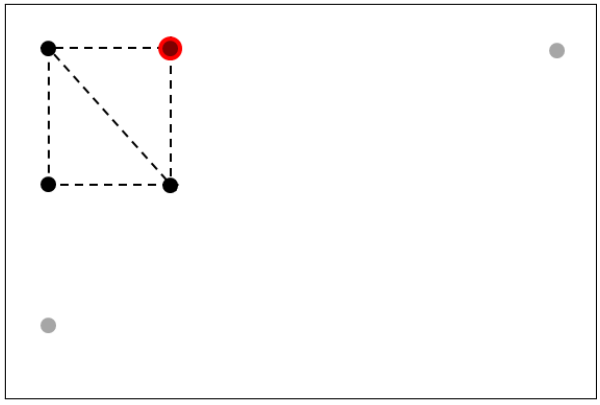
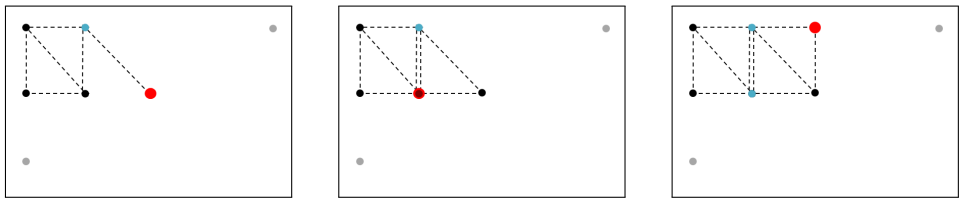


Figure 6.12: The starting point of the new quad has same coordinates as the last vertex of the first quad.

Figures 6.16a - 8.2b visualize how the remaining second quad's vertices are also added to the list. Relative to the manipulated starting point, we again move along the width *and* length vector for the second vertex of the quad (6.16a). The third vertex is below the starting point (6.16b), while the fourth and final vertex of the quad is on the right-handed side of the starting point (8.2b). The blue points indicate the duplicated vertices, which are located at the sides of the quads where the quads touch.



(a) Second vertex is again positioned on the right and below the starting point.

(b) Third vertex is below the starting point.

(c) Fourth vertex is positioned on the right next to the starting point.

Figure 6.13: Definition of all vertices for the second heat map cell.

For each new quad, the starting point will be moved until the right side of the rectangle is reached with the last quad's vertices. In case of the distance between the last starting point and the top right corner vector being less than the quad size, a special edge treatment along the width is necessary. When moving along the width vector, we do not move as many units as the quad size. Instead, we only move as many units as the distance between the last starting point and the top left corner. This needs to be considered when creating the vertices for the top right (equation 6.5) and bottom right corner of the quad (equation 6.6).

$$\overrightarrow{V2_{rightEdge}} = \overrightarrow{SP} + \overrightarrow{TopRight} - \overrightarrow{SP} + \overrightarrow{NLV} * QuadSize \tag{6.5}$$

$$\overrightarrow{V4_{rightEdge}} = \overrightarrow{SP} + \overrightarrow{TopRight} - \overrightarrow{SP} \tag{6.6}$$

When moving along the length vector of the bottom right and bottom left corner, we still move as many units as the quad size. As shown in figure 6.14, this concludes the first row of quads.

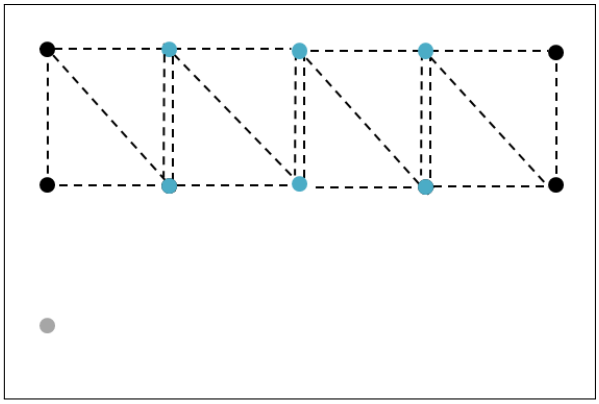


Figure 6.14: Completion of the first row of heat map cells.

After finishing a row, the starting point is manipulated again. It is first set to the top left corner of the rectangle. Depending on the number of the current row, we simply move as many quad size units along the length vector. The equation can be seen in equation 6.7:

$$\overrightarrow{SP} = \overrightarrow{TopLeft} + i * QuadSize * \overrightarrow{NLV} \tag{6.7}$$

where:

i = number of current row

We now start creating quads along a row again, as described above. This leads to duplicated vertices along the bottom line of vertices in the previous row and along the top line of vertices in the current row. This is again needed, since these vertices belong to distinct quads that may have different colors.

When getting to the last row of quads, we need to consider whether a full sized quad fits within the boundaries of the rectangle height-wise. This is only the case if the length of the rectangular bound is a multiple of the quad size. We then simply create the last row of quads in the same way as we created all previous rows. If this is not the case, a special treatment for the lower edge is necessary just as a special treatment for the right edge of the bound can be necessary as described above. In this case, when creating vertices where moving along the length vector is necessary - which are the bottom right and bottom left vertex of the quad - we again cannot move as

far along the length vector as a quad size unit. Instead, we use the distance between the starting point of this row of quads and the bottom left corner vector of the rectangular bound. For the second vector of the quad, equation 6.8 is used and for the third vector of the quad equation 6.9 resembles the correct position calculation.

$$\overrightarrow{V2_{bottomEdge}} = \overrightarrow{SP} + \overrightarrow{NWV} * QuadSize + \overrightarrow{BottomLeft} - \overrightarrow{SP} \tag{6.8}$$

$$\overrightarrow{V3_{bottomEdge}} = \overrightarrow{SP} + \overrightarrow{BottomLeft} - \overrightarrow{SP} \tag{6.9}$$

Calculating all vertices leads to the vertices being evenly distributed, as portrayed in figure 6.15. All created vertices are saved in the mesh's list of vertices.

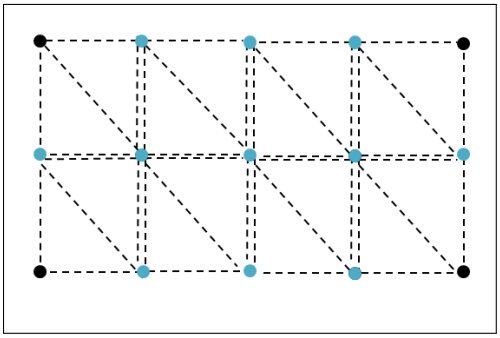
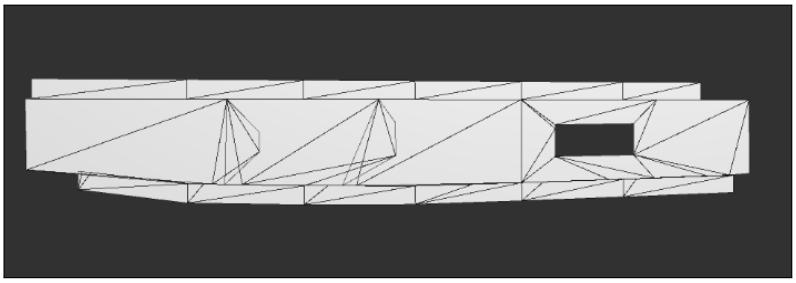
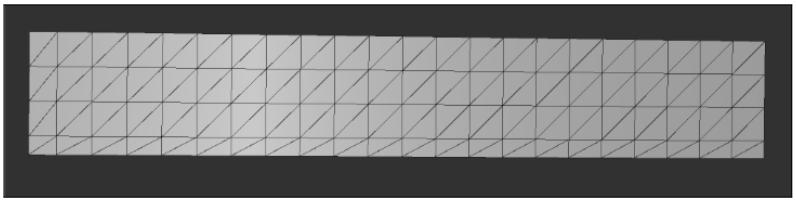


Figure 6.15: Completion of the whole heat map grid.

Concrete results of the heat map meshes with realigned vertices can be seen in figures 6.16 and 6.17.

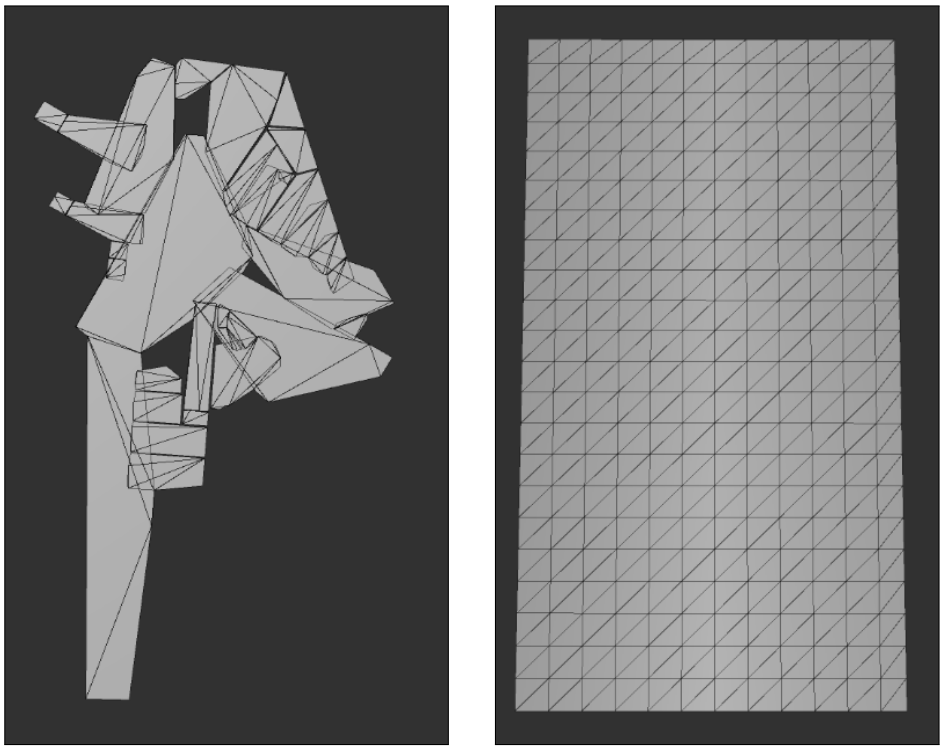


(a) Example mesh 1 - before.



(b) Example mesh 1 - after.

Figure 6.16: Example mesh 1 before and after realigning vertices.



(a) Example mesh 2 - before.

(b) Example mesh 2 - after.

Figure 6.17: Example mesh 2 before and after realigning vertices.

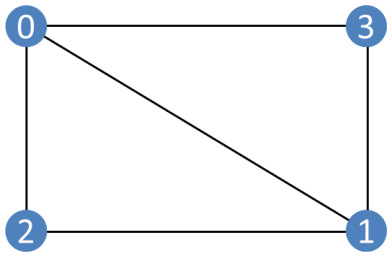
TRIANGLES

As mentioned in 2.1, the three vertices that form a triangle need to be in a consecutive order inside the Mesh.triangles list. A single quad comprises two triangles. Since two vertices are shared among the triangles, the index of only four distinct vertices have to be added but still six entries in total are needed. Figure 6.18a displays the indices of the vertices within the same quad. When connecting the vertices clockwise, the first triangle is formed by the vertices at the indices 0, 1 and 2. The vertices for the second triangle have the indices 0, 3 and 1. The order of the entries can be seen in figure 6.18b.

For each consecutive quad, the six affected indices will have the same difference in between, but per additional quad an addition with four will be made to each entry's value. Accordingly, the indices for the second quad in the Mesh.triangles list will be 4, 5, 6, 4, 7 and 5. This means that every sixth entry in the triangles list has an offset of four until the last triangles vertices have been addressed. Filling the triangles list concludes connecting all vertices to triangles.

UV COORDINATES

We need to take the part of the texture that reflects the default behavior of the heat map as the UV coordinate. A heat map cell will only react by changing its color if a pedestrian is placed on top of it. This means that the



(a) Order in which vertices of a heat map cell are defined.

triangles[0]	0
triangles[1]	1
triangles[2]	2
triangles[3]	0
triangles[4]	3
triangles[5]	1

(b) Extract of the first six entries of the triangles list.

Figure 6.18: Positioning of heat map cell vertices and their order within the triangles list.

default behavior is that no pedestrian is on the quad and therefore it should not be colored in any way. The first pixel of the texture - which starts at the coordinate (0, 0) - is a transparent pixel and its alpha value is set to 1. This will be the UV coordinate of every single mesh's vertex during the mesh creation. Details concerning the structure of the texture will be discussed below in section 6.22.

6.3 STORING AND MANAGING HEAT MAP MESHES

After creating the vertices, triangles and UV coordinates of heat map meshes, we need to define a structure that we can use for storing all heat map meshes, i.e. both floor and stair meshes. In addition to storing the meshes, we also need to manage the heat map meshes. This involves mapping pedestrian positions to concrete heat map cells and tracking how many pedestrians are located on which heat map cell. The first section will cover the storage of heat map meshes, while the second section will cover the management of heat map meshes.

STORAGE OF HEAT MAP MESHES

Before storing the heat map meshes, we need to assign them to concrete Unity `GameObjects` for them to be rendered by the game engine[28]. Moreover, we can now apply the heat map material to the GameObject. The material that we apply comprises a texture that will be described later in section 6.4.

Given that the heat map meshes have the exact same coordinates as the meshes to which they belong, Unity does not know which mesh to prioritize when rendering the scene. This leads to digital artifacts where depending on the angle of view on some patches the heat map becomes visible to the user but on other patches the floor meshes are visible instead. A solution to this

problem is to slightly translate the mother GameObject in the scene graph which contains all heat map meshes. All changes made to the mother object are also made to its children objects within the scene graph. This ensures that the heat maps are floating above their respective floor meshes, making it continuously visible to the user. Translating it by 0.001 into the positive y-direction - i.e. slightly upwards - is sufficient and does not cause a gap that makes the heat map disconnect from the floor mesh for the human eye.

In order to identify on which heat map cell pedestrians are located, it is necessary to know the coordinates of where each cell lies. For this purpose, we created a `HeatmapData` class. A HeatmapData object is created for every single heat map mesh, namely for both heat map floor meshes as well as every stair mesh. The HeatmapData object stores the mesh itself and it contains information to the heat map mesh. The information comprises several vectors such as the three vectors describing the bounding box, i.e. the top left, top right and bottom left vector, the width and length vector and the number of quads along the width and along the height of the heat map mesh. Storing all of this information makes it later possible to calculate the correct heat map cell position of a pedestrian easily and - more importantly - also quickly. Moreover, it stores a plane object that is created from the top left corner vertex, width and length vector. The plane object is later used to check whether a pedestrian is currently on the mesh that is currently considered height-wise. Finally, a HeatmapData object holds a list of integers. The number of integers in this list is equal to the number of cells that the mesh comprises. This list will be used to track how many pedestrians are on each cell of the heat map mesh.

Additionally, a `HeatmapHandler` class is implemented. This class is instantiated only once, and it holds a list of all HeatmapData objects. Each HeatmapData object that has been recently created will be immediately added to the list. As soon as a pedestrian moves, we need to check whether the new position of the pedestrian also means that he is placed on a different cell. That different cell may or may not be within the same heat map mesh. Therefore, we need to iterate through the entire HeatmapData list of the HeatmapHandler to hopefully find the heat map mesh on which the pedestrian is now located as quickly as possible.

Given that the iteration through the list of HeatmapData objects will happen multiple times in a frame, it makes sense to pay attention to which HeatmapData objects are put into the list first, since they will be considered first when trying to find the mesh on which a pedestrian is now located. In most cases, floor meshes are larger than stair meshes and use a much larger space. It is therefore very likely that pedestrians are on floor meshes most of the time. It is also likely that pedestrians are on stairs, but this is not due to stair meshes being large and able to hold many pedestrians. It is more likely due to the fact that with most geometries there are many stair meshes inside the

geometry. Therefore, it makes sense to look through the small number of floor meshes first to get the matching mesh to the pedestrians new positions, rather than looking through stair meshes first. Hence, the HeatmapData objects that contain floor meshes are put into the list first and afterwards stair meshes are loaded into the list.

MANAGING OF HEAT MAP MESHES

In order to later make the heat map cells change their color according to the amount of pedestrians on them, we first need to identify the heat map cells on which the pedestrians are located.
The `PedestrianSystem.cs` class of the SumoVizUnity project has been created prior to this work. Among other tasks, it iterates through every single pedestrian entity and gets the two closest positions to the current frame of the simulation visualization. Hereupon, an interpolation of the positions is made based on where the current frame is compared with the frames that belong to each taken position. This leads to the most probable position that the pedestrian might be on. By doing so, the pedestrian can be animated smoothly from frame to frame without having them suddenly "teleporting" to their new location. The calculated position is the position that is also needed for the heat map, since it describes the position of the pedestrian in the current frame. This position can now be used to find the cell on which the pedestrian is.

To check whether a pedestrian is on an existing mesh, several checks are made. The first step is to iterate through all meshes and check whether one of the meshes match the pedestrian's position. This is achieved by simply using the Unity function `GetDistanceToPoint(Vector3 point)` from the `Plane` class[30]. As a parameter, we pass the current position of the pedestrian. The result of the function returns a positive value if the point is on the side of the plane towards which the normal of the plane is facing and negative if it is on the opposite direction of the plane. By taking the absolute value we can now check whether the point is on the same height as the mesh. Within a tolerance of 0.01, the position is considered to be on the same height as the mesh. The next check is to determine whether the position is within the boundaries of the mesh. This is accompanied by calculating which concrete cell of the heat map mesh is affected. The UVs of the corresponding vertices needs to be manipulated. As previously mentioned, the meshes of the stairs are not axis-aligned to the x- and z-axis. Therefore, a calculation with vectors is again needed for the solution to work for both floor meshes and stair meshes. To ascertain on which cell of the mesh the currently-considered position is located, several vector calculations have to be performed.

We can obtain all of the information that we need for the calculations from the HeatmapData objects. We need the top left, width and length vector as well as the vector describing the current position of the pedestrian. All vectors are shown in figure 6.19.

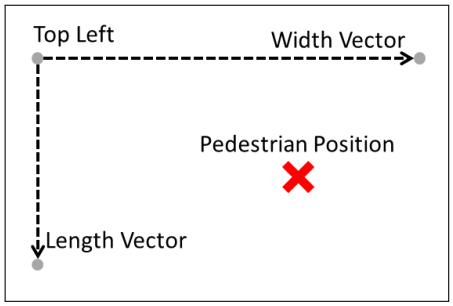


Figure 6.19: Positioning of top left, width and length vector. The red cross represents the pedestrian position.

By subtracting the top left vector from the position we obtain the vector that points to the position, starting at the top left. The black vector in figure 6.20 represents this newly-calculated vector.

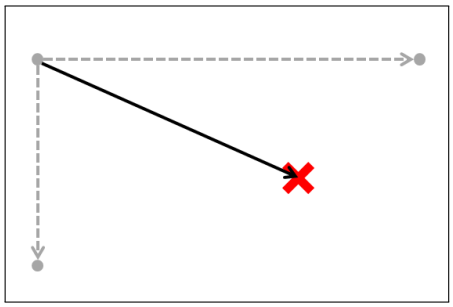


Figure 6.20: New vector is spanned between the top left corner and pedestrian position..

Unity provides us with a `Project` function [32]. The first parameter of the function is the vector onto which the vector passed as the second parameter will be projected. Figure 6.21 shows how we use the function to project the vector that was calculated in figure 6.20 and project it onto both the width vector and length vector.

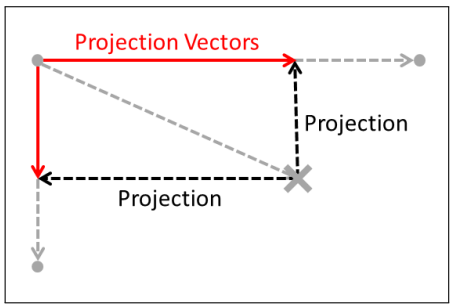


Figure 6.21: Projection of previously-calculated vector onto the width and length vector.

Since the vertices of the heat map mesh are first created column-wise and row-wise, we can now calculate the number of column and row of the cell on which the pedestrian is currently located. This is achieved by comparing the magnitude of the width vector with the magnitude of the vector that has been projected onto the width vector and comparing the magnitudes of the length vector width the magnitude of the vector that was projected onto the

length vector. A quick verification whether either one of the projected vectors' magnitudes are larger than the magnitudes of the vectors onto which they have been projected shows whether the point is outside of the considered mesh. If this is the case, a different mesh within the list of all possible meshes might fit.

In case the point lies within the bounds of the mesh, the index of the quad within the list can now be calculated by dividing the magnitude of the projection vectors by the quad size. Afterwards, the decimals need to be cut off. The result when taking the projection onto the length vector tells us in which row the quad lies, while the result of the projection vector along the width tells us in which column the quad can be found. Multiplying the first result with the number of quads per row and afterwards adding the second result, gives us the index of the affected cell.

As mentioned above, each HeatmapData also contains a list that tracks the number of pedestrians per cell. Now that we know which mesh the pedestrian is located on and the exact index of the cell, we can increase the number of pedestrians for this cell by one. The actual coloring of the heat map through manipulation of the UVs will be explained in the following section.

6.4 HEAT MAP COLORING

The coloring of the heat map cells is undertaken immediately after calculating the index of the cell on which the currently-handled pedestrian is located. The coloring is undertaken according to the number of pedestrians that are currently on the cell. The first section describes how the coloring of cells that have pedestrians on them will be made. In the second section, we implement the influence of single cells on their neighbors.

6.4.1 *Coloring of Single Cells*

For the heat map colors, a texture is used, as can be seen in figure 6.22. It has the resolution 64x1, which means that it comprises 64 colored pixels along the x-axis. It does not matter which y-value is taken for the texture, since there is only one pixel along the y-axis. The left-most pixel is transparent and the first non-transparent pixel is green. The color then shifts from yellow over orange to finally red on the very right. Manipulating the UVs of a mesh leads to the affected vertex taking a different pixel of the texture as their color and thus causing a color change.

Figure 6.22: Heat map texture from green to red. The first value is transparent.

Initially, the UV coordinates of every single heat map mesh have an x-value of zero, since the default will be that most quads do not have any pedestrians on them. This leads to all quads being transparent in the beginning.
The user of the program can set a number for the maximum number of pedestrians per heat map cell. If the number of pedestrians on a single cell reaches that number, the cell will be colored red. With a number of pedestrians on a cell in between one and the number set by the user, the UV will be accordingly interpolated to access the color that proportionally fits to the number of pedestrians that are currently on the cell.

In the previous section, we have calculated the index of the cell on which the pedestrian is located. We have also extracted the total number of pedestrians that are on this cell by reading the number from the list that is tracking the number of pedestrians per cell. Now we can change the UVs of the four affected vertices.

When we use the texture, we need to consider that the first pixel of the texture is transparent. Consequently, when we want to use the colored part of the texture, we need to take be careful about which texture coordinate we take for the second pixel. As mentioned in section 2.1, texture coordinates are normalized into a range from 0 to 1. Since there are 64 pixels along the width, the second pixel starts at the texture coordinate $1/64$. However, this is not the correct texture coordinate for the green color, since this is simply accessing the edge from the first pixel to the second pixel. As a result, this color will be a mixture of the green second pixel and the transparent first pixel, leading to a green color that is halfway transparent. Instead, we need to access the *middle* of the second pixel. Given that one pixel has a width of $1/64$, we need to half this value and add it onto the texture coordinate, in which the second pixel starts. This results in the texture coordinate $1/64 + 1/128$, which equals $3/128$. On the other hand, when accessing the first and last pixel of the texture, we do not have to ensure to access the middle of the pixel. This is due to the fact that texture coordinates smaller than zero will get the first pixel's value, while texture coordinates higher than one will get the last pixel's value.

If the number of pedestrians on the quad is between one and the set value for max pedestrians, we need to map the number accordingly to an interval of

$$[3/128, 1] \tag{6.10}$$

We can now set this value as the UV for all four affected vertices, so that the entire quad changes its color according to the number of pedestrians.

6.4.2 *Coloring of Affected Cells' Neighbors*

We implemented a visualization of the impact of density-caused forces on a densely-populated cell on their neighboring cell prototypically. This section describes how the implementation is undertaken.

Each cell that contains at least one pedestrian will cause their neighbor cells to get an additional pedestrian count of one. This simulates a transfer of possible forces from one cell to other cells. In this implementation, we consider the N4-neighborhood of a cell, which means that the top, bottom, right and left neighbors are affected. Figure 6.23 shows a N4 neighborhood visually. The N4 neighbors of the middle cell are colored in blue tint.

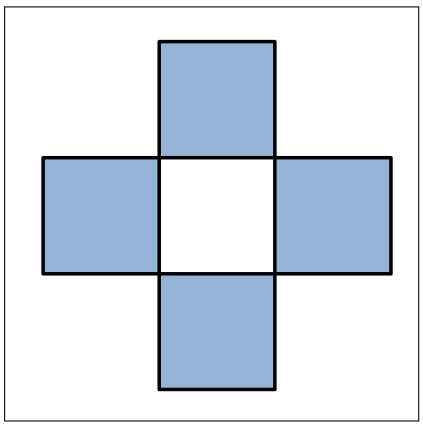


Figure 6.23: The N4 neighborhood of the center cell is colored blue.

To obtain the index of these four neighbors, we need to use the index of the currently-affected cell index. Within the mesh, the grid cells are arranged to form a two-dimensional grid where the cells are ordered row-wise from left to right. On the other hand, the list that contains the pedestrian count per cell is not two-dimensional, but rather it is a one-dimensional list. However, through the index of the current cell we can calculate in which row and column the cell is located, since we have saved the information about the number of cells that are inside one row inside the HeatmapData object. For a heat map with ten rows, with each row having five cells this means, that the index zero to four will contain the first five cells while the second row contains cells with the index five to nine etc. By calculating the index of the current cell modulo the number of cells per row, we obtain the column number of the current cell as a result.

$$r \equiv i \mod n \tag{6.11}$$

where:

i = index of the cell

n = number of cells per row

r = resulting column number of the cell

We can now determine whether the cell has all N4 neighbors and if so determine their indices:

- If $r - n >= 0$: the *top* neighbor has the index $i - n$

- If $r + n < m * n$: the *bottom* neighbor has the index $i + n$
- If $r + 1 < n$: the *right* neighbor has the index $i + 1$
- If $r - 1 >= 0$: the *left* neighbor has the index $i - 1$

where:

i = index of the cell
m = number of cells per column
n = number of cells per row
r = resulting column number of the cell

For all true conditions, the resulting neighbor index will be taken and the pedestrian count for that cell will be incremented by one. Afterwards, we update the affected cell's UVs, which makes the neighbor cells update their cell color according to their new pedestrian count.

6.5 RUN-TIME OPTIMIZATION

As mentioned above, the `PedestrianSystem.cs` class of the SumoVizUnity project calculates the new position of every single pedestrian once in every frame. Is is possible to take these new pedestrian positions and apply them to the heat map cells newly for every single frame. In order to keep the heat map information about how many pedestrians are on which cells up to date, we would also have to clear the pedestrian count list of every single HeatmapData object in every single frame. Clearing the heat maps to fill them from scratch again in every single frame is very costly and causes a lower frame rate. Moreover, it is highly unusual for a pedestrian to move so quickly that he/she changes the heat map cell on which he is placed within a frame or even a second.

A better solution to this problem is to only update the HeatmapData when the new position of a pedestrian causes a change in the heat map, e.g. the pedestrian moves to a different cell or a different heat map mesh. We implemented a new system to update the heat map UVs on demand rather than per frame.

To implement the on-demand-system, we created two lists inside the PedestrianSystem class. Both lists are as large as the number of pedestrians in the current simulation. One list will track the *index of the heat map mesh* on which the pedestrian was last, while the other list tracks the *index of the cell* on which the pedestrian was seen last. The first pedestrian will be tracked by using the first index of the lists, the second pedestrian will be assigned to the second index of the lists, etc. As soon as the new pedestrian position has been calculated, we check whether the heat map mesh index and cell index still match the previously-recorded indices.

If either one of the indices is not up to date, we decrement the pedestrian count of the previous cell by one to take away the current pedestrian, since he/she no longer seems to be located on that cell. Now we can update the mesh and cell index for this particular pedestrian and finally increment the pedestrian count of the correct and new cell by one. In this way, the heat map UVs are only updated on demand, which makes it possible for the application to run with a higher frame rate, since a lot of unnecessary calculation costs can be prevented in this way.

7

RESULTS

In this section, we present the results of the heat map, with different settings.

The user can determine the size of the heat map cells. The dimensions of the coordinate system resembles a ratio of 1:1 meter, which means that a line of the length 1 inside the visualization resembles one meter. The user can also set the maximum number of pedestrians on a single cell. If this number is reached or exceeded, the heat map cell will be colored red to indicate dense areas. This setting will further be called "max pedestrians".

In figures 7.1-7.3, the size of the heat map cells is 2x2. In figures 7.1 and 7.2, max pedestrian is set to three. In figure 7.1, we can see an area where pedestrians are distributed quite densely. Cells that are only populated by one pedestrian are green, while cells with two pedestrians are orange and all cells that have at least three pedestrians located on them are colored red.

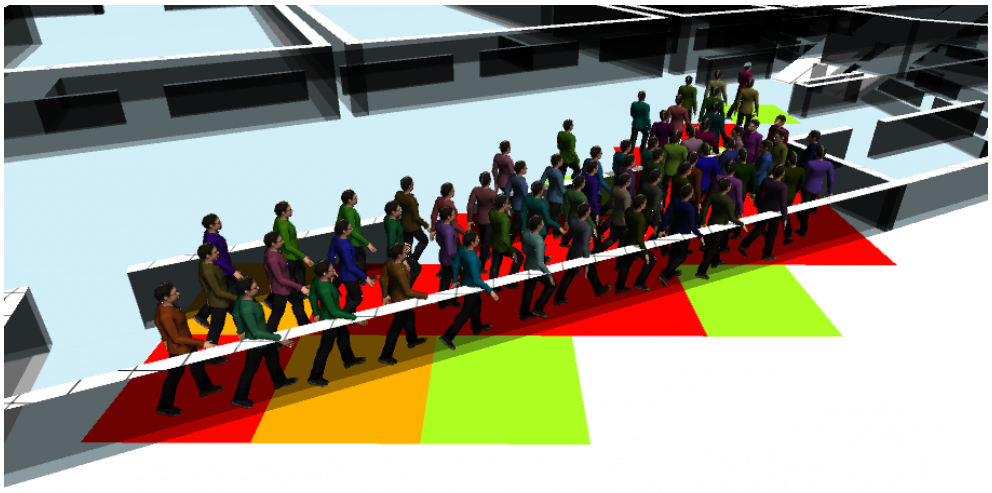


Figure 7.1: Densely-populated area with cell sizes of 2x2. Max pedestrian is set to 3.

Figure 7.2 shows an area where pedestrians are positioned less densely and move freely.

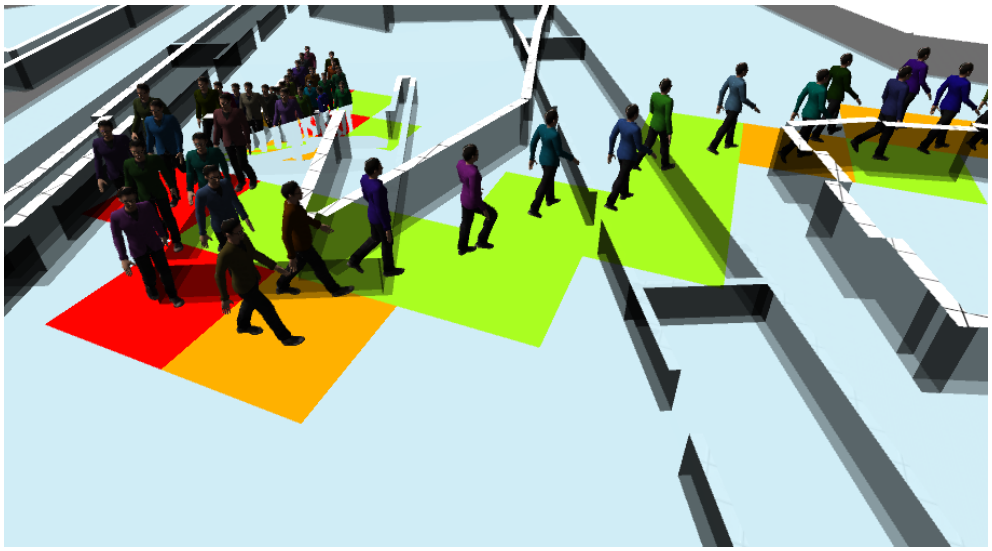


Figure 7.2: Less-densely populated area with cell sizes of 2x2. Max pedestrian is set to 3.

Figure 7.3 shows the area where pedestrians are positioned densely again, although the max pedestrian setting is set to nine here.

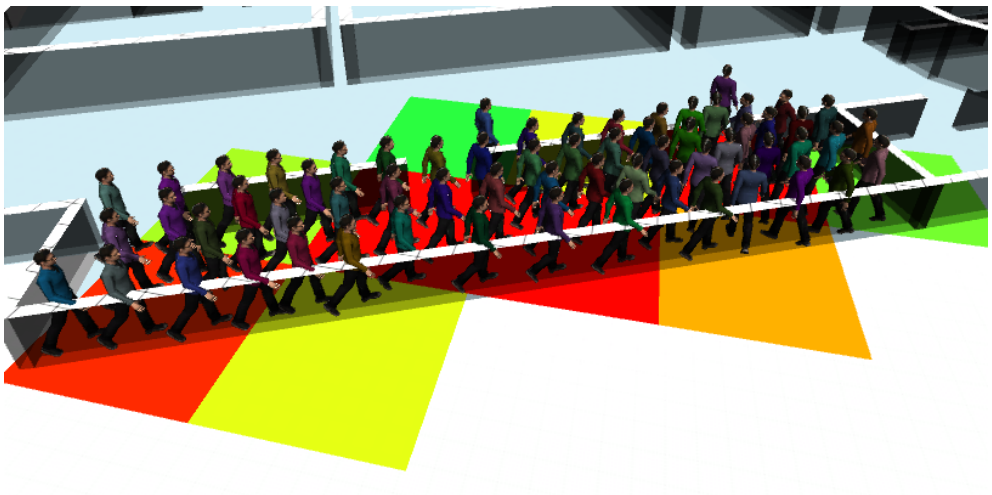


Figure 7.3: Densely-populated area with cell sizes of 2x2. Max pedestrian is set to 9.

We can see that by increasing the setting of the max pedestrians, a higher range of colors from the texture will be used. Instead of only seeing green, orange and red, we also get to see a more saturated green and yellow in figure 7.3. This is due to the fact that a setting three for of max pedestrians, only three sections of the texture will be used, resulting in only three distinct colors, i.e. one for one pedestrian and one for two pedestrians and one for three pedestrians that populate a single heat map cell. Increasing this number also increases the necessary coding of pedestrian amounts, since for a max pedestrian setting of nine we also need to use nine distinct sections of the texture.

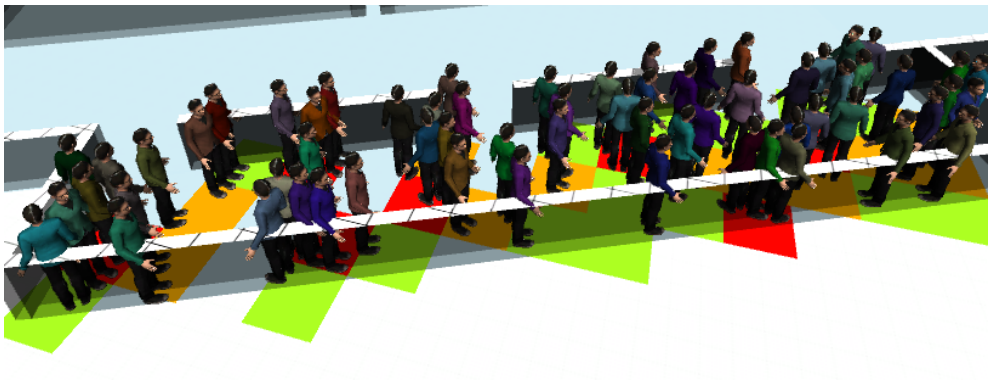


Figure 7.4: Densely-populated area with cell sizes of 1x1. Max pedestrian is set to 3.

Figures 7.4 and 7.5 show examples where the heat map grid size is set to 1x1. The maximum pedestrian setting is set to three again in these examples. In the dense area of figure 7.4, we can again see some heat map cells where the max number of pedestrians are located on the same quad.

Figure 7.5: Less densely populated area with cell sizes of 1x1. Max pedestrian is set to 3.

Compared with figure 7.5, we can see in figure 7.5 that the smaller cell size causes not all cells to light up. This is due to the fact that the space between two pedestrians in a low-density area can be larger than the size of a single heat map cell.

Figure 7.6 shows what the heat map looks like on top of stairs. The settings inside the figure use a cell size of 1x1 and a max pedestrian count of three.

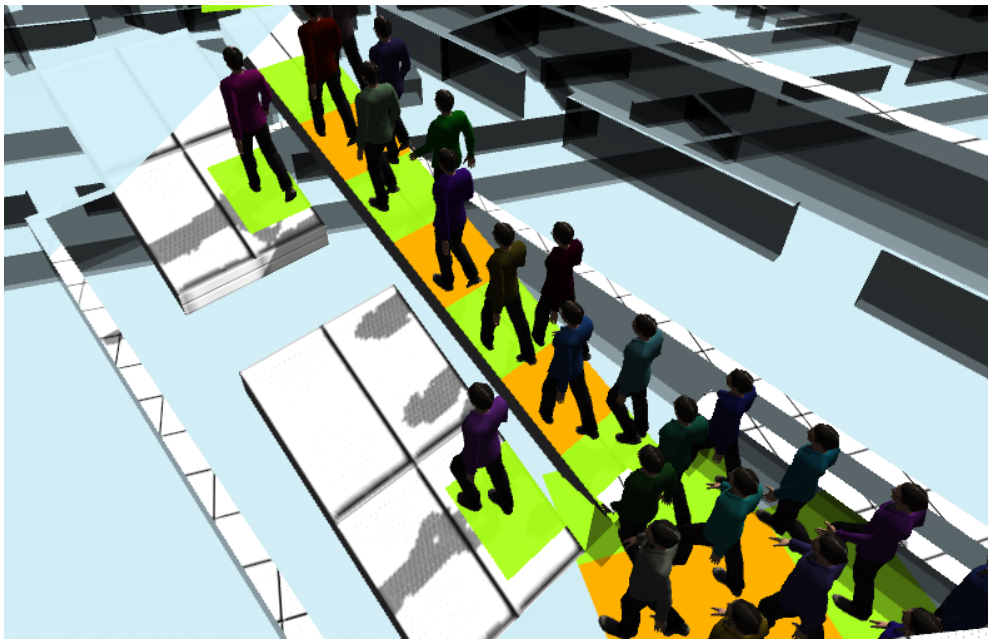


Figure 7.6: Heat map on top of stair meshes. Cell size is set to 1x1, max pedestrian is set to 3.

As mentioned in the previous sections, we also implemented the influence of populated heat map cells to their immediate N4 neighborhood. Figures 7.7 and 7.8 both show an example. In both figures, the heat map cell size is 1x1 and max pedestrian is set to three. The first figure shows the high-density area, while the second figure again shows a low-density area. We can see that the additional manipulation of neighboring heat map cells leads to fewer gaps and a more continuous heat map.

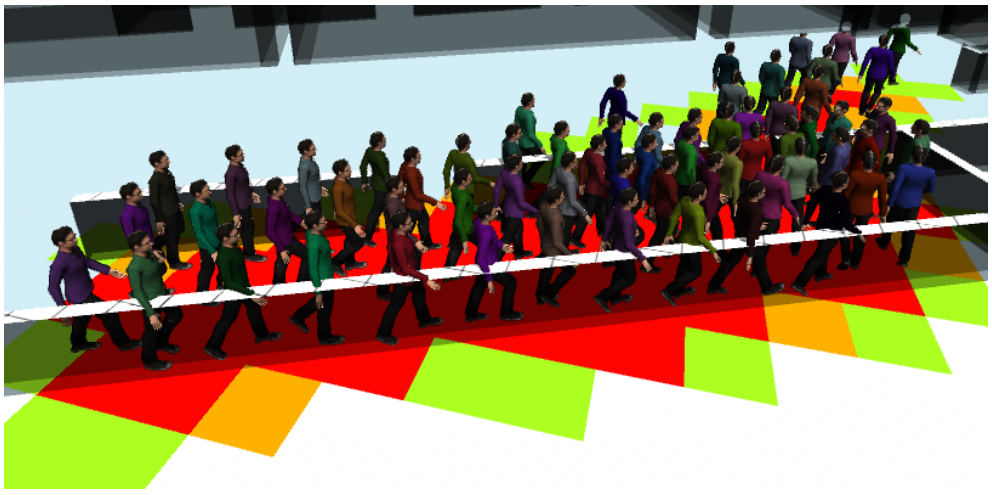


Figure 7.7: High-density area example of pedestrian count of heat map cells spreading to their N4 neighbors. The heat map cell size used is 1x1 and max pedestrian count is 3.

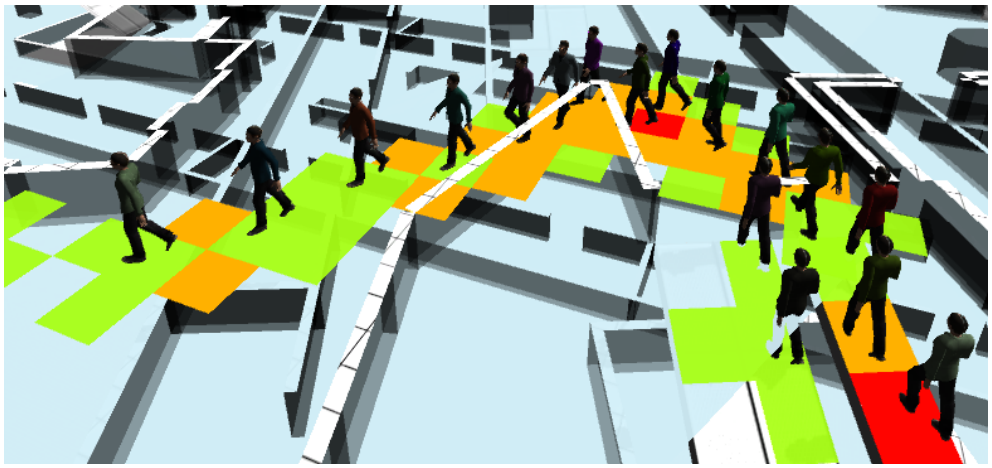
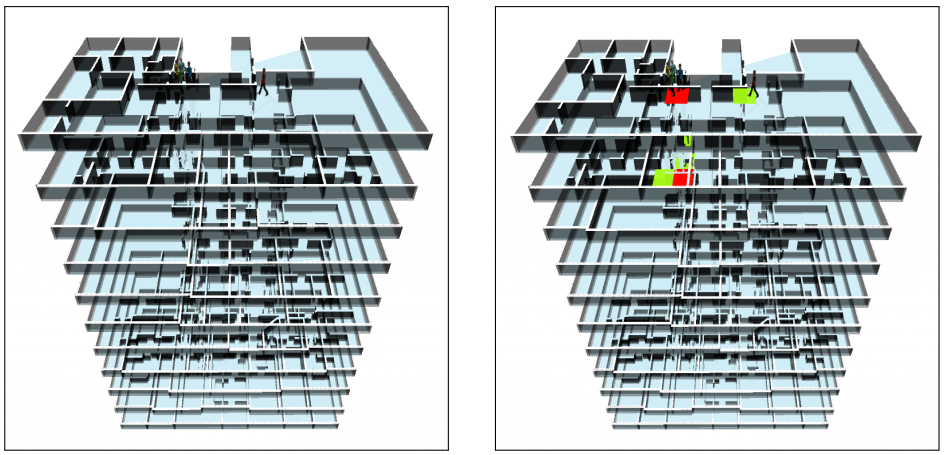


Figure 7.8: Low-density area example of pedestrian count of heat map cells spreading to their N4 neighbors. The heat map cell size used is 1x1 and max pedestrian count is 3.

8 DISCUSSION

This section discusses of the results obtained in the previous section.

The main goal of the heat map visualization was achieved, namely for the user to gain a quick visual impression of the distribution of pedestrians and potentially dangerous areas. Figure 8.1 shows an example of a tower building geometry. In the first figure, the heat map is not active. Especially with the pedestrians that are inside the building, it is not possible to get a quick impression of dense areas when zoomed out.



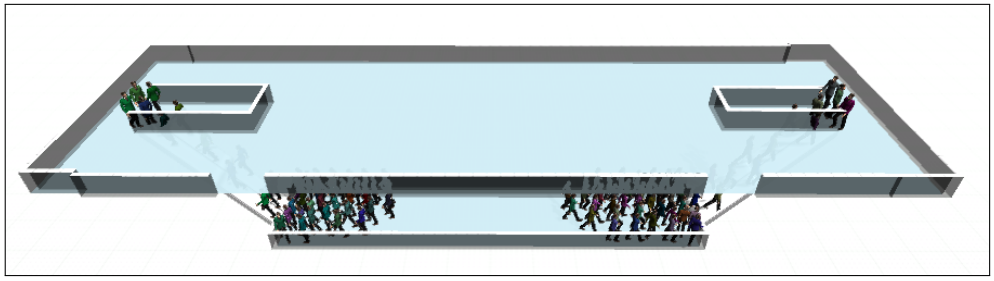
(a) Without heat map.

(b) With heat map.

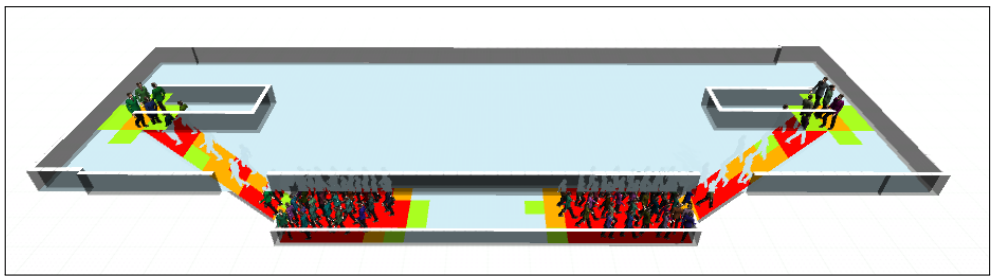
Figure 8.1: Example of a tower building, with and without heat map.

In the second figure, we can see exactly where the pedestrians are located, even when zoomed out and even if the view onto pedestrians is blocked by heat map meshes. We can also see where dense areas are located.

Moreover, in figure 8.2 we can see a geometry that comprises two stairs that are each connected to two platforms. The figure that shows the result with the heat map turned on shows that high-density areas are those areas that are at the beginning and end of a stair, as well as the areas leading to a stair entry.



(a) Without heat map.



(b) With heat map.

Figure 8.2: Example of a geometry with two stairs, with and without the heat map.

It is also again possible to see the distribution of pedestrians even if they are in between meshes, which makes the pedestrian models difficult to see.

The results show that depending on the selected size of the heat map cells, the setting of how many pedestrians on a single cell will make the cell turn red should be adapted accordingly. When choosing a cell size that is rather large and can therefore hold a larger amount of pedestrians, the max pedestrian setting should be set to a higher number. Choosing a small number would make the heat map cells turn red very quickly, since a large size of grid cells will hold many pedestrians. Our results show that a cell size of 2x2 can hold more than nine pedestrians. Choosing a max pedestrian setting of three would probably not help the viewer to identify interesting areas inside the geometry. If a small size is chosen for the heat map cells, e.g. a cell size of 1x1, not as many pedestrians might fit onto one cell. Choosing a smaller value for max pedestrians such as three would cause more interesting results with the heat map. Of course, the users can vary the heat map cell sizes and experiment with the visualization to find the settings that help them to analyse the density of pedestrian however they desire.

These results and figures show that the goal of the heat map to quickly reveal how pedestrians are distributed and where to find dense areas, was fulfilled. However, the implementation of the heat map still has room for improvement in some places.

When taking small cell sizes and only coloring the cells on which pedestrians are located without influencing their N4 neighbors, the heat map can look very sparse and slightly disconnected from pedestrians in low-density

areas. An example of this phenomenon can be seen in figure 7.5. This is due to the fact that we are not checking, whether a pedestrian model is hovering over multiple cells when he/she reaches the edge of a cell. Instead, we map the position of a pedestrian on a single point, which resembles the center of mass of a pedestrian model. A possible approach to solve the seemingly disconnect of pedestrian positions to heat map cells would be to color multiple cells for one pedestrian if the whole model is hovering over multiple cells.

Since we merge all floor meshes that are on the same level, we cannot consider any possible walls of the geometry when creating the heat map cells. In figure 8.3, we used a small cell size. Here we do not really encounter the problem of cells starting at one side of the wall and stretching over to the other side of the wall. However, when taking a larger cell size like 2x2, which we can see in figure 8.4, heat map cells might reach over a wall and consider pedestrians that are located in different rooms at the same time. This can be seen when looking at the lower and top right orange cell. In reality, this would not be an accurate representation of the density in this area. The calculation of the density would have to stop considering the part of an area that surpasses a wall.

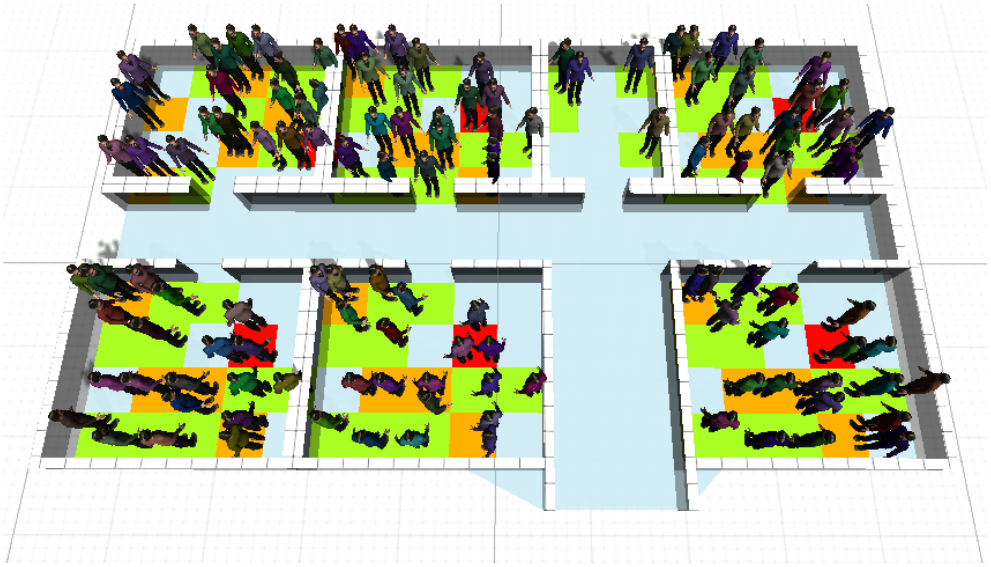


Figure 8.3: Example of a geometry with walls and small heat map cells.

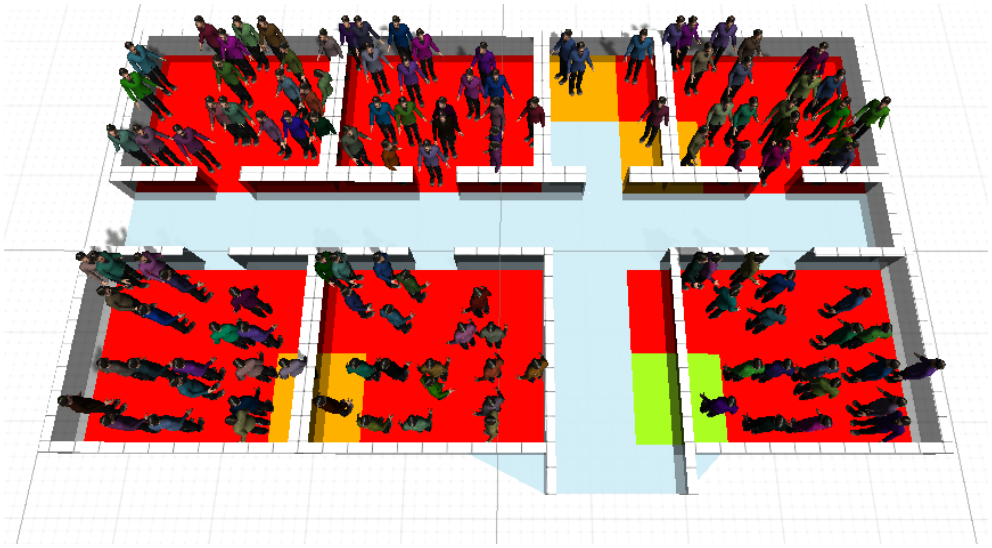
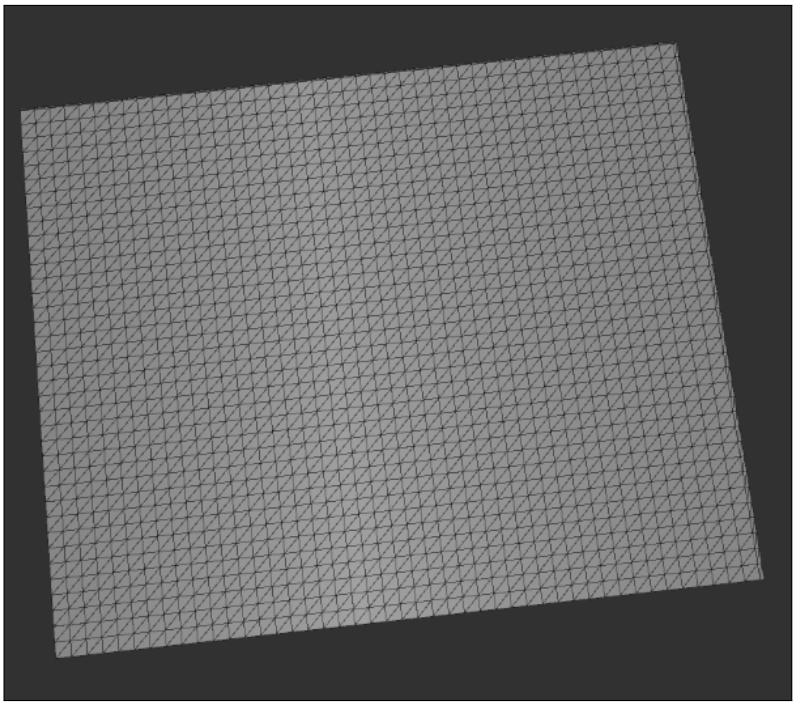


Figure 8.4: Example of a geometry with walls and large heat map cells.

Depending on the distribution of floor meshes per geometry, it is possible that floor meshes are on the same height but have a large distance in between. An example of two distant floor meshes that have the same y-value can be seen in figure 8.5a. Given that we use a bounding box to create a heat map grid that floats over both floor meshes at one, we will have a heat map grid that is much larger than the actual floor areas that it tries to cover, as can be seen in figure 8.5b. The user will not able to see that the heat map mesh is too large, since heat map cells that are have no pedestrians on them will be transparent, but the redundant size of the heat map might cause performance issues if a geometry comprises many similar floor meshes. Unity would still unnecessarily continue to render these transparent meshes. However, in order to truly cause performance issues the geometry would have to be generated in a strongly disadvantageous way.

(a) Both floor meshes in their original form.



(b) Both floor meshes combined to a single floor mesh.

Figure 8.5: Example of floor meshes that have the same y-value but are far apart.

The influence of populated heat map cells on their neighboring cells is implemented prototypically by incrementing the pedestrian count on neighboring cells by one. In order to correctly resemble the physical impact of densely-populated cells on their neighbors, a more accurate implementation is necessary. The improved implementation should not only have an influence on the N4 neighborhood of a cell, but instead the whole N8 neighborhood of a cell should be influenced. Moreover, instead of incrementing the pedestrian count by one in each neighboring cell, this should be replaced by a factor-based solution in which the number of pedestrians in the center cell impacts how strong the impact on neighbor cells is, instead of adding a fixed size of one pedestrian each time. Finally, the improved implementation should not realize the impact by increasing the pedestrian counts on the cells. Instead, an additional variable should be introduced for the sole purpose of tracking

the transferred forces from neighboring cells.

The final point concerning aspects that can be improved on the heat map is the fidgety nature of the heat map. If two pedestrians are on the same heat map cell and within less than a second a new, third pedestrian enters the same cell and short afterwards the foremost pedestrian leaves the cell, the heat map cell could change its color from orange to red and back to orange within a very short time. This and similar scenarios can happen at multiple locations, which causes the heat map to seem very erratic to the viewer. There are two possible approaches to make the heat map seem more calm to the user. First, it would be possible to work with mean values over a certain time, whereby short-time outliers of pedestrian counts would not influence the heat map as much and cause it to be more calm. A different approach is not to update the heat map in every single frame. With 200 frames per second, it is possible for the heat map to change 200 times within a second, which would not help its readability for a human at all. Instead, a possible solution could be to update the heat map less often, e.g. once or twice per second. Consequently, the heat map could be more comfortable to analyse for the user.

9

CONCLUSION AND FUTURE WORK

Increasing urbanization in several countries leads to an increasing number of pedestrians living in urban areas, which causes high population densities. Especially infrastructurally-relevant areas such as walkways and train stations - which can be densely populated at times - need to be designed in a way to ensure the safety of pedestrians. Pedestrian simulation programs can be used to simulate how different numbers of pedestrians move in areas where moving freely is not possible due to the limitations through walls and stairs. Simulation programs take geometry files as input and make it possible for the user to make configurations to the simulation program, such as the start and destination of pedestrians and their movement speed. In order to extract information about the safety of pedestrians inside a visualization of pedestrian simulation data, analysis tools can be used.

In this work, we have proposed concepts in relation to several possible visualization techniques, which include - but are not limited to - visualizing the pedestrian movement speed, the numeric value of pedestrian flow, pedestrian flow visualizations that visualize patterns in the movement of pedestrians through arrows and the visualization of density in certain areas.

Due to the time constraints of this thesis, we chose to implement a heat map to visualize the density in areas that are populated by pedestrians. We used a grid-based approach for the heat map, where each cell that is populated by a pedestrian will signal the density on that particular heat map cell through its coloring. A red color indicates a high density, while colors ranging from orange over yellow up to green signal a lower density in that area.

The main goal of the heat map was achieved, namely to provide the user a quicker overview of the dense areas of the geometry, was achieved. However, the implementation still leaves room for improvements of already implemented features in future work.

Additional features, that could be implemented in future work are other visualization concepts beside the heat map that have been presented in this thesis, such as visualizing the movement speed of pedestrians, the numeric value of the flow in certain areas and the visualization of movement patterns of pedestrians. Moreover, the implementation of the heat map could be extended by visualizing the density in certain areas by not only coloring heat map cells but also by elevating cells according to their density value. This could make a very interactive impression on the viewer if the elevation of heat map cells is combined with an animation that makes the cell rise slowly

upon experiencing a change of the number of pedestrians that are located on it. An improved future realization of the heat map could also be implemented in a way to consider walls that are close to pedestrians. Walls could be considered as a dense area, since pedestrians that are located near a wall have no possibility to move through an area that comprises a wall.

BIBLIOGRAPHY

[1] *BR Wissen Fußball-Spielanalyse: Das gläserne Spiel*. https://www.br.de/wissen/spielanalyse-fussball-forschung100.html. Accessed: 2021-05-24.

[2] *Balu Ertl Voronoi Diagram*. https://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg. Accessed: 2021-05-25.

[3] Daniel Büchele. "Visualisierung von Fußgängersimulationsdaten auf Basis einer 3D-Game-Engine." MA thesis. Fakultät für Bauingenieur- und Vermessungswesen, Technische Universität München, June 2014.

[4] *Daniel Büchele. SumoVizUnity GitHub*. https://github.com/danielbuechele/SumoVizUnity. Accessed: 2021-05-27.

[5] *Diercke Weltatlas*. https://diercke.westermann.de/content/erde-physische-%C3%BCbersicht-978-3-14-100770-1-6-1-0. Accessed: 2021-05-26.

[6] *Global Human Settlement - GHSL Homepage - European Commission*. https://ghslsys.jrc.ec.europa.eu/index.php. Accessed: 2021-05-27.

[7] Armel Ulrich Kemloh Wagoum, Mohcine Chraibi, and Gregor Lämmel. "JuPedSim: an open framework for simulating and analyzing the dynamics of pedestrians." In: Transportation Research Group of India (3rd CTRG). 2015. URL: https://www.researchgate.net/publication/289377829_JuPedSim_an_open_framework_for_simulating_and_analyzing_the_dynamics_of_pedestrians.

[8] *Kieran Hunt Flow Visualization*. https://stackoverflow.com/questions/51843313/flow-visualisation-in-python-using-curved-path-following-vectors. Accessed: 2021-05-25.

[9] Toussaint Loua. *Atlas statistique de la population de Paris*. Paris: J. Dejey cie, 1873.

[10] *MORPHOCODE Visualizing Pedestrian Activity in the City of Melbourne*. https://morphocode.com/visualizing-pedestrian-activity-city-melbourne/. Accessed: 2021-05-27.

[11] *Maps of World What is the Average Height of Males around the world?* https://www.mapsofworld.com/answers/regions/average-height-males-around-world/. Accessed: 2021-05-26.

[12] *Marvin Lars Weisbrod. SumoVizUnity (VR) GitHub*. https://github.com/nachtmarv/SumoVizUnity. Accessed: 2021-05-27.

[13] *Matt Daniels Human Terrain: Visualizing the World's Population, in 3D*. https://pudding.cool/2018/10/city_3d/. Accessed: 2021-05-27.

[14] *Miguel Andrade Cluster Heat Map*. https://en.wikipedia.org/wiki/File:Heatmap.png. Accessed: 2021-05-24.

[15] Alexandre Perrot, Romain Bourqui, Nicolas Hanusse, Frederic Lalanne, and David Auber. "Large interactive visualization of density functions on big data infrastructure." In: *2015 IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, Oct. 2015. DOI: 10.1109/ldav.2015.7348077. URL: https://doi.org/10.1109/ldav.2015.7348077.

[16] Wojciech Pokojski and Paulina Pokojska. "Voronoi diagrams – inventor, method, applications." In: *Polish Cartographical Review* 50.3 (Sept. 2018), pp. 141–150. DOI: 10.2478/pcr-2018-0009. URL: https://doi.org/10.2478/pcr-2018-0009.

[17] *PrimeLocation Heatmap of UK property values*. https://www.primelocation.com/heatmaps/. Accessed: 2021-05-24.

[18] *Statista China - Urbanisierung bis 2019*. https://de.statista.com/statistik/daten/studie/166163/umfrage/urbanisierung-in-china/. Accessed: 2021-05-29.

[19] *Statista Frankreich - Urbanisierung bis 2019*. https://de.statista.com/statistik/daten/studie/167202/umfrage/urbanisierung-in-frankreich/. Accessed: 2021-05-29.

[20] *Statista Grad der Urbanisierung in Deutschland*. https://de.statista.com/statistik/daten/studie/662560/umfrage/urbanisierung-in-deutschland/#:~:text=Die%20Statistik%20zeigt%20den%20Grad,der%20Gesamtbev%C3%B6lkerung%20Deutschlands%20in%20St%C3%A4dten. Accessed: 2021-05-28.

[21] *Statista Großbritannien - Urbanisierung bis 2019*. https://de.statista.com/statistik/daten/studie/167289/umfrage/urbanisierung-in-grossbritannien/. Accessed: 2021-05-29.

[22] *Statista Russland - Urbanisierung bis 2019*. https://de.statista.com/statistik/daten/studie/171395/umfrage/urbanisierung-in-russland/. Accessed: 2021-05-29.

[23] B. Steffen and A. Seyfried. "Methods for measuring pedestrian density, flow, speed and direction with minimal scatter." In: *Physica A: Statistical Mechanics and its Applications* 389.9 (May 2010), pp. 1902–1910. DOI: 10.1016/j.physa.2009.12.015. URL: https://doi.org/10.1016/j.physa.2009.12.015.

[24] *SumoVizUnity JuPedSim GitHub*. https://github.com/chraibi/SumoVizUnity. Accessed: 2021-05-27.

[25] *SumoVizUnity accu:rate GitHub*. https://github.com/accu-rate/SumoVizUnity. Accessed: 2021-05-27.

[26] *Textures in OpenGL*. http://www.c-jump.com/bcc/common/Talk3/OpenGL/Wk07_texture/Wk07_texture.html. Accessed: 2021-05-26.

[27] *Unity - Manual: Example - Creating a quad*. https://docs.unity3d.com/Manual/Example-CreatingaBillboardPlane.html. Accessed: 2021-05-27.

[28] *Unity - Scripting API: GameObject*. https://docs.unity3d.com/ScriptReference/GameObject.html. Accessed: 2021-05-27.

[29] *Unity - Scripting API: Mesh.bounds*. https://docs.unity3d.com/ScriptReference/Mesh-bounds.html. Accessed: 2021-05-27.

[30] *Unity - Scripting API: Plane.GetDistanceToPoint*. https://docs.unity3d.com/ScriptReference/Plane.GetDistanceToPoint.html. Accessed: 2021-05-27.

[31] *Unity - Scripting API: Vector3.Normalize*. https://docs.unity3d.com/ScriptReference/Vector3.Normalize.html. Accessed: 2021-05-27.

[32] *Unity - Scripting API: Vector3.Project*. https://docs.unity3d.com/ScriptReference/Vector3.Project.html. Accessed: 2021-05-27.

[33] *Unity - Scripting API: Vector3.magnitude*. https://docs.unity3d.com/ScriptReference/Vector3-magnitude.html. Accessed: 2021-05-27.

[34] *Unity-Manual: Transforms*. https://docs.unity3d.com/560/Documentation/Manual/Transforms.html. Accessed: 2021-05-26.

[35] *Unity Technologies. Unity Real-Time Development Platform*. https://unity.com/. Accessed: 2021-05-27.

[36] Marvin Lars Weisbrod. "Visualisierung von Fußgängerdaten in Virtual Reality." MA thesis. Fachbereich Informatik, Hochschule Darmstadt, Aug. 2020.

[37] Leland Wilkinson and Michael Friendly. "The History of the Cluster Heat Map." In: *The American Statistician* 63.2 (May 2009), pp. 179–184. DOI: 10.1198/tas.2009.0033. URL: https://doi.org/10.1198/tas.2009.0033.

[38] *accu:rate GmbH - Institute for crowd simulation*. https://www.accu-rate.de/en/. Accessed: 2021-05-27.