St. Pölten University of Applied Sciences

/ medien & digitale technologien

/fh/// st.pölten

# Crowd Simulation

## Real-time system for simulating human behaviour in a virtual 3D environment

Bachelor Thesis

Bachelor Course on Media Technology
at St. Pölten University of Applied Sciences

Submitted by:

**Lukas Felbinger**

1510261027

**Advisor**: FH-Prof. Dipl.-Ing. Dr. Markus Seidl, Bakk.

St. Pölten, 14.05.2018

# Declaration

- The attached research paper is my own, original work undertaken in partial fulfillment of my degree.

- I have made no use of sources, materials or assistance other than those which have been openly and fully acknowledged in the text. If any part of another person's work has been quoted, this either appears in inverted commas or (if beyond a few lines) is indented.

- Any direct quotation or source of ideas has been identified in the text by author, date, and page number(s) immediately after such an item, and full details are provided in a reference list at the end of the text.

- I understand that any breach of the fair practice regulations may result in a mark of zero for this research paper and that it could also involve other repercussions.

..... .....

St. Pölten, 14.05.2018 Signature

# Abstract

Crowd simulation is the approach of simulating the behaviour of a large number of pedestrians, used in game development, architecture, traffic planning as well as evacuation simulation. Essential elements of crowd simulations are a convincing behaviour model of the agents, satisfactory rendering and variety, a well-defined and semantically modelled environment. This thesis focuses on the behaviour of agents in the field of evacuation simulation.

Initially, the term *simulation* in general is examined, followed by the advantages of modern computer simulations. Afterwards, an overview of the different types of simulation is presented as well as the major field of application. Further, the specific simulation model called *crowd simulation* and its main characteristics is elaborated.

Subsequently, to determine the most suitable development platform for the implementation of this work, freely accessible game engines are evaluated, according to their accessibility and features. Additionally, the term *game engine* is explained to provide a better understanding of the difference between a game and its engine. Likewise, the essential components of a state-of-the-art 3D game engine are stated.

The crowd simulation is implemented using the Unity game engine and tested in a straightforward 3D model of a fictional public building. Before the development process is described, the minimum viable product is defined in the conceptual design to present the minimum requirements and limits of the desired crowd simulation. Finally, occurred problems during the implementation and their corresponding solutions are given. This work will be closed by discussing its accomplishments in addition to the prospects of the implemented crowd simulation.

# Kurzfassung

Crowd Simulation ist der Ansatz, um das Verhalten einer größeren Menschenmenge zu simulieren und wird oft in der Spieleentwicklung, Architektur, Verkehrsplanung sowie Evakuierungssimulation verwendet. Wesentliche Bestandteile von Crowd Simulationen sind ein überzeugendes Verhaltensmodell der Agenten, befriedigende Darstellung und Vielfalt, eine gut definierte und semantisch modellierte Umgebung. Diese Arbeit konzentriert sich auf das Verhalten der Agenten im Wesen der Evakuierungssimulation.

Anfangs wird der Terminus *Simulation* behandelt, gefolgt von den Vorteilen von modernen Computersimulationen. Anschließend wird ein Überblick von den verschiedenen Arten von Simulationen präsentiert, sowie das Hauptanwendungsgebiet dieser. Weiters wird auf den spezifischen Anwendungsbereich Crowd Simulation eingegangen und dessen tragende Charakteristiken ausgearbeitet.

Desweiteren, um die am besten geeignete Entwicklungsumgebung für die Implementierung zu bestimmen, werden frei zugängliche Game Engines, gemäß ihrer Zugänglichkeit und Features, evaluiert. Zusätzlich wird der Terminus Game Engine näher erklärt um ein besseres Verständnis über die Unterschiede von einem Spiel und dessen zugehöriger Spiele Engine zu geben. Gleichermaßen werden die wesentlichen Komponenten einer State-of-the-art 3D Game Engine aufgezeigt.

Die Crowd Simulation wird mithilfe der Unity Game Engine implementiert und in einem vereinfachten 3D Modells eines fiktionalen öffentlichen Gebäudes getestet. Vor dem Entwicklungsprozess wird das Minimum Viable Product im konzeptionellen Design definiert, um die minimalen Anforderungen und Grenzen der gewünschten Simulation aufzuzeigen. Abschließend werden aufgetretene Probleme und deren zugehörige Lösungen angegeben. Diese Arbeit wird mit einer Diskussion über die geleisteten Errungenschaften, sowie den Zukunftsaussichten der implementierten Crowd Simulation abgeschlossen.

# Table of Contents

**Declaration** II

**Abstract** III

**Kurzfassung** IV

**Table of Contents** V

**1 Introduction** 1

1.1 Motivation 1

1.2 Objective 2

1.3 Methods 2

1.4 Structure 3

**2 Related Work** 4

**3 Simulation** 5

3.1 Definition 5

3.2 Classification 6

3.2.1 Discrete-Event and Continuous Simulations 6

3.2.2 Dynamic and Static Simulations 7

3.2.3 Stochastic and Deterministic Simulations 8

3.3 Field of Application 8

3.4 Crowd Simulation 9

3.4.1 Sociological Model 9

3.4.2 Crowd Model 10

3.4.3 Environment Model 10

3.4.4 Agent Attributes 10

**4 Game Engine Evaluation** 11

4.1 Definition 11

4.2 Architecture 13

4.2.1 Resource Manager 14

4.2.2 Rendering Manager 14

4.2.3 Audio Manager 15

4.2.4 Input Manager 15

4.2.5 Scene Manager 15

4.2.6 Error Manager 15

4.2.7 Physics Manager 16
4.2.8 Networking Manager 16
4.2.9 AI Manager 16
4.3 Popular Game Engines 17
4.3.1 Quake Engine 17
4.3.2 Source Engine 17
4.3.3 Unity 18
4.3.4 Unreal Engine 19
4.3.5 CryENGINE 19
4.4 Field of Application 20
4.4.1 Game Development 20
4.4.2 Film Production 20
4.4.3 Architectural Visualization 20
4.4.4 Educational Training 21
4.4.5 Crowd Simulation 21
4.5 Discussion 22
**5 Implementation 24**
5.1 Conceptual Design 24
5.1.1 Agents 25
5.1.2 Classes and Relationships 25
5.1.3 Environment 26
5.1.4 Graphical User Interface 26
5.2 Development 27
5.2.1 Unity Editor 27
5.2.2 Environment 28
5.2.3 3D Models 29
5.2.4 Logic and Behaviour-Model 30
5.2.5 User Interface 35
5.3 Problems and Solutions 36
5.3.1 Performance 36
5.3.2 Collision Detection 36
5.3.3 Idle Activity 36
5.4 Testing and Results 37
5.4.1 First Test 37
5.4.2 Second Test 38
**6 Conclusion 40**
6.1 Conclusion 40
6.2 Future 41
**References 42**

**List of Figures** **43**

**List of Tables** **45**

**List of Listings** **46**

# 1 Introduction

This chapter will introduce the reader to the topic of this thesis, through explaining the background and the motivation of this work. Moreover, the objective will be clarified by presenting the research questions. Finally, the methods, which were used in the process of finding reasonable literature will be given, followed by a brief overview of this work´s structure.

## 1.1 Motivation

The concept of applying computer supported simulations of pedestrian behaviour in the process of planning buildings is becoming more approved in architecture, as it is more cost-effective to organize space utilization for humans. This approach is often referred to as *agent-based simulations*.

Crowd simulations with a focus on evacuation situations are a field of application of agent-based simulations, which virtually analyse and test the amount of time needed for humans to evacuate a building during a catastrophe. When the maximum amount of time needed to evacuate the environment is not met, additional exits can be arranged before the actual construction process of the building has begun. Environments tested with evacuation systems are usually well-defined and closed areas, such as airplanes, subways, ships as well as the inner areas of buildings.

Besides emergency situations, agent-based simulations are often applied for everyday situations in urban areas. The flow of traffic is a crucial element of large cities and is therefore often analysed to design better road networks beneficial towards more safety and less traffic jam. In addition to simulating the traffic on roads and sidewalks, agent-based simulations are also used to give knowledge about the traffic density in shopping centres for the evaluation of customer frequencies and average shopping times.

## 1.2 Objective

The aim of this thesis is to implement an agent-based simulation in a state-of-the-art 3D game engine with focus on evacuation situations. To determine the most suitable game engine, modern and freely accessible engines will be stated and subsequently evaluated. Throughout this work, following research questions will be answered.

**Research Questions:**

1. How can human behaviour in a public building be simulated during an emergency situation to evaluate space utilization and the placement of exits in real-time?
2. How can conventional human behaviour-models be implemented with a state-of-the-art 3D game engine in a visually straightforward, yet appealing way?
3. How can the planning of public buildings be supported in crowding the virtual models with humanly behaving agents in terms of safety and cost-efficiency?

## 1.3 Methods

The required literature for the theoretical research was gathered by using the following search engines:

- OBVSG (Austrian Library Network)
- Google Scholar

In addition to the digital research, public libraries were examined for topic related papers and books. The acquired literature in combination with the personal knowledge concluded to the outcome of this work. Moreover, to get reliable information about the covered game engines in this research paper, the official websites were visited.

## 1.4 Structure

**Chapter 1 – Introduction:**

The first chapter introduces the reader to the topic of this thesis, through explaining the background and the motivation of this work. Moreover, the objective will be clarified by presenting the research questions in addition to the methods used to answer those questions.

**Chapter 2 – Related works:**

In the second chapter, different approaches of other works and literature that relate thematically to this work´s subject are highlighted.

**Chapter 3 – Simulation:**

Chapter 3 clarifies the term *simulation,* gives an overview of the different types of simulation as well as the major field of application. Furthermore, the application area called *crowd simulation* and its main characteristics are described in more detail.

**Chapter 4 – Game Engine Evaluation:**

The fourth chapter explains the term *game engine* and surveys the architecture of a state-of-the-art 3D game engine. Additionally, freely accessible game engines are evaluated and the field of application is determined. Eventually, the most suitable game engine for this work´s implementation is resolved.

**Chapter 5 – Implementation:**

In chapter 5, the implementation process of this work´s crowd simulation is documented. First, the conceptual designs of environment and behaviour-model are presented, followed by the practical implementation in Unity. Finally, major problems and solutions will be stated, as well as the results of the final testing.

**Chapter 6 – Conclusion:**

Chapter 6 closes this work and summarizes the findings of the research along with those of the practical approach. In the end, future plans to further improve the simulation are dispalyed.

# 2 Related Work

The master´s thesis *Crowd simulation for virtual environments in Unity* by Jaros is an approach to create an agent-based simulation in a 3D model of the Westbahnhof in Vienna. While the complexity of the human behaviour-model and the virtual environment are more complex compared to this work´s approach, the concept is still very similar. The concept is to fill a 3D environment with a large number of realistically behaving agents and analyse the results.

The books *Game Engine Architecture* from Jason Gregory and *Game Engine Design and Implementation* from Alan Thorn cover the explanation of the term *game engine* and moreover the required components. While Gregory displays the architecture of a game engine in a complex manner, Thorn tries to break down the architecture into numerous segments called ‘managers’. Gregory additionally treats some popular game engines and how they have evolved over the time. By consulting these two books, along with some smaller works, most of the general, but important, aspects of a game engine could be clarified.

# 3 Simulation

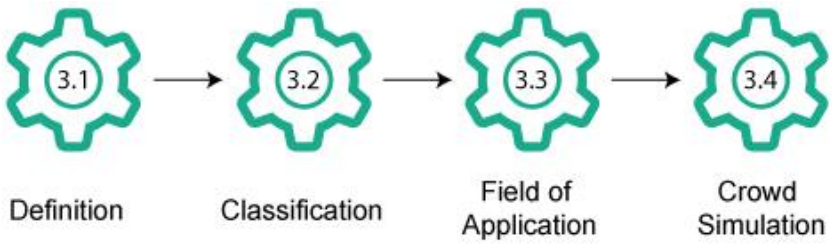



*Figure 1:* Graphical overview of chapter 3

This chapter examines the term *simulation* and states the advantages of using a computer in the simulation process. Additionally, an overview of the different types of simulation will be presented as well as the major field of application. At the end of this chapter, the specific simulation model called *crowd simulation* and its main characteristics will be described.

## 3.1 Definition

Simulation is the technique of imitating the behaviour of a certain situation or also known as *source system*. The two major goals of simulations are gaining information more efficiently called an *analytic simulation* as well as training and entertaining personnel called a *virtual environment simulation*. The analytic simulation is a quantitative analysis based on exact data and should be executed in an as-fast-as-possible manner. In contrast, a virtual environment simulation is executed in a scaled real-time within a virtual environment. When human interaction is involved in the simulation, it is referred to as *constructive simulation* and one without any human interaction *autonomous simulation*. (Choi & Kang, 2013, p. 6).

With the appearance of computers in 1940, numeric models were converted into computer-based solutions, reducing human computation errors and being able to operate at a much higher speed. Additionally, computer simulations provide cost-effective and risk-free solutions when compared to real-life experimentations (Wainer, 2017, p. 2).

## 3.2 Classification

In their book, Choi & Kang classify the simulation of dynamic systems into three main types: *discrete-event simulation*, *continuous simulation* and *quantum simulation*. Quantum simulations provide insight about specific physics problems but won´t be treated any further in this work. Another famous type of simulation is the *Monte Carlo simulation*, which is static system simulation (Choi & Kang, 2013, p. 6). Furthermore, it is important to know if the simulation contains random variables called a *stochastic simulation* or no randomness at all called a *deterministic simulation*.

There are 3 central categories to classify a simulation:

- Discrete-event or continuous
- Dynamic or static
- Stochastic or deterministic

### 3.2.1 Discrete-Event and Continuous Simulations

“Discrete-event simulation is a computer evaluation of a discrete-event dynamic system model where the operation of the system is represented as a chronological sequence of events” (Choi & Kang, 2013, p. 9).

In a discrete-event simulation the essential elements are events and variables of which we constantly keep track. There are three fundamental variables that are often used for this sort of simulation: the time variable to determine the amount of simulated time that has passed, the counter variables to keep count of the number of times a certain event has taken place and the system state variable to specify the current state of the system. Those variables change or update their value on any occasion of an “event” (see Figure 2) and all relevant data will be received as output (Ross, 2013, p.112).

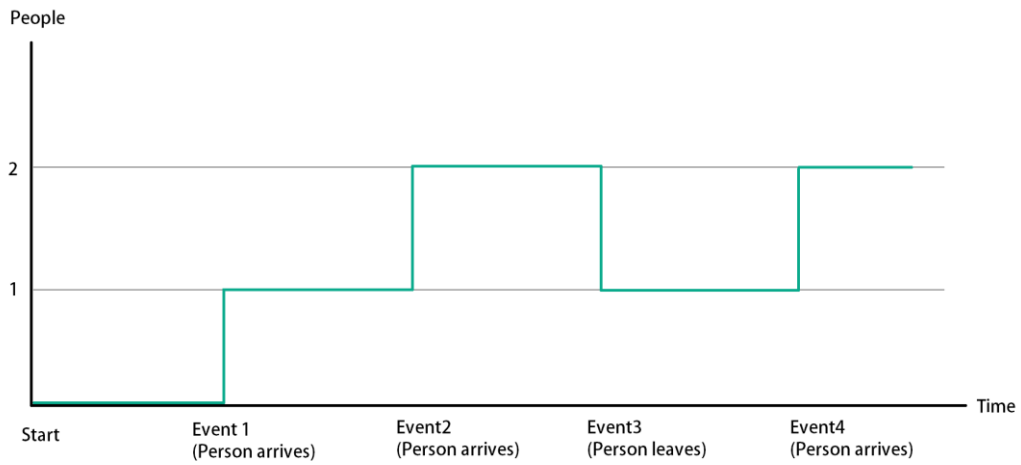



*Figure 2:* State changes caused by discrete-events over time

In contrast to discrete-event simulations, the variables of continuous simulations change continuously as a function over time (see Figure 3). A continuous simulation model is usually described by using differential equations.

A *derivative function* is used to determine the rate of change of the state variables instead of specifying the next state directly. Given an input and a state, only the rate of change of the state at any particular time instant can be seen on the time axis. Accordingly, the issue is to compute the state of any point in the future, which can be solved by *numerical integration methods*. (Zeigler, Praehofer, & Kim, 2000, p.49-51).

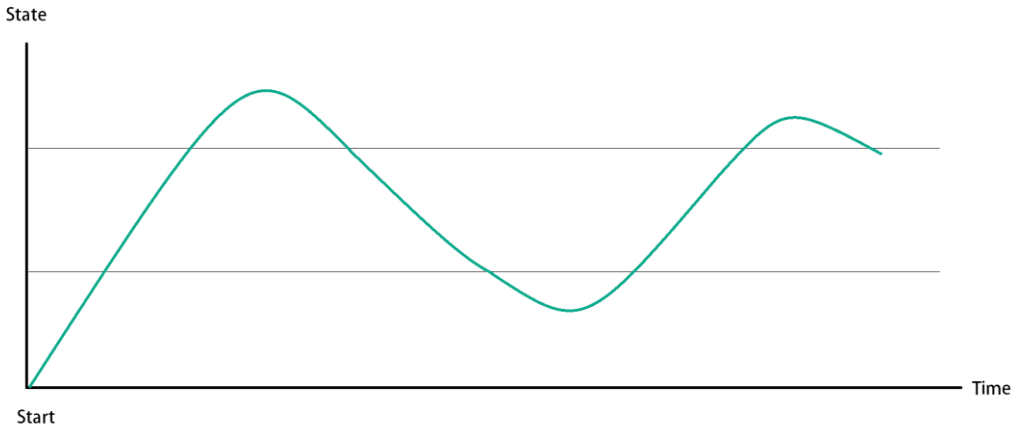



*Figure 3:* Continuous changes of a state variable over time

### 3.2.2 Dynamic and Static Simulations

Dynamic simulations look at state changes that occur over time. Therefore, time is an important aspect of dynamic simulations, on the contrary, static simulations are not based on time. While the previously covered simulation types are categorized as dynamic system simulations, the Monte Carlo simulation is a static system simulation.

Numerical integrations of functions, which are impossible to evaluate with direct analytical methods, are computed by a class of computational algorithms that rely on repeated random sampling. This technique has been used in forecasting models in order to understand the impact of risk and uncertainty (Choi & Kang, 2013, p. 9).

### 3.2.3 Stochastic and Deterministic Simulations

Simulation models can be based on stochastic or deterministic values. In a stochastic simulation model at least one variable is stochastic in nature, whereas in a deterministic model, variables, both input and output, are not random at all. (Bandyopadhyay & Bhattacharya, 2014, p.17).

Deterministic simulations always acquire constant input values and return constant output values. Stochastic simulations acquire one or more random input values and produce random outputs (see Figure 4). For this reason, a deterministic simulation will always behave the same, given the same input values, no matter how often the simulation is started.

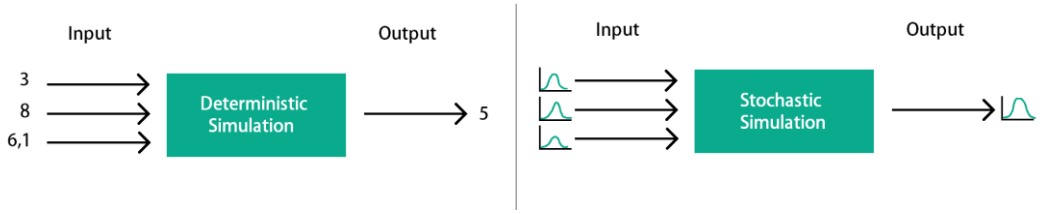



*Figure 4:* Input and output examples of stochastic and deterministic simulations

## 3.3 Field of Application

In the fields of science and technology, simulations are widely applied at this time. The Winter Simulation Conference (WSC) publishes simulation related research papers every year, therefore their online archive is a reliable source to learn about the recent application areas of simulations (Bandyopadhyay & Bhattacharya, 2014, p.5).

The following list of application areas is obtained from the website of the WSC from the year 2017:

- Agent-based simulation
- Architecture and Construction
- Healthcare applications
- Military applications
- Manufacturing applications
- Environment and Sustainability Applications

## 3.4 Crowd Simulation

The simulation of crowds is a common application area of agent-based simulations. The aim is to simulate interactions and actions of autonomous agents as individuals as well as in groups to analyse the virtual behaviour in the simulation and reflect it onto the real world.

Crowd behaviours have especially been modelled in the field of architecture with a main focus on crowd evacuation simulators. The virtual environments, which are usually used for this simulation model, are well-defined and closed areas, such as airplanes, subways, ships as well as the inner areas of buildings (Thalmann & Musse, 2012, p.2).

In the work of Jaros M., he developed a realistic crowd simulation within a virtual replication of the Westbahnhof, which is an essential train station in Vienna. In the beginning of the application´s run-time, multiple agents with different strategies and activities are spawned. While strategies (e.g. go to shopping) are constant for an agent´s lifetime, activities (e.g. to buy food, to buy a newspaper) are chosen and prioritized dynamically by every agent (Jaros, 2015).

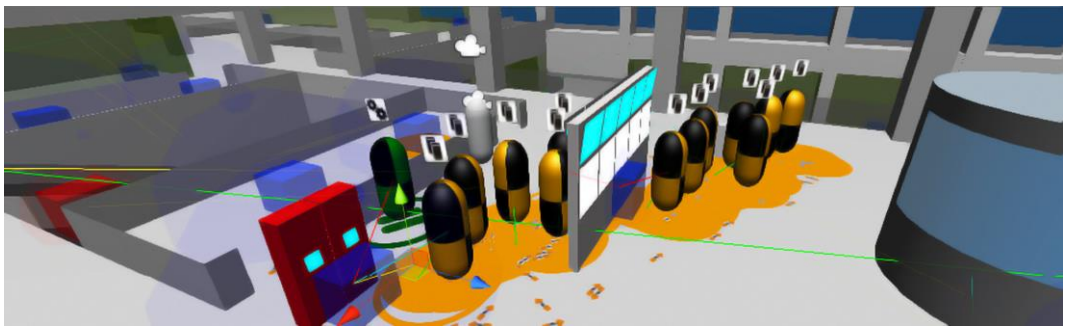


*Figure 5:* Demonstration a simulation in a train station[1]

### 3.4.1 Sociological Model

Although sociology is mainly interested in researching collective behaviour, only a few works on crowd simulation have been done in this field. There are several different approaches of implementing sociological models in crowd simulations. An approach by McPhail to model a crowd is based on *perception control theory* where separate individuals maintains their relationship to others by trying to control their experience. In this case it is a spatial relationship to others in a group (Thalmann & Musse, 2012, p.7).

[1] http://mj1.at/projects/crowd-simulation-for-virtual-environments-in-unity (04.05.2018)

### 3.4.2 Crowd Model

Essential parts of crowd modelling are a convincing behaviour and a satisfactory rendering, which both should be treated equally important. If the rendering is not suitable, even the best model of behaviour will not be convincing. Furthermore, if there is no appropriate behaviour model, even a simulation with the best rendering will look dull quickly (Thalmann & Musse, 2012, p.8).

### 3.4.3 Environment Model

In addition to the previous essential elements, the surrounding environment must be graphically and semantically modelled, for a behaviour to make sense. An appropriate environment model ensures that characters behave believably, by performing actions in the right places (Thalmann & Musse, 2012, p.7).

### 3.4.4 Agent Attributes

Every agent has a set of attributes and variables representing characteristics of a real human. There are static attributes, such as name and gender, but also dynamic ones like current position (coordinates), state and health. Depending on the duration of the simulation, attributes like age, height and weight can be either static or dynamic. Moving speed and reaction time are variables. which are calculated by using other values, being for example health condition and age.

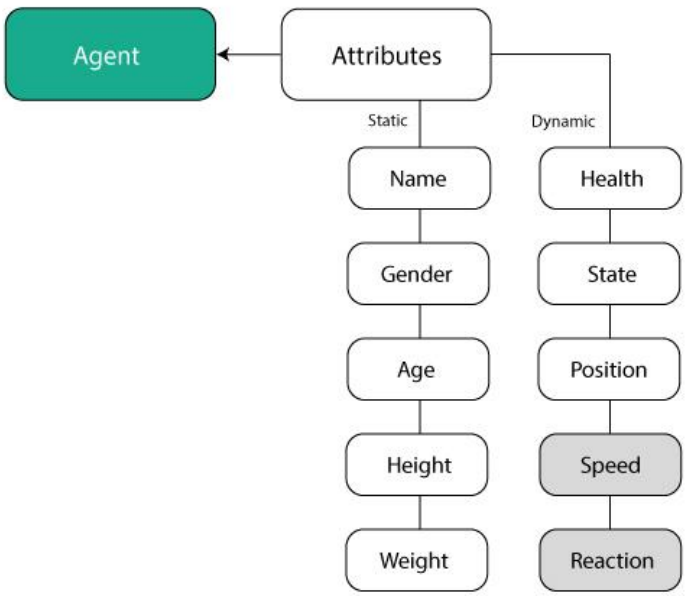



*Figure 6:* Attributes of an individual agent in a crowd simulation

# 4 Game Engine Evaluation

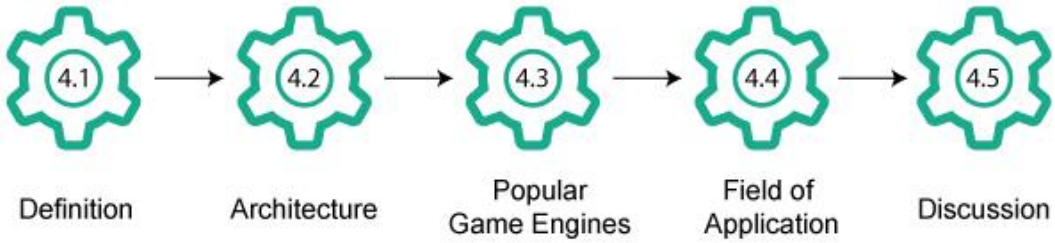



*Figure 7:* Graphical overview of chapter 4

The aim of this chapter is to explain the term *game engine* and present an overview of the architecture of a state-of-the-art 3D game engine. Furthermore, the advantages and disadvantages of popular and freely accessible game engines will be individually investigated and afterwards compared to each other. At the end of this chapter, the field of application will be determined and the most suitable game engine for this work´s implementation will be discussed.

## 4.1 Definition

The term game engine describes a software framework, which provides the essential components needed for almost any video game in order to reduce the development time of the game production. Moreover, further time can be spent on the actual creative part of the production and the idea itself. Before the concept of game engines was created, every game development started from scratch, which basically means that every component like physics or sound management had to be renewed every time.

The game engine can´t be seen as the motor that powers the video game, but as the essential centerpiece, without which the development or the execution of the game would not be possible. Additionally, the game engine has a major impact on how files are arranged and imported during the process of the game production, but even further on how the workflow unfolds during the development (Thorn, 2010, p. 3-4).

Since the game engine contains practically all generic components that can be found in a game, likewise there must be specific content, which makes the end-product unique. This content is mostly exclusive to visual and acoustic elements that varies the result from previous or upcoming productions (Thorn, 2010, p. 5).

The bridge between the abstract game engine and the specific content are tools, which connects the static content to the active engine (see Figure 8). An example for a development tool could be a map editor that provides all functionalities to create an appealing world for the end user and is then fed into the game engine (Thorn, 2010, p. 6).

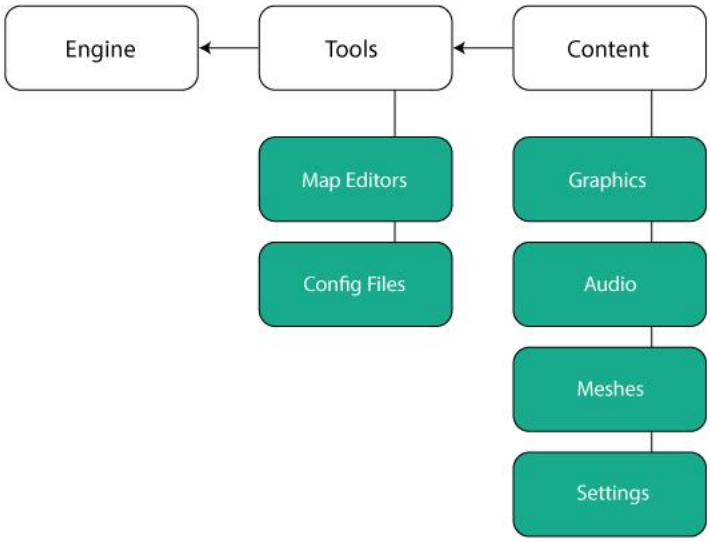



*Figure 8:* Relationship of the parts of game development[2]

Even though unique content and content creation tools are crucial for the development process, this chapter will focus on the actual game engine itself.

The term itself derived about 1995, when first-person shooters like *Doom* by id Software gained enormous population. Since *Doom* was created with a clear separation between the core components and the individual game assets, people were able to change the game experience by adding new assets with only nominal changes to the engine (Gregory, 2014, p. 11).

[2] (Thorn, 2010, p. 7)

## 4.2 Architecture

The architectural design of a game engine often varies, because there is no strict standard that defines an ordinary game engine. Therefore, different game engines have a different set of components, in order to fulfil their different purposes.

Although there are no formal or informal standards, there are some essential components also called managers every typical engine holds (see Figure 9). The engine programmer divides the engine into many different managers, according to their functional status (Thorn, 2010, p. 9).

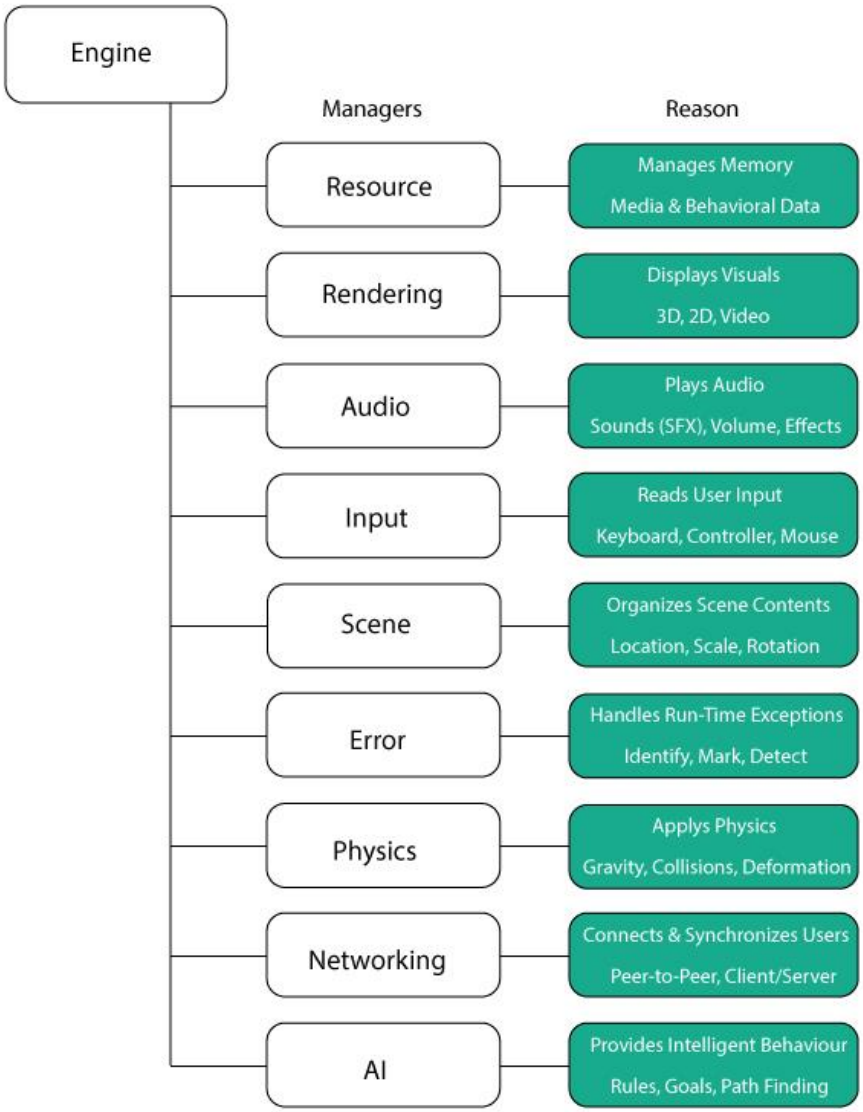



*Figure 9:* Relevant managers of a modern game engine and their function

### 4.2.1 Resource Manager

The resource manager is an essential part of a game engine, considering that any accessing of media data or behavioural data is done with it. While media data is referring to resources, such as graphics, audio, materials, animations and everything that defines how a game looks, behavioural data can be simple text files, which determine how the game runs, like window size, resolution or even subtitles. As well as accessing resources, the resource manager also has the function to load and unload resources in and out of the memory during run-time (Thorn, 2010, p. 10).

*Table 1:* Media file formats managed by game engines

| Image | Audio | Video | Models/Animations |
|---|---|---|---|
| BMP, TIF, TGA, JPG, PSD | MP3, OGG, WAV, AIFF | MOV, AVI, MPG, MP4 | EG, MAX, BLEND, MB, MA |

### 4.2.2 Rendering Manager

To display the desired visual assets to the monitor it isn´t enough to just load them into the memory with the resource manager, in addition, there needs to be a graphical component that is able to draw the loaded resources onto the screen.

This Rendering manager is the combination of a graphics rendering library or API (application programming interfaces), such as DirectX or OpenGL and a series of functions and tools created by the engine developer. Further, the rendering manager must decide how and where pixels are drawn to the screen (Thorn, 2010, p. 11).

*Table 2:* APIs for rendering managers

| API | Founder |
|---|---|
| **Direct3D** | Microsoft |
| **OpenGL** | Silicon Graphics |
| **Vulkan** | Khronos Group |
| **Mantle** | AMD, DICE |

### 4.2.3 Audio Manager

Equivalent to the rendering manager, the audio component has the job to address the acoustical elements loaded by the resource manager to the proper audio source of the system and playback them as adjusted (volume, effects).

Although the use of a pre-existing audio engine saves time, nevertheless custom software development, integration work, fine-tuning and attention is beneficial to produce high-quality audio in the final product (Gregory, 2014, p. 49).

### 4.2.4 Input Manager

Every game engine has a proper input management to handle and process incoming signals from a human interface device (HID) including keyboard and mouse, gamepads, or other specialized controllers like dance pads, steering wheels and fishing rods. The Input manager reads the raw data coming from the device, debounces button inputs to avoid spurious open/close transitions, detects button events, interprets accelerometer inputs and often allows the user to customize the mapping between physical inputs and logical functions (Gregory, 2014, p. 48).

### 4.2.5 Scene Manager

The scene usually refers to a level in a game or a stage in a movie where all currently needed characters and props come together. Since it is not enough to just load resources from files and display them randomly on the screen, an additional manager is necessary in between to decide how resources are associated with each other and the world. Furthermore, this so-called scene manager coordinates the active assets and their motion, regardless of them being graphics or audio as well as keeps track of time with the ambition of firing events (Thorn, 2010, p. 13-14).

### 4.2.6 Error Manager

Profiling and debugging tools of the error manager usually include methods for debug drawing, such as an ingame menu system or a console, as well as a facility for displaying the current profiling statistics while the game is running. Additionally, the manager provides functions to debug custom print statements, along with tools to analyse and dump memory usage for the sake of performance optimization. Some engines are even capable of recording short events, which basically means the developer not only sees the error message when an application crashes, but also the video of what exactly happened during the crash (Gregory, 2014, p. 44-45).

### 4.2.7 Physics Manager

Although not all games feature realistic physics, they still require reasonable collision detection, in favour of avoiding solid objects moving through other solid objects. Moreover, it is required to even be able to interact with the virtual world in a plausible manner. When collisions are detected by the physics system, they are almost always resolved as a part of the physics integration, since collision and physics are usually coupled. Considering that there are already completely coded physics engines on the market, companies tend to integrate such a third-party SDK into their engine, instead of writing their own (Gregory, 2014, p. 46).

### 4.2.8 Networking Manager

Games often rely on cooperative and competitive gameplay in which two or more human players perform within a single virtual world. While local multiplayer only requires one hardware machine with two or more input devices and takes place on either a single screen or a split-screen with multiple cameras, online multiplayer requires a mass of logic revolving around communicating with different servers or even other client computers (Gregory, 2014, p. 49-50).

There are three main methods to establish a connection between multiple players over the network:

- Peer-to-Peer: Participants are simultaneously server and client; All Players each share equal authority and networking tasks.
- Client-Server with Client as Host: Standard networking method of most game engines. A single Player is the host and responsible for the connection.
- Client-Server with Dedicated Server: An external dedicated server connected to the internet provides the player network.

### 4.2.9 AI Manager

Artificial intelligence is the attempt to digitally simulate the human mind in terms of learning and problem solving.

Formerly, artificial intelligence wasn´t considered as a part of game engines. Companies like Kynogon developed AI SDKs, such as *Kynapse*, which contributed most of the technology required to construct commercially profitable AI. *Kynapse* provides navmesh generation, path finding, static and dynamic object avoidance and analysing of game spaces, to find strategical significant spots (Gregory, 2015, p. 53).

## 4.3 Popular Game Engines

There is a wide variety of freely accessible 3D game engines on the market that are suitable for a professional development far beyond video games. Criteria for choosing a suitable game engine are commonly the following: target platform, genre, usability, licensing, application size, programming language and community.

### 4.3.1 Quake Engine

The Quake Engine was developed in 1996 by id Software for the 3D first-person shooter game *Quake*. This engine was one of the first ones that used three dimensional figures instead of two dimensional sprites, additionally lightmaps and dynamic light sources were featured. However, the main reason why *Quake* accomplished such popularity was due to the opportunity to customize a game completely, without creating everything from scratch. In consequence, the modding community was born.

Even though, the Quake Engine became obsolete, the technology has been used to create many other games and even other engines, e.g. Valve´s Source Engine has distant roots to the former Quake Engine (Gregory, 2014, p. 26 - 27).

### 4.3.2 Source Engine

The Source Engine is a high-quality engine developed by Valve Cooperation in 2004 and is most notably used in games like *Half-Life 2*, *Portal* and *Team Fortress 2* (Gregory, 2014, p. 28).

Even though the original *Half-Life* and *Counter Strike* were initially powered by the former Goldsource/GoldSrc Engine, Valve ported them to their current engine, leading to *Half-Life: Source* and *Counter-Strike-Source.* As mentioned before, the Sources Engine has distant roots to the Quake Engine, which derives from the fact, that the GoldSrc Engine was a heavily modified Quake Engine.

Advanced artificial intelligence management for NPCs (Non-Player Characters) is fully featured by the Source Engine, in order to provide sophisticated character navigation and behaviour, such as running, jumping and climbing stairs as well as ladders. Moreover, the engine is highly modular, which allows users the modification of Source-based games to a very fine degree. This is supported by the Valve Developer Community that arranges references and tutorials beneficial towards developing mods. Further advantages are realistic physics, lip synchronization, facial expression technology, detailed character textures, advanced rendering features and a material system that is able to define the

texture an item uses, but also what it is made from, which will affect the sound it produces when motion occurs along with its mass

The Source SDK is freely available, after purchasing a Source-based game in the digital distribution platform called Steam. The main benefit of Steam is that once a product is developed with the Source Engine, it can directly be submitted to Steam´s workshop, where community members can rate and download the result.

In accordance with Valve´s terms of agreement, games and modifications developed by the Source Engine have to be free; otherwise companies like Havok need to be paid a licensing fee of $25,000 for their physics engine[3]. Supported scripting languages include Squirrel and LUA.

### 4.3.3 Unity

Unity is a powerful game development and run-time engine developed by Unity Technology in 2005 and well-known for supporting a wide range of platforms. The use of Unity allows deploying games on mobile platforms, consoles, desktop computers and Smart TVs. In addition, it supports a Webplayer in favour of deploying on the majority of all web browsers. The main design intention of Unity is to ease the development and multi-platform game deployment. An integrated, straightforward environment editor in which you are able to create and manipulate the assets and entities in the game world and preview the changes ad hoc directly in the editor or on the target hardware (Gregory, 2014, p. 30).

Moreover, Unity comes with a powerful animation system that supports animation rearranging, involving the ability to change an animation authored for one character on a different character. Unity´s capability to analyse and optimize the end-product on each target platform in order to manage the performance quality on each deployment platform is provided by a powerful suite of tools. The supported scripting languages are JavaScript, C# and Boo, also networked multiplayer games are possible (Gregory, 2014, p. 30).

Unity has four different subscription options: personal, plus, pro and enterprise. The personal edition is freely accessible, although if a commercial product reaches annual gross revenues in excess of $100,000, you need to change your subscription to a fitting commercial one[4].

[3] https://partner.steamgames.com/documentation/source_games (04.05.2018)

[4] https://unity3d.com/de/legal/terms-of-service/software (04.05.2018)

### 4.3.4 Unreal Engine

The Unreal Engine is a powerful prototyping tool developed by Epic Games in 1998 for the first-person shooter game *Unreal*. As a result of this, Unreal Engine has become a major competitor to *Quake* in the FPS genre. Further famous games powered by the Unreal Engine are *Unreal Tournament 2004*, *Gears of War* and *Tekken 7*. Even though the Unreal Engine is famous for its first-person related games, third person is supported too. Nevertheless, this game engine isn´t perfect, thus most developers have to modify it in several ways in order to run their product optimally on the target platform (Gregory, 2014, p. 27).

Since 2015 Epic Games announced that their game engine is freely accessible for everyone. When you ship a game or an application, you pay a 5% royalty on gross revenue after the first $3,000 per product, per quarter.

### 4.3.5 CryENGINE

The CryENGINE was originally developed as a Nvidia tech demo by Crytek in 2002 and, as soon as the potential of this technology was noticed, Crytek used the engine to turn it into the game *Far Cry*. Over the years, many games have been made with CryENGINE including Crysis, Codename Kingdoms, Warface and Ryse: Son of Rome. Additionally, Crytek´s platform contributes powerful asset-creation tools and high-level real-time graphics (Gregory, 2014, p. 28-29). Supported target platforms include, Xbox One, PlayStation 4, Windows, Linux and Oculus (https://www.cryengine.com/features/platforms, 05.05.2018). The scripting languages encompass C++, C# and Lua.

In 2016, Crytek announced their new ‘Pay What You Want’ model for the latest CryEngine V, which makes the engine completely free and furthermore it´s 100% royalty free[5].

[5] https://www.cryengine.com/news/introducing-cryengine-v-a-letter-to-our-community (05.05.2018)

## 4.4 Field of Application

Although game engines are mostly used in the process of making video games, state-of-the-art 3D game engines are additionally used for film productions, scientific simulations, interactive architectural building, complex data visualizations or even learning experiences for educational purposes.

### 4.4.1 Game Development

The original intent of game engines was the creation of video games, with the purpose to separate unique game content from abstract components, which can be reused for another game, such as physics.

Most video games are examples of what computer scientists would call ‘soft real-time interactive agent-based computer simulation’. This phrase means that a game is a dynamically changing virtual world, where several distinct entities collaborate, user input is processed in real-time and missed deadlines are not catastrophic (Gregory, 2014, p. 8-9).

### 4.4.2 Film Production

Conventional animated films are usually pre-rendered, which means that the whole digital content is rendered only once with fix settings, camera angles and effects.

In contrast to pre-rendering, real-time rendering processes each frame at the precise instant you see it. At 30 FPS (frames per second) a single frame takes 1/30th of a second to render. Since it´s in real-time, camera movement and user engagement are possible while the run-time action continues to occur. Moreover, because the created visual resources for the film are typical 3D models, the assets can be reused for further projects, such as a video game. Several short films, for instance Unity´s *Adam*[6] has already been composed and rendered using a game engine.

### 4.4.3 Architectural Visualization

Recently, engineers, architects and constructors started to use real-time visualization tools, instead of conventional CAD software to attain more interactivity. This allows the builders and customers to move within the desired construct and change elements such as interior colours, light settings and object materials during the presentation. Additionally, already implemented structures

[6] https://unity3d.com/pages/adam (06.05.2018)

can be used for further simulations, for instance evacuations and crowd dynamics, since the game engine provides all required tools for such extensions.

Rontsinsky M. accomplished a realistic visualization of Vienna´s Semper-Depot by using Unity. The building is named after one of the architects Gottried Semper and was built in 1877. One intention of his application was to evaluate necessary architectural changes in real-time, such as arranging modern fire protection without violating the monument´s protected status (Rontsinsky, 2012).

### 4.4.4 Educational Training

The same as our society has developed in terms of technology, teaching and learning have evolved in the same manner. This leads to modernized methods in the process of education like the usage of entertainment software instead of traditional teacher-centred instruction. An approach to the development for educational games is the usage of game engines. This allows the creation and modification of the learning application suited for the intended use, for example virtual driving lessons.

An example of a learning experience through gaming is *ThIatro*, developed in Froschauer´s approaches to address cultural heritage playfully. *ThIatro* is a level-based 3D game created with Unity to combine learning and fun in order to make the communication of art historical facts more entertaining (Froschauer, 2012).

### 4.4.5 Crowd Simulation

The crowd simulation by Jaros M. presented in chapter 3 (see Page 9) is implemented using the Unity game engine and tested in a realistic 3D model of a train station. By changing the virtual surrounding, adjusting the strategies and activities, the behavioural model can be reused for other environments.

## 4.5 Discussion

Since all covered game engines are technically capable to provide enough functionality to be considered as a development tool for a crowd simulation, the focus of this discussion lies on the accessibility. While they are all completely free, some of them have an additional subscription model in which the user gets further benefits from paying a monthly salary. These benefits include special training provided by the engine´s developers, exclusive insights into the source code or even less response time on support requests.

*Table 3:* Accessibility of all covered game engine

| | **Quake Engine** | **Source Engine** | **Unity** | **Unreal Engine 4** | **CryENGINE V** |
|---|---|---|---|---|---|
| **License** | GNU[7] | Free | Free (Personal) | Free | Free (Pay what you want) |
| **Royalty** | Free | Free (Non-profit use) | Free | 5% (After $3000) | Free |
| **Subscription** | No | No | Pro<br>Premium<br>Enterprise | Yes | Base<br>Premium |
| **Scripting** | QuakeC | Squirrel, LUA | C#, Boo, JavaScript, | C++ | C++, C#, LUA |

Scripting is another important aspect when discussing the accessibility of a game engine. Game engines that support different scripting languages allow users to choose the most suitable language for their application. Furthermore, multiple scripting languages give users a higher chance to code with a language, they are already familiar with.

[7] http://www.gnu.org/licenses/gpl.html (06.05.2018)

As mentioned, all the covered game engines are basically capable of powering a fully functional crowd simulation and they are also all recommendable in terms of accessibility (see Table 3). Therefore, a further, more detailed rating is required, to determine the most suitable game engine.

To rate the five game engines clearly in a single visualization, a table is used where:

- Red (-) equals bad
- Orange (~) equals neutral
- Green (+) equals good

*Table 4:* Rating of game engines according to the current state-of-the-art

| | Quake Engine | Source Engine | Unity | Unreal Engine 4 | CryENGINE V |
|---|---|---|---|---|---|
| **Graphics** | – | ~ | + | + | + |
| **Scripting** | – | ~ | + | ~ | + |
| **Tutorials** | + | ~ | + | + | ~ |
| **Support** | – | ~ | ~ | ~ | – |
| **Learning Curve** | ~ | ~ | + | ~ | ~ |
| **Accessibility** | + | ~ | + | ~ | + |
| **Maintenance** | – | – | + | + | + |

Considering all mentioned elements which are important for a capable game engine, Unity is the most suitable development tool for this work´s implementation. While the accessibility might be equally good compared to the CryEngine, there are many tutorials available for Unity projects due to its immense community. Moreover, the learning curve benefits from its straightforward interface and the graphics are completely sufficient for the visual presentation of a decent 3D crowd simulation.

# 5 Implementation

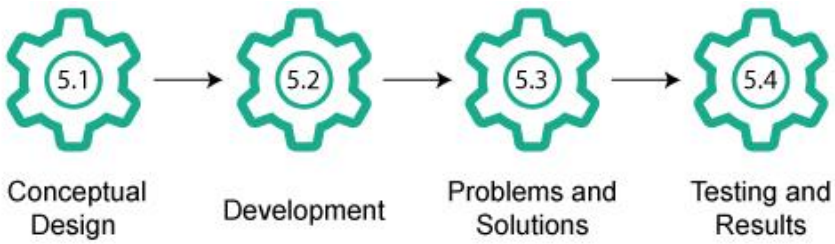



*Figure 10:* Graphical overview of chapter 5

In this chapter all previous researches will be taken into account and a crowd simulation will be developed Unity. First, the conceptual design of the simulation will be presented, including defining the minimum viable product, determining the required classes and their relationships as well as creating a rough sketch of the 3D environment. Afterwards, the whole environment and its interior will be realized, followed by the implementation of the behaviour-model. Finally, major problems and their solutions will be stated, as well as the final testing results.

## 5.1 Conceptual Design

The conceptual design is the initial phase of large projects after the idea has been found. Visualising the potential end-product helps to get less surprised by upcoming difficulties in the development process. Furthermore, this phase helps to manage resources, such as time more efficiently. The confectioning process also helps to determine the *minimum viable product* (MVP) also referred to as the *minimum viable simulation* (MVS) in this thesis.

The minimum viable simulation is the functioning end-simulation with just enough complexity and features to produce relevant results, which can be analysed afterwards. The following sections within the chapter conceptual design will show the limits and requirements of the MVS.

### 5.1.1 Agents

Humans, represented by programmed agents need to be visually and characteristically unique. The most straightforward approach to make them visibly distinguishable is to give each agent a different random colour. The unique characteristics of each human are defined by personal attributes such as name, age and weight. Further variables could be a burden to represent luggage, health to display the medical condition as well as moving-speed. To make use of all the activities in the virtual surrounding area, a set of different states is required. Each state results in a different activity related to an object in the environment (see Figure 11) and changes at completion.

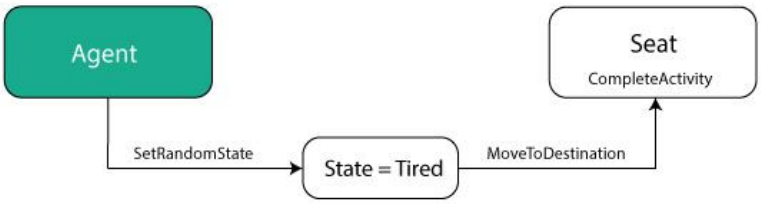



*Figure 11:* Relationship between agents and activity-objects

### 5.1.2 Classes and Relationships

As previously stated, the simulation already requires two different classes, one to represent humans and one to represent properties. Since the simulation will include numerous different properties such as seats and shops, the property-class can be an abstract parent class. Moreover, a class to store all initial values called “Initializer” and a class to handle the functionality during run-time named “Manager” will be implemented.

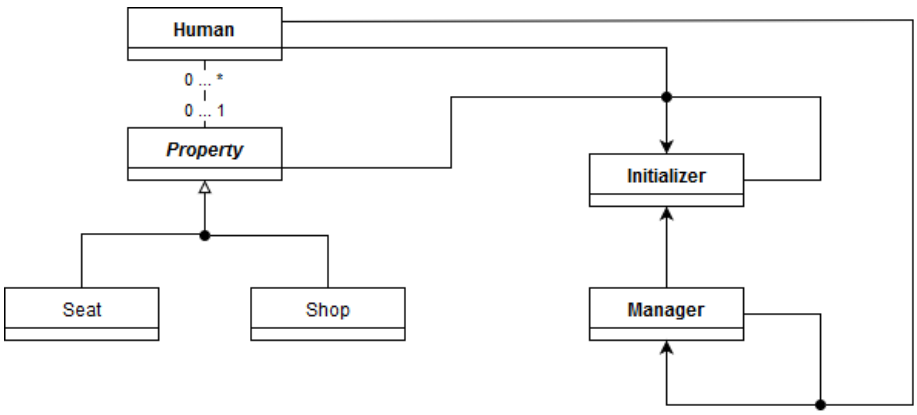



*Figure 12:* Relationship of the essential classes

Every human can be associated to exactly 0 or 1 properties and every property can have 0 or more (*) humans depending on the capacity. For instance, the human can be in only one shop at a time, but every shop can contain multiple people. The two classes “Initializer” and “Manager” will be implemented with the singleton design pattern, meaning there can only ever be one instance of those classes. While the class diagram will get much more complex during the development process, the current version is sufficient to start programming.

### 5.1.3 Environment

The virtual environment will have at least two floors connected with stairs on which the agents can move freely. Furthermore, the minimalistic building shall provide multiple activities for the agents to interact with, such as seats and shops, as well as different entries/exits to enter and leave the building from various positions.

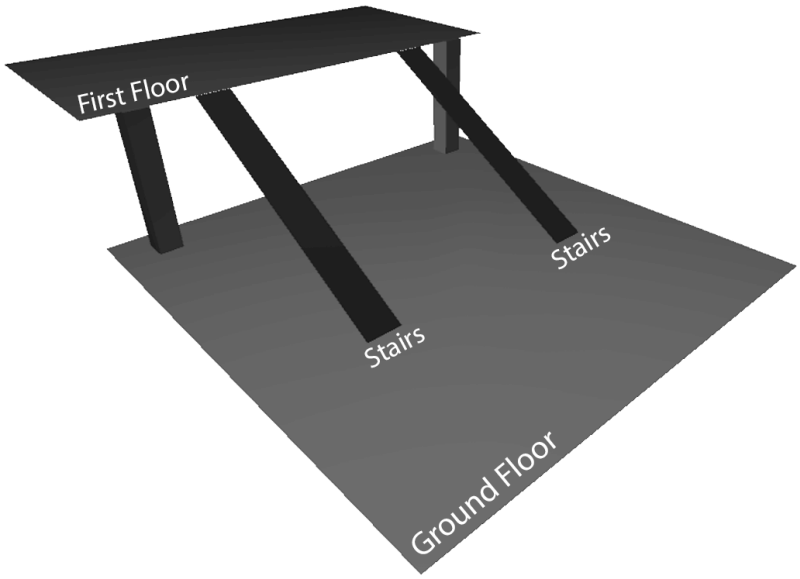



*Figure 13:* Model of a simple simulation environment

### 5.1.4 Graphical User Interface

The last important requirement for the minimum viable simulation is the graphical user interface (GUI). With the GUI, the user is able to observe meaningful simulation values in real-time, for instance the number of active agents, running-time of the simulation, characteristics of individual objects and current state of the simulation. Additionally, the GUI should provide the possibility to restart and exit the simulation at any given time.

# 5.2 Development

Considering the previous evaluation of the game engines and the conceptual design of the crowd simulation, all requirements for the start of the development phase are met. From this point, every practical step, such as programming the behaviour-model as well as modelling the environment, takes place in Unity

## 5.2.1 Unity Editor

The Unity editor is made up of multiple tabbed panels called Views, which all have specific purposes. Before the implementation starts, a new project with a customized tab-layout is created (see Figure 14).

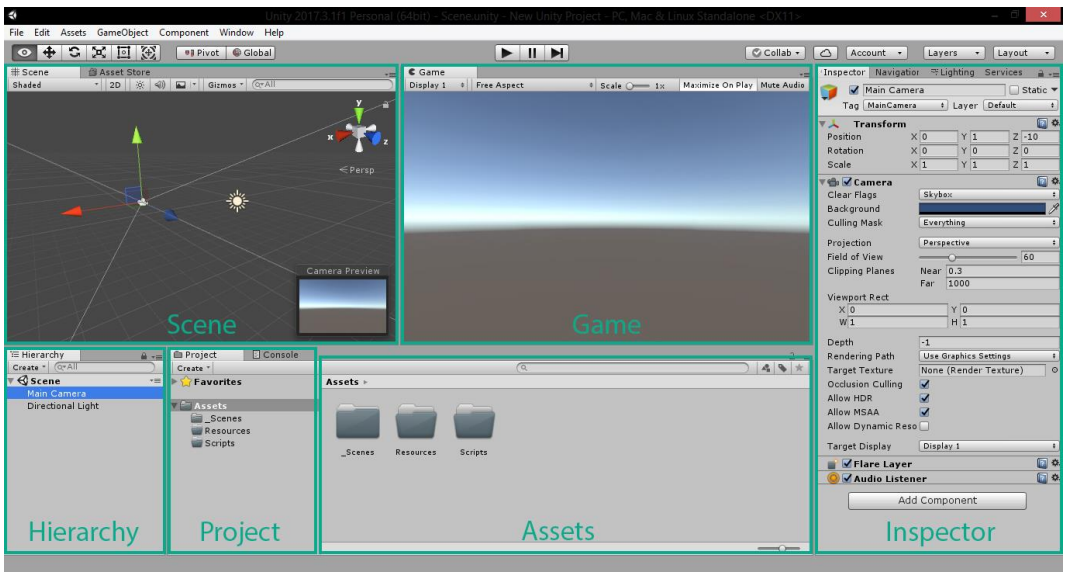



*Figure 14:* Interface of the Unity editor with customized layout

- **Scene**: The scene view is used to place, select and position gameobjects, such as the camera, lights and models.
- **Game**: The game view is rendered from the camera in the game. It represents the final, published game.
- **Inspector**: The Inspector displays detailed information about the currently selected gameobject, including all attached Components like Colliders, Scripts and Audio Sources.
- **Hierarchy**: The Hierarchy contains every gameobject in the current Scene.
- **Project**: In the Project Browser, all assets that belong to the project can be accessed and managed.
- **Assets**: The Asset Browser provides a more detailed view of the assets within the project.

### 5.2.2 Environment

The foundation of the environment is completely modelled in the Unity editor, using simple 3D objects like planes and cubes. Furthermore, the simulation-area is not enclosed by walls.

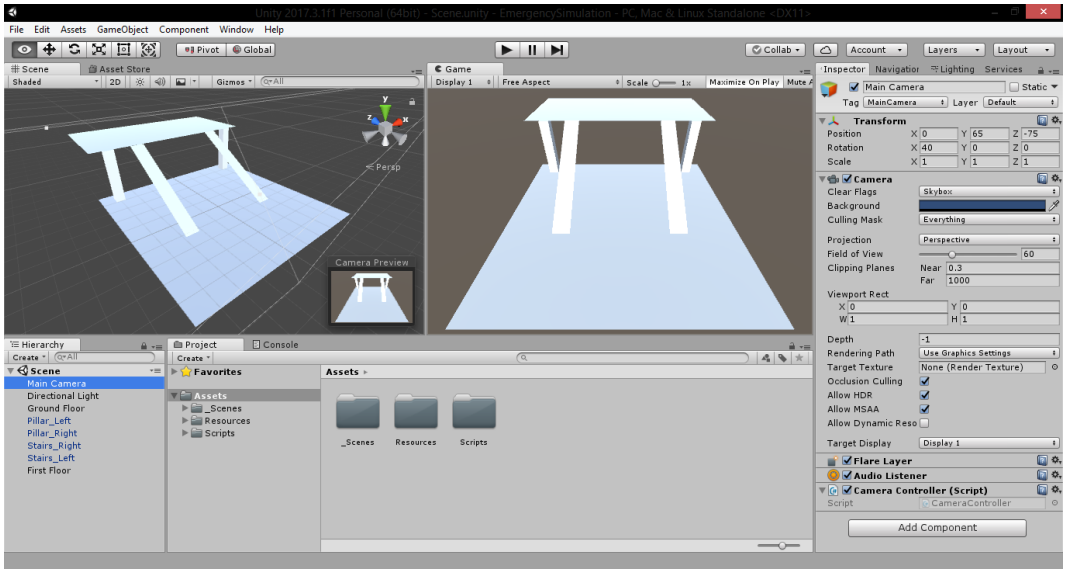



*Figure 15:* Unity editor after the base-environment is implemented

All walkable areas are implemented as planes, since they are easy to place and adjust. Decorative elements like pillars are just arranged for visual purposes and realized with cubes.

*Table 5:* 3D objects used to create the base-environment

| Name | 3D Object | Description |
|---|---|---|
| Ground Floor | Plane | Largest area for agent-activity (walkable) |
| Pillar Left | Cube | Decorative element (not walkable) |
| Pillar Right | Cube | Decorative element (not walkable) |
| Stairs Left | Plane | Connects the two floors on the left side (walkable) |
| Stairs Right | Plane | Connects the two floors on the right side (walkable) |
| First Floor | Plane | Additional small area for agent-activity (walkable) |

Unity´s navigation system allows to determine areas as “walkable” or “not walkable”, which are specifically defined for agents to move intelligently around the game world by applying navigation meshes that are created automatically from the scene geometry. The settings for the navigation system can be found in the tab “Navigation”, additionally the static navigation meshes must be baked, before it will be applied.

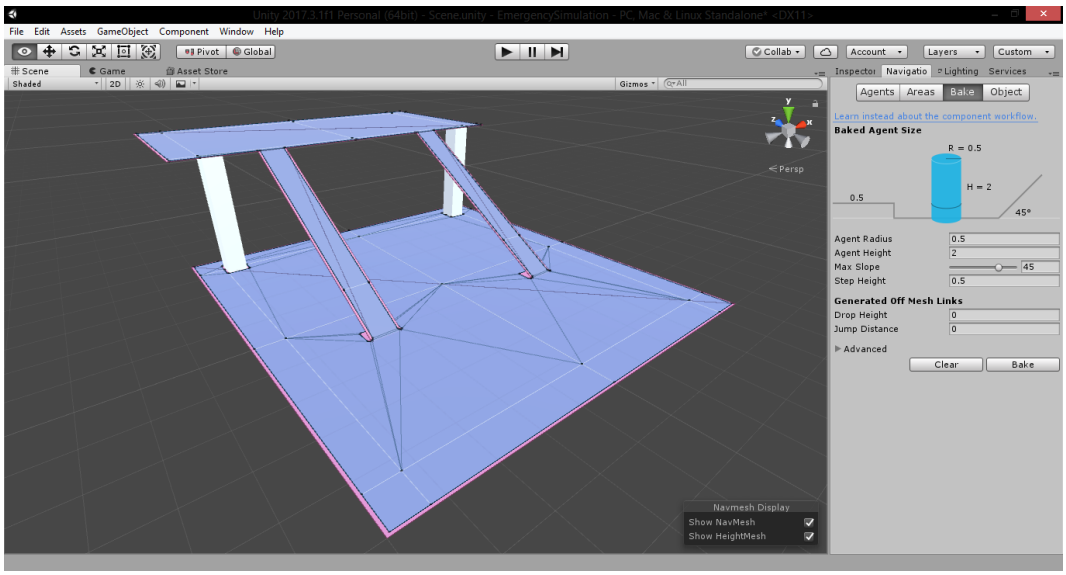



*Figure 16:* Scene view of the environment with applied navigation meshes

Walkable areas are visualized in blue, contrary to not walkable areas, which are displayed in purple. Since the default settings of the navigation-baking process are fine for now, the foundation of the environment is completed. Essential settings for the baking process include:

- **Agent Radius**: How close to the edge navigation meshed will be placed.
- **Agent Height**: How much vertical clearance space must exist.
- **Max Slope**: Maximum slope the agents can walk up to.
- **Step Height**: Maximum height the agents can climb over.

### 5.2.3 3D Models

Further models are required to fill the empty building. Those models include agents, entries/exits and properties, being seats and shops for now.




*Figure 17:* Essential models for the simulation

After implementing the initial environment elements and applying the navigation-properties, the entire navigation meshes are updated by re-baking the whole scene.

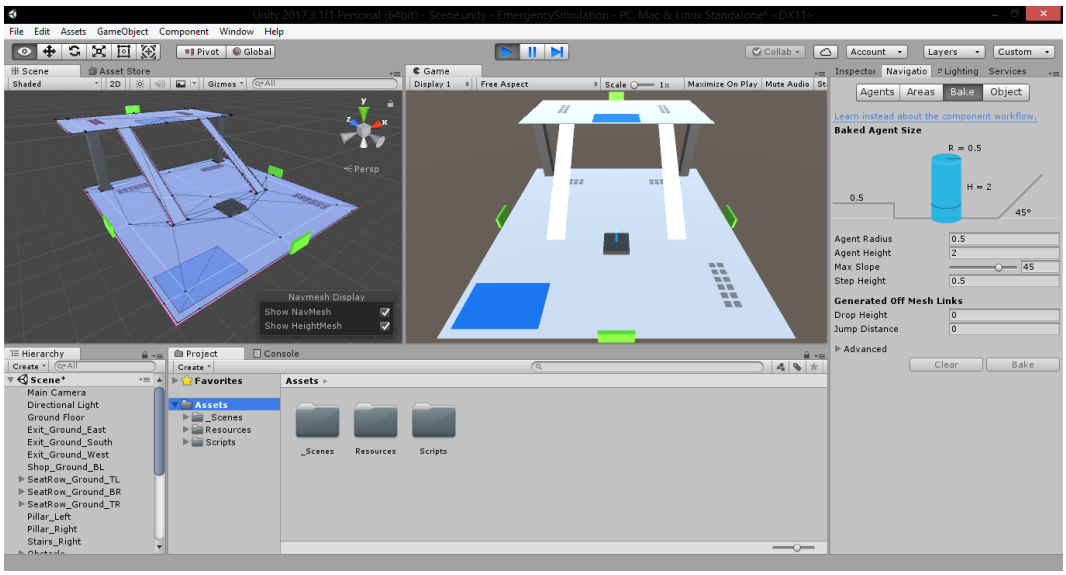


*Figure 18:* Environment after all essential models are placed

There are no agents in the initial scene, because they will be spawned during the run-time by the manager script. An additional obstacle, representing a fountain, was placed in the middle of the scene to increase the dynamic and the appearance of the simulation.

### 5.2.4 Logic and Behaviour-Model

#### *5.2.4.1 Initializer*

The first created script is the Initializer, which stores all initially required values centralized and contributes them to all classes. Moreover, all private fields in the Initializer are decorated with the inspector-attribute called "`[SerializeField]`". This attribute ensures that variables are set- and changeable in the inspector, allowing the adjustment of values on-the-fly, without accessing the script. To make the private fields accessible by other classes, a public property, containing a getter is added.

*Listing 1:* Definition and accessibility of variables in the Initializer

```
[SerializeField] private float minAgentSpawnTime;
[SerializeField] private float maxAgentSpawnTime;

public float AgentSpawnTime
{
    get { return Random.Range(minAgentSpawnTime, maxAgentSpawnTime); }
}
```

The benefit of the public getter is that a random value between the two related variables can be straightforwardly returned, leading to the desired stochastic behaviour of the simulation. This is because the other classes don´t need to know the minimum and the maximum value of the range, they only need a random value within that range.

Further, the Initializer is implemented with the singleton design pattern, meaning that there can only be one instance of the class. The singleton-pattern ensures that classes which require access to the Initializer always access the same instance every time.

*Listing 2:* Approach to implement the Initializer with a singleton pattern

```
public class Initializer : MonoBehaviour
{
    private static Initializer instance;

    public static Initializer GetInstance()
    {
        if (instance == null)
            Debug.LogError("No Initializer was created!");

        return instance;
    }

    void Awake()
    {
        if (instance != null)
            Debug.LogError("Only one instance of Initializer is
allowed!");

        instance = this;
    }
}
```

In addition to store initial values, the Initializer also has lists of all initial gameobjects of the scene, such as floors, exits, properties, due to the reason that their number isn´t changing during the simulation. Likewise, supportive methods to return a random gameobject from an associated list are provided, for instance GetRandomExitPos() (see Listing 3).

*Listing 3:* Method to return a random exit from the list

```
public GameObject GetRandomExitGameobject()
{
    return exits[Random.Range(0, exits.Count)];
}
```

It is also possible to change the method-type to Vector3 and directly return the position of the object, but by returning the complete gameobject, components can be easily accessed by other classes.

### *5.2.4.2 Manager*

The Manager has much in common with the Initializer like the implementation as a singleton and the holding of globally accessible methods and lists, except it stores values and manages events during the run-time.

A coroutine in the Manager instantiates agents in random intervals at one of the given entries. Those intervals are defined by the AgentSpawnTime (See Listing 1) and the random exit is returned by the GetRandomExitGameobject() function (see Listing 3), declared in the Initializer. Coroutines are essentially functions declared with a return type of IEnumerator and with a yield return statement in the function. The yield return line is the point at which execution will pause and be resumed. To run the coroutine, the method StartCoroutine() is used.

*Listing 4:* Spawning coroutine for agents

```
private IEnumerator SpawnCoroutine()
{
    while (!catastropheActive)
    {
        GameObject entry = init.GetRandomExitGameobject();
        GameObject agentClone = Instantiate(init.AgentPrefab,
entry.transform.position, Quaternion.identity);

        if (agentClone.GetComponent<Human>() != null)
        {
            Human h = agentClone.GetComponent<Human>();
            h.EntryObject = entry;
        }

        agents.Add(agentClone);

        yield return new WaitForSeconds(init.AgentSpawnTime);
    }
}
```

This coroutine will be repeated at defined intervals as long as there is no active catastrophe. Furthermore, the instantiated agent is stored in the intended agent-list and the Manager accesses each agent´s Human component to save the entry-gameobject from which the agent entered the simulation environment. The Human component is the script that contains the human behaviour-model for the agent, which is explained in the next section.

The second essential task of the Manager, besides instantiating agents, is to keep track of the running time and activate a catastrophe after a certain amount of time. The Manager also determines whether the agents are all evacuated, or the panic is still ongoing.

#### *5.2.4.3 Human*

*Human* is the central class of this implementation. Even though realistic behaving entities in simulations are often referred to as agents, in this implementation the behaviour-logic is stored in a class called Human.

The Human class has two different types of personal variables. The first type includes values defined in the Initializer which are then initialized in the Human class like name, age, weight and burden. The second type of values is calculated trough those initial variables, such as speed and reaction time. While speed is determined by age, weight and burden, the reaction time is only based on age. Additionally, each agent stores information about the entry it used to enter the environment, the current destination object and the property it is currently using.

Therefore, the essential elements of an agent are:

- Name: The name of each agent is a random string provided by a name-array which contains 5493 different names. The name-array is created by a static class called FileReader.
- Age: Random integer value between 10 and 90 years.
- Weight: Random float value between 40 and 120 kg.
- Burden: Random float value between 0 and 20 kg.
- StatesEnum: Contains the five different states TIRED, HUNGRY, BUSY, IDLE and PANIC.
- Speed: Calculated float value.
- Reaction time: Calculated float value.

In addition to the Human component, the agent also has a NavMeshAgent component and a Collider component. While the NavMeshAgent allows to navigate the scene using the previously implemented navigation meshes on the environment, the Collider component enables to detect collisions with other gameobjects when moving within the environment. Through combining these two components the agent is able to move intelligently to a given destination and furthermore recognizes when the destination is triggered.

The behaviour-model is achieved through states. Every agent gets a state from the Human class, which then starts the state-loop (see Figure 19) until whether the agent leaves or the Manager starts a catastrophe.

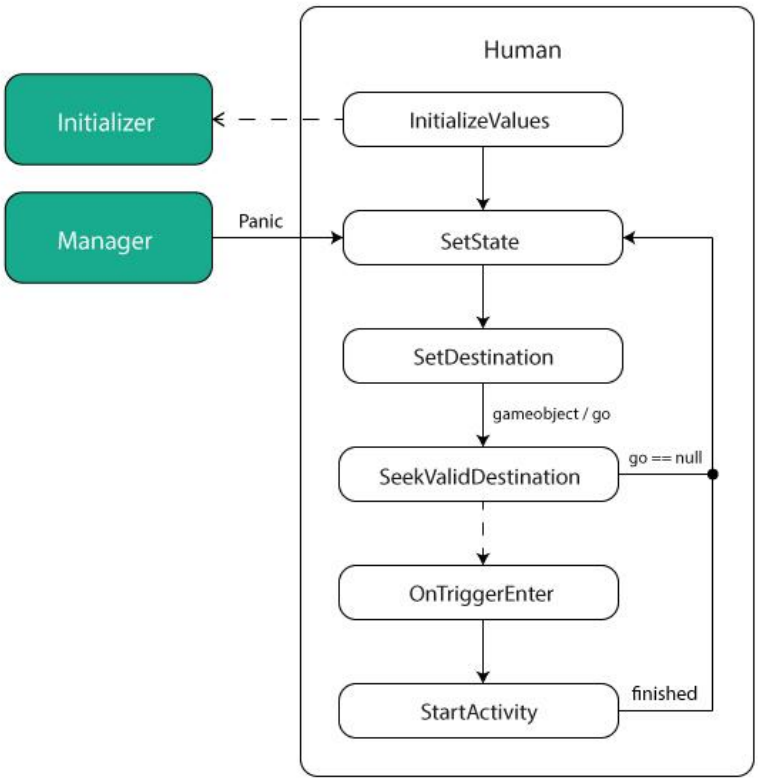



*Figure 19:* State-loop of the agents

Firstly, the InitializeValues function initializes all required variables from the Initializer and calls the SetState method to give the agent an initial state. After the random state from a defined StatesEnum is set, the method SetDestination is started, which hands over a gameobject to the SeekValidDestiation method, according to the current state. SeekValidDestiation then checks if the given gameobject is still available or null. For instance, if a Property is already occupied by the maximum number of Human objects, the method would return null and the loop would restart with a new state that is not the same as the old state. In case the destination is not null, the agent will move to the destination-object. As soon as the agent arrives the destination, a collision is detected which will then enable the StartActivity function and restart the loop once the activity is finished.

Additionally, the manager can break the state-loop by using the SetPanic method when a catastrophe is enabled. This will stop all current activities after a short amount of time, defined by the agent´s reaction time and sets the states of all agents to panic.

#### *5.2.4.4 Property*

Property is an abstract parent class for classes such as Seat and Shop. It provides the logic to determine by how many agents it´s currently used and if there is still capacity left for more users. The Property class stores a list of all Human objects that are currently using it. By counting the elements within the user-list and checking if the number is lower than the maximal capacity provided by the Initializer (see Listing 5).

*Listing 5:* Check if the Property object is completely occupied

```
public bool IsFull()
{
    if (users.Count < capacity)
        return false;

    return true;
}
```

### 5.2.5 User Interface

The whole UI consists out of three different panels:

- Simulation Panel: Shows the running time and current number of agents in the scene.
- Human Panel: Displays all relevant values of an agent when clicked, such as name, age, weight, burden and current state.
- Evacuation Panel: Shows the amount of time needed to evacuate the building and is only visible during a catastrophe.

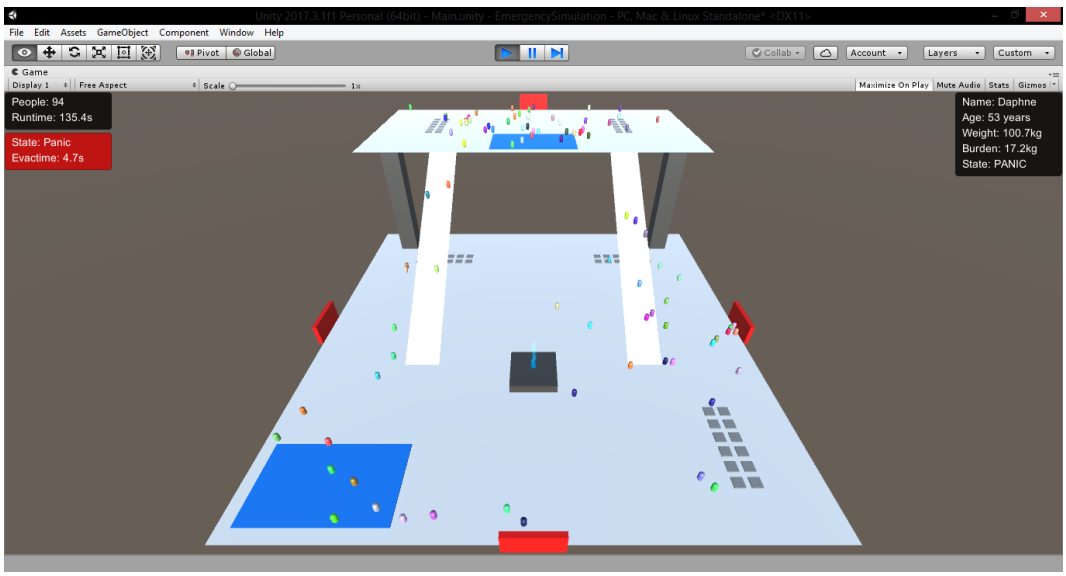



*Figure 20:* Finished simulation during a catastrophe

## 5.3 Problems and Solutions

The following section presents problems that occurred during the implementation phase and describes the applied solution.

### 5.3.1 Performance

A leading problem during the development of crowd simulations is the performance due to the high number of active agents in the scene. The approach in this work was to avoid Unity´s Update method as much as possible, since it is called every frame. Instead of checking positions every frame, the agents use a trigger event system to check if the destination has been arrived. To achieve this, all included object must have a Collider component, which is set to trigger.

### 5.3.2 Collision Detection

During the early testing of the simulation, the collision detection of the trigger event system often failed. Therefore, the agents never arrived their destination object, even though the Collider components were correctly adjusted. The problem was that the agents floated on the navigation mesh and the objects never collided because of the elevation. To prevent the floating behaviour of the agents, the option “Height Mesh” in the advanced settings of the Navigation tab has to be checked and the navigation mesh has to be baked again.

### 5.3.3 Idle Activity

The idle activity was the most problematic activity to implement, because there is no idle object in the scene and therefore no Collider to check if the agent has arrived at its destination. To keep the current logic with the trigger event system, an empty gameobject with a Collider was created. Since it makes no sense to initially place the idle objects at random spots, the agent instantiates its own idle object and destroys it when the idle activity is completed. Additionally, the child class Idle was created which extends the abstract class Property, because the idle activity was already fairly similar to the other property activities.

Another problem with the idle activity was the fact that the idle object is randomly instantiated within the environment. Consequently, the idle object could be created in a non-walkable destination, meaning the agent will never be able to reach the object. Moreover, the idle object could also be instantiated over a seat, which means that there are two activity objects at the same position with only space for one agent. The answer to this problem was to extend the Idle class and add a function that checks if is overlapping with an obstacle or property. The agent instantiates new idle objects and destroys old ones until no overlap is occurring.

## 5.4 Testing and Results

There will be two major tests with developed simulation to prove that environmental changes are directly affecting the amount of time needed for a successful evacuation. In both tests, the catastrophe will occur after exactly five minutes. Important findings include elements like run-time, highest amount of agent, evacuation-time and problems.

### 5.4.1 First Test

The first testing environment will have two entries and one stairway.

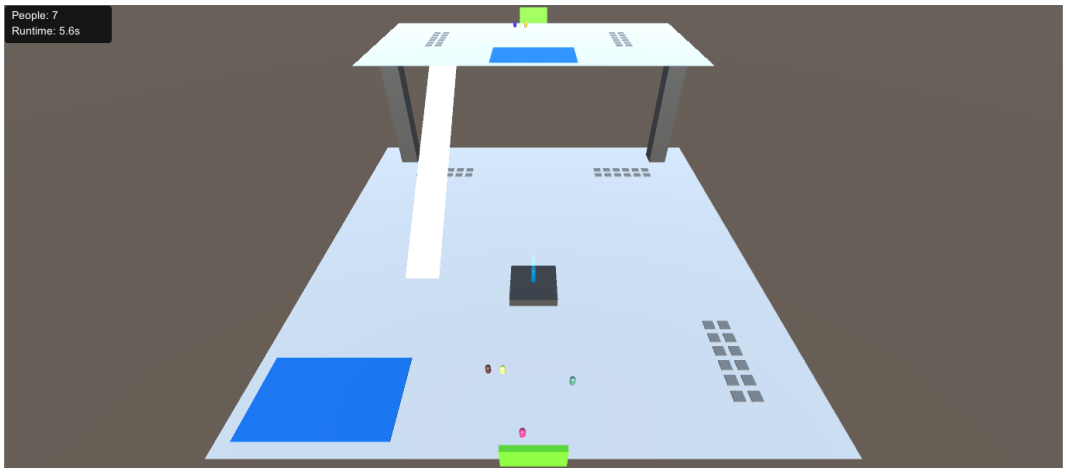



*Figure 21:* Scenario of the first simulation testing

#### *5.4.1.1 Traffic on stairway*

Since there was only one stairway, the traffic between the ground floor and the first floor was so concentrated that it led to crowding.

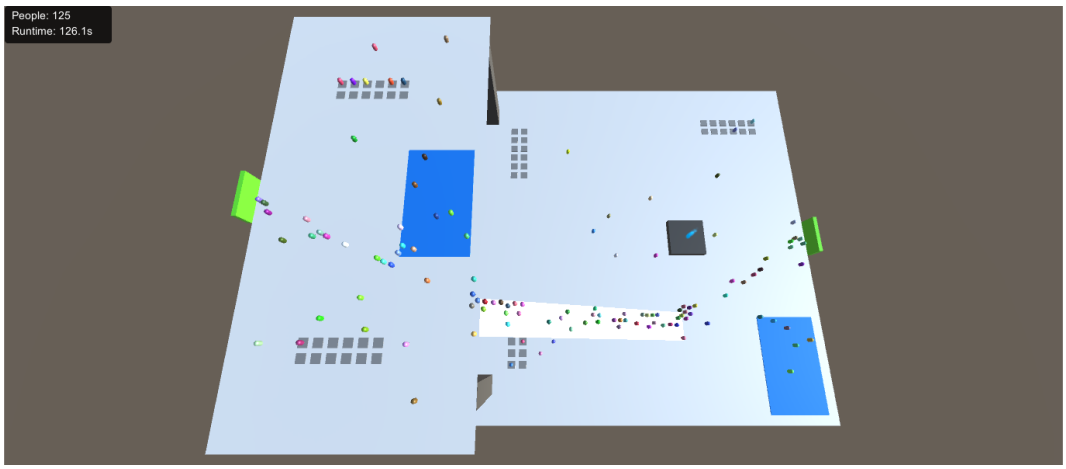



*Figure 22:* High traffic on the stairway in the first test

#### *5.4.1.2 Crowding at exits*

The low amount of exits which were placed far from each other, led to crowding during the evacuation.

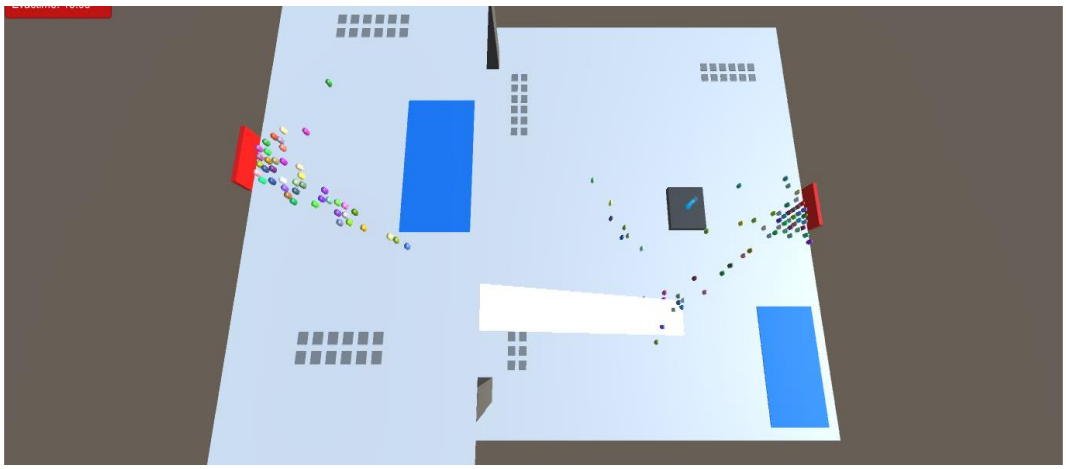


*Figure 23:* Problematic crowding at the exits in the first test

#### *5.4.1.3 Results*

*Table 6:* Results of the first simulation testing

| Run-time | Maximal Agents | Evacuation-time | Problems |
|---|---|---|---|
| 363.5s | 179 | 63.5s | 2 |

### 5.4.2 Second Test

The second testing environment will have four entries and two stairways.

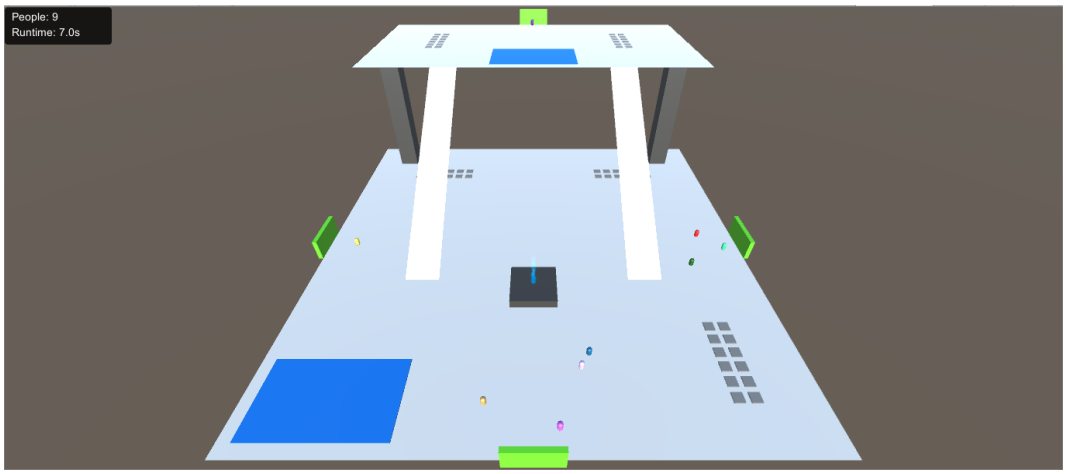



*Figure 24:* Scenario of the second simulation testing

#### *5.4.2.1 Crowding at exits*

Nevertheless the evacuation-time was far better in this testing scenario, there was still some crowding at the exits, especially on the first floor.

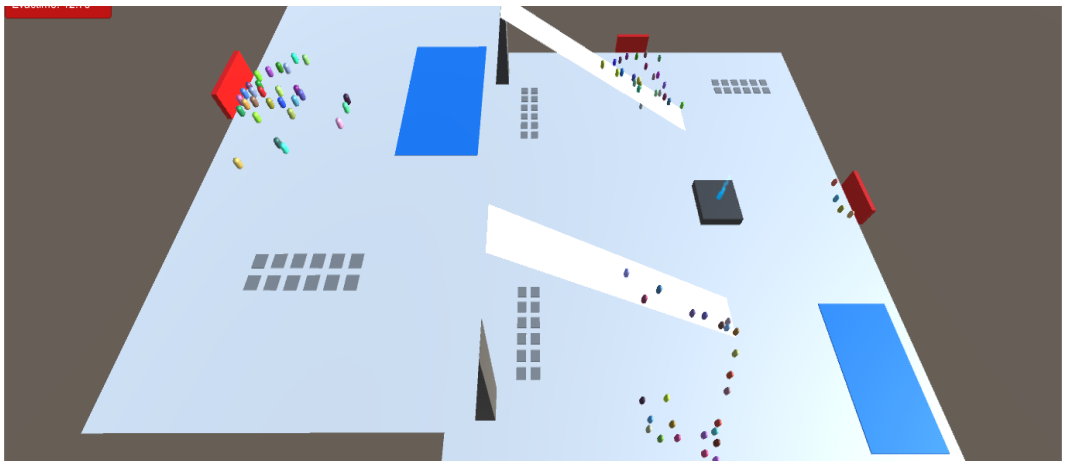


*Figure 25:* Problematic crowding at the exits in the second test

#### *5.4.2.2 Results*

*Table 7:* Results of the second simulation testing

| Run-time | Maximal Agents | Evacuation-time | Problems |
|---|---|---|---|
| 331.8s | 144 | 31.8s | 1 |

Although the modelled environment and the implemented behaviour-model are still very straightforward compared to real-life, the developed simulation is able to provide valuable results. By adding two additional exits and one staircase, the amount of time needed to complete the evacuation is less than the half time required during the first test. The complete testing without documentation took about twelve minutes, including the environmental changes, yet presented crucial differences between the two areas, during the evacuation. This proves the statement that agent-based simulation can support the planning of public buildings and areas in terms of safety and cost-efficiency.

# 6 Conclusion

## 6.1 Conclusion

An agent-based simulation to evaluate virtual environments during an emergency within the terms of the minimum viable simulation has been developed. Humans are represented by coded agents which have similar characteristics defined by variables, such as name, age and weight. They enter the building trough one of the available entries in the scene and receive an initial state that directly relates to an activity. Activities achieve the desired dynamic appearance of the simulation through using a loop where agents start new activities once the current activity is finished.

The software environment for the implementation of the behaviour-model was determined by focusing on specific software frameworks called game engines. Theoretical research within this field and a practical evaluation of different game engines lead to the conclusion of using Unity as the development platform. Using Unity´s integrated navigation system in combination with the implemented behaviour-model, agents are able to move and behave realistically within the abstract environment.

The testing results demonstrated the relevance of crowd simulations with a focus on emergency situations. Even in an abstract environment, the testing showed valuable results by changing essential surrounding elements like entries and stairways. Therefore, the simulation model can be applied in further different settings through changing the environment and adapting the behaviour-model.

## 6.2 Future

Due to the limited resources of this bachelor thesis, there are multiple section which allow room for further improvement.

The current behaviour-model only simulates individual agents and contains no logic for relationships or groups. In the real world, the possibility of meeting a related person randomly is given and therefore social interaction is happening. The implementation of relationships would make the simulation more realistic, since there would be additional influences of how the agents act during an emergency. Few people would leave the evacuation area alone without paying attention to related human beings.

The environment is still just an abstract building with no correlation to a real area found in the world. Through implementing a 3D model of a building that already exists or is currently planned, the results of the simulation would have a much higher value.

Currently there is only one type of catastrophe with no specific characteristics implemented. Diverse catastrophes could lead to different conditions of the agents, such as visual impairment and breathing issues through of smoke. Additionally, the origin of the catastrophe is essential for the agents to plan their evacuation path realistically. Another influential aspect is how agents are able to notice a certain emergency. If the catastrophe is only noticeable acoustically, deaf people will require a higher amount of time to realize the emergency incident.

# References

Choi, B. K., & Kang, D. (2013). *Modeling and simulation of discrete-event systems*. Hoboken, NJ: John Wiley & Sons Inc.

Wainer, G. A. (2017). *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press.

Ross, S. M. (2013). *Simulation (Fifth edition)*. Amsterdam: Academic Press.

Bandyopadhyay, S., & Bhattacharya, R. (2014). *Discrete and Continuous Simulation: Theory and Practice*. CRC Press.

Zeigler, B. P., Praehofer, H., & Kim, T. G. (2000*). Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press.

Thalmann, D., & Musse, S. R. (2012). *Crowd Simulation*. Springer Science & Business Media.

Jaros, M. (2015). *Crowd simulation for virtual environments in Unity*. Wien: Technische Universität Wien

Thorn, A. (2010). *Game Engine Design and Implementation: Foundations of Game Development*. Sudbury, Mass: Jones & Bartlett Publ Inc.

Gregory, J. (2014). *Game Engine Architecture, Second Edition (2nd edition. Revised.)*. Boca Raton: Taylor & Francis Ltd.

Rontsinsky, M. (2012). *Echtzeit-Rendering-Techniken für die Planung und Visualisierung von Architektur-Projekten*. Wien: Technische Universität Wien

Froschauer, J. (2012). *Serious heritage games: playful approaches to address cultural heritage*. Wien: Technische Universität Wien

# List of Figures

Figure 1: Graphical overview of chapter 3 ..... 5
Figure 2: State changes caused by discrete-events over time ..... 6
Figure 3: Continuous changes of a state variable over time ..... 7
Figure 4: Input and output examples of stochastic and deterministic simulations . 8
Figure 5: Demonstration a simulation in a train station ..... 9
Figure 6: Attributes of an individual agent in a crowd simulation ..... 10
Figure 7: Graphical overview of chapter 4 ..... 11
Figure 8: Relationship of the parts of game development ..... 12
Figure 9: Relevant managers of a modern game engine and their function ..... 13
Figure 10: Graphical overview of chapter 5 ..... 24
Figure 11: Relationship between agents and activity-objects ..... 25
Figure 12: Relationship of the essential classes ..... 25
Figure 13: Model of a simple simulation environment ..... 26
Figure 14: Interface of the Unity editor with customized layout ..... 27
Figure 15: Unity editor after the base-environment is implemented ..... 28
Figure 16: Scene view of the environment with applied navigation meshes ..... 29
Figure 17: Essential models for the simulation ..... 29
Figure 18: Environment after all essential models are placed ..... 30
Figure 19: State-loop of the agents ..... 34
Figure 20: Finished simulation during a catastrophe ..... 35
Figure 21: Scenario of the first simulation testing ..... 37
Figure 22: High traffic on the stairway in the first test ..... 37
Figure 23: Problematic crowding at the exits in the first test ..... 38

Figure 24: Scenario of the second simulation testing ..... 38

Figure 25: Problematic crowding at the exits in the second test ..... 39

# List of Tables

Table 1: Media file formats managed by game engines ..... 14

Table 2: APIs for rendering managers ..... 14

Table 3: Accessibility of all covered game engine ..... 22

Table 4: Rating of game engines according to the current state-of-the-art ..... 23

Table 5: 3D objects used to create the base-environment ..... 28

Table 6: Results of the first simulation testing ..... 38

Table 7: Results of the second simulation testing ..... 39

# List of Listings

Listing 1: Definition and accessibility of variables in the Initializer ..... 30

Listing 2: Approach to implement the Initializer with a singleton pattern ..... 31

Listing 3: Method to return a random exit from the list ..... 31

Listing 4: Spawning coroutine for agents ..... 32

Listing 5: Check if the Property object is completely occupied ..... 35