Fachhochschule Aachen
Campus Jülich

Field of study: Scientific Programming
University department: Medizintechnik und Technomathematik

Developing a Multiuser Platform for Virtual Experiments in Pedestrian Dynamics

Bachelor Thesis by

Julia Valder
Mat.Nr.: 4006095

Jülich, August 16, 2016

Declaration of Originality

I hereby declare that I have produced and written this bachelor thesis on the topic

> **Developing a Multiuser Platform for Virtual Experiments in Pedestrian Dynamics**

on my own. Only the mentioned aids and sources have been used.

Name: Julia Valder

Mat. Nr.: 4006095

August 16, 2016

Student's signature

This thesis has been created at the Jülich Supercomputing Centre (JSC) at Forschungszentrum Jülich and has been supervised by:

1. examiner: Prof. A. Terstegge, FH Aachen

2. examiner: Carsten Karbach, M.Sc., JSC

Contents

1 Introduction and Motivation

This thesis describes the development of a multiuser platform for virtual experiments in pedestrian dynamics (henceforth referred to as *the multiuser platform*). Real life experiments in pedestrian dynamics are conducted to understand the behavior of large crowds and improve safety facilities in buildings and large-scale events. Since these experiments are high in costs and effort, experiments are also conducted in virtual environments. The goal of the multiuser platform is to mitigate the problems of past virtual experiments. Its main feature is that all pedestrians connect to the same virtual experiment over a network. This means that all pedestrians in the virtual experiment are controlled by experiment participants instead of some pedestrians being controlled by the computer. Another advantage is that virtual pedestrians can be exposed to dangerous situations such as fire, which is not possible in real life experiments.

The following chapters explain the development of the multiuser platform as well as the finished product in detail. This chapter introduces important terms and motivates development of the multiuser platform.

1.1 Pedestrian Dynamics

Figure 1.1: Open Air Concert at "Rock am Ring" 2016

The term *pedestrian dynamics* combines the study of crowd dynamics, which are the behavior of large crowds and the flows within them, and the optimization of pedestrian facilities[1]. They can be observed in many situations in everyday life, such as concerts (see 1.1) or football matches, and play an important role in the planning of public buildings and infrastructures. Prof. G. Keith Still describes crowd dynamics as *"the study of the how and where crowds form and move above the critical density of more than one person per square metre"* [10]. Pedestrian dynamics, on the other hand, also apply to harmless situations in everyday life where the behavior of pedestrians is important, such as optimizing corridor corners. When designing buildings or planning public events, understanding pedestrian dynamics is of utmost importance to maintain a safe environment. In the past, large scale events have caused accidents and injuries, e.g. the Love Parade 2010 in Duisburg

[1]see www.fz-juelich.de/ias/jsc/EN/Research/ModellingSimulation/CivilSecurityTraffic/PedestrianDynamics/_node.html (retrieved August 16, 2016)

(Germany) where 21 people died and many more were injured due to a mass panic and insufficient exit routes.

There are many questions that have to be answered to ensure the security of a building or an event: Where should exits be placed so that crowd density will not rise too high in the event of evacuation? How broad must a hallway be to let a certain amount of people pass through in a given time? Can this building or event be evacuated quickly enough? Researchers conduct experiments in pedestrian dynamics in order to learn about the behavior of pedestrian groups and answer these questions.

Real life experiments usually take place in large halls where provisional rooms are constructed. While moving through the experiment, participants are tracked with special cameras and software. This tracking produces a so called trajectory for every participant, which represents the participant's path through the setup. A trajectory indirectly contains information about velocity and significant deviations from a predicted path. When evaluating all participants' trajectories, they also show crowd density and identify potential points of danger.

Figure 1.2: *BaSiGo* experiment in Düsseldorf

A typical experiment is the *Exit-Choice-Experiment*. The participants enter a room and choose between a number of exits. Their goal is to leave the room as fast as possible. Such experiments on exit choice behavior were already conducted in 1989 in order to ensure safety in buildings (see [4]). While there are many other experiment setups, the Exit-Choice-Experiment is used as an example throughout this thesis.

The goal of experiments in pedestrian dynamics is to learn about crowd behavior. That way, crowds can be simulated when planning events or buildings in order to determine necessary safety measures and increase security.

Figure 1.2 shows a pedestrian experiment which was conducted in Düsseldorf as part of the *BaSiGo* project in 2013[2]. The experiments were a joint cooperation between Forschungszentrum Jülich and the Universität Siegen. The goal of *BaSiGo* is to increase overall security for large scale events. Over 1000 people participated in the experiments. There were multiple experiment runs in which parameters, such as the width of an exit, were varied to study the change of dynamics in the pedestrian stream. The experiments were successful but also high in costs, effort and time. As a consequence researchers started looking into virtual experiments in order to reduce the amount of resources needed.

1.2 Virtual Experiments

The expression *virtual experiments* used throughout this thesis describes experiments in pedestrian dynamics conducted on a computer. Experiment participants control a figure in a virtual world. With the figure, also called *pawn* or *character*, they participate in experiments in pedestrian dynamics which are set up in the virtual environment.

Trajectory tracking in real life experiments improved throughout the years but still faces a significant number of problems, especially in high density crowds. In [2] Boltes identifies common difficulties as people overlapping on pictures due to height differences and objects falsely being classified as people.

Additionally to the aforementioned problems in tracking, real life experiments take considerable effort to conduct. Participants have to be invited, a venue has to be prepared and data has to be categorized. Real life experiments cannot provide pedestrian dynamics data from real danger situations either since it would be immoral to expose experiment participants to dangers such as fire.

In an attempt to solve these problems, researchers have considered the possibility of virtual experiments. In a virtual world, participants would merely expose their controlled pawns to a dangerous situation. Another advantage includes easy tracking since the position of the controlled pawns can be determined via a simple method call.

It is still in question whether virtual experiments provide the same reliable data as real life experiments. If future works confirm this, virtual experiments could replace real life experiments as they are less expensive in costs and effort.

1.2.1 Recent Virtual Experiments

Virtual experiments in pedestrian dynamics were conducted at Forschungszentrum Jülich in March 2016 by the Bergische Universtät Wuppertal (henceforth referred to as *Wup-*

[2]see www.basigo.de/aktuelles/artikel/article/experimente-zur-fussgaengerdynamik-forscher-betrachten-bis-zu-1000-personen-gleichzeitig.html (retrieved August 16, 2016)

pertal experiments). The goal is to determine whether virtual experiments provide data similar to real life experiments. Figure 1.3 shows one of the virtual experiments which are set up with the software Vizard 5, a tool for creating three-dimensional environments. The participant wore an Oculus Rift Head Mounted Display (HMD) (see 1.4) and used the keyboard to navigate in the virtual world. The experiment was an Exit-Choice-Experiment in which the participant controlled a person at the end of a group. The other pedestrians in the group were controlled by a computer and moved on a predetermined path. This path was created by using trajectories from real life experiments.

Figure 1.3: Virtual experiments in Vizard 5

Figure 1.4: Oculus Rift HMD

The HMD was used to increase natural behavior in the virtual experiment. The use of an HMD creates immersion, which means that the participant feels like he is 'in the scene'. It is likely that participants show more natural behavior when feeling immersed.

Some aspects of virtual experiments can be improved by a multiuser approach. Since the other pedestrians were computer-controlled in recent experiments, they did not react to collisions with the person controlled by the participant. In real life, a reaction to a collision might influence the participant's path. The participants were told to keep their distance to other pedestrians and stay at the end of the group, which influenced the participant's behavior to be less natural than in real life where the participant might walk faster than other pedestrians. These problems can be mitigated by using a multiuser approach and allowing participants to react to each other.

The evaluation of the Wuppertal experiments is still in progress but the multiuser platform can be used in various ways despite the result. If the data of the Wuppertal experiments turns out to be of small significance it can be tested whether a multiuser approach provides more significant data. If the data is significant it can be used to conduct more virtual experiments.

1.3 Unreal Engine 4

A virtual pedestrian dynamics experiment is similar to a video game in many ways. The participants can be seen as players who connect to the experiment structure, the level, over a network. As in a game, each player has a goal. In most pedestrian experiments the goal is to leave the experiment room as quickly as possible without running over other pedestrians.

Because of these similarities it seems only natural to use a game engine to create the multiuser platform. Unreal Engine 4 (UE4) is a widely used game engine by Epic Games which is known for its high graphical quality rendering. It is free and open source. A royalty is charged for commercially released projects which does not apply to virtual experiments in pedestrian dynamics in research.

As evaluated in an earlier paper, UE4 brings many advantages over the software Vizard 5, which was used in previous experiments [11]. UE4 can be programmed using C++ as well as *event graphs*, a visual programming method native to Unreal Engine. Combining these two methods effectively makes setting up a virtual experiment in UE4 easier than in Vizard 5. UE4 also comes with a vast library of game related functions. This especially includes multiplayer functionality. Because of all the assistance it offers, UE4 is used in this thesis to create the multiuser platform.

1.3.1 Event Graphs

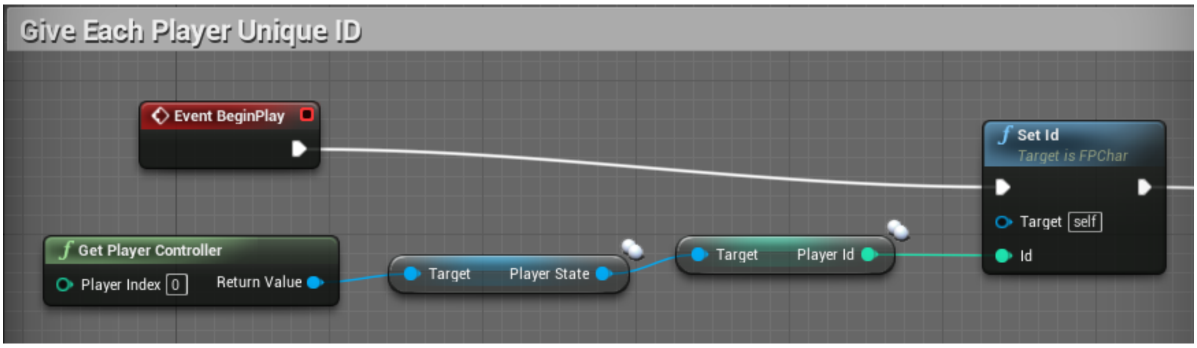


Figure 1.5: Event graph example in UE4

Figure 1.5 shows an example event graph in UE4. Event graphs are composed of different nodes (red, blue and green) which are similar to C++ method calls. Red nodes mark the beginning of an execution flow and represent C++ method declarations. White lines mark the execution flow and connect different execution nodes. In the example the method *BeginPlay* starts executing and then calls the method *Set Id*. Blue nodes, such as *Set Id*, are callable methods which are connected to other execution nodes (white) and are usually used for writing rather than reading operations.

Just like method calls in C++ nodes can have input parameters and return values. Parameters are indicated by round connectors, like *Target* and *Id* for the *Set Id* node. These parameters can be given by other method calls or call chains. Green nodes are functions which return values but do not contain writing functionality. They only read values and pass them to other methods as parameters. In the example, the function *GetPlayerController* returns a player controller which is passed to the next reading node as a parameter. In the end, the chain results in the player's *ID* which is then set. Apart from red, blue and green nodes event graphs can also contain black or grey nodes which represent control structures such as branches or switches.

1.4 Serious Games

Virtual experiments in pedestrian dynamics are similar to video games but their main purpose is research and not entertainment. Games *"in which education (in its various forms) is the primary goal, rather than entertainment"* [6, p. 17] are called serious games. They were first introduced by Abt in 1970 as educational games mostly based on paper [1]. The definition was updated by Sawyer in 2002 to include the new video game industry [9]. Sawyer influenced recent definitions such as the one above. Example uses of serious games include skill training [6, pp. 155], tackling medical issues [6, p. 180] and general education [6, pp. 112].

Figure 1.6: Promise - a serious game for training employees in the oil and gas industry

In this case the goal is to 'educate' the researchers conducting the experiment by exposing data about the behavior of pedestrian crowds. Therefore, the multiuser platform could be described as a serious game by the above definition.

On the other hand, although not mentioned in the definition, serious games are usually intended to educate the player rather than gather research data. They are supposed to make learning or training a fun experience. The purpose of virtual experiments in pedestrian dynamics is not training the participant but gathering data. At first, this seems to clash with the common view of how serious games are used. But a more elaborate explanation of the term *serious game* includes that the game aspect is supposed to motivate players to learn. While serious games are not always a voluntary activity, they can still be entertaining [6, p. 21]. Applied to virtual experiments in pedestrian dynamics, developing the multiuser platform with a game aspect can motivate participation.

Beyond the simple similarities to a video game (see 1.3), this justifies the use of a game engine to create the multiuser platform. Even if the multiuser platform differs from the usual understanding of serious games, keeping the idea of serious games in mind provides a more defined route for development.

Video game technology is already being used for scientific purposes in multiple ways. Apart from serious games, another example is the Oculus Rift HMD, a virtual reality device developed mainly for games and entertainment. But the Oculus Rift can also be used to visualize scientific data, e.g. Drouhard and Steed use the Oculus Rift to explore crystal structures and neutron data [3].

1.5 Thesis Goal and Structure

The goal of this thesis is developing a multiuser platform for virtual experiments in pedestrian dynamics which mitigates the problems of past virtual experiments and provides a suitable basis for future virtual experiments.

The main feature of the software is that multiple participants can connect to the same experiment over the internet. Since participants do not have to travel to an experiment venue, time and costs of experiments are reduced. Automatic trajectory tracking and time measurement simplify data gathering and evaluation.

This thesis describes the functional requirements for the multiuser platform as well as a general concept of the software's architecture. It offers a detailed review on the software's components and its development. The thesis also focuses on how to extend the multiuser platform in the future, not only by replacing the map but also by adding new features.

After the above introduction and motivation, the second chapter contains the requirements for the multiuser platform. It lists desirable features as well as expectations regarding quality and performance. Some features are described as optional giving an outlook on further development in the future.

The main part of this thesis is the program description which can be found in chapter 3. It describes all classes and assets of the software in great detail. It shows how different components of the platform work together and in which order methods are called. The chapter also explains how to set up a functioning network for the platform.

The fourth chapter focuses on the extensibility of the multiuser platform. It explains what kind of software setup is necessary to develop the program and gives details about important interfaces. Adding new experiment structures is an important part of this chapter but it also shows how to develop new logic. This chapter is mainly intended for future developers extending the multiuser platform.

Chapter 5 contains a list of test cases for the program and their results. It describes normal cases, special cases, error cases and how the software handles them. The performance of the program is also tested and analyzed.

The last chapter gives a summary of the thesis by enlisting important features and achievements of the software that was developed. It also describes prospects for possible extensions and future use cases.

1.6 Distinction and Limits

No virtual experiments were conducted during this thesis but the developed software can be used for such experiments in the future. The thesis is not concerned with the significance of virtual experiments in pedestrian dynamics since it can also be used to test this significance and thus is useful either way. Since the focus of this thesis is the software itself and not the experiment setup, the geometry used is a simple exit choice setup, which is merely an example and can be easily replaced.

The choice of game engine in this thesis is based on a previous paper about the comparison between Unreal Engine 4 and Vizard 5 (see [11]). In that paper, the two tools were compared in their ability to create virtual pedestrian experiments. These insights are now used to actually create a software for virtual experiments. This thesis advances from previous works on virtual experiments by adding the ability to conduct virtual experiments with multiple participants at the same time.

1.6.1 Similar Works

In 2012, Ribeiro et al. developed a serious game to train evacuation behavior with the Unity game engine [8]. Their program is closer to the original understanding of serious games since they aim to train the player for evacuation scenarios. Only one player is trained at a time and the only data collected is the time taken. The software's goal is to train people for evacuation situations while this thesis focuses on experiments for understanding natural dynamics in evacuation situations. This thesis also differs from [8] by applying a multiuser approach.

Figure 1.7: Serious game for evacuation training (see [8])

Many other papers, such as [5] are concerned with simulating evacuation processes with models derived from past pedestrian dynamics experiments. They use data from real life experiments in order to anticipate crowd movement at large-scale events or evacuations. There are very few papers on virtual experiments in pedestrian dynamics and none applying a multiuser approach. This way, the multiuser platform fills a space of need for virtual applications in pedestrian dynamics.

1.7 Use Cases

When designing the multiuser platform, it is important to keep future use cases in mind. This section explains intended use cases as well as alternative use cases for which the software does not have to be changed.

The main purpose of the multiuser platform is conducting virtual experiments in pedestrian dynamics. In contrast to real life experiments, virtual experiments are low in cost and effort and could eventually be conducted with hundreds of participants. Researchers can distribute the platform to all participants and conduct an experiment over the internet without having to invite the participants to a specific venue. Researchers can create a geometry that fits their needs either within UE4 or in an external tool. They can recompile the multiuser platform with the new geometry and then use it to conduct the experiment and gather data about crowd and pedestrian behavior in different situations. This use case will mainly be applied by experts and experiment supervisors in the field of pedestrian dynamics.

Next to the main purpose the multiuser platform can also be used to validate results of virtual experiments in pedestrian dynamics. This is especially important since data significance has not been proven yet. Researchers can recreate real life experiments and compare the data of virtual and real life experiments. This can be used to find out which kinds of virtual experiments are relevant and can be conducted in the future. If real life experiments still need to be conducted, the multiuser platform can be used to test experiment setups before building them in real life. This is similar to companies which simulate a new product's capabilities before building a prototype.

Once the validity of virtual experiments has been proven the multiuser platform can be used in a variety of situations. When planning buildings or evaluating existing buildings, experiments can be conducted with the multiuser platform to test planned evacuation routes and ensure their sufficiency. An important advantage of the multiuser platform is that it can be used to conduct experiments in danger situations such as fire. These experiments, which are not possible in real life, can provide insightful data for researches to ensure the safety of buildings and public events.

The multiuser platform can also be used for evacuation training without having to change it. It is similar to the evacuation training software described in [8] but offers additional functionality like trajectory tracking. When not using this additional functionality, the geometry setup and time measurement capability can be used to train people to more efficiently evacuate from buildings or events. It advances from [8] since it can be used to train multiple people at once because of its network features.

2 Requirement Analysis

2.1 Overview

The multiuser platform developed in this thesis advances previous software for virtual experiments and mitigates problems such as limited freedom of movement (see 1.2.1). It is supposed to work like a video game, meaning it starts with a main menu where players can choose to connect to a server hosting an experiment. Players should be able to join a server from home. After joining an experiment, the experiment setup level is loaded and after completing the experiment goal, which means leaving the room, a game end message appears. The platform should be flexible to allow easy addition of background processing and in-game logic.

The next section contains a list of features the multiuser platform should provide. Next to important key features, some features are not strictly required. They might not be implemented during this thesis but are still viable extension ideas for further development.

The section about performance is concerned with the multiplayer functionality and stability of the software. It lists desired frame rates and an approximate number of clients the software should be able to handle without loss of stability.

Section 2.5 describes expectations for graphical quality and realism of the experiment.

2.2 Features

The multiuser platform requires a few key features to make it functional for its purpose. Other features are optional and could also be implemented in the future. The following sections 2.2.1 to 2.2.8 describe a variety of features which are suitable for the multiuser platform. Most features were designed in cooperation with experts in pedestrian dynamics and in virtual experiments whereas some stem from game design ideas, such as the main menu.

The key feature, which is multiplayer functionality is not described separately but is a part of all presented features, which means all feature descriptions contain mentions of multiplayer functionality.

2.2.1 User Interface

To make a simple work flow for the platform, the first thing participants should see when starting the program is a menu. Figure 2.1 shows a sketch of what the main menu might look like. It should offer options to host a new experiment or connect to an experiment server. The connection can be established by typing an IP address or by consulting an automatically loaded list of available servers. The main menu should also have a button to quit the program.

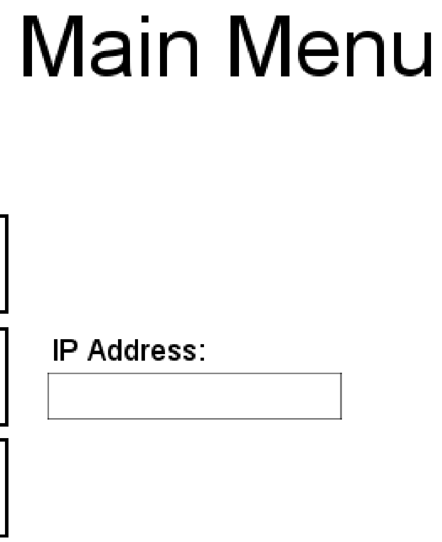


Figure 2.1: Sketch for the main menu

When hosting a new experiment, the server user should be transported to a lobby which shows the number of connected participants. The lobby should display buttons to either quit the game and return to the main menu or to start the experiment for all connected participants. When joining a server the user should also be brought to a lobby but without the player count and without the option to start the game, since this can only be done by the server. The user should still have the option to quit while in the lobby. When starting the experiment, all connected clients should be brought to the experiment map.

This menu brings a work flow to the software and allows the server to wait for participants to connect before starting the actual experiment. As an optional feature, a loading screen for the experiment could be implemented. There could also be an instruction screen showing details of the upcoming experiment.

2.2.2 Server List

To avoid confusion with IP addresses, the main menu could display a list of available servers. The servers should be searched for automatically. The user then should have the option to connect to one of the found servers by clicking on them.

This feature is optional because experiment supervisors have to pass information about the server to the participant in any case, so passing an IP address does not take additional effort. The server list is easy to use if only one server is active at the time since the list will find only one server for the user. But when the multiuser platform becomes widely used multiple servers could be active at the same time. In this case, passing an IP address is easier to ensure users connect to the right server.

2.2.3 Experiment Run

When the server starts the experiment, all participants are transported to the same experiment setup. The setup should be easily exchangeable to make flexible experiments possible. Experiment setups should have a defined starting point for all participants. As an example for the first implementation, a simple Exit-Choice-Experiment is used.

Since all participants are connected to the same server and experiment structure, they can see each other's pawns and movements. Pawns should be able to move freely through the experiment structure. They should be able to turn in any direction as well as walk forwards, backwards or sideways. However, they should not be able to walk through walls. Gravity should be in effect so that players cannot move upwards without sufficient structures to walk on, such as stairs or ramps. Pawns should not be able to walk through other players. In case of a collision, pawns should push each other away. Participants move their pawns through the experiment towards a defined 'finish line'. After passing the finish line the users should see a message that the experiment is over.

As an optional extension, pawns could be able to crouch and jump in reaction to danger situations such as fire. This feature is optional because currently pedestrian experiments do not allow jumping or crouching. With the possibility of virtual experiments, this might change in the future. When such experiments are planned, the feature could still be implemented.

2.2.4 Time Measurement

For possibly using the time for evaluation, the program should track the time any participant takes to complete the experiment. The time measurement should start as soon as the experiment setup is loaded and end when the pawn passes the finish line.

As an optional feature the time could be displayed on screen for every participant. Displaying the time might enforce the experiment goal to get to the finish line as quickly as possible but might also distract participants from the experiment. Therefore, if a time display is implemented, developers should be able to turn it off easily.

2.2.5 Trajectory Tracking

In order to simplify the evaluation of virtual experiments over real life experiments the software should automatically track and save trajectories of all participants. The trajectory data should be stored in a suitable manner (see 2.3) to allow output files similar to those of real life experiments. Trajectory tracking should start at the beginning of the experiment and end as soon as the pawn passes the finish line.

Automatic trajectory tracking is one of the main features of the multiuser platform because it is one of the greatest advantages over real life experiments. The location of a pawn does not have to be determined by complex image processing but can be retrieved with a simple method call.

2.2.6 Output Files

After completing an experiment, trajectories and other evaluations should be saved into output files. These files should only be created on the server. The output files should be generated in accordance with files from real life experiments (see 2.3). The output file for a participant's trajectory should be created after this participant passes the finish line.

In the future, other evaluation methods could be implemented such as calculating density from all participants' trajectories. It might also be possible to connect external tools for trajectory evaluation, e.g. *JuPedSim*[1], to the multiuser platform. These options are not implemented in this thesis in order to keep the first version of the multiuser platform of general purpose and highly flexible.

2.2.7 Spectator Mode

Participants might be interested in watching the rest of the experiment after they have already finished. Also, experiment supervisors should be able to overview the experiment on the server computer without actually participating. Therefore, the program should offer a spectator mode. The spectator mode should be available for the server computer at any time and for participants after completing the experiment. A possible way of doing this would be to allow the participant to return to the experiment as a spectator by pressing a certain key on the keyboard or a button on the screen.

Spectators should be able to move around the scene freely without the application of

[1] see www.jupedsim.org/ (retrieved August 16, 2016)

gravity. They should be invisible to participants which are still in the experiment. Multiple spectators watching the experiment cannot see each other and can move through each other.

2.2.8 Fire

One advantage of virtual experiments is being able to expose participants' pawns to dangers such as fire without causing real physical damage. Developers should be able to place a fire within the experiment structure and define its size. As an optional feature the fire could expand over time.

In order to make participants stay away from the fire, they should have a certain amount of health which decreases when stepping into a fire. When reaching 0 health, the experiment should end for this participant.

2.3 Trajectory Data

Trajectories should be stored in accordance to trajectory files from real life experiments. Trajectory files consist of commentary lines and lines of trajectory points. This data format is a requirement because it is used by evaluation software such as *JuPedSim*. While not implemented in this version of the software, future versions could include evaluations made by external tools like *JuPedSim*. Therefore, output files generated by the multiuser platform should use this data format.

Listing 2.1: File with pedestrian trajectory

```
# id frame x/cm y/cm z/cm rot.x rot.y time/s
257 1 2950.480 1560.760 161.093 -1.000 0.000 0.266
257 2 2950.480 1560.760 161.093 -1.000 0.000 0.283
257 3 2950.480 1560.760 161.093 -1.000 0.000 0.300
257 4 2950.480 1560.760 161.093 -1.000 0.000 0.316
257 5 2950.480 1560.760 161.093 -1.000 0.000 0.333
257 6 2950.480 1560.760 161.093 -1.000 0.000 0.350
257 7 2949.856 1560.483 161.093 -0.913 -0.406 0.366
257 8 2948.609 1559.928 161.093 -0.913 -0.406 0.383
257 9 2946.737 1559.096 161.093 -0.913 -0.406 0.400
```

Each line begins with the ID of the trajectory which is the participant's unique ID. The next column contains the frame number for this line. The next three columns are location coordinates. The first is the coordinate on the x-axis in centimeters which is used as a unit because it is the usual level of detail real life experiments work with. The second column is for the coordinate on the y-axis and the third for the z-axis. The z-coordinate is usually the height of the participant which does not change throughout the experiment. In virtual experiments it can also be used as the z-coordinate in a three-dimensional world since virtual experiments can have different height levels in structures. As an optional addition, the trajectory point may contain two floating point numbers which form a two-dimensional vector representing the shoulder rotation when viewed in a two-dimensional coordinate system. The coordinate system presents a birds eye view onto the pawn's head which lies in the coordinate origin. The shoulder rotation vector then points into the direction the pawn is turned towards. The last column contains the time stamp for each trajectory data point. It is used to synchronize trajectories in case of different frame rates on clients. The column is not used in real life experiment files since no frame rate drops are encountered when recording with a camera. Evaluation programs simply ignore this column if files which contain it are evaluated. In the future, time stamps can be used to interpolate trajectory points in case of frame rate drops to make them equidistant if necessary.

2.4 Performance

This section covers requirements on performance such as frame rate and the number of clients that should be handled. These requirements are guidelines for development and can be met within acceptable ranges.

The *frame rate*, or *frames per second (FPS)*, determines how smoothly the game runs. As Read explains in [7] human eyes perceive a sequence of images as motion at about 15 images per second (= frames per second). Movies are usually made with 24 FPS, which provides visual continuity but still retains the 'unrealistic' feel of a movie while movies made with 48 FPS are often perceived as 'too realistic'.

Video games, on the other hand, use higher frame rates because they are linked to monitor refresh rates. Monitors usually refresh at 60 Hz to avoid flickering. Read says in [7] that the human eye can distinguish 48 flashes of light per second. Any lower refresh rate would cause humans to perceive flickering which is distracting for computer users. Most modern video games run at 60 FPS to match the usual monitor refresh rates. Older games often run at 30 FPS, showing one frame for two consecutive refresh cycles. While a game with 60 FPS runs a lot smoother, it also requires twice as much rendering as a 30 FPS game which can require significantly more powerful hardware. If the hardware is not sufficient, the frame rate might occasionally drop which is distracting. In this case, a stable game with 30 FPS is better than an unstable game with 60 FPS, as also mentioned in [7].

When using an Oculus Rift for virtual experiments, a higher frame rate is key to reduce motion sickness and flickering. Therefore, 60 FPS are a desirable goal for the multiuser platform so that the Oculus Rift can be used easily.

In order to successfully use the multiuser platform for virtual experiments in pedestrian dynamics, the software should be able to handle a certain number of clients. Usually, experiments have 20-30 participants, e.g. the Wuppertal experiments, but occasional experiments might have more, such as the *BaSiGo-Experiments* (see 1.1). The prototype application should be able to support about 30 connected clients while future improvements could increase performance for even more clients.

The frame rate and possible number of clients highly depend on the used hardware. Usually, an unstable game can be stabilized by adding more GPU and CPU power. Thus, this requirement is mainly about the used game engine being able to create stable games with the required frame rate and number of connected clients.

2.5 Quality

In order to increase the feeling of immersion and thus make participants behave more naturally the multiuser platform should provide high graphical quality. The scene should be displayed smoothly and the participant should not be able to see individual pixels on screen. The lighting should be natural. There are no specifications for the materials that should be used for walls or floors because in real life experiments these are not always natural materials either. Any kind of simple material can be used for walls and floors.

Pawns should look like humans and also be animated accordingly. They should have idling, walking and running animations which are chosen according to the pawn's current speed. There should be different visual representations of pawns, use of which should be chosen randomly.

While this section describes the desired graphical quality and realism of the software, it is not concerned with an exact setup for a pedestrian experiment. The geometry used in this prototype is merely an example and can be replaced easily for future use cases.

3 Program Description

This chapter aims to describe the multiuser platform and its workflow in detail. It lists individual classes and how classes interact with each other. Thereby it gives an overview of the whole program.

3.1 Architecture

A program developed with UE4 consists of three main components: *maps*, *actors* and *UObjects*. Maps contain the game environment which is the world the player can interact with. They can be menus as well as game structures. Multiple maps can be defined per game as they represent levels between which the player can travel. Actors are assets that are placed on game maps. Actors can represent anything that is relevant to the game such as a wall or the player's pawn. Generally, every object in the game environment can be classified as an actor, which can be either static or dynamic. The term *UObjects* describes classes in UE4 derived from the *UObject* class. These classes are used to assist the game but are not placed directly in the game environment. An example of this are widgets for player head-up displays (HUDs), which display important information to the player. These widgets assist the game but are not part of the game world since they are an overlay on the player's screen. Another example for *UObjects* are trajectory data points, since those are not placed in the game environment.

Maps, actors and *UObjects* can use C++ programmed functionality as well as event graphs. For all classes in this program, a C++ class was created first, then a *blueprint class* was derived from it. A blueprint class contains event graphs as well as the visual representation for the actor. This ensures that both C++ programming and visual event graphs can be used to create logic for the new actor. Each map has a *level blueprint* which contains event graphs that affect the whole map such as placing a new component at the beginning of the game. While a blueprint class contains event graphs and a visual representation for an actor, the level blueprint consists only of event graphs that affect the whole level rather than one specific class.

When looking at the proposed features for the multiuser platform, one can identify different modules of the program, such as different maps and classes. The implemented multiuser platform consists of four maps, of which one is the experiment structure and the other three are menus. Figure 3.1 shows the implemented map flow. The program starts with a main menu in which the user can choose to host an experiment or join an existing one using the server's IP address. When hosting or joining, the user is placed in the lobby, which is used as a waiting area. The hosting user can see how many participants have joined and can choose to start the experiment. When the hosting user clicks *Start*, all connected users travel to the experiment level and the experiment begins. Compared to the maps derived from the feature list in chapter 2 a new map has been added for the game end screen. The player travels to this map after completing the experiment by passing the finish line and may then return to the game level as a spectator. The whole program flow is shown in the sequence diagram in figure 3.3. It also includes the participant walking into a fire which is not intended but still belongs to the multiuser platform's functionality. To keep it simple, the sequence diagram does not have lifelines for the main menu and the lobby.

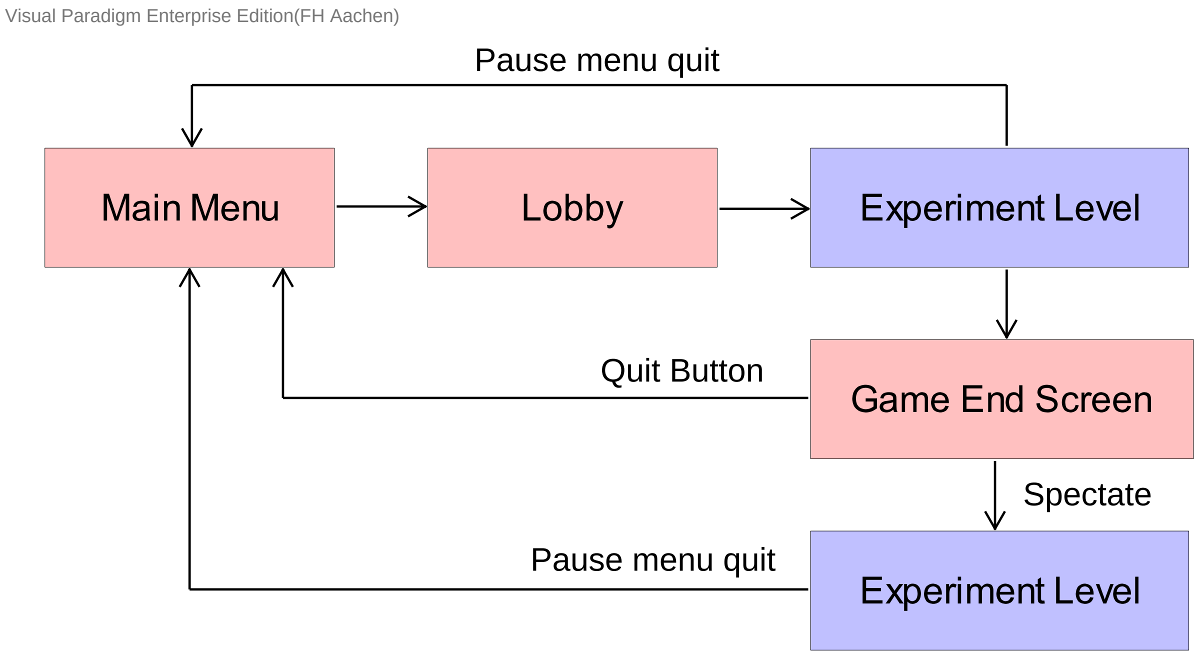


Figure 3.1: Map flow

The program can be divided into two parts which are menus (red) and the experiment level (blue). The two parts are separate and no information is exchanged between them. They are merely connected by travel commands. The menu is used to connect the players to the experiment server while the experiment level stages the experiment itself and processes background data. Both parts can be identified in figure 3.2 which shows the general architecture of the program. The parts are explained in greater detail in the following sections.

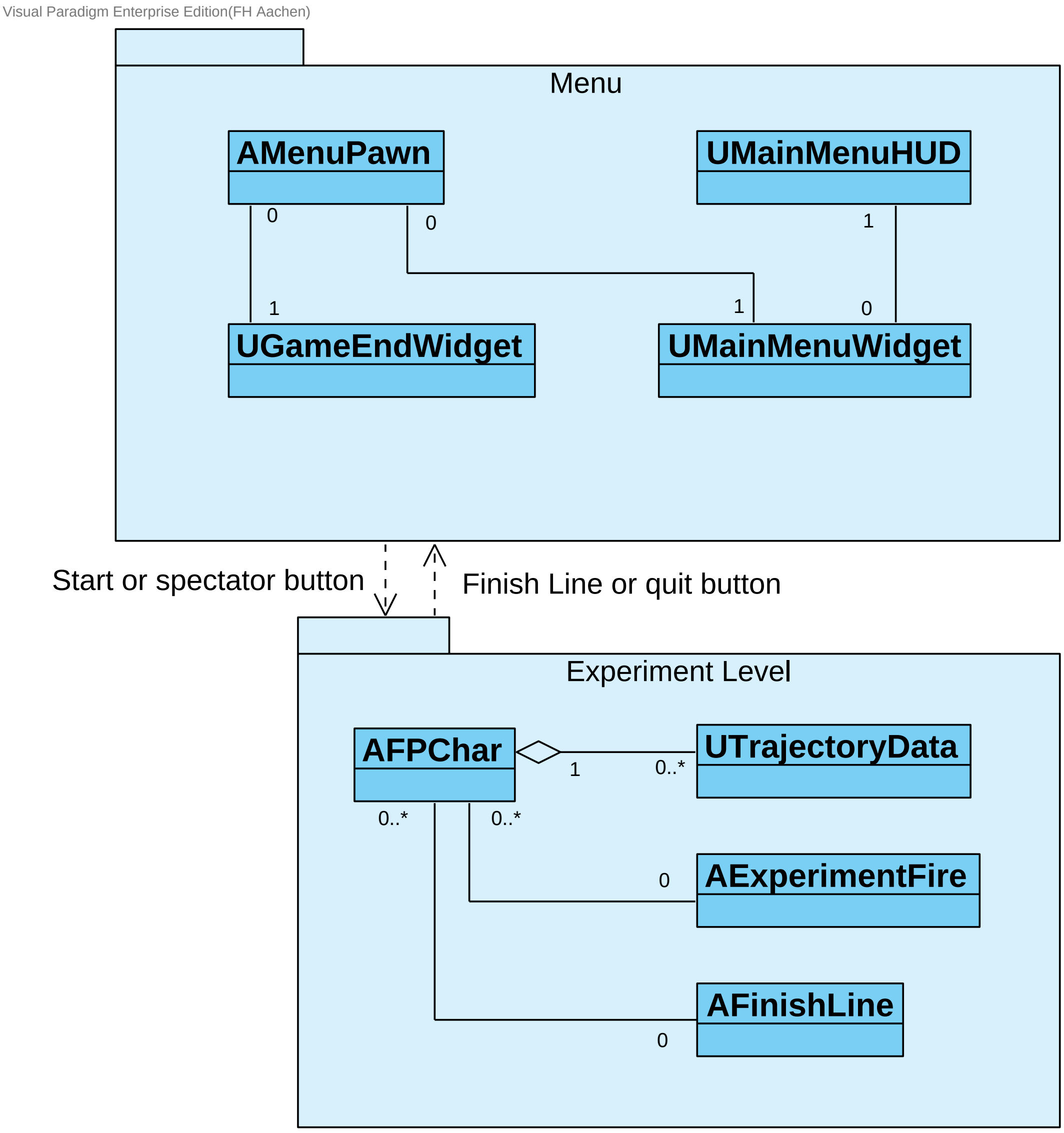


Figure 3.2: General program architecture

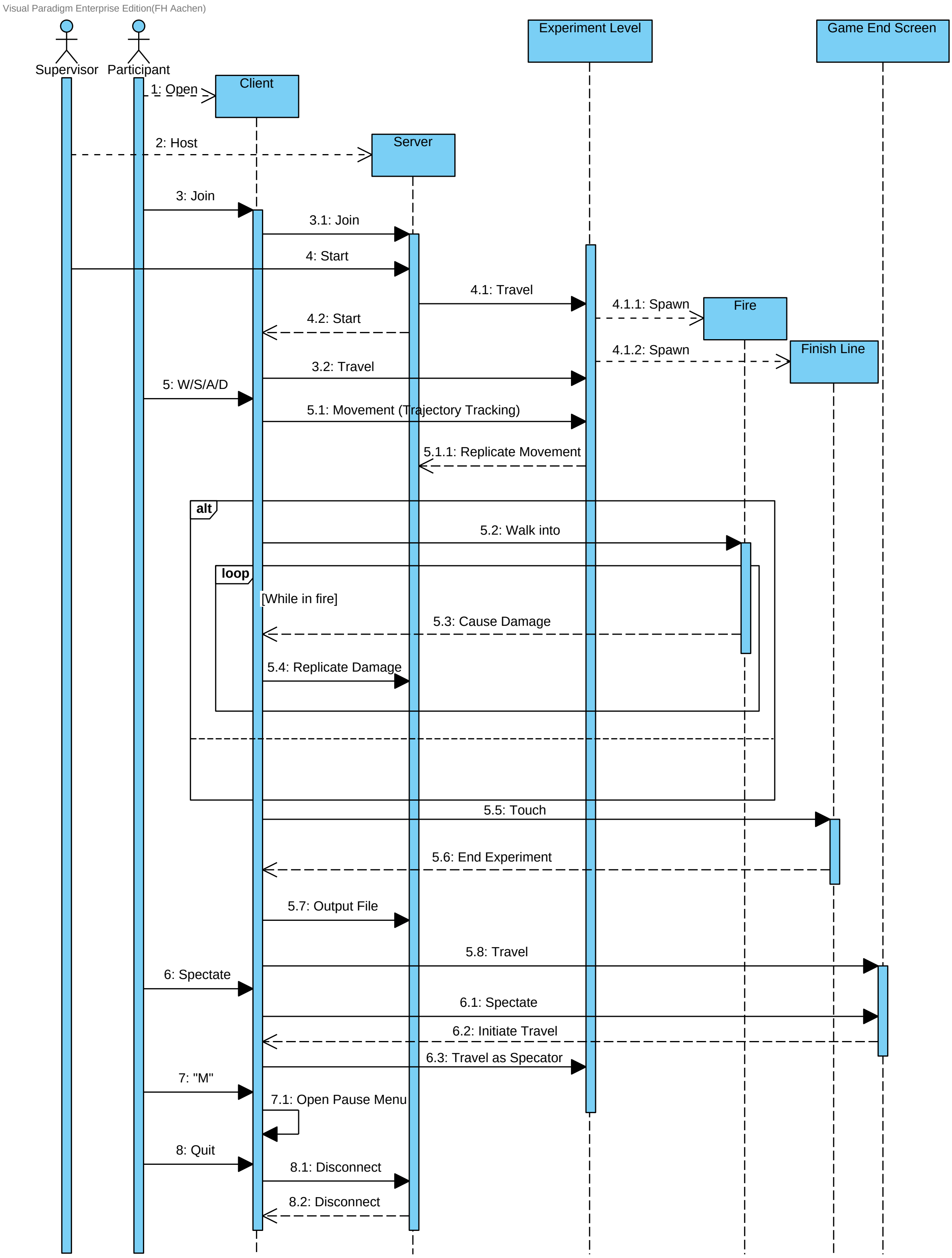


Figure 3.3: Simplified sequence diagram

The feature list introduces a number of components that are necessary for the experiment setup. These can be viewed as classes which are implemented within the multiuser platform and which can hold variables. An important class is the player's pawn which holds all player specific values. Other classes are the *Finish Line*, which opens the game end screen upon collision with a player, and a fire component, which decreases a pawn's health while in contact with it.

The program works with a client-server-architecture for networking functionality. Each participant controls one client which connects to the main experiment server over the internet. The network functionality is described in greater detail in section 3.6.

3.2 Controls

The user interface and the experiment itself are navigated with mouse and keyboard or with a gamepad. With a keyboard, participants can use the *W/S/A/D* keys to walk forward (*"W"*), backwards (*"S"*) or sideways (*"A"* and *"D"*) during the experiment. The mouse can be used to turn or look up during the experiment. When pressing *"W"* while looking up the participant still walks forward and not upwards since gravitiy applies. This is different for spectators since they can move without gravitiy. If a spectator looks up or down and then presses *"W"* he moves exactly in the direction he is looking at.

When using a gamepad the right stick works in the same way as the mouse and the left stick functions as the *W/S/A/D* keys.

3.3 User Interface

The main menu and the lobby use three classes to be functional. Figure 3.4 shows the class diagram for all menu objects including a fourth class which is used for the game end screen. The *MenuPawn* is a pawn the player uses to navigate the menu. In this case it is the mouse cursor. The *MenuPawn* is also used in the game end screen. *UMainMenuHUD* is the class that generates the menu background. The menu map itself is completely empty until the *MainMenuHUD* displays something visible. The HUD also administrates whether the main menu or the lobby should be shown. The menu is handled by the *MainMenuWidget*. The widget is added to the player's screen by the *MenuPawn* when the program starts and contains all interactable objects such as buttons and textboxes.

The *MainMenuWidget* is used for both the main menu and the lobby. To decide which buttons and texts should be shown it calls upon the *UMainMenuHUD* for its attributes *MainMenuActive* and *LobbyActive*. These are set by the level blueprints of the main menu map and the lobby map. UE4 offers automatic binding of certain attributes to functions. When the widget is added to the viewport all clients and the server call the

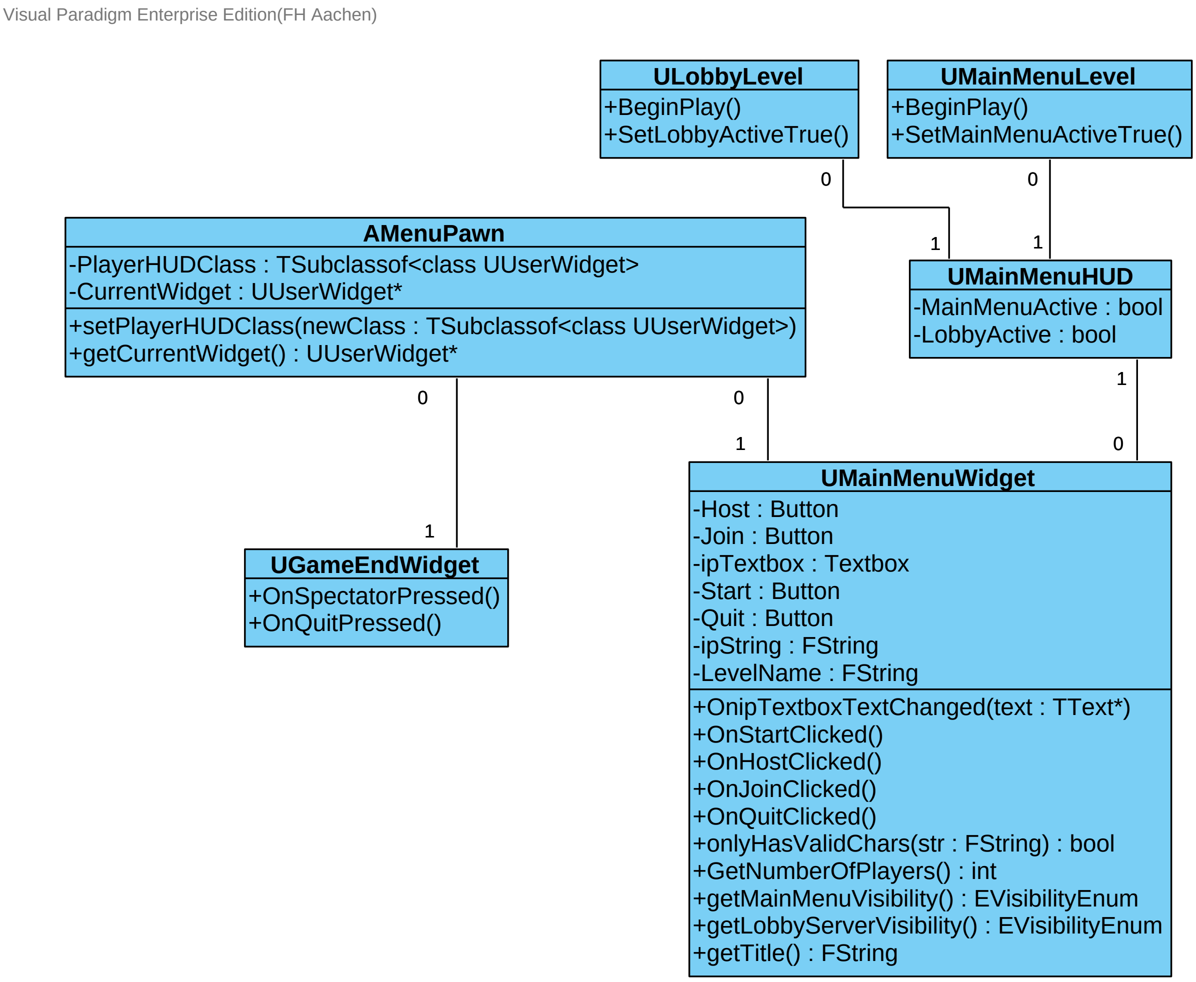


Figure 3.4: Class diagram of menu classes

bound functions to determine whether interface objects are visible. This way the visibility of the main menu buttons is bound to the function *getMainMenuVisibility* which returns visible if *MainMenuActive* is true. The *Start* button's visibility is bound to *getLobbyServerVisibility* which returns visible when *LobbyActive* is true and the calling player is the server.

In the main menu the user has the option to either host a new experiment, join an existing one or quit the application (see figure 3.5). When the *Host* button is clicked, the user travels to the lobby as a server (see figure 3.6). If the *Join* button is clicked, the IP address entered in the *ipTextbox* is used to connect to an existing server. Before connecting, a simple syntax check is performed scanning the IP address for invalid characters. Numeric characters as well as *"."* and *":"* are considered valid. If the entered string does not contain invalid characters a connection is attempted. Since the server is currently in the lobby, the user will also travel to the lobby on successful connection

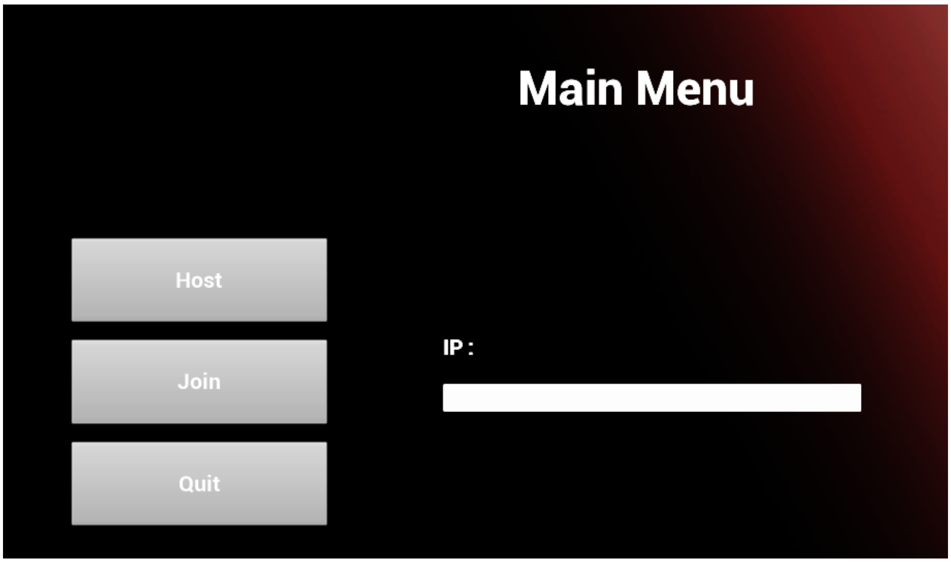


Figure 3.5: Main menu of the multiuser platform

but will see less UI elements than the server. The IP used to connect to the server will be saved in a *SaveData* blueprint for future use. Only the server can see how many participants are connected and choose to start the experiment. If the server user clicks *Start*, all connected clients travel to the experiment level and start the experiment. A client can only see the *Quit* button in the lobby. Clicking the *Quit* button closes the program. If a client clicks *Quit* it is disconnected from the server. If the server user clicks *Quit* all clients are disconnected from the server.

The *MenuPawn* is used in the main menu, the lobby and the game end screen. The game end screen is used after a participant finishes the experiment (see figure 3.7). Since the game end screen uses a different widget, the *GameEndWidget*, the *MenuPawn* offers methods to change the current widget by using the method *setPlayerHUDClass*. The game end screen uses the same HUD to draw the background, but both *MainMenuActive* and *LobbyActive* are set to false. The *GameEndWidget* only displays text and offers an interaction to return to the game level as a spectator by clicking a button. Alternatively, the player can quit the program by clicking another button. When returning to the game level as a spectator, the program loads the IP which was saved when connecting to the server in the main menu from the *SaveData* blueprint.

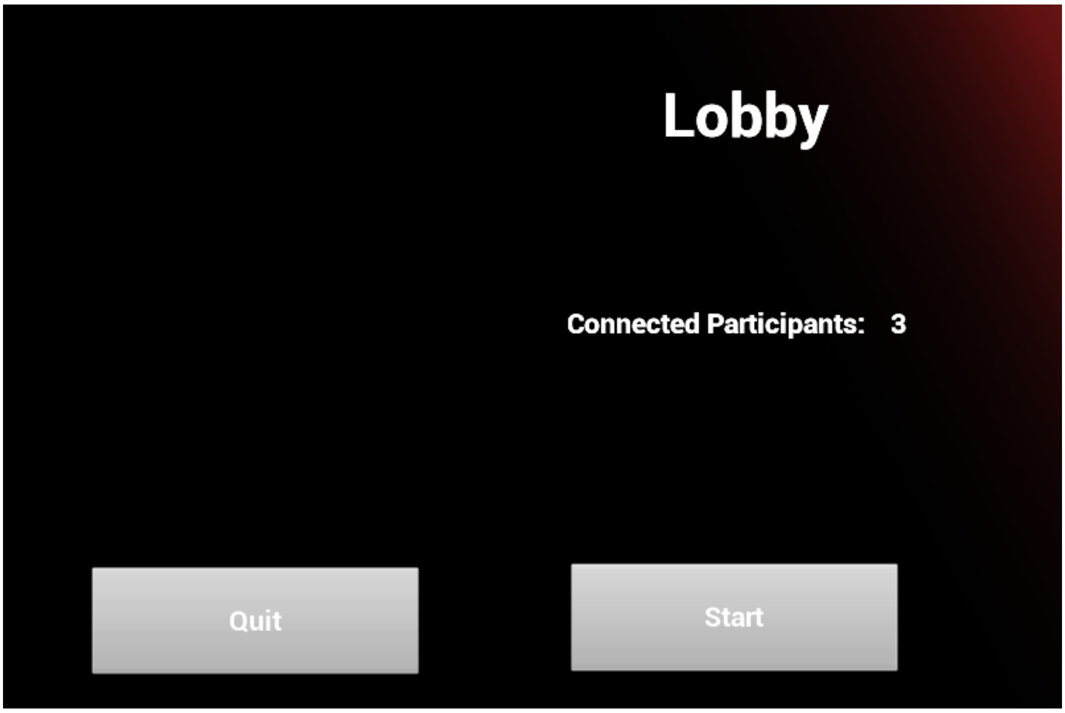


Figure 3.6: Multiuser platform lobby as server

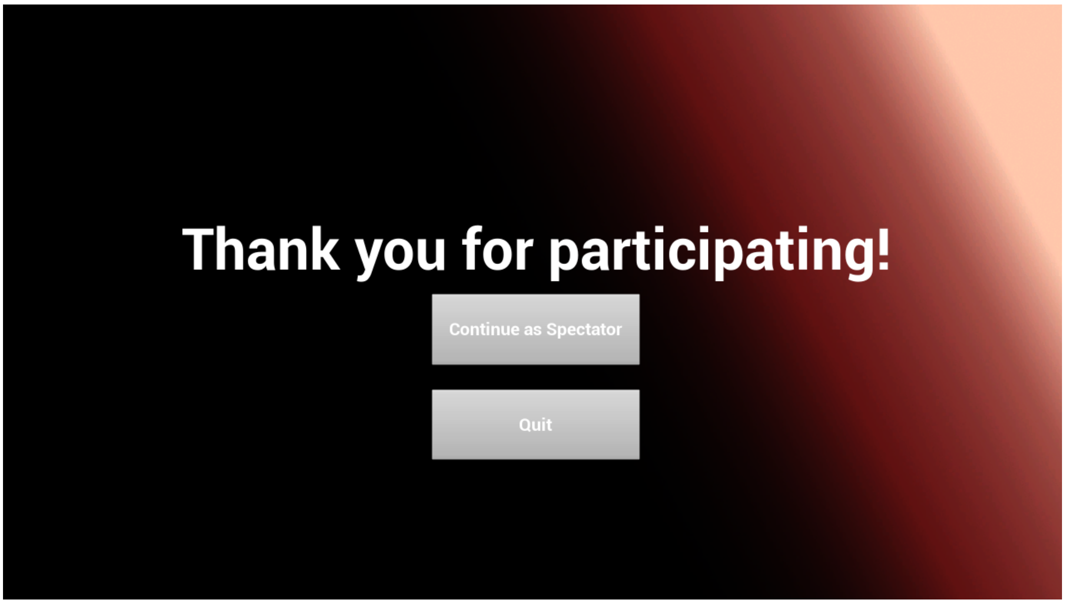


Figure 3.7: Multiuser platform game end screen

3.3.1 Pause Menu

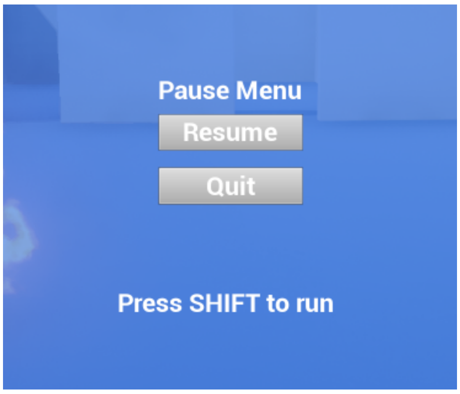


Figure 3.8: Pause menu

During the experiment the participant is able to open a pause menu by pressing *"M"*. The menu lets the participant either resume the experiment or quit the program (see figure 3.8). If the participant clicks *Quit* he is disconnected from the server and returns to the main menu. Character movement is disabled while the game is paused but the participant's character is still visible to other participants. Time measurement continues while the pause menu is open but the player's HUD does not update until the pause menu is closed again. By clicking *Resume* movement is re-enabled and the participant can continue the experiment.

This functionality is implemented in the class *BP_FPChar* which stores a reference to the *PauseMenuWidget* and adds it to the player's screen when the *"M"* key is pressed. The functionality of the buttons is implemented in the *PauseMenuWidget*.

3.4 Experiment Level

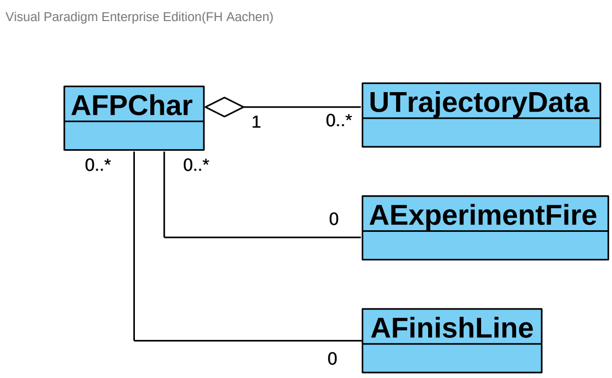


Figure 3.9: Simplified class diagramm for the experiment level

A simplified class diagramm of all classes relevant to the experiment level is shown in figure 3.9. Detailed diagrams for the individual classes are given in the respective sections below. The experiment level starts after the server clicks the start button in the lobby. All connected clients are transported to the experiment level and the experiment starts immediately. Every participant gains control of a pawn which is of the class *AFPChar*. The only exception is the server player who enters the experiment level as an invisible spectator and does not control an *FPChar* (first person character). This class stores a

TrajectoryData point in a list every frame. From this list an output file with the participant's trajectory can be generated at the end of the experiment. When an *FPChar* walks into an *ExperimentFire*, the fire starts deducting health from the character. If a character walks through the invisible *FinishLine*, the finish line will trigger the end of the experiment after a fixed amount of time. This includes saving the particpant's trajectory into a file and transporting the participant to the game end screen.

The example geometry is modeled with an *Exit-Choice-Experiment* in mind. This means there is a room with one entrance on one side and two exits on the other side (see figure 3.10). The room can be open towards the sky or have a ceiling. There are gathering areas in front of the entrance and behind the exits so that participants stay in a closed system and cannot leave the experiment setup. The entrance and exits in this setup are openings in the walls and do not have doors.

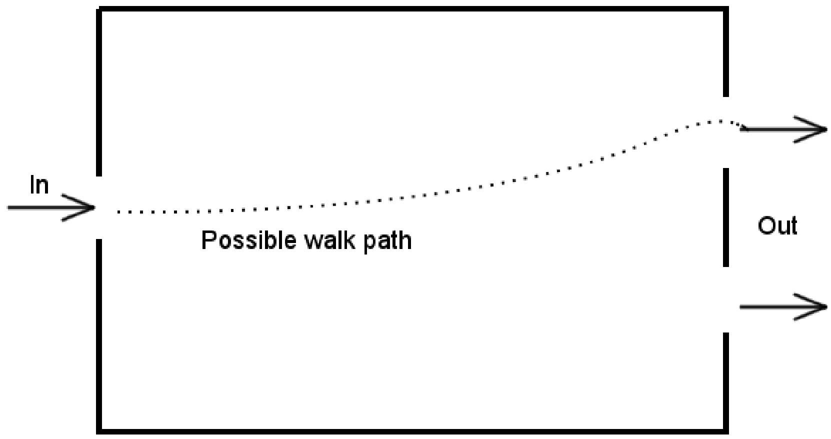


Figure 3.10: Exit-choice-experiment (sketch)

3.4.1 First Person Character

AFPChar is the most important class of the project since it contains the basis for all the main features. The C++ class is called *AFPChar* because it is an ***A**ctor*. The derived blueprint class is called *BP_FPChar* and can be found in the *Blueprints* folder (see 4.2 for folder hierarchy). Since the platform works with first person view, a participant cannot see their own pawn, only those of other participants. Therefore the mesh used is chosen randomly for every participant. In the editor, the pawn's skeletal mesh is set as female. At the beginning of the experiment every pawn has a 50% chance of being turned into a male by changing its skeletal mesh during runtime.

Figure 3.11: Male character

Figure 3.12: Female character

AFPChar inherits from the Unreal Engine class *ACharacter*. Thus, it comes with a movement component which simplifies setting up movement. The user's input is taken in on defined axes which are usually bound to keys on the keyboard or joystick inputs. Axes are set up in the project settings. The method *SetupPlayerInputComponent* is automatically called when constructing the character. It binds the methods *MoveForward*, *MoveRight*, *Yaw* and *Pitch* to the defined axes. *MoveForward* and *MoveRight* both retrieve the current *Forward-* or *RightVector* and scale it with the amount given as a parameter as well as with the attribute *velocityScale*.

VelocityScale is used to adjust the maximum velocity by multiplying it with the movement input vector. It can be modified to make the character run faster. It is defined as a *UPROPERTY* so it can be edited in the FPChar details panel as well as in event graphs. For the multiuser platform the default value is 0.3 but it is increased to 0.6 while the player is pressing one of the *"Shift"* keys. When the *"Shift"* key is released, *velocityScale* is set back to 0.3.

The scaled *Forward-* or *RightVectors* are used as input for the movement component which then moves the character according to the input vector. The methods *Yaw* and *Pitch* pass the amount they are given as parameter directly to a method of the movement component which lets the player rotate or look up. The amount is scaled by a fixed float value of 180.0f, which defines the sensitivity of the according axes, and by the amount of seconds passed since the last *Yaw/Pitch* update.

Each player is given a unique ID by the system at the beginning of the experiment. The ID is retrieved from the system by the *BeginPlay* method in *BP_FPChar* and stored in the *id* variable. It is later used to generate a file name for the participant's trajectory file and also in the ID column within the trajectory file.

Figure 3.13: Class diagramm for the first person character

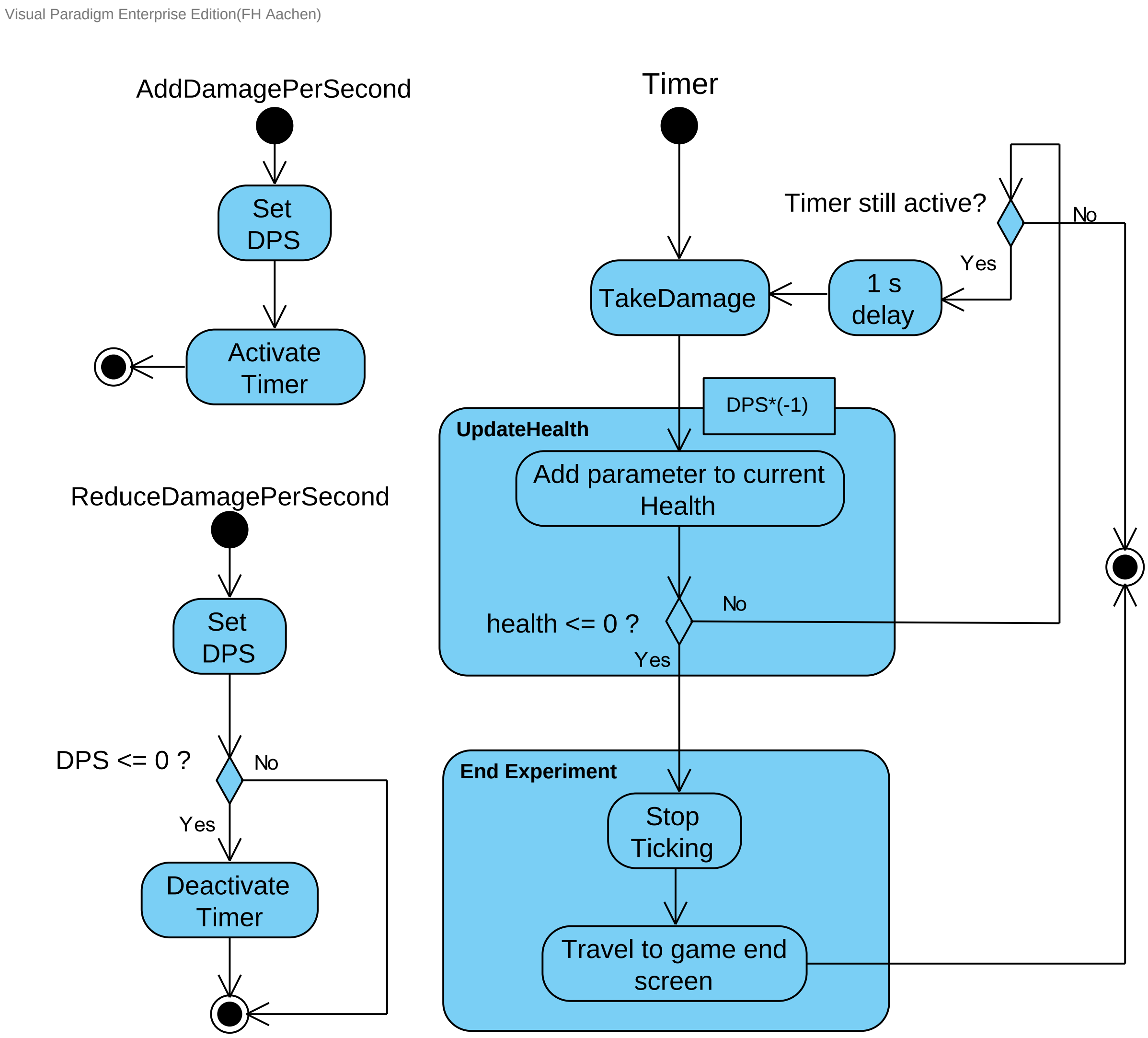


Figure 3.14: Flow diagram for damage system

The class *AFPChar* implements a simple health system in order to make use of danger situations within experiments. The flow of the damage system is shown in figure 3.14. Initially, each character has a health value of 100. The *DPS* variable (damage per second) stores the current amount of damage per second applied to the character. Damage sources can cause damage by calling the method *AddDamagePerSecond*, which adds the parameter to *DPS*. This way, multiple damage sources can add their damage but still only a single timer is used. *AddDamagePerSecond* starts a timer which causes the current amount of damage per second to the character by using the *TakeDamage* method which is a private method which is called every second by the damage timer. When attempting to start a timer that is already active, the timer simply remains active. *TakeDamage* multiplies *DPS* by -1 and passes it to the *UpdateHealth* method.

UpdateHealth adds the parameter value to the character's current health and checks if the character's health has reached 0. If this is the case, the end of the experiment is initiated by calling the method *EndExperiment* which stops trajectory tracking and immediately transports the player to the game end screen. This system is in place to 'punish' a participant for bringing their character into potentially dangerous situations as the goal is to avoid damage. Damage sources can use the method *ReduceDamagePerSecond* to stop causing their amount of damage to the character. If the remaining amount of damage per second is ≤ 0 the damage timer is stopped.

Listing 3.1: *Tick* method in *AFPChar*

```
// Called every frame
void AFPChar::Tick( float DeltaTime )
{
  Super::Tick( DeltaTime );

  //If the ticking hasn't been stopped
  if (ticking) {

    //Update Frame number and the trajectory array
    frame++;
    UpdateTrajectory();
  }
}
```

The *Tick* method in *AFPChar* (see listing 3.1) is used to save trajectory data points. This feature is explained in greater detail in section 3.5. The *Tick* mechanism can be stopped by setting the *ticking* variable to *false*. For example, the *Tick* method is stopped when a character passes the finish line or dies in a fire. Stopping the Tick method does not automatically save the trajectory to an output file. It merely stops the recording of further data points. The method *saveData* can be called to save the trajectory to a file. It is called by the finish line component when the participant completes the experiment.

3.4.2 Fire

The fire component created for this program consists of a fire particle system, which comes in the starter content package of UE4, and a collision sphere surrounding it. The collision sphere is invisible during program execution. When a character steps into the collision sphere, the method *OnComponentBeginOverlap* is triggered. If the character who stepped into the fire is a *FPChar* the fire calls *BP_FPChar*'s method *AddDamagePerSecond*. The damage value which is applied each second is stored in the float variable *DPS* and passed to *AddDamagePerSecond* as a parameter. It can be modified in order to make the fire cause more or less damage. It has to be a positive value since *AFPChar*'s *TakeDamage* method always multiplies the *DPS* value with -1. When

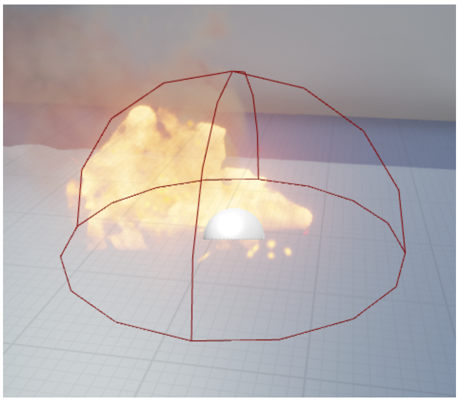


Figure 3.15: ExperimentFire actor in UE4 editor

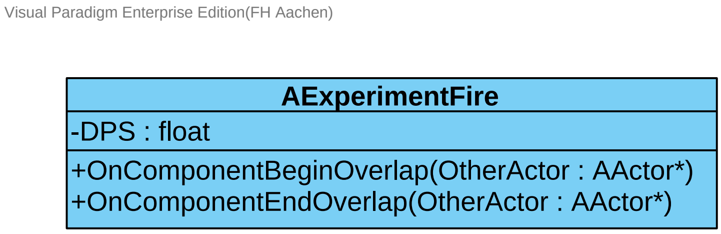


Figure 3.16: Class diagramm for ExperimentFire

the character steps out of the fire, the fire component calls the *ReduceDamagePerSecond* method on the character in order to stop causing damage.

3.4.3 Finish Line

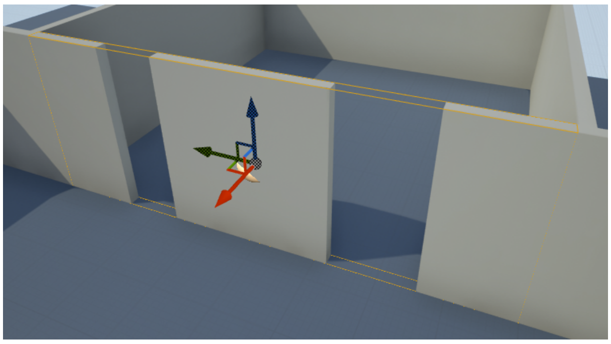


Figure 3.17: FinishLine actor (yellow) in UE4 editor

The *FinishLine* actor is responsible for ending the experiment for a participant. It consists of a collision box component which is invisible during program execution. When a character touches the collision box, the method *OnComponentBeginOverlap* is invoked. It initiates the saving of all relevant data such as the trajectory by calling *saveData* on the character given it is an *FPChar*. The finish line stops trajectory tracking for the character by setting its *ticking* attribute to *false*. On screen, the participant will see the time display stopping in their HUD. *FinishLine* then starts a timer which transports the character to the game end screen after 3 seconds. The delay is in place so that the participant can see their timer stopping and can prepare for the end of the experiment.

Multiple finish lines can be placed in an experiment level. They are meant to be placed at room or building exits. The finish line can be scaled in all directions. It does not matter when a character leaves the finish line collision box, which means that only the *OnComponentBeginOverlap* method is needed.

3.5 Background Data Gathering

During the experiment run, data is collected. In the version developed for this thesis this includes a trajectory for each participant and the time passed since the start of the experiment. Both are collected using the *Tick* method of the class *AFPChar*.

3.5.1 Time Measurement

The time since the start of the experiment can be retrieved at any time using the method *GetTimePassed* in *AFPChar* which uses a UE4 native function to get the time since the start of the experiment from the server. This means that the time passed is synchronized across all clients. This is important since it enables time stamps to be used to synchronize trajectories recorded at different frame rates.

The time passed since the start of the experiment is always displayed on the participant's screen. This is implemented with a widget called *PlayerHUDWidget*. The widget is added to every player's viewport at the beginning of the experiment. The text of the textbox is bound to a method which retrieves the passed time in a readable format from *AFPChar* using the *getTimePassedReadableFormat* method which converts the float value returned by *GetTimePassed* into a string.

The *PlayerHUDClass* attribute of *AFPChar* determines what kind of widget is added to the player's screen. *PlayerHUDClass* is also a *UPROPERTY* which means it can be edited in the details panel of *BP_FPChar* in the UE4 editor. In order to disable the time display one simply has to set the *PlayerHUDClass* attribute to *"None"*.

3.5.2 Trajectory Tracking

During the experiment a trajectory is tracked for each participant. It is saved as a list of *UTrajectoryData* objects. Every *UTrajectoryData* object represents one line in the trajectory output file (see 2.3). The data points are recorded with the *Tick* method in the class *AFPChar*. If *ticking* is set to *true* the method *UpdateTrajectory* is called which creates a new *UTrajectoryData* object. It stores the character's ID, the current frame number, the character's location, its shoulder rotation and a timestamp to synchronize different trajectories. Location and rotation are retrieved with methods native to UE4. The unit used is centimeters.

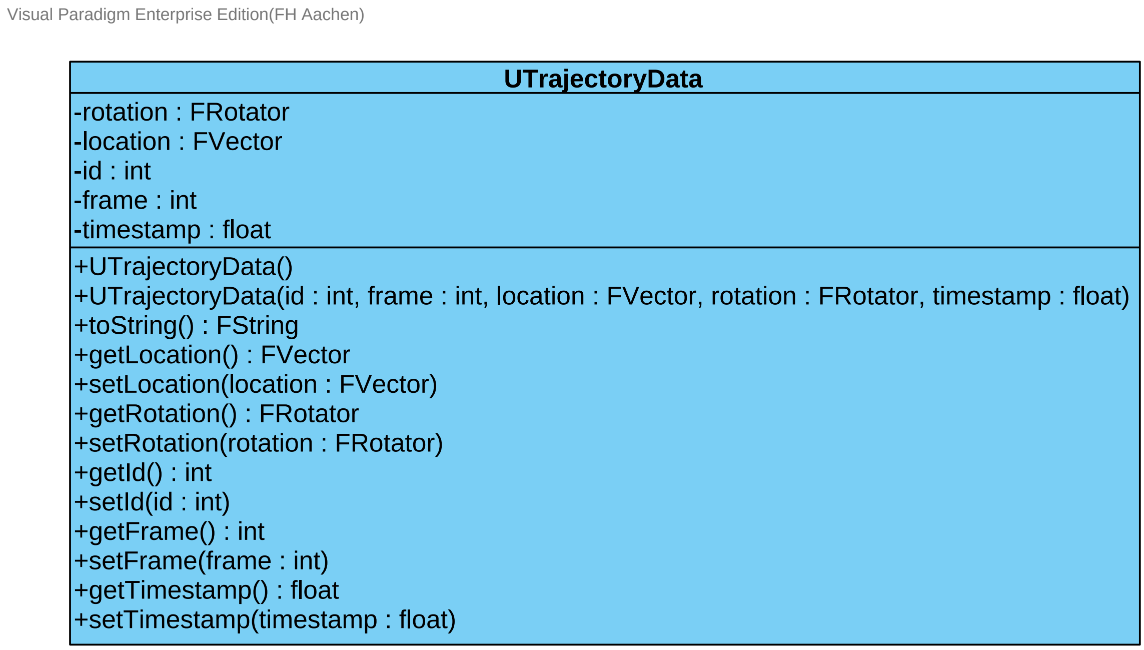


Figure 3.18: Class diagramm for UTrajectoryData

All coordinates refer to the level's coordinate origin. In the multiuser platform, this is one of the corners of the floor. The coordinate origin cannot be moved. Developers should rather build their level around it to make sure the program uses the desired point of reference for coordinates. Location coordinates are stored as a three dimensional vector of floats and the shoulder rotation as a two dimensional vector of floats. The elements of the three dimensional vector are in the same order as in the file. The first element is the x-coordinate, the second is the y-coordinate and the third is the z-coordinate. *AFPChar* also contains methods to save the list as an output file.

In the future the trajectories recorded at low frame rates can be interpolated to make them equidistant by using the time stamps saved with the data points.

3.5.3 Output File

The method *saveTrajectoryToFile* in *AFPChar* (see listing 3.2) saves the list of *UTrajectoryData* objects to an output file. The filename is generated by using the player's unique ID. The *toString* method of *UTrajectoryData* is called to create a string which is then written into the output file as one line with standard C++ file output tools. If an error is encountered while writing into the output file, the error message is displayed to the user.

Listing 3.2: Saving the trajectory in *AFPChar*

```
void AFPChar::saveTrajectoryToFile_Implementation(const FString& filename)
{
  std::ofstream outFile;
  FString line;
  outFile.open(TCHAR_TO_UTF8(*filename));

  if (!outFile) {
    FString error = "Failed to create output file at " + filename + "\n";
    GEngine->AddOnScreenDebugMessage(-1, 10.f, FColor::Red, error);
  }

  try
  {
    for (int32 i = 0; i < trajectory.Num(); i++)
    {
      line = trajectory[i]->toString();
      line.Append(TEXT("\n"));
      outFile << TCHAR_TO_UTF8(*line);
    }
  }
  catch (std::exception& e)
  {
    GEngine->AddOnScreenDebugMessage(-1, 10.f, FColor::Red, FString(ANSI_TO_TCHAR(e.what())));
  }

  outFile.close();
}
```

3.6 Network

This section explains the program's networking functionality and also describes how to setup a functioning server. Section 3.6.3 shows how the software handles network errors.

3.6.1 Client-Server-Model

Multiplayer functionality in UE4 uses an authoritative client-server-model and works with a system called *Replication*[1] which means that the server owns all of the actors and *replicates* them to all the clients. Basically, the client's representations of actors are copies which mirror actions of the server's representations. The server's representation is considered the 'correct' one while clients use an approximation since actors might have changed on the server since the client last received a replication.

[1]see https://wiki.unrealengine.com/Replication (retrieved August 16, 2016)

Movement and physics are automatically calculated on the server and replicated to all clients without the developer having to make adjustments for multiplayer. Most of the multiplayer functionality is built in with UE4. The developer does not need to program networking functionality itself but some functions have to be flagged to make sure they either run on the server or on the owning client. For example, traveling to a new map should be executed on the owning client so that only this one client will travel instead of all clients. Methods that deal with damage, on the other hand, should run on the server to prevent manipulation on client side.

In the multiuser platform, some methods of *AFPChar* are set to run on the server. Such are *AddDamagePerSecond*, *ReduceDamagePerSecond* and *UpdateHealth*. This makes sure the private *health* variable can only be manipulated by public server functions, and thus that the server has effective control over a player's health.

Listing 3.3: Server functions in *AFPChar*

```
UFUNCTION(Server, Reliable, WithValidation, BlueprintCallable)
  void UpdateHealth(float deltahealth);

UFUNCTION(Server, Reliable, WithValidation, BlueprintCallable,
   Category = "Collision")
  void saveData();

UFUNCTION(Server, Reliable, WithValidation, Category = "Trajectory
   ")
  void saveTrajectoryToFile(const FString& filename);
```

Another function which is executed on the server is *saveTrajectoryToFile* because trajectory output files should be generated on the server computer and not on clients. Since changes to the actor are replicated to clients but the method call itself is not, no files are created on client computers.

The method *EndExperiment* in *AFPChar* is a client function because it initiates travel. Setting this as a server function would cause all clients to always travel at the same time which is not desirable for the multiuser platform, e.g. when one participant finishes the experiment while others are still in the level, only this one client should travel to the game end screen.

3.6.2 Network Setup

Both client and server initially start the same executable on their machines. When the main menu appears, one of the computers has to choose the *Host* option in order to become a server. Only after one of the computers starts hosting other clients can connect to the server. They can do this by entering the server's IP address in the designated field and click *Join*. The IP address used is the global IP of the server and can be found in the computer's network settings. The standard port for the multiuser platform is 7777. In order for the connection to work this port has to be accessible. The firewall also has to be configured to allow connections from the multiuser platform by opening an inbound and outbound port for the program's executable.

In order to implement a session-based rather than an IP-based system one has to replace all console commands with relevant session commands. Session commands such as *Host Session* or *Join Session* are native to event graphs. This could be used to implement a server list in the main menu rather than using IP addresses.

3.6.3 Error Handling

In order to sufficiently handle network errors, the multiuser platform uses a custom *GameInstance* which can be found in the *Blueprints* folder. It implements the method *OnNetworkError* which handles all kinds of connection issues by printing an informative message to the user. The method is automatically called by the program in case of a network error. The error type is passed as a parameter so that a suitable message can easily be shown.

The most common network errors are IP errors and loss of connection. An IP error occurs when the user attempts to connect to a server with the wrong IP address, meaning no valid server can be found under the IP the user entered. In this case the message *"Network error! Connection could not be established. No valid server found under this IP address"* is printed to the user's screen. The message disappears after 10 seconds and the user remains in the main menu.

A client can lose connection to the server if either the server unexpectedly shuts down or the client loses internet connection. In this case the message *"Network error! Connection was lost. Server shut down or internet connection lost"* is displayed on the user's screen for 10 seconds.

4 Extensibility

In order to enable future extensions and flexibility, this chapter describes possible ways of extending and modifying the multiuser platform for specific needs. It does not give suggestions for possible extensions but rather explains the means and prerequisites for adding new features. The descriptions are of general nature and do not describe steps to implement specific components. It is mainly intended for future developers. The most important section in this chapter is 4.2 since it explains how to replace the experiment structure which is crucial when conducting virtual experiments other than an Exit-Choice-Experiment. In terms of logic and features, the multiuser platform developed during this thesis is fully functional for virtual experiments. But in the future, more features might be needed for more elaborate experiments. Therefore, this chapter gives an overview of important methods and practices.

Section 4.1 describes the tools needed in order to modify the multiuser platform and explains how to set them up. More specific documentation on the usage of these tools is available on the respective tool's website. The chapter gives an overview of important components of the UE4 editor for future reference in other sections.

The section about adding new geometries also describes components which should always be placed in new setups as well as the folder hierarchy within the project and how to use starter content to create new structures.

Section 4.3 explains how to add new characters to the multiuser platform. This refers to the visual appearance of the participant's pawn. It describes how to import characters and touches on setting up existing animations for new characters. Another part of this section is how to integrate new characters into the random character chooser which determines a pawn's appearance.

The last section in this chapter is about how to extend the experiment logic rather than just its appearance. It describes best practices on adding new components and important functions which can be used to add logic.

4.1 Software Setup

The two tools used for developing the multiuser platform are *Unreal Engine 4* (UE4) and *Visual Studio 2015* (VS15) for Microsoft Windows. UE4 is used for visual editing and programming with event graphs while VS15 is used to program C++ functionality.

The multiuser platform is developed on Microsoft Windows because software setup is considerably easier than on Linux. It is also the most common operation system which many people use at home. Since the multiuser platform is intended to be used by participants from home it is sensible to compile the program for Windows which is easiest when developing on Windows as well.

The UE4 version used is 4.11.2. In order to expand the project, the same version should be used for development as it is not compatible with other versions. To upgrade the engine version, the newest UE4 version has to be downloaded. The developer can then create a copy of the project for the new version by attempting to open it in the new version. Some features might have to be refined in new engine versions since they might not work as developed.

UE4 is available for free on the Unreal Engine website[1] and includes the full visual editor. To program for UE4 in C++, VS15 is also needed. It is available for free under

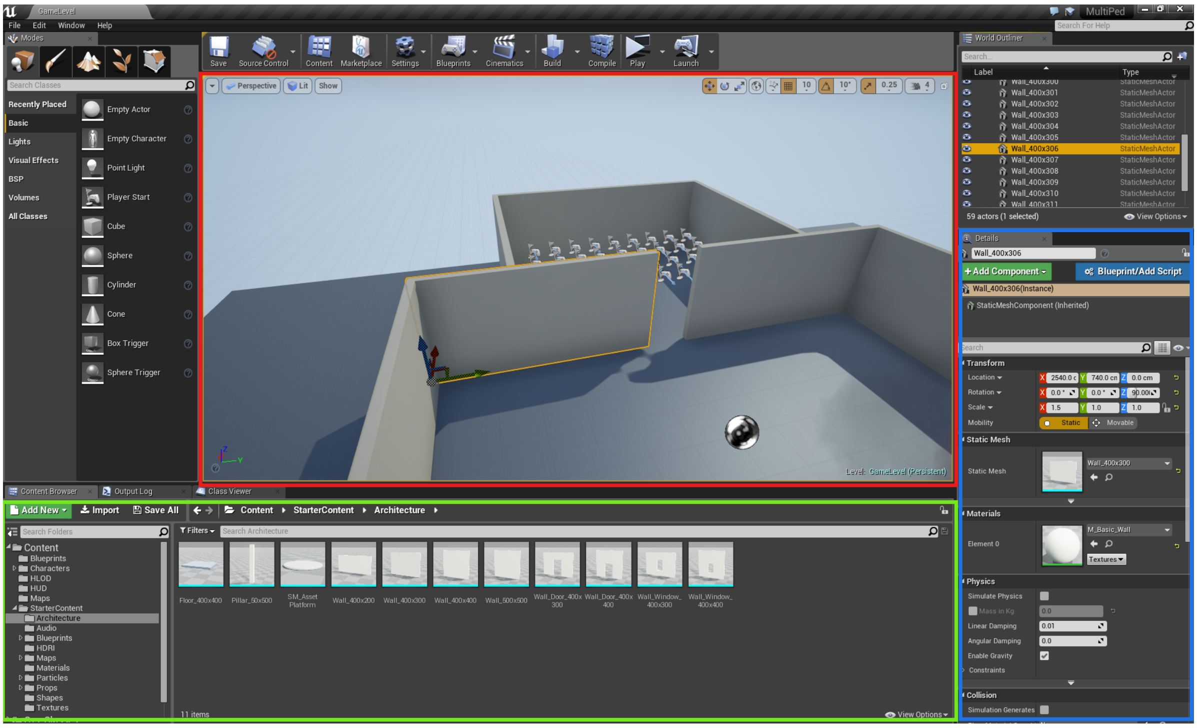


Figure 4.1: UE4 visual editor with viewport (red), content browser (green) and details panel (blue)

a community license after registering an account with Microsoft. This version of *Visual Studio* is required when working with UE4 4.11.2.

[1] http://www.unrealengine.com/ (retrieved August 16, 2016)

To install UE4, the Epic Games Launcher has to be downloaded as well and an Epic Games account has to be registered. During the installation, it can be chosen where to store UE4 project files. All assets and C++ classes associated with a project are stored in a subfolder of that location.

When installed, the two tools are linked and work together. The developer does not need to link the tools manually. This is done automatically if they are installed on the same computer. Creating a new C++ classnin UE4 automatically opens VS15. C++ code can be compiled from the UE4 editor. VS15 also has all UE4 libraries available which help with auto-completion on UE4 specific code.

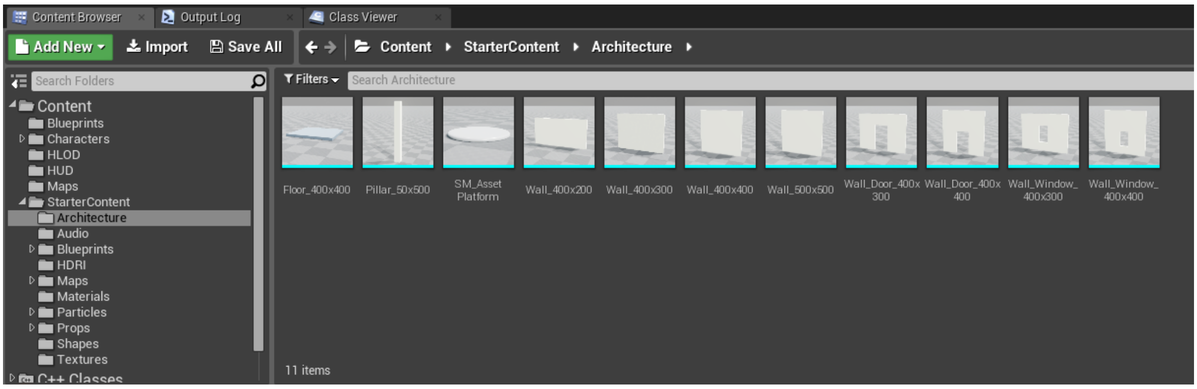


Figure 4.2: UE4 content browser showing architecture assets

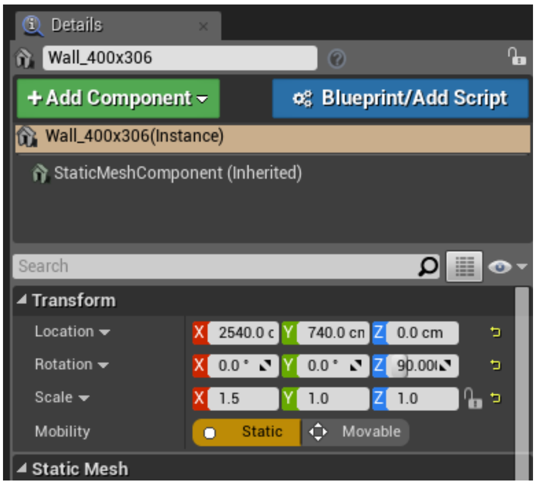


Figure 4.3: Part of the details panel in the UE4 editor

UE4 offers a *content browser* to overview all assets of the current project. This browser also displays any C++ classes added to the project.

The central part of UE4's editor is the *viewport*. It shows the current level and allows the developer to edit visually. Within the viewport, assets can be dragged to move, scale or rotate them. The viewport also directly mirrors any changes to attributes in the editor or in VS15. Assets can be dragged into the viewport from the content browser to place them.

When clicking on an asset in the viewport, UE4 opens a details panel on the right which contains important attributes of the current asset. As described in section 4.4 the developer can modify the panel by adding new C++ attributes declared as *UPROPERTY*.

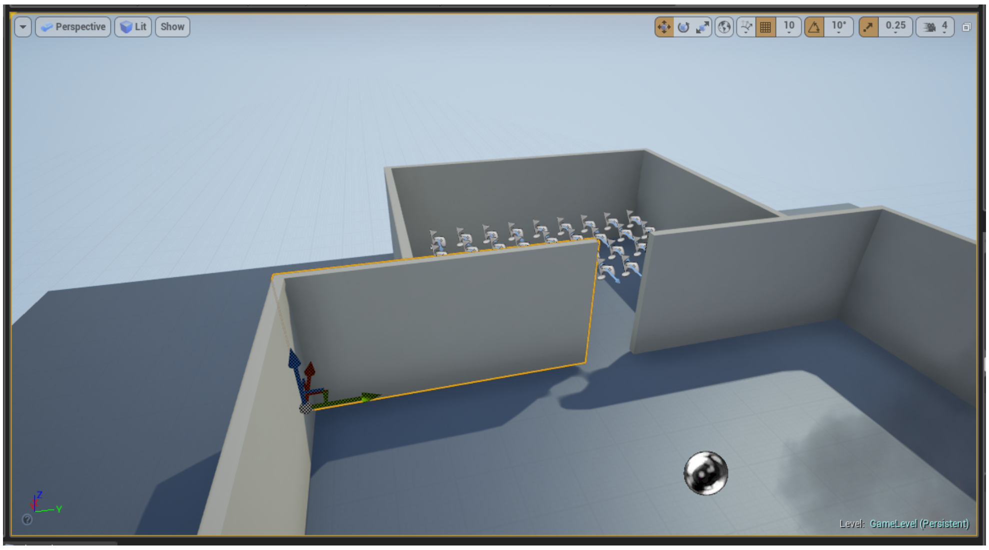


Figure 4.4: Viewport in the UE4 editor

The details panel can be used to scale, rotate or move an actor. This is often more precise than dragging in the viewport. It can also be used to add new components to an actor, e.g. a collision box. When double clicking on an asset in the content browser it can be edited and the details panel will show class defaults for this asset.

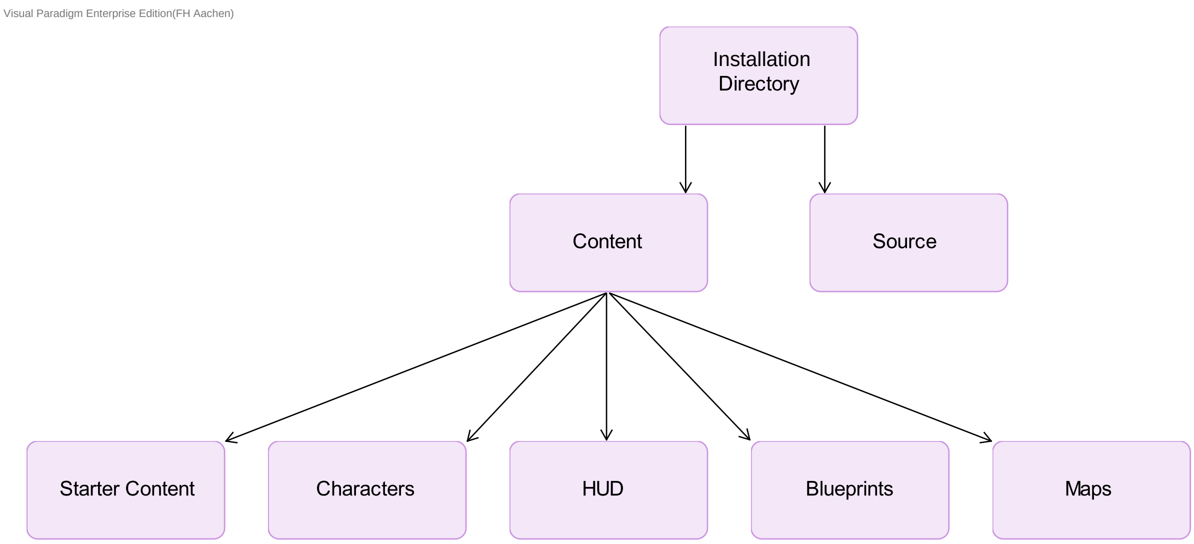


Figure 4.5: Simplified folder hierarchy of UE4 project

The project is distributed through the author's gitlab repository under

```
https://gitlab.version.fz-juelich.de/valder1/bachelorthesis.git
```

and can be cloned by anyone. Figure 4.5 shows important parts of the folder hierarchy within the project folder. After downloading the source code, the *.uproject* file in the main folder can be opened in UE4 which sets up the project for development. The git repository also contains an executable for Windows. Executables can be created by packaging the project in the UE4 editor.

4.2 Virtual Experiment Setup

In order to conduct virtual experiments in pedestrian dynamics, the experiment structure usually has to be changed while execution flow and functionality remain the same. This section explains in detail how to replace the example geometry with a new one and add all essential functionality.

The map that contains the experiment structure is called *GameLevel* and can be found in the *Maps* folder. This map must be edited to customize experiment geometries. The most important assets for doing so are walls which are located in *StarterContent/Architecture*. The walls can be dragged into the viewport from the content browser. They have defined heights and widths - as indicated by the file name - but can be scaled in each direction using the scale values in the editor. All sizes are defined in centimetres and scaling works with relative values. Assets can also be scaled by selecting the scale tool and dragging them to the desired size in the viewport. This method is suitable for creating easy setups. For more complex setups, geometries can be imported from 3D software like *3D Studio Max* or *Maya*. These structures should be imported into their own subfolder by choosing *Import* in the content browser. The recommended file extension for imports is *.fbx* (**F**ilm**b**o**x**) though UE4 also accepts *.obj* files. FBX is a file format by Autodesk designed specifically for digital content creation tools[2]. When exporting assets from other programs into the *.fbx* format some tools even offer a version of *FBX* optimized for Unreal Engine which should always be chosen, if possible, since it mitigates compatibility problems and graphical errors.

To make the experiment fully functional a few components have to be placed within the structure. The first one is a built-in component called *Player Start*. It can be found by searching the asset browser in the editor. *Player Starts* determine where pawns spawn when the experiment begins. There should be at least as many *Player Starts* as participants otherwise the program will try to adjust the spawn location which might lead to pawns spawning outside of the structure. Therefore, it is advisable to place more *Player Starts* than necessary to avoid this behavior.

[2]see http://www.autodesk.com/products/fbx/overview (retrieved August 16, 2016)

The most important component to place is *Finish Line* which can be found in the *Blueprints* folder labeled as *BP_FinishLine*. This component initiates the end of the experiment for a participant when touched as described in 3.4.1. Since it only triggers methods on a pawn and does not store variables itself it is possible to place multiple *Finish Lines* within an experiment structure, e.g. when having multiple exits.

The fire component is called *BP_ExperimentFire* and can be found in the *Blueprints* folder. It is simple and uses UE4's built-in fire particle system. The fire can be resized and moved but behaves in a pre-determined way. It will not react to walls or pawns. This behavior is slightly unrealistic but easy to use. It could be changed by implementing a custom particle system. This would be a considerable effort and is not done during this thesis but could be an option for future extension. In order to make the fire expand over time the developer can implement the *Tick* method either in the *AExperimentFire* C++ class or in the *BP_ExperimentFire* event graph. The *Tick* method can be implemented to manipulate the scale or location values in order to make the fire expand or move.

4.3 Character Meshes

The multiuser platform implemented in this thesis comes with two different skeletal meshes (= visual representations) for pawns, a male and a female. The characters in this project have been created with *Adobe Fuse CC* but any characters in *FBX* format can be used.

As with other assets, new skeletal meshes can be imported in the *.fbx* file format. Skeletal meshes are located in the *Characters* folder. Each skeletal mesh and its animations should be placed in its own folder to keep the assets easy to overview.

In order to use animations for the new character, an *animation blueprint* must be created. The animation blueprint defines which animations should be used at what time. A simple animation blueprint uses only three animations: idle, walking and running. This animation blueprint only has one variable, which is *speed*. Whenever an animation update is required, the speed is set by looking at the length of the current input vector of the character. This is implemented in the event graph of the animation blueprint. The speed variable is used as input for a one-dimensional *blend space*. The blend space smooths animations together and uses a parameter - in this case *speed* - to determine how much of each animation to use. The blend space is then used as the only state machine within the animation blueprint using the speed variable as its input. This causes the character to smoothly transition from idle to walking and then to running when gradually increasing the speed. Animations used for animation blueprints in UE4 should always be *in place* meaning the character is not translated during the animation because the movement is already handled by the character movement component.

The random character chooser is located in *BP_FPChar* in the *Blueprints* folder. It is part of the method *Begin Play* which is called at the start of the experiment. In order to add a new character to the random character chooser, it has to be added to the switch which evaluates the generated random number as shown in figure 4.6. The random chooser first generates a random integer between 0 and 99. This integer is then evaluated by checking if it is greater than 49. If so, the character's skeletal mesh is changed to the male version. Other distributions of characters can be achieved by adding more branches after the first one returns false and adjusting threshold numbers in the *Greater-Than* nodes.

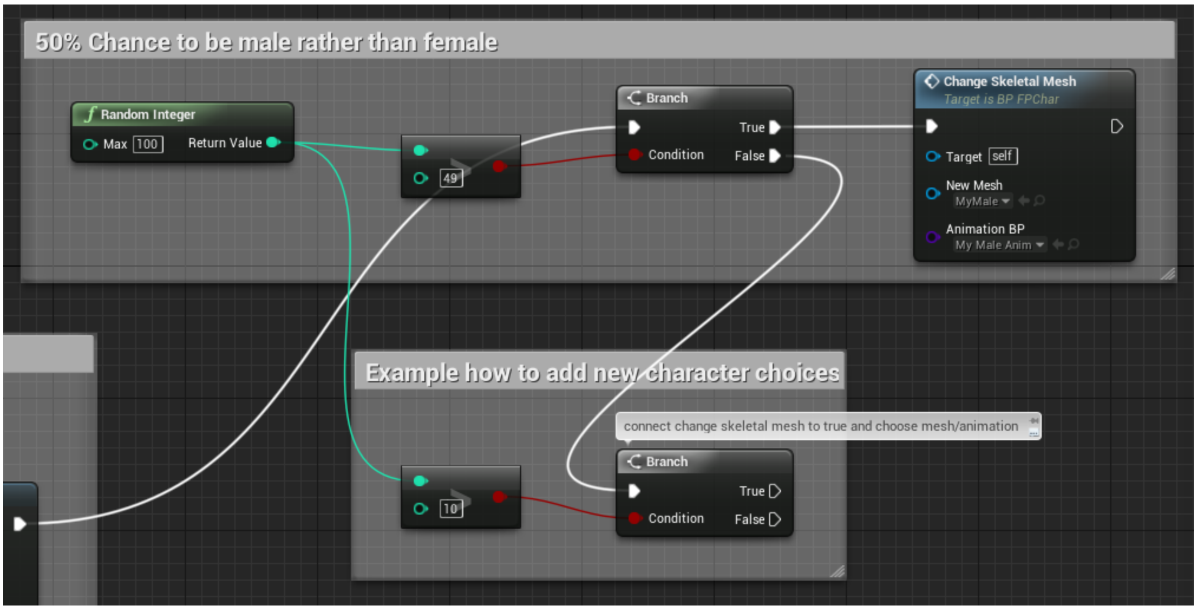


Figure 4.6: Random character chooser as event graph

4.4 Experiment Logic

In the future, developers can add experiment logic to the multiuser platform by creating new components or changing behavior of exisiting actors. A good practice for adding new actors is to first create a C++ class, then derive a blueprint class from it. Combining C++ programming and event graphs makes creating logic simple and effective. The derived blueprint class is also best used for creating the visual representation of the actor by editing it in the *viewport* tab. This can be done in C++ also but the *viewport* tab allows for easier editing while seeing direct results.

Functions and variables added to the C++ class can easily be exposed to event graphs by declaring variables as *UPROPERTY* and functions as *UFUNCTION*. Both declarations have various optional attributes. A detailed listing of all attributes can be found on the

Unreal Engine website. It is important to note that a *UPROPERTY* variable that has been set to *EditAnywhere* or *BlueprintReadWrite* will also show in the editor's side bar when clicking on an object of this class in the viewport. This means that the developer can actively change UE4's visual editor by programming a component in C++ which is an additional assistance.

Variables and functions exposed to blueprints can be used in event graphs. Two important attributes for *UFUNCTIONs* are *BlueprintPure* and *BlueprintCallable* (see listing 4.1 and figure 4.7). *BlueprintPure* functions do not have an execution node and are used to access variables and pass them as parameters to executable functions. *BlueprintCallable* functions have an execution node and are often more complex procedures which e.g. modify the actor, initiate travel or produce output. It is also notable that *BlueprintPure* functions always have return values while *BlueprintCallable* functions might not.

```cpp
UFUNCTION(BlueprintPure)
  bool IsDead();

UFUNCTION(Server, Reliable,
    WithValidation,
    BlueprintCallable)
  void UpdateHealth(float
      deltahealth);
```

Listing 4.1: BlueprintCallable and BlueprintPure functions in C++

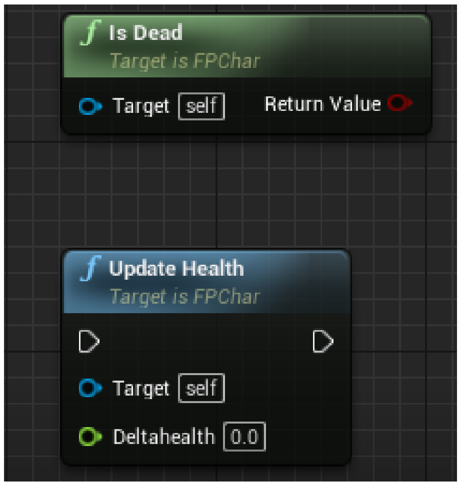


Figure 4.7: BlueprintPure (green) and BlueprintCallable (blue) functions in Event Graphs

UFUNCTION attributes can also be used to declare whether a function should run on the server or on the owning client. Functions which run on the server are those which modify gameplay variables that are important for all players such as health. Traveling to the game end screen on the other hand should be executed on the owning client because when one participant finishes the experiment, other participants should not travel to the game end screen as well.

When implementing functionality there are a few important methods that can be used. The method *BeginPlay* is automatically called when the actor starts existing in the game. It is not the same as the constructor since the actor might be constructed when starting the program but does not exist in the game world until later. There is no way of knowing when an object will be constructed so the constructor in C++ should only be used for variable default values. *BeginPlay* on the other hand is always called when the component starts to exist and be relevant to the game. For example, the *BeginPlay* method of the first person character is called when the actual experiment starts though it may have been constructed long before. *BeginPlay* is accessible in event graphs as well as in C++ programming.

To add recurring or automatic functionality the *Tick* method can be used which is called every frame. In the multiuser platform this method is used for saving trajectory points as well as other recurring functionality. Using the *Tick* method can be resource intensive and it should only be used for simple procedures. In case of complex procedures the developer should consider setting up a timer in the *BeginPlay* method instead of using *Tick*.

Some components of the multiuser platform have functionality that is triggered by collision, such as the *Finish Line*. To implement collision functionality the actor needs a collision box component. The methods *OnComponentBeginOverlap* and *OnComponentEndOverlap* can be implemented to make the actor react to collision. These methods are called when collision with another actor begins or ends. The other actor is passed as a parameter and can be used to implement logic. Another example of this in the multiuser platform is the experiment fire which starts decreasing a pawn's health on collision and stops doing so when the collision ends.

5 Test and Analysis

This chapter describes tests which were conducted after completing development of the prototype application. Test cases are given for normal program execution, special cases, which have to be handled, and expected errors. The sections below present the different test cases, which contain user input, the program's reactions and possible output. In error cases it is described how the multiuser platform handles the error and presents it to the user.

Section 5.2 recaps performance guidelines stated in 2.4 and analyzes how the multiuser platform meets performance goals.

5.1 Test Cases

Every test case below consists of a description which functionality is being tested and what kind of reaction from the program is expected. It is then stated what kind of input was used and what kind of output or behavior was observed.

Test cases follow the *White-Box-Test* technique which means test cases were designed with consideration of the multiuser platform's inner structure. The goal of these tests is to ensure the functionality of all features which were implemented during this thesis. Errors and special cases should be handled appropriately to avoid unexpected behavior. Test cases result in success when the program behaves as expected.

5.1.1 Normal Cases

Normal cases are those cases which test the intended program flow. In a normal case, the software reacts as expected and the test does not result in any errors. For the multiuser platform, these normal cases describe the intended procedure of a virtual experiment. First, a server is hosted, then clients join and the experiment starts. Characters are able to move around freely. The time display HUD works as intended and is paused when using the pause menu. When passing the finish line, the participant is transported to the game end screen but can return to the level as a spectator. A trajectory file is created on the server whenever a participant passes the finish line.

The following normal cases also refer to the feature requirement list in 2.2. If a test case focuses on a certain feature it is noted whether this feature works as intended in the requirements.

Hosting a server

Tested functionality: Hosting a new server by clicking the *Host* button in the main menu.

Expected behavior: The user is transported to the lobby as a listen server and can see how many participants are connected as well as choose to start the experiment.

User input: Starting the software and clicking the *Host* button in the main menu.

Program reaction/output: The lobby is loaded and the user can see that 0 participants are connected. The button to start the experiment is shown. During this test, output logging was activated. The log confirmed that a new server was started which is listening on port 7777, as expected.

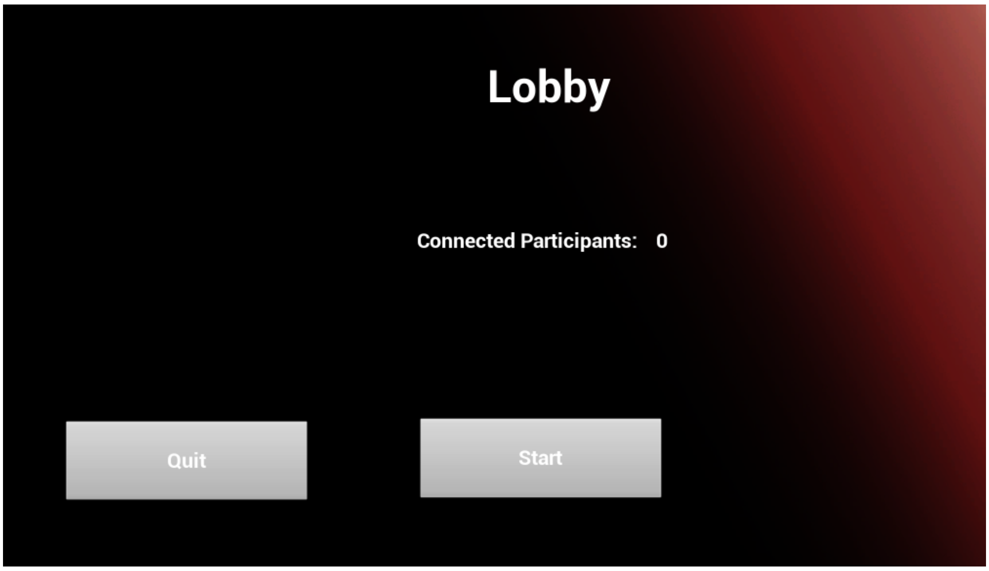


Figure 5.1: Successful server creation

Figure 5.2: Output log after clicking the *Host* button

Test result: Success. This feature works as intended by the requirements for the main menu in 2.2.1.

Joining a server

Tested functionality: Joining an existing server by entering the server's IP address and clicking the *Join* button in the main menu.

Expected behavior: The user is transported to the lobby as a participant. The user sees the *Quit* button but not the *Start* button or how many participants are connected. The server sees that 1 participant is connected.

User input: Starting the software and entering an exisiting server's IP address into the textbox, then clicking the *Join*.

Program reaction/output: The lobby is loaded and the user can see the *Quit* button but not the server's UI. On the server, 1 participant is listed as connected.

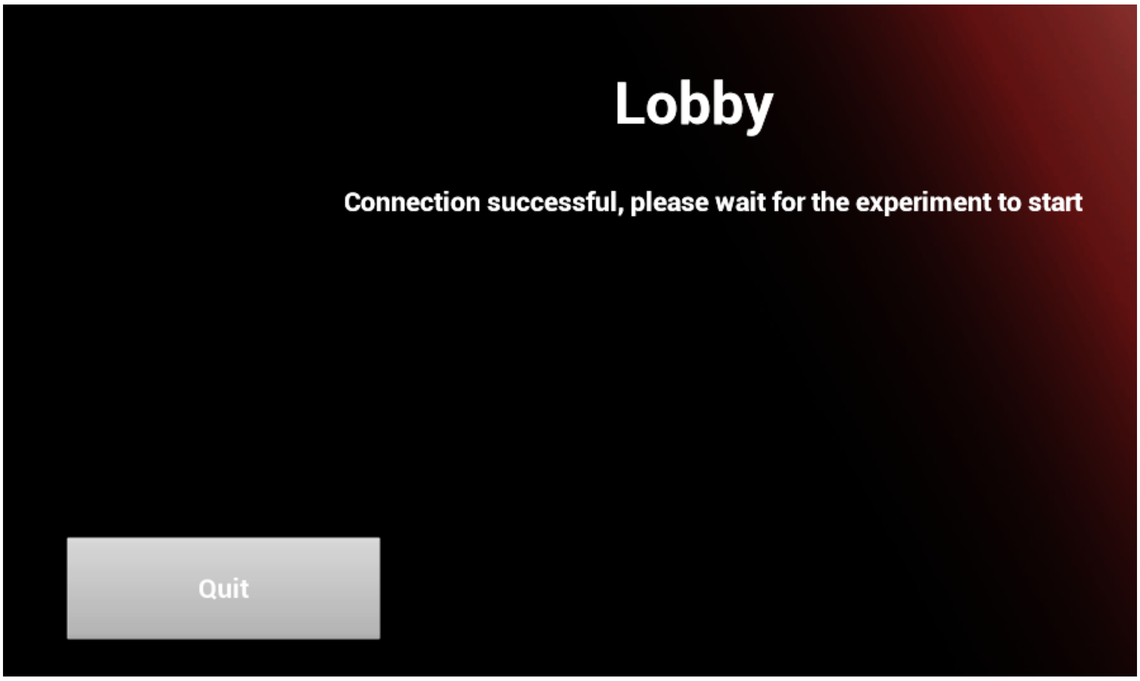


Figure 5.3: Successful join on client side

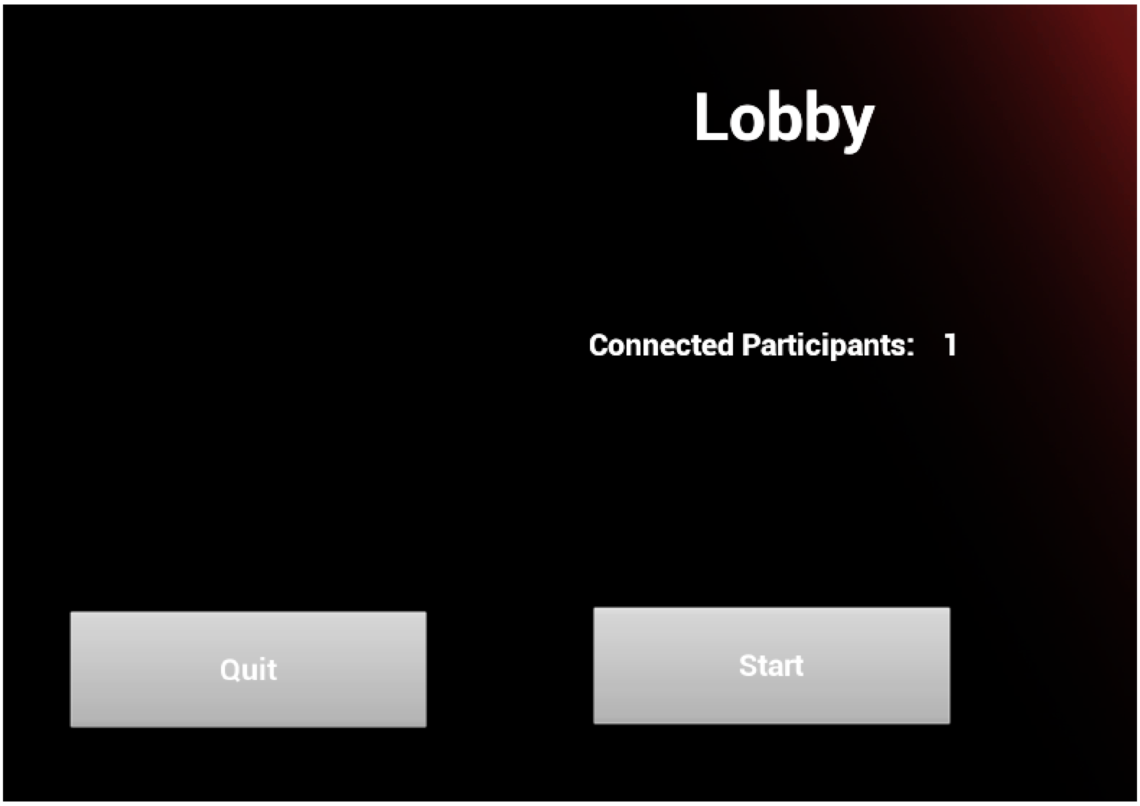


Figure 5.4: Successful join on server side

Test result: Success. This feature works as intended by the requirements for the main menu in 2.2.1.

Starting the experiment

Tested functionality: The server starting the experiment and the random character chooser distributing random meshes to all characters.

Expected behavior: All clients are transported to the game level and are visually represented by a charcter which is randomly male or female. All clients spawn at a player start in the waiting area of the experiment structure. Clients do not spawn on top of each other, outside of the structure or inside walls. The server spawns as a spectator without a visual representation.

User input: The server clicking the *Start* button in the lobby after 15 clients have joined.

Program reaction/output: The game level is loaded and the server spawns as a spectator. All clients spawn in the waiting area of the experiment structure. Every client is represented by a character either male or female. Of 15 connected clients, 8 are represented by a female character and 7 by a male which is approximately the desired distribution of 50:50.

Figure 5.5: 15 clients connected

Figure 5.6: Experiment started successfully

Test result: Success. Traveling to the game level from the lobby works as intended by the requirements in 2.2.1.

Character Movement

Tested functionality: Characters walking and running through the experiment structure while being suitably animated.

Expected behavior: After loading the game level, characters are able to move freely through the experiment structure. They are able to move forwards, backwards or sideways but not upwards or downwards without sufficient structures to walk on. Characters are able to turn in every direction including upwards and downwards. Characters run faster if the user presses *Shift* on the keyboard. Correct animations apply depending on the character's speed.

User input: After connecting to the server and being transported to the game level, using the *W/S/A/D* keys to move around and the mouse to turn in different directions. Pressing the *Shift* key while moving to run faster. Walking against walls and other participants to test the collision system.

Program reaction/output: Characters are able to move around freely and can turn in every direction by using the mouse. Idle and walking animations work correctly. Characters are unable to pass through walls or other characters. When pressing *Shift* and moving, the character runs faster and suitable running animations are used.

Figure 5.7: Moving character, gravity applies

Figure 5.8: Character being unable to walk through a wall

Figure 5.9: Character running while pressing *Shift*

Test result: Success. Movement works as intended by the requirements for character movement in 2.2.3.

Time Display and Pause Menu

Tested functionality: The time passed since the start of the experiment being displayed on screen. Time display being paused when using the pause menu. Using the pause menu to pause and resume the experiment.

Expected behavior: When starting the experiment, every participant can see the time passed since the start on their screen. When pressing *"M"* a pause menu is opened and the time display is paused. When clicking *Resume* in the pause menu the menu is closed and the time display starts updating again. The user can still see other characters move while their own game is paused.

User input: Pressing *"M"* after starting the experiment. Pressing *Resume* in the pause menu.

Program reaction/output: After the experiment starts, a timer starts ticking in the bottom left corner of the screen. It shows the time passed since the start of the experiment. After pressing *"M"* a pause menu is opened and the timer stops ticking. Other actors and characters can still be seen as moving. When clicking the *Resume* button the timer starts ticking again and the pause menu disappears.

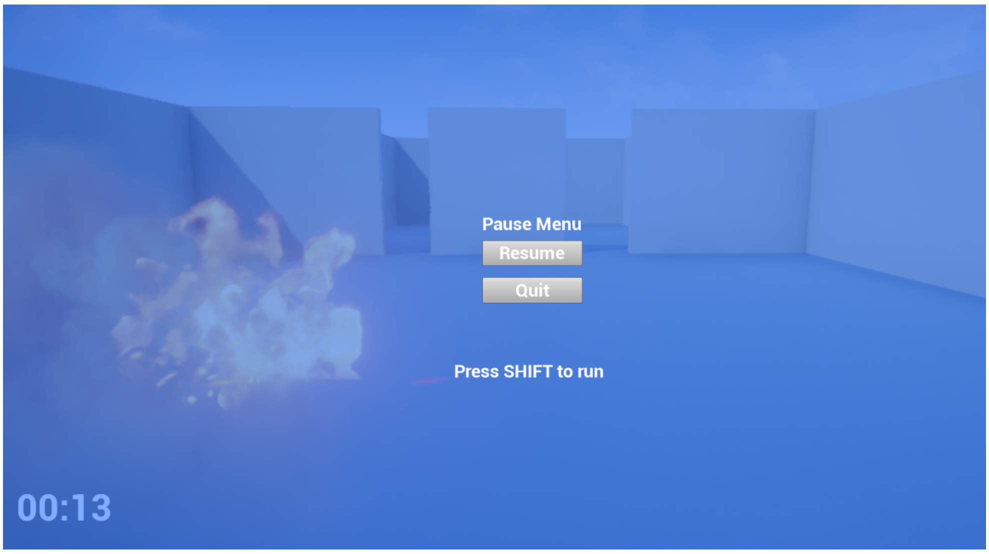


Figure 5.10: Pause menu and time display

Test result: Success. The time display feature works as intended by the requirements for time measurement in 2.2.4.

Ending the Experiment

Tested functionality: Ending the experiment by passing the finish line.

Expected behavior: When passing the finish line, the time display in the bottom left corner stops ticking. After 3 seconds, the participant is transported to the game end screen, their visual representation in the experiment level is destroyed. On the game end screen, the user sees an option to return to the game as a spectator or quit the program.

User input: Walking past the finish line with a character.

Program reaction/output: The time display stops ticking as soon as the character passes the finish line. After a while, the game end screen appears and shows a button *Continue as Spectator* and a *Quit* button. The server who is still spectating the experiment level confirms that the visual representation of the participant has been destroyed.

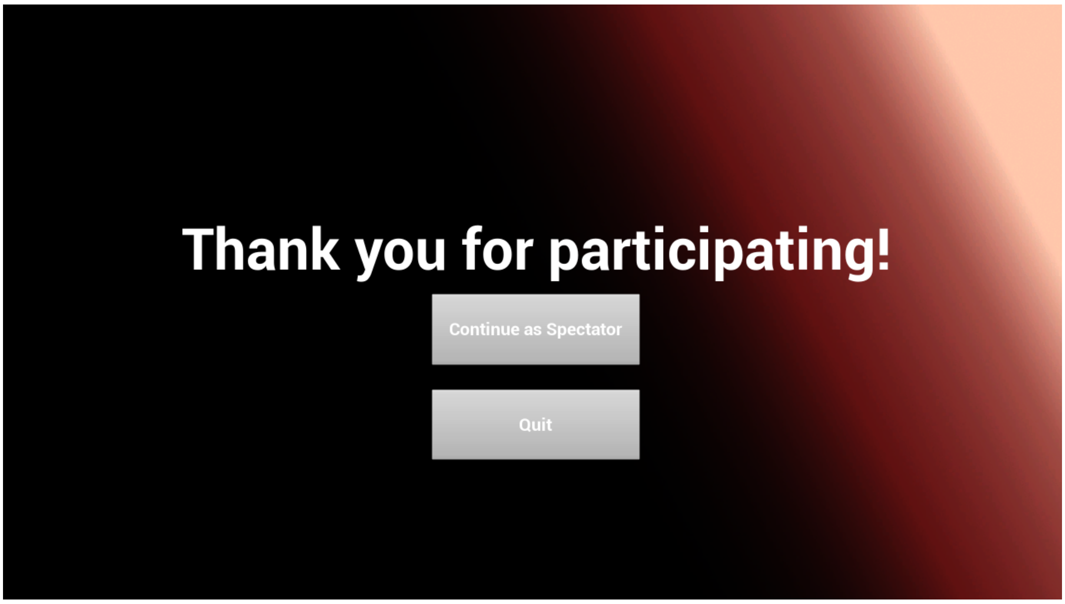


Figure 5.11: Game end screen

Test result: Success. This feature works in accordance with the requirements in 2.2.3.

Spectator Mode

Tested functionality: Returning to the experiment level as a spectator after completing the experiment.

Expected behavior: When clicking the *Continue as Spectator* button the experiment level is loaded again. The user can then fly around as an invisible spectator without gravity being applied. He can watch other clients who are still in the experiment.

User input: Pressing the *Continue as Spectator* button in the game end screen and using the *W/S/A/D* keys and the mouse to move around the experiment level as a spectator.

Program reaction/output: The experiment level is loaded and the user moves around the experiment level as a spectator. He can watch but is invisible to clients still in the experiment. The user can move without gravity being applied.

Figure 5.12: Spectating client after experiment completion

Test result: Success. This feature is working as intended by the requirements in 2.2.7.

Output File Generation

Tested functionality: Trajectory output file being generated after a character passing the finish line.

Expected behavior: After a character passes the finish line, the character's trajectory is saved to a file on the server. The file format is the same as for real life experiments. The trajectory file is named using the player's unique ID.

User input: Passing the finish line with a character in the experiment level.

Program reaction/output: Whenever a character passes the finish line, a trajectory file is generated on the server in the program's installation directory. The files contain the participant's trajectories in the file format described in 2.3. The files are distinct and named after the participants' unique IDs.

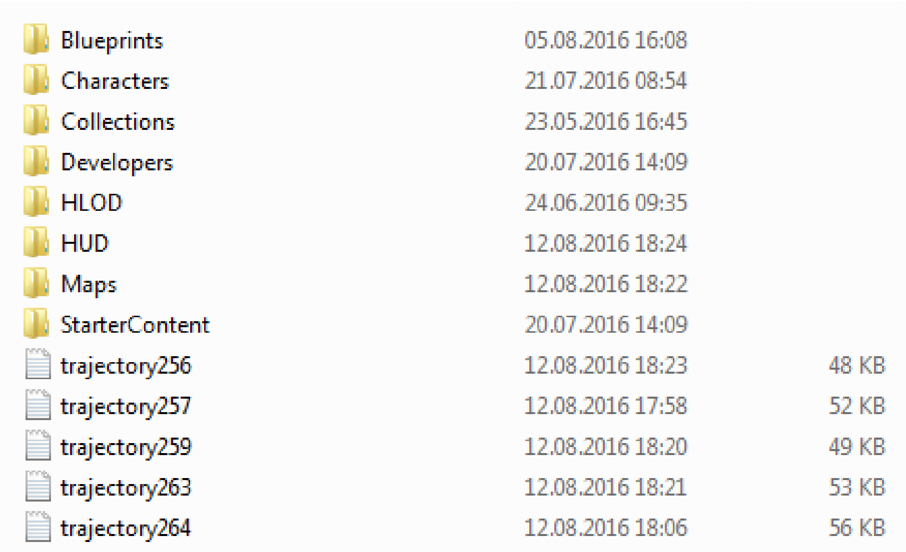
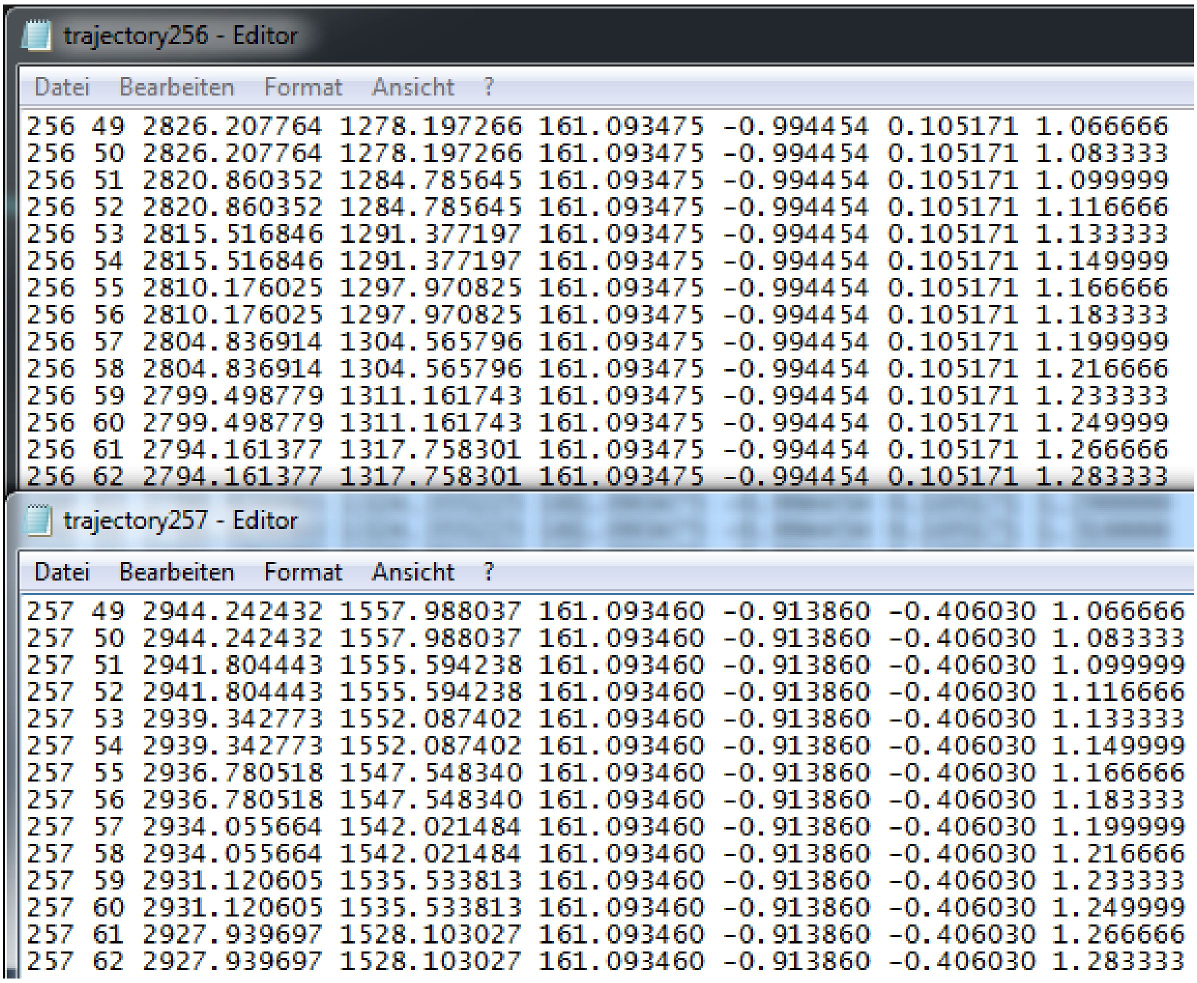


Figure 5.13: Successful output file generation in installation directory



trajectory256 - Editor

Datei Bearbeiten Format Ansicht ?

```
256 49 2826.207764 1278.197266 161.093475 -0.994454 0.105171 1.066666
256 50 2826.207764 1278.197266 161.093475 -0.994454 0.105171 1.083333
256 51 2820.860352 1284.785645 161.093475 -0.994454 0.105171 1.099999
256 52 2820.860352 1284.785645 161.093475 -0.994454 0.105171 1.116666
256 53 2815.516846 1291.377197 161.093475 -0.994454 0.105171 1.133333
256 54 2815.516846 1291.377197 161.093475 -0.994454 0.105171 1.149999
256 55 2810.176025 1297.970825 161.093475 -0.994454 0.105171 1.166666
256 56 2810.176025 1297.970825 161.093475 -0.994454 0.105171 1.183333
256 57 2804.836914 1304.565796 161.093475 -0.994454 0.105171 1.199999
256 58 2804.836914 1304.565796 161.093475 -0.994454 0.105171 1.216666
256 59 2799.498779 1311.161743 161.093475 -0.994454 0.105171 1.233333
256 60 2799.498779 1311.161743 161.093475 -0.994454 0.105171 1.249999
256 61 2794.161377 1317.758301 161.093475 -0.994454 0.105171 1.266666
256 62 2794.161377 1317.758301 161.093475 -0.994454 0.105171 1.283333
```

trajectory257 - Editor

Datei Bearbeiten Format Ansicht ?

```
257 49 2944.242432 1557.988037 161.093460 -0.913860 -0.406030 1.066666
257 50 2944.242432 1557.988037 161.093460 -0.913860 -0.406030 1.083333
257 51 2941.804443 1555.594238 161.093460 -0.913860 -0.406030 1.099999
257 52 2941.804443 1555.594238 161.093460 -0.913860 -0.406030 1.116666
257 53 2939.342773 1552.087402 161.093460 -0.913860 -0.406030 1.133333
257 54 2939.342773 1552.087402 161.093460 -0.913860 -0.406030 1.149999
257 55 2936.780518 1547.548340 161.093460 -0.913860 -0.406030 1.166666
257 56 2936.780518 1547.548340 161.093460 -0.913860 -0.406030 1.183333
257 57 2934.055664 1542.021484 161.093460 -0.913860 -0.406030 1.199999
257 58 2934.055664 1542.021484 161.093460 -0.913860 -0.406030 1.216666
257 59 2931.120605 1535.533813 161.093460 -0.913860 -0.406030 1.233333
257 60 2931.120605 1535.533813 161.093460 -0.913860 -0.406030 1.249999
257 61 2927.939697 1528.103027 161.093460 -0.913860 -0.406030 1.266666
257 62 2927.939697 1528.103027 161.093460 -0.913860 -0.406030 1.283333
```

Figure 5.14: Two distinct trajectories

Test result: Success. This feature works as intended by the requirements for output files in 2.2.6. It can be inferred that trajectory tracking works as intended by the requirements in 2.2.5.

5.1.2 Special Cases

Special cases test scenarios which are not intended to appear during a regular program run but are still within limits of the program's functionality. A special case may or may not result in errors. Usually the developer defines whether a special test case should produce an error or not.

Concerning the multiuser platform two special cases can be identified: A character being reduced to 0 health and thus dying, and a client quitting the program during the experiment through the pause menu. Both cases do not produce errors but are part of the multiuser platform's functionality.

Character's health reduced to 0

Test scenario: A character is reduced to 0 health by a damage source, in this case a fire. The experiment subsequently ends for the controlling participant.

Expected behavior: When being reduced to 0 health, the participant is transported to the game end screen. His visual representation in the scene is destroyed and his trajectory is not saved in an output file. The participant then has the option to return to the experiment as a spectator.

User input: The participant walks into the fire's damage radius and stays there until his health is reduced to 0.

Program reaction/output: About 10 seconds after walking into the fire, the participant travels to the game end screen. The server spectating the experiment confirms that the visual representation of the participant has been destroyed. No output file is created on the server for this participant. On the game end screen, the participant is shown the options to either return to the experiment as a spectator or quit the program.

Test result: Success. This feature works as intended by the requirements in 2.2.8.

Client Quits

Test scenario: A client quits the program through the pause menu during the experiment.

Expected behavior: When a client quits through the pause menu it is transported back to the main menu. The character's visual representation in the experiment is destroyed. The server is not affected by the client quitting. All other participants continue the experiment normally.

User input: Entering the pause menu by pressing *"M"* during the experiment level, then pressing the *Quit* button.

Program reaction/output: The participant is transported back to the main menu. The server output log confirms that a client has disconnected from the server. The character's visual representation in the experiment is destroyed. The server runs normally and is not further affected by the client quitting, and neither are the other participants.

Figure 5.15: Server output log after a client quits

Test result: Success

5.1.3 Error Cases

Error cases are cases which result in the display of an error message and interrupt the intended program flow. They are used to test whether the program correctly displays error messages instead of shutting down or showing unexpected behavior when encountering an error.

In the multiuser platform errors can occur when network connections cannot be established, are lost, or when the output fails. All of these cases are handled by the multiuser platform and informative error messages are displayed.

Invalid IP Address

Test scenario: A user enters an IP address which contains invalid characters and clicks *Join* in the main menu.

Expected behavior: An error message is displayed at the top left corner of the screen informing the user that the entered IP is invalid. The user does not leave the main menu.

User input: The user enters the string *"abc"* into the textbox for the IP address and clicks *Join*.

Program reaction/output: The syntax check for the IP address detects invalid characters. The error message *"abc is not a valid IP"* is displayed in the top left corner. The user does not leave the main menu.

Figure 5.16: Error message after entering an invalid IP address

Test result: Success

Wrong IP Address

Test scenario: The user tries to connect to an IP address which does not belong to a currently open experiment server.

Expected behavior: The user does not leave the main menu, instead an error message is displayed in the top left corner informing the user that no valid server was found under the entered IP address.

User input: The user enters a wrong IP address into the textbox for the IP address and clicks *Join* in the main menu.

Program reaction/output: The error message *"Network error! Connection could not be established! No valid server found under this IP address"* is displayed in the top left corner for 10 seconds. The user does not leave the main menu.

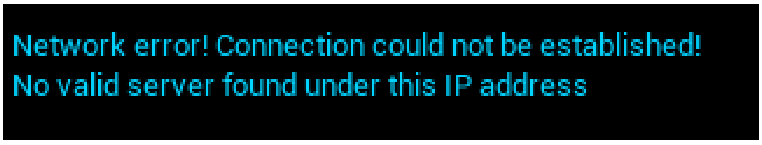


Figure 5.17: Error message after entering a wrong IP address

Test result: Success

Server Shut Down

Test scenario: The server unexpectedly shuts down while clients are connected.

Expected behavior: If the server unexpectedly shuts down, all clients that were connected are disconnected and transported to the main menu. An error message is displayed in the top left corner informing the user that the connection was lost.

User input: The server computer shuts down.

Program reaction/output: After the shut down, all clients are disconnected and transported to the main menu. The message *"Network error! Connection was lost! Server shut down or internet connection failure"* is displayed for 10 seconds.

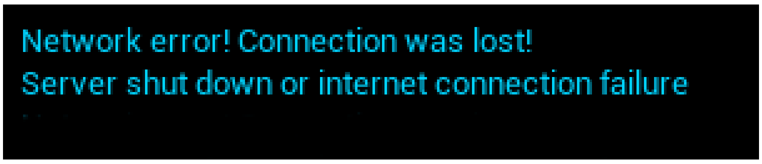


Figure 5.18: Error message in client after server shut down

Test result: Success

Output File Error

Test scenario: The server computer encounters an error while trying to create a trajectory output file.

Expected behavior: If the server encounters an error while trying to create or write into a trajectory output file, an error message is displayed on screen informing experiment supervisors about the error. Participants still get transported to the game end screen after passing the finish line. The program does not shut down.

User input: A participant passes the finish line. The filepath for output files is marked as read-only.

Program reaction/output: The program displays an error message on the server's screen which contains details about the error taken from the C++ exception description.

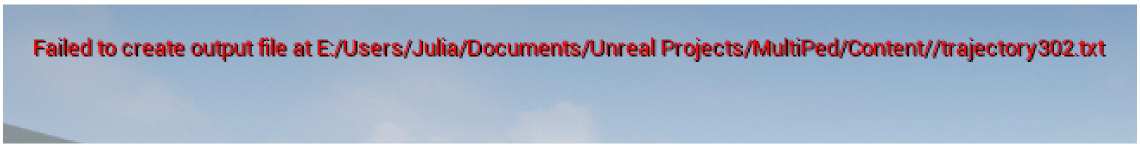


Figure 5.19: Error message after output file creation error

Test result: Success

5.2 Performance Analysis

The requirements for performance in 2.4 state that the multiuser platform should be able to handle at least 30 clients and have a frame rate of 60 FPS.

A test conducted during this thesis used a server and 30 connected clients, of which 15 each were running on the same computer. Since many clients were handled by the same computer the frame rate and graphical quality was low for this test. The goal was to ensure that clients stayed connected and that they did not experience any frame rate drops. From tests with three clients it can be inferred that graphical quality and frame rate greatly improve when one computer only handles one client.

During the test, clients all stayed connected to the server and did not experience any frame rate drops. Therefore, this test can be seen as a successful example of the multiuser platform handling up to 30 clients. Other multiplayer games made with UE4 such as Unreal Tournament prove that UE4 made games exist which can handle at least 32 players[1] while maintaining quality and frame rate. In the future, the multiuser platform could be optimized towards handling more clients.

The frame rate is highly dependent on the used hardware. During this thesis, the multiuser platform was tested with three different computers. The main development computer features a *GeForce GTX 730* GPU. When one host and one client run on the main development computer the client displays an avarage frame rate of 69 FPS. When connected to the main computer over a network, a client with a *GeForce GT 730* works with an avarage frame rate of 61 FPS. Both scenarios fulfill the performance goal of 60 FPS. The frame rate on a client is highly dependent on its hardware since UE4 requires sufficient GPU power. For example, the multiuser platform was tested with a laptop with Intel on-board graphics. This client showed grevious graphical problems and an avarage frame rate below 30 FPS. Therefore, it is recommended to use at least a *GeForce GT 730* or similar GPU when running the multiuser platform as a client. However, the frame rate of other clients is not influenced by one slow client. It is more important that the server maintains a constant frame rate of at least 60 FPS to not slow down clients.

The smoothness of the program is also dependent on internet connection speed. If it takes too long for a server to send data updates to the clients, delays and abruptly moving characters can be observed on clients. Next to a sufficient GPU the server also needs a sufficient CPU to handle client requests and calculate movement or other data. In this test, the server used an *Intel i7-3770* CPU with 3,4 GHz and 16 GB of RAM. The server was able to handle client requests without clients experiencing significant delays. The server's performance is not dependent on client connectivity as proven in test 5.1.2.

[1]see http://www.tldp.org/HOWTO/archived/Game-Server-HOWTO/ut.html

6 Summary and Outlook

6.1 Summary

This thesis describes the development of a multiuser platform for virtual experiments in pedestrian dynamics. These experiments are used to gather data about the behavior of pedestrian crowds in danger situations as well as in everyday life scenarios. Eventually, the goal is improving security at large scale events and optimizing pedestrian facilities. The multiuser platform developed during this thesis provides a functional basis to conduct virtual experiments. It is a client-server-application which is distributed free and open source. The similarity of virtual experiments to video games justifies the use of the game engine *Unreal Engine 4* to develop the program. In a previous analysis UE4 proved to be a viable option for creating virtual experiments since it provides convincing functionality, such as easy visual level editing and multiuser support.

Chapter 1 of this thesis provides an overview of pedestrian dynamics and virtual experiments. It explains what pedestrian dynamics are used for and how virtual experiments can mitigate problems of real life experiments. The term *serious game* is defined and it is assessed whether the multiuser platform can be seen as one. The thesis is distinguished from similar works and it is shown how the multiuser platform advances previous software for virtual experiments.

The second chapter analyses what features are required to make the multiuser platform functional and which are optional for future development. The requirements were discussed and determined with experts in pedestrian dynamics. The chapter also gives guidelines for graphical quality and performance of the multiuser platform.

Chapter 3 describes the finished product developed during this thesis in detail. It explains how the program is structured and how classes work together. It also visualizes the program's work flow and states how it is intended to be used.

The focus of chapter 4 lies on developing new features for the multiuser platform in the future. It describes important interfaces and explains best practices for adding new features, replacing the experiment geometry or importing new characters.

The thesis is concluded by tests conducted in chapter 5. Test cases cover all feature requirements of chapter 2 and also test how the multiuser platform handles errors. It is analyzed whether the multiuser platform meets set performance goals.

When starting the program, the user can choose to host an experiment or join an existing one. During the experiment, participants control a character which they can move through the experiment. Characters are animated according to their speed. The example implemented in this thesis is an exit choice experiment which means that participants enter a room through an opening on one side and have to choose between two exits on the other sides. The exits are marked with a finish line component so that the program recognizes when a participant is finished.

A fire component can be placed in the experiment structure which damages characters that walk through it. This can lead to the experiment ending prematurely for the controlling participant. The experiment ends regularly when a character walks through an exit. The character's trajectory, which is recorded throughout the experiment, is then saved to an output file on the server. After passing the finish line, the experiment ends for the participant but he can return to the experiment as a spectator.

The multiuser platform is suitable for a variety of use cases, including conducting virtual pedestrian experiments to plan buildings, testing evacuation routes, testing behavior in danger situations and more. The program is highly flexible and can be easily customized by using the free *Unreal Engine 4* editor. It can be adapted for different virtual experiments and other use cases by replacing the experiment geometry or programming new features using the existing interfaces.

The multiuser platform also mitigates a number of problems that can be observed in real life experiments. For example it simplifies tracking by automatically saving trajectories in the character class. Using the multiuser platform to conduct experiments also reduces costs and effort since participants can connect to the experiment from any computer and do not have to be invited to a specific venue.

During development of the multiuser platform, UE4's visual editor made constructing the experiment geometry easy. Components can be easily placed using the UE4 viewport. The details panel and viewport tools simplified scaling and positioning compared to setting scale and location values in code. The multiuser functionality was considerably aided by UE4 native methods. Movement is automatically replicated over the network and few changes had to be made compared to a single user approach. However, synchronizing trajectories over the network posed a significant problem since it cannot be guaranteed that all clients run with the same frame rate. This problem was solved by saving a time stamp with all trajectory points. The time used for the time stamp is a provided by the server, thus trajectories can be synchronized across all clients.

6.2 Outlook

The multiuser platform implements all key features that are required to make it functional. However, more features can be added in the future to adapt the multiuser platform for specific scenarios or new types of virtual experiments.

The requirements propose a server list as a feature of the main menu. A server list would replace the current system of typing in the server's IP address to connect. Instead, a list would display all currently available servers of the multiuser platform. The participant could then select a server from the list and connect to it.

Another prospect for development is improving the fire component. It currently uses UE4's built-in particle system for fire. This works well in terms of graphical quality but lacks in realism as the fire does not react to walls or pedestrians. It behaves in a pre-determined way and does not expand over time. This can be changed in the future by either adding the expand functionality to the existing component or creating an entirely new particle system that supports interactive behavior and reacts to structures and people. Another idea is to use fire simulation data as input. Apart from expanding the fire's functionality, developers can also add more danger scenarios and sources of damage, e.g. earthquakes or collapsing buildings. Damage causing components can utilize the interface of the *FPChar* class to cause damage to characters.

In order to conduct large scale experiments with hundreds of participants the multiuser platform can be optimized in terms of bandwidth usage. Since UE4 does not cap the number of clients able to connect to a server, the multiuser platform can be used on servers with more powerful hardware to enable experiments with possibly hundreds of participants. When the validity of virtual experiments is proven, the multiuser platform could be used as an effective aid to real life experiments. By partially replacing and optimizing real life experiments, virtual experiments reduce costs and effort for pedestrian experiments considerably.

In the future, the multiuser platform should be established as a viable tool for conducting virtual experiments in pedestrian dynamics. Since it is free and open source, the platform can be offered to different experts in the field to test their own experiments. JSC can provide extensive support and help with the setup of these experiments. For example, the Wuppertal experiments with about 30 participants could be run again using the multiuser platform since having all pedestrians controlled by real people significantly increases the comparability to real life experiments. In this context, the Oculus Rift can be used to create more immersion and make the participants behave more naturally.

To make trajectories usable for evaluation tools such as *JuPedSim*, post-processing of trajectories could be implemented to make trajectories recorded at different frame rates equidistant. This can be done easily by interpolating trajectory data points using the saved time stamps.

If the validity of virtual experiments cannot be proven, the multiuser platform can be used to visualize and replay real life experiments by adding features for automatic movement of characters according to existing trajectories. Trajectories could also be visualized in a structure after a virtual experiment or from existing data.

In summary, the multiuser platform developed in this thesis can be easily customized and provides multiple options for future use cases and further development.

References

[1] C. Abt. *Serious Games*. The Viking Press, 1970.

[2] M. Boltes. *Automatische Erfassung präziser Trajektorien in Personenströmen hoher Dichte*. Forschungszentrum Jülich, 2015.

[3] M. Drouhard, C. A. Steed, S. Hahn, T. Proffen, J. Daniel, and M. Matheson. *Immersive Visualisation for Materials Science Data Analysis using the Oculus Rift*. In *2nd Workshop on Advances in Software and Hardware for Big Data to Knowledge Discovery 2015 at IEEE Big Data 2015*, 2015.

[4] M. Kimura and J. Sime. *Exit Choice Behaviour During The Evacuation Of Two Lecture Theatres*. In *Fire Safety Science Symposium 2*, 1989.

[5] A. Kirchner. *Simulation of evacuation processes using a bionics-inspired cellular automation model for pedestrian dynamics*. Universität zu Köln, 2002.

[6] D. Michael and S. Chen. *Serious Games: Games That Educate, Train, and Inform*. Thomson Course Technology, 2005.

[7] P. Read. *Restoration of motion picture film*. Butterworth-Heinemann, 2000.

[8] J. Ribeiro, J. Almeida, R. Rossetti, A. Coelho, and A. Coelho. *Using serious games to train evacuation behaviour*. In *7th Iberian Conference on Information Systems and Technologies (CISTI)*, 2012.

[9] B. Sawyer and D. Rejeski. *Serious games: Improving public policy through game-based learning and simulation*. Woodrow Wilson International Center for Scholars, 2002.

[10] G. Still. *Crowd Dynamics*. University of Warwick, 1989.

[11] J. Valder. *Comparison between Vizard VR Toolkit and Unreal Engine 4 as platforms for virtual experiments in pedestrian dynamics using the Oculus Rift*. Forschungszentrum Jülich, 2015.

Figure Sources

If not mentioned in the table below, figures are screenshots of the author's work or diagrams created by the author. All diagrams were created with the software *Visual Paradigm 12.2* using the FH Aachen academic license.