UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Meelis Perli

Delta Building Visualization – Agent Logic

Bachelor's Thesis (9 ECTS)

Supervisor: Raimond-Hendrik Tunnel, MSc

Tartu 2019

Delta Building Visualization – Agent Logic

Abstract:

The work in this Bachelor's thesis is a continuation of the Delta Building Visualization project. Firstly, the initial state of source code and its refactoring is described. The main part of this thesis is about the implementation of new agent logic. The new agent logic was implemented to make the agents behavior more realistic and to improve the performance of the visualization. It was done by implementing two layered pathfinding, modifying the precalculated paths and agent grouping. Lastly, the testing of the performance and the stability of the project is described.

Keywords:

Visualization, pathfinding, agent behavior, Unity, computer graphics, multi-agent system, optimization

CERCS: P170 Computer science, numerical analysis, systems, control

Delta õppehoone visualisatsioon – agentide loogika

Lühikokkuvõte:

Käesolev bakalaureusetöö kirjeldab Delta õppehoone visualisatsiooni edasiarendust. Töös esmalt kirjeldatakse lähtekoodi algseisu ja selle parandusi. Töö põhiosas antakse töö käigus implementeeritud agentide loogika kirjeldus. Uus agentide loogika implementeeriti, et muuta agentide käitumist realistlikumaks ja parandada visualisatsiooni jõudlust. See saavutati kahekihilise rajaleidmise algoritmi implementeerimisega, eelnevalt arvutatud radade mõjutamisega ja agentide rühmitamisega. Töö lõpus kirjeldatakse arendatud rakenduse jõudlust ja töökindlust.

Võtmesõnad:

Visualisatsioon, rajaleidmine, agentide käitumine, Unity, arvutigraafika**,** mitme agendi süsteem, optimeerimine

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

The Delta Building Visualization (DBV) project, as the name describes, is a visualization of the Delta building, the building in 2020 will house some of the University of Tartu's institutes, including the Institute of Computer Science. The visualization is created to motivate students in the building, improve the environment in a fun way [1], aid the spectators of this visualization to navigate and see what is going on in the building [2]. It is not just a display of the building's floor plan, but a real-time 3D virtual environment. There are humanoid figures, who will be referred to as *agents* in this thesis, but *actors* in the code. These agents mimic students and educators by doing different emotes and taking part in classes according to the real-life data. The visualization also includes a real-time timetable and every room has a clearly visible room number. This helps spectators to see what classes are in which rooms.

The project was started in 2017/2018 by Andrei Voitenko and Aleksander Nikolajev. During their BSc theses they managed to do a lot, but there are many things that still need to be done or improved. This year the project is continued in this thesis and in Einar Linde's and Daniel Kütt's BSc theses. The main goal of this thesis is to improve the agents. Linde improved the lighting in the visualization and created new visual effect [3]. Kütt created an admin tool that makes it easier to control and test what is happening in the visualization [4].

Chapter 2 gives an overview of the prior state of the code and its refactoring. The latter makes the implementation of new features easier. All three theses started with refactoring the previous code. Here the refactoring was mainly focused on the agents because these included the features improved during this thesis. Timetable classes were also refactored, because initially the plan was to finish the timetable integration, that Voitenko started in his thesis [2]. It was not done in this thesis, because the Study Information System 2.0 developers could not find the time to implement the necessary parts of the interface on their side.

Chapter 3 describes the methods used in this thesis to optimize and make the agents in the visualization behave more naturally. Initially they moved in single files, could move through each other, in some cases teleported to their seats and had some other issues as well. The improvements include modifying the precalculated paths, adding a new higher-level navigation system and making some agents move in groups.

Chapter 4 covers the testing of the new functionality. The frame times and memory usage in the DBV are measured on a similar computer as used in previous theses and compared to the results in Nikolajev's thesis [1]. The visualization was also shown to some viewers, who evaluated the new features.

Appendix I contains the Glossary. Appendix II contains the build of the project, measurements, Python programs, that were used to convert the measurements to usable values, and other files.

The reader is expected to know the terminology covered by the Bachelor's program of Computer Science. New terms introduced in this thesis are covered in the Glossary.

2 Legacy Code

When starting a new software project, it is often difficult to foresee how the project's code structure will turn out. Usually there will be situations where there is a need to quickly implement a new feature or something is left for the last moment. In these situations, there often is not enough time to think about the code design and as a result the code might rot[1]. A way to combat this problem is to find and fix the code smells[2]. Well written code usually will not make it run faster, but it will make it easier to read and implement new features [5].

Code smells existed in the DBV project too, so it was important to fix those before adding new features. Firstly, the structure of the project's file tree had to be improved. The previous developers had separated their work by creating folders that were named after themselves. It can be useful when you want to show what you did but makes the project structure more confusing for other developers (see Figure 1).

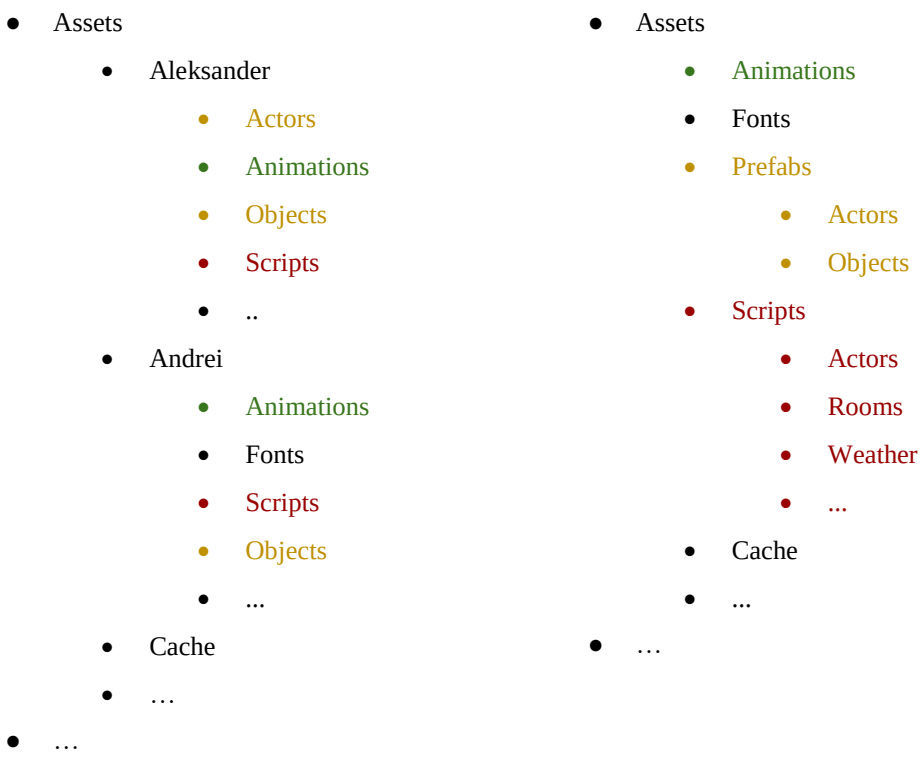

Figure 1. Initial file structure (left) and the refactored file structure (right). Colors, except black illustrate similar folders

The previous developers had a similar file structure inside their individual folders. Thus, the files could be and were easily merged together to a single top-level namespace. After that some files still required further grouping, so they would be easier to find. For example, the

[1] https://en.wikipedia.org/wiki/Software_rot

[2] https://en.wikipedia.org/wiki/Code_smell

script files required further grouping. A lot of files were already grouped by the semantics of their use, so the files that do similar things would be in one folder (see Figure 1).

While relocating files, first of the code smells caused an error. It was due to the *timetable.json* file location had been hardcoded and its location had changed. This was fixed by creating a new folder called *Data* for files that hold data and changing file paths in the scripts.

The next subchapters cover the initial state of the code in the classes that are relevant to this thesis and fixes to the code smells in the said classes. Subchapter 2.3 also covers the initial state of the agent behavior.

2.1 Timetable

The most common indicator of bad code is duplicated code [5]. There was a lot of it in the *LoadTimetable* class. Most of it was in the form of duplicated *if*-statements. For instance, there was an *if*-statement for every day and for every possible lesson beginning time. A section of the *if*-statements blocks before the refactor:

```
1.  if  (day == “Monday”) {
2.  foreach (Room a in timetable.Monday.Rooms) {
3.    if (CompareTime_8_15 < CompareTime && CompareTime < CompareTime_10_15) {
4.      foreach (Subject sub in a.Subject) {
5.        if (sub.Start == “08:15 UTC+2”) {
6.          setSubject(sub, a);
7.    }}}
8.    if (CompareTime_10_15 < CompareTime && CompareTime < CompareTime_12_15) {
9.      foreach (Subject sub in a.Subject) {
10.       if (sub.Start == “10:15 UTC+2”) {
11.         setSubject(sub, a);
12.   }}}
13.   if (CompareTime_12_15 < CompareTime && CompareTime < CompareTime_14_15) {
14.     …
15.     }
16.   …
17.   }
18. …
19. }
```

To fix this, three new methods were created. Firstly the method `getRoomsOfDay(DateTime day)` returns all the lessons of the current day. The method `getCurrentClassStartTime()`

finds the time when the currently ongoing lessons started. Finally the method `getSubjectAt(string startTime)` combines the results of previous two and finds only the currently ongoing lessons. The following code shows the change.

```
1.  DateTime currentTime = DateTime.Now;
2.  foreach (RoomData room in timetable.getRoomsOfDay(currentTime.DayOfWeek)) {
3.    Subject sub = room.getSubjectAt(getCurrentClassStartTime());
4.    if (sub != null)
5.      setSubjects(sub, room);
6.  }
```

Deciding which course names should be shown in the graphical user interface is also done in the *LoadTimetable* class. That code has the same problem. The course names can be in 2 different languages and for both there is a separate *if*-statement block that shortens the lesson's name if it is longer than 40 symbols. In this state it would be tedious to make changes to this class. The applied fix for this code smell was to create a new method that trims the name and is language-independent. This way it is much easier to change the name's maximum length and to add new languages.

2.2 Agents

One of the main objectives in this thesis was to optimize the agents. Unity has a great built-in tool for detecting performance issues called Profiler[3]. Although it was found to be inaccurate[4] for measuring the real frame time, due to the Profiler's overhead, it is still great for finding performance bottlenecks. Using this tool, it turned out that all the moving agents' paths were updated after every few frames. In addition to that, the agents also took more resources when checking if they are near a door or which floor they are on. Optimizing these made the frame time faster by a few milliseconds. Chapter 3 will deal with agent behavior and cover these problems in depth, so optimizing these was not a priority while refactoring. Agents get their paths from the *RoomManager* class, which will be covered in the next subchapter.

[3] https://docs.unity3d.com/Manual/Profiler.html

[4] https://answers.unity.com/questions/33369/profiler-fps-vs-stats-fps-vs-timedeltatime.html

2.2.1 RoomManager

RoomManager is a class that calculates, stores and gives agents paths to the rooms, the spawn and the despawn locations. The problem in this class was the use of data structures. A lot of layers in a collection variable makes it difficult to use.

An example of the described problematic variable from the code:

```
1. Dictionary<Transform, Dictionary<Transform, NavMeshPath>> cachepaths;
```

This is a representation of a directed graph. A better way to do this is to create a separate class for the nodes and let the nodes remember their destinations and other info they might need. This allows the creation of different types of nodes and makes pathing more malleable.

The operations on the multi-layered dictionary field variables in this class were difficult to understand, because quite often multiple *for*-loops were used to access or modify the data within these variables. It became clear that the class tried to accomplish too much and thus was bloated. In such a case it is advised to extract a new class [5]. A node class called *Destination* was extracted from *RoomManager*. This class is a superclass to every other class that can be a destination in an agent's path.

The following variables were also problematic:

1. List<Transform> rooms
2. Dictionary<GameObject, List<GameObject>> roomToActors
3. Dictionary<Transform, List<Dictionary<Transform, int>>> roomToSeats

These variables were often used near each other. In such cases Fowler *et al.* suggest extracting a new class [5]. The new class is called *Room*, it holds the data about the chairs in a room and is used to distribute paths within the rooms. Figure 2 illustrates how the newly created *Room* and the *RoomManager* classes are linked.

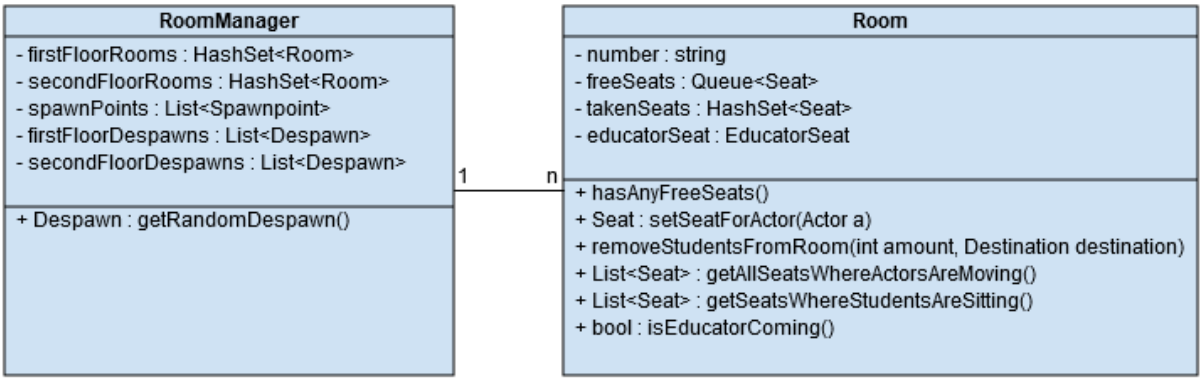

Figure 2. UML class diagram of *RoomManager* and *Room* classes after the refactor.

The previous solution of getting a free seat was to check the list of seats if any are free. Now a queue is used instead to give a free seat and upon freeing the seat it is put back into the queue. The *Seat* class had some other issues as well, which are covered in the next subchapter.

2.2.2 *Seat* and *EducatorSeat*

Both *Seat* and *EducatorSeat* classes are a part of the agents' paths. These classes were used to delete an agent once it sat down on the seat, because the sitting animation is played on the seat instead of on the agent itself. Once the agent had to stand up, a new agent was created to replace the old one. Creating new objects allocates memory but deleting objects does not instantly free it. Once there is a lot of unused allocated memory, a process called *garbage collection* in c# makes it available again. According to Unity's documentation on garbage collection it can cause drops in performance. The proposed fix for this is to pool objects [6]. This way the agents and their memory can be reused. New agents only need to be created when the pool is empty.

If an object is needed, it is taken out of the pool, given its parameter values and made active. Once the object is not needed anymore, it is set inactive and put back into the pool [7]. In the current case it is possible to simplify this object pooling cycle. Instead of putting the agent back to the pool, a reference of the agent is given to the seat. Now once the agent sits on the seat, the agent is set inactive and when the agent stands up, it is set active again. Doing so will also ensure that the agent will have the same position and rotation upon standing up as it had before sitting down.

2.2.3 Actor Pool

Seats were not the only place where agents were deleted. It was also done in the despawn locations, which were called "remove points" in the previous theses. In this case actually pooling the agents was necessary, because the despawn and spawn locations do not share the same coordinates and a spawn location is randomly selected when spawning an agent. For implementing this pool, a queue was used for the same reasons as described in chapter 2.2.1. For an illustration on how the pool works, see Figure 3.

New scripts for spawn and despawn locations were also created. These classes deal with spawning and removing agents and are a part of the agents' path. The spawning and removing part could also be done in the pool script, but in the future, there might be a need to spawn different types of agents and thus it would be easier to spawn them from one script.

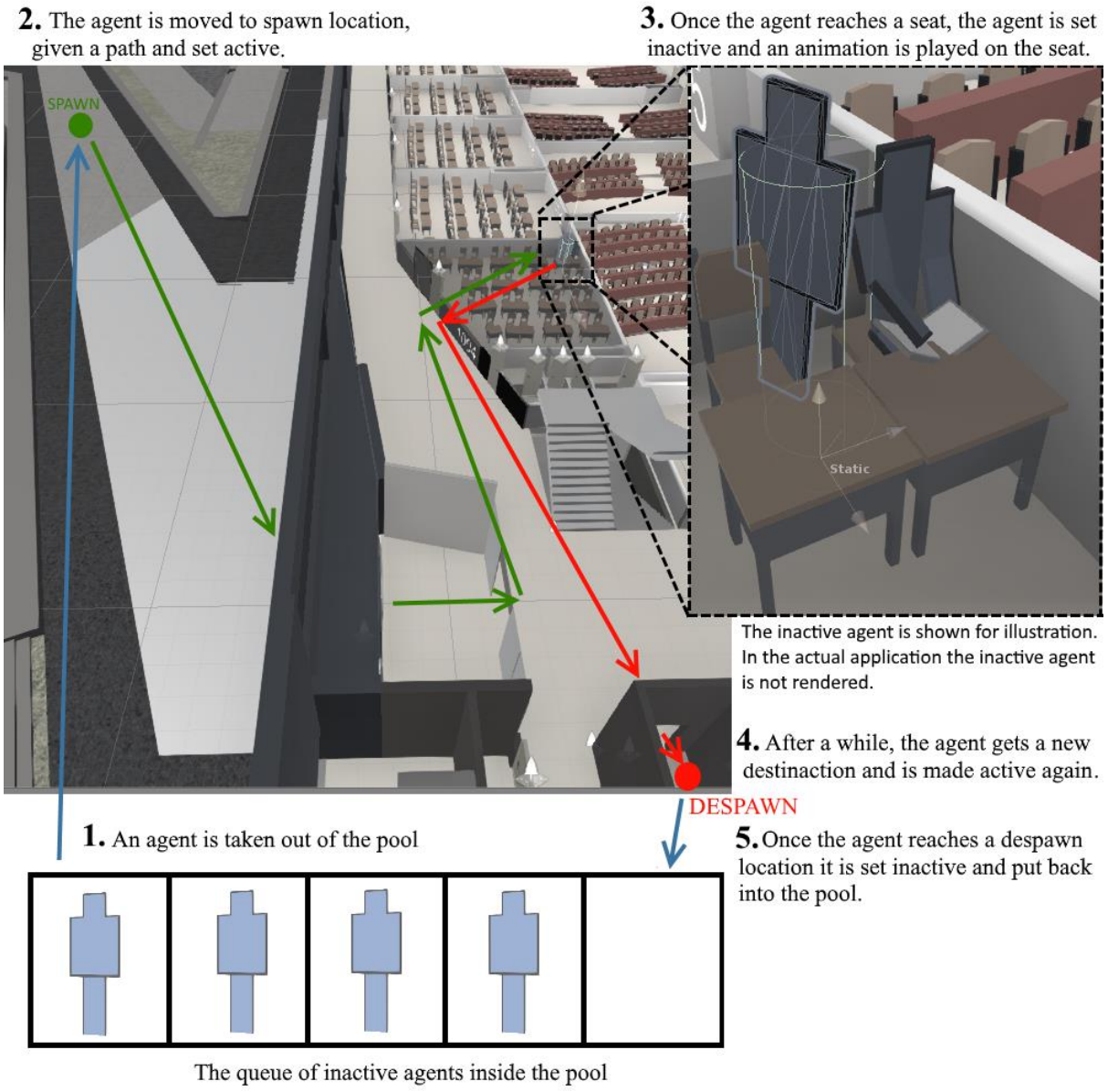

Figure 3. Agent's journey from the pool to a seat and back to the pool.

2.3 Initial State of the Agent Behavior

The path-finding solution made by the previous developers was made using Unity's built-in navigation and path-finding system called NavMesh[5]. The system uses an abstract data structure that represents the area where the agents can move around. That area is precalculated for a static scene. It makes calculating paths within it much quicker, because then there is no need to check for static obstacles anymore.

The initial pathfinding solution has a few problems. First, the agents tended to move in single files resulting in unrealistic crowd patterns (see Figure 4). Agents in the NavMesh have a parameter called *obstacle avoidance radius*, which creates a circular area for every agent on the *navmesh* where the other agents cannot enter. Nikolajev set that radius to 0.1 to increase performance [1]. For reference 0.5 is agent's width at its shoulders. This made agents move really close together. Simply increasing this value would cause excessive loss in performance.

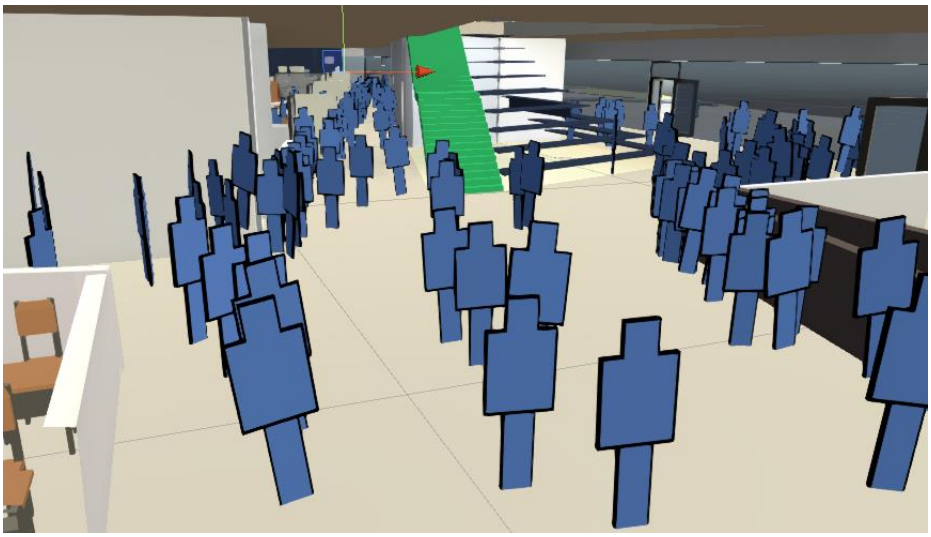

Figure 4. Agents moving in a single file and are too close to each other.

Poor *navmesh* mapping of the scene resulted in weird paths. For instance, there was a place where the agents could move through the stairs (see Figure 5). Furthermore, some areas on the second floor were not walkable due to the walls on the first floor (see Figure 6). Both issues were fixed by remapping the geometry of the scene.

[5] https://docs.unity3d.com/ScriptReference/AI.NavMesh.html

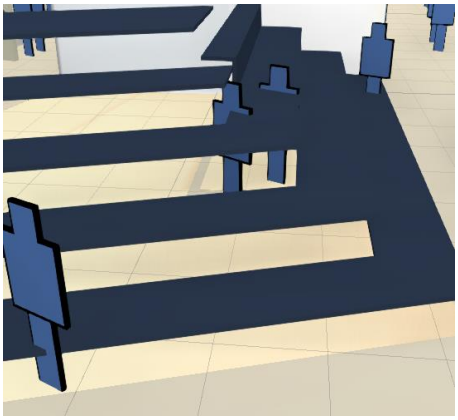

Figure 5. Agents walking through the stairs

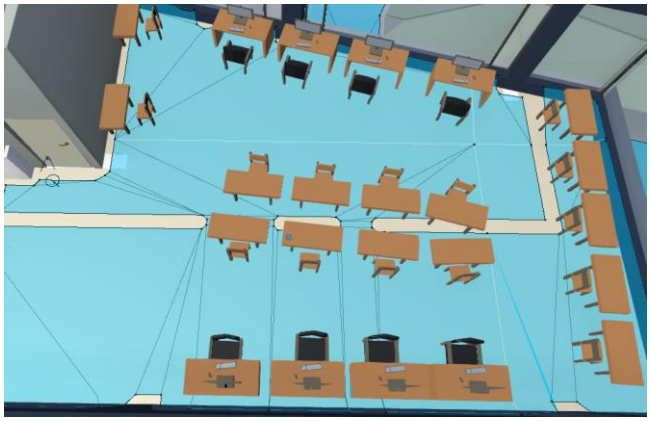

Figure 6. Navmesh on the first floor. Blue areas are walkable.

Agents also perform emotes, so they could mimic real people better. They have designated areas where they can do emotes. Once an agent starts to perform, it is disabled, and a new object is used to display the animation instead. In some cases, agent's size changes due to that (see Figure 7Figure 7). The fix was just to make the emoting agents smaller.

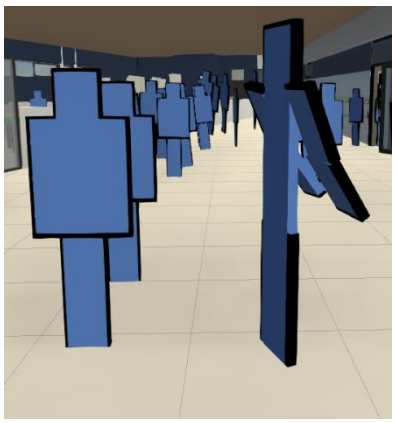

Figure 7. Size differences

The agents moving in a single file issue requires deeper insight and thus the fixes to that problem are described in the next Chapter.

3 The Agent Behavior

The main purpose of the agents in this visualization is to make the visualization seem lively. They do so by mimicking the real students and educators. As described in chapter 2.3, the agents initially had several problems. The previous developers left some suggestions for the future developers regarding improving the agents, who are referred to as actors in their theses and in the code [1]:

1. *Improve the Actor behavior in a way to make the visualization more natural.*
2. *Further improve the performance of the visualization. Different methods, such as grouping Actors together, can potentially improve the visualization.*

The next sub-chapters cover these suggestions. In chapter 3.1 the alternatives to the initial pathfinding solution are discussed and then the implementation of the new pathfinding solution is given. Chapter 3.2 describes how a higher-level navigation system was created using different waypoints. Chapter 3.3 covers the agent groups and chapter 3.4 describes some additional issues that came out while improving the agents.

3.1 Pathfinding

There are a lot of different algorithms to find a path from one location to another [8]. For creating a dynamic multi-agent system[6] (MAS) that can contain up to 2000 members, it is important that the used algorithm is computationally cheap. Furthermore, the agents cannot just take the shortest path, because that can cause the agents to move in unnatural single file formations. The following subchapter covers some of the potential algorithms that could be used for such pathfinding.

3.1.1 Related Algorithms

Many different methods have been used for creating a multi-agent system. Toll *et al.* used real time crowd density information to make agents take less crowded routes. Using crowd density information in the DBV project did not seem reasonable, because Toll *et al.* described that it took them 2 ms to update a path for a single agent [9].

Flow tiles can also be used to create a dynamic MAS [10]. These tiles tell the agents in which direction they should move. The direction could be changed if something is in the

[6] https://en.wikipedia.org/wiki/Multi-agent_system

way or another agent just went in the same direction. It is difficult to implement this if the agents have different destinations and may not go through each other.

Kavraki *et al.* generated a probabilistic roadmap for robot path planning and path modification if a robot was nearing a collision [11]. This method is also described to be computationally cheap. Although this seemed like a plausible method, it was not used in the DBV, because the paths it generates are too jagged. Though the inspiration for the method used in DBV came from their method. The next subchapters describe the methods used in this thesis to improve the pathfinding in the DBV.

3.1.2 Agent Obstacle Avoidance Radius

While refactoring it was discovered that agents' paths were updated every frame. It turned out that every time an agent's destination was not equal to the position it was heading, the agent's path was calculated again. The following if-statement was initially created because sometimes the newly spawned agents did not start to move [1].

```
1. if (navMeshAgent.destination != destination.transform.position)
2.     navMeshAgent.SetDestination(destination.transform.position);
```

The given condition was almost always true, thus reducing the performance. NavMeshAgents have a Boolean variable called *hasPath*, that was used in this thesis to fix the condition. Figure 8 shows the performance difference before and after this change.

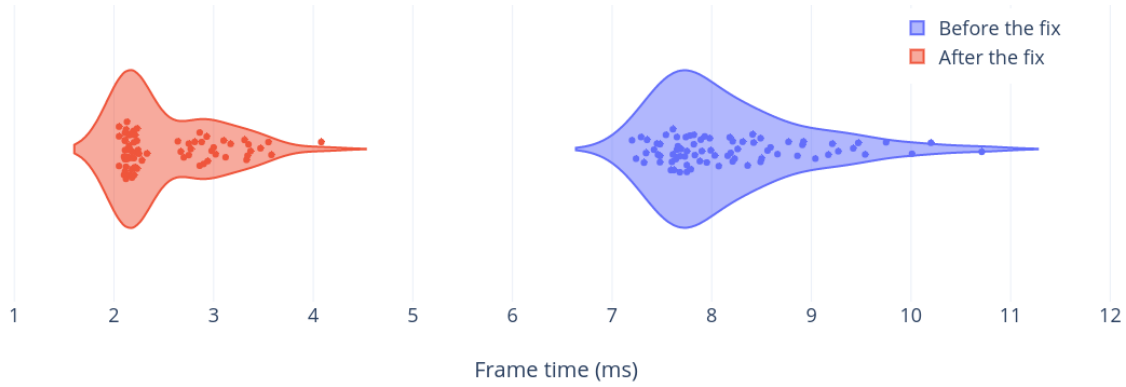

Figure 8. The comparison of an *if-statement*'s execution time before and after fixing it.

This change in performance allowed to increase the NavMeshAgent's obstacle avoidance radius (OAR) to 0.25 and to have better performance than before. The performances of different configurations are shown in Figure 9. OAR makes the agents keep greater distance

from each other. Thus, the agents are less likely to go through each other. Figure 10 shows the difference that higher OAR value has on performance.

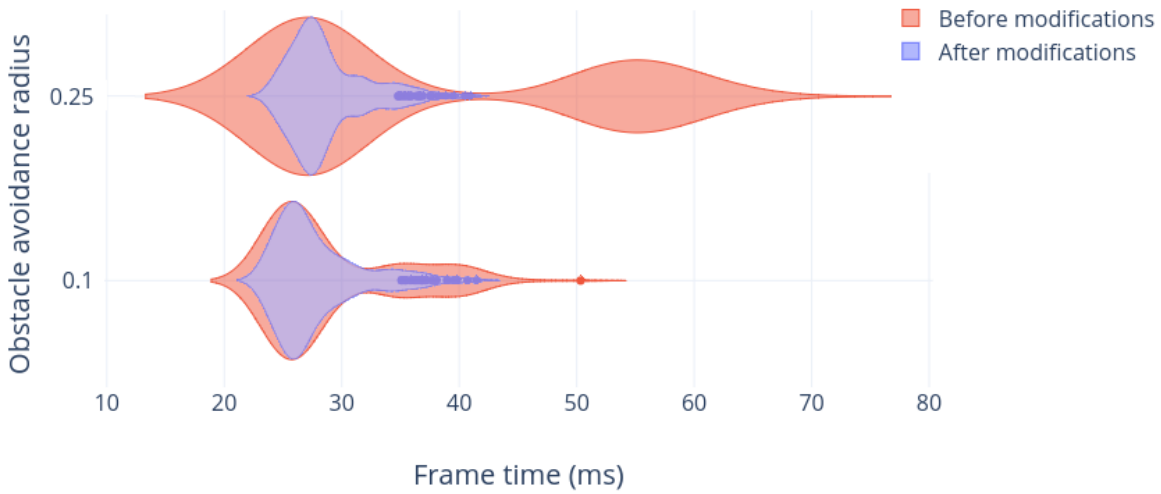

Figure 9. The comparison of total frame time with different obstacle avoidance radii before and after the modifications

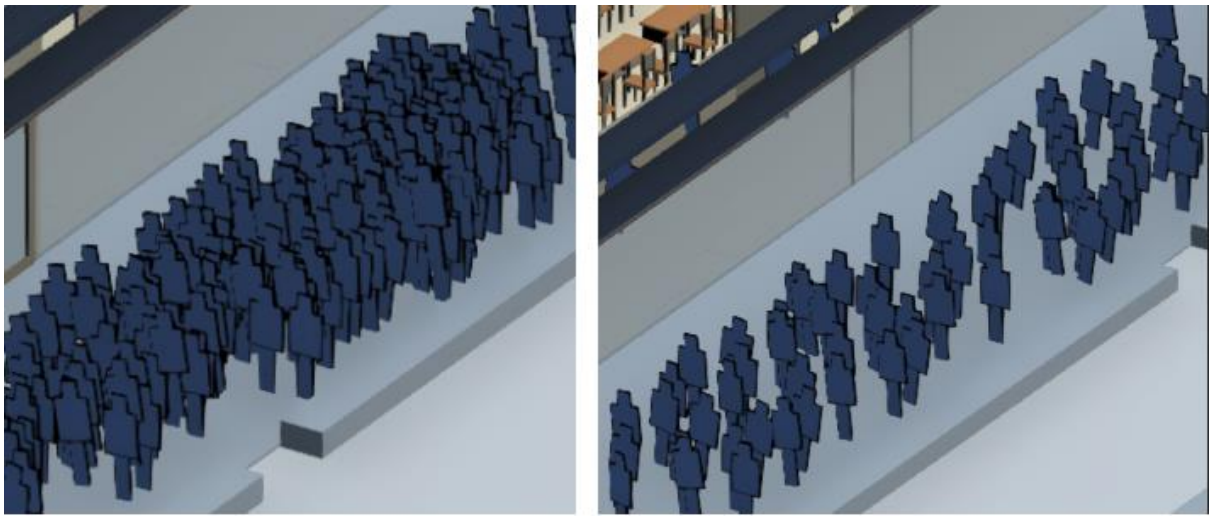

Figure 10. Agents with OAR value 0.1 (right) and with OAR value 0.25 (left).

The previous change made agents move better in crowded areas but did not improve agents elsewhere. The agents still formed single file formations in less populated areas. The next subchapter covers how the precalculated paths were improved.

3.1.3 Modifying the Precalculated Paths

After seeing that Unity's NavMesh performs quite well even with 2000 agents, implementing a new pathfinding system did not seem necessary. For debugging purposes, a debugging tool was created in this thesis. One of its functionalities is to visualize the precalculated paths. The precalculated paths near one of the entrances to the Delta building

are shown as red lines in Figure 11. In some locations the lines seemed to spread out or to join, which indicated that there were a lot of overlapping paths.

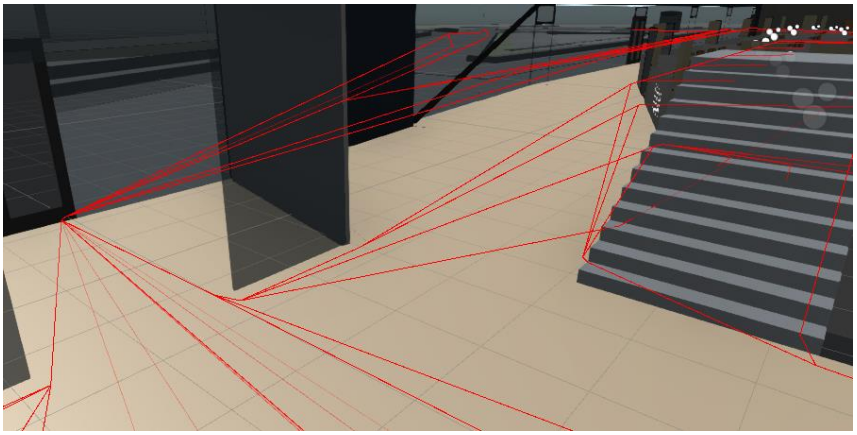

Figure 11. Precalculated NavMeshPaths.

The solution was to spread out the paths. Unity's NavMeshPath[7] class stores path's vertices as a list of 3-dimensional vectors and these can be modified. Instead of implementing an algorithm for calculating the spread-out paths, objects called *path influencers* were created.

Path influencers are objects that move the path's vertices closer or further away from the influencer. A randomly generated number from 0 to 1 is used to spread out the vertices. This number is generated per path, so the paths would not get too jagged. For controlling the distance, the influencers also have maximum and minimum force variables. Both the force variables can have any real number value. Positive values for these variables make the influencer pull the path vertices towards the influencer. Negative values push them away. Each influencer has a maximum radius, so it would not interfere with vertices that are too far away. The maximum radius also affects the distance that the vertices are moved. The effect that the path influencers have on paths can be seen in Figure 12.

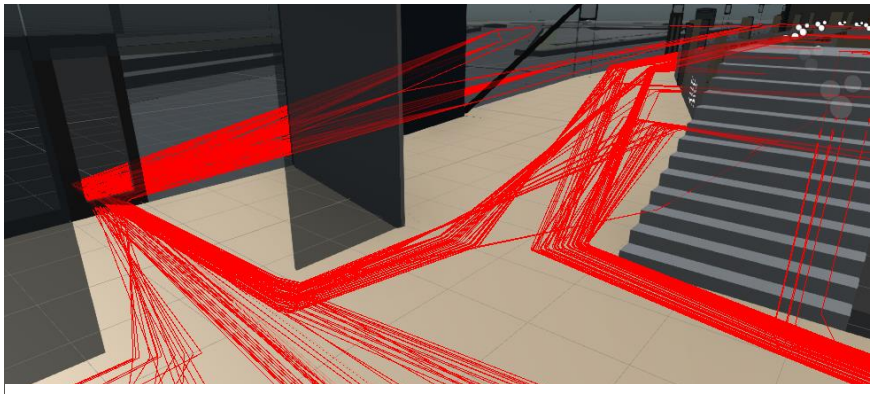

Figure 12. Precalculated paths after using influencers.

[7] https://docs.unity3d.com/ScriptReference/AI.NavMeshPath.html

Figure 13 illustrates how path influencers work. The black ring is the range of the influencer. The red ring illustrates the maximum strength and the blue ring is for the minimum force. B marks the initial location of a path's segment ABC vertex, that is in range of the influencer. Given influencer has a range of 4, a maximum force of 0,5 and minimum force of 0,25. This means that the vertex B is moved to at least point B_{min} and can be moved up to the point B_{max}. The brown area indicates where the affected paths can go. Figure 14 displays a similar influencer, but with negative force values.

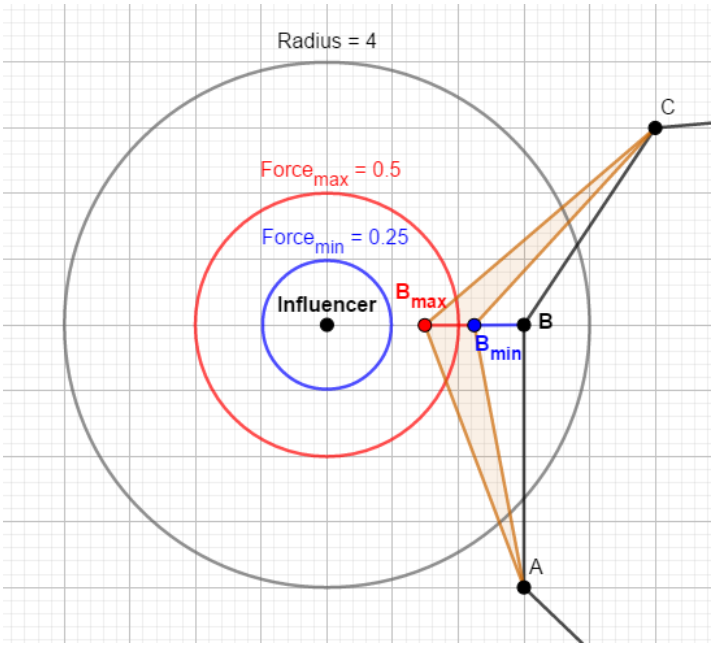

Figure 13. Path influencer with positive force value displacing a vertex

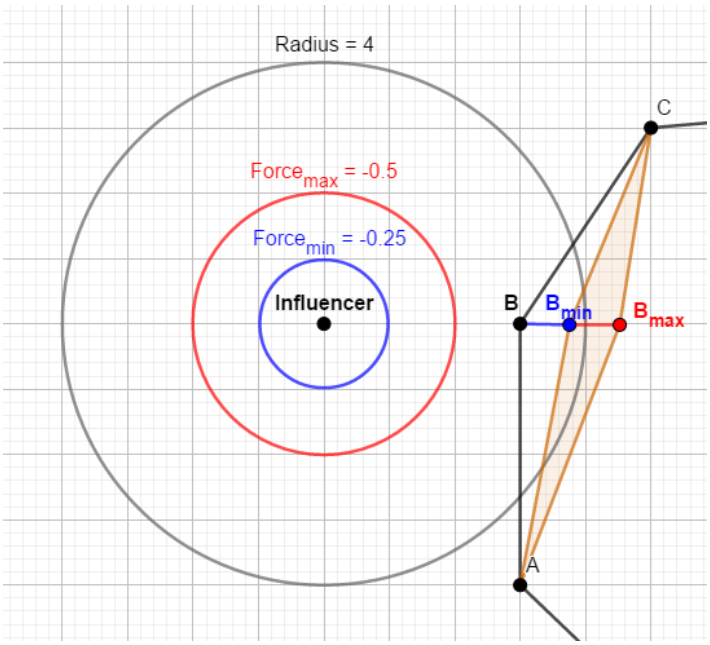

Figure 14. Path influencer with positive force value displacing a vertex

The path influencers were then put everywhere inside the DBV where the paths overlapped. Figure 15 and Figure 16 show the difference that the influencers made on the first floor of the DBV.

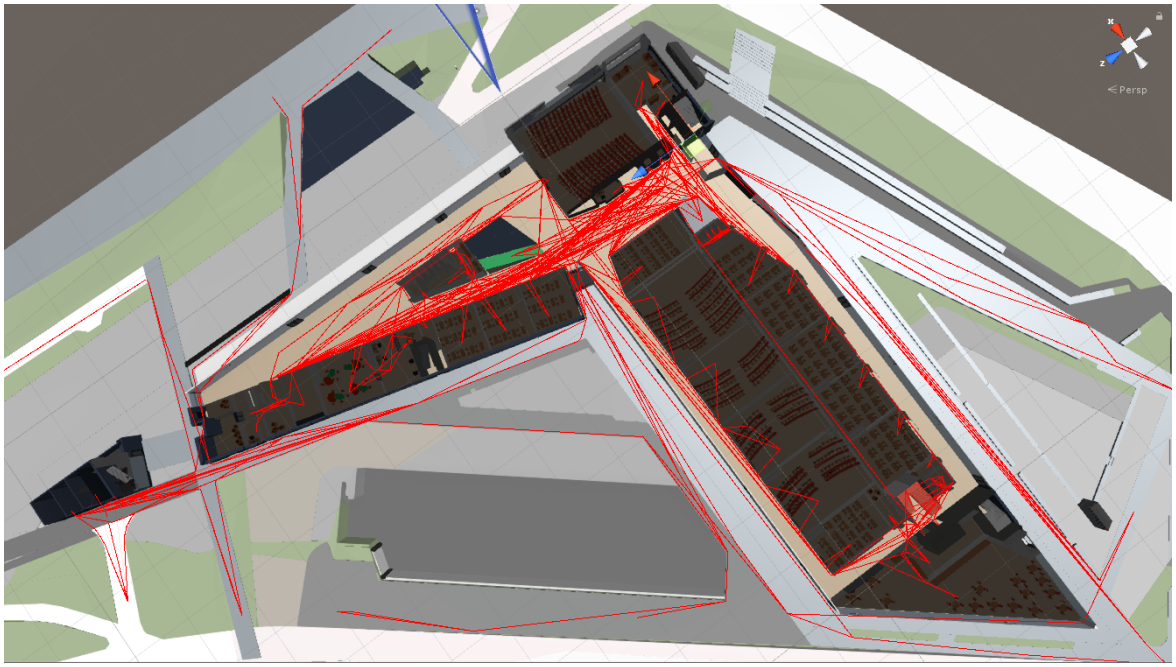

Figure 15. Precalculated paths on the first floor before the influencers.

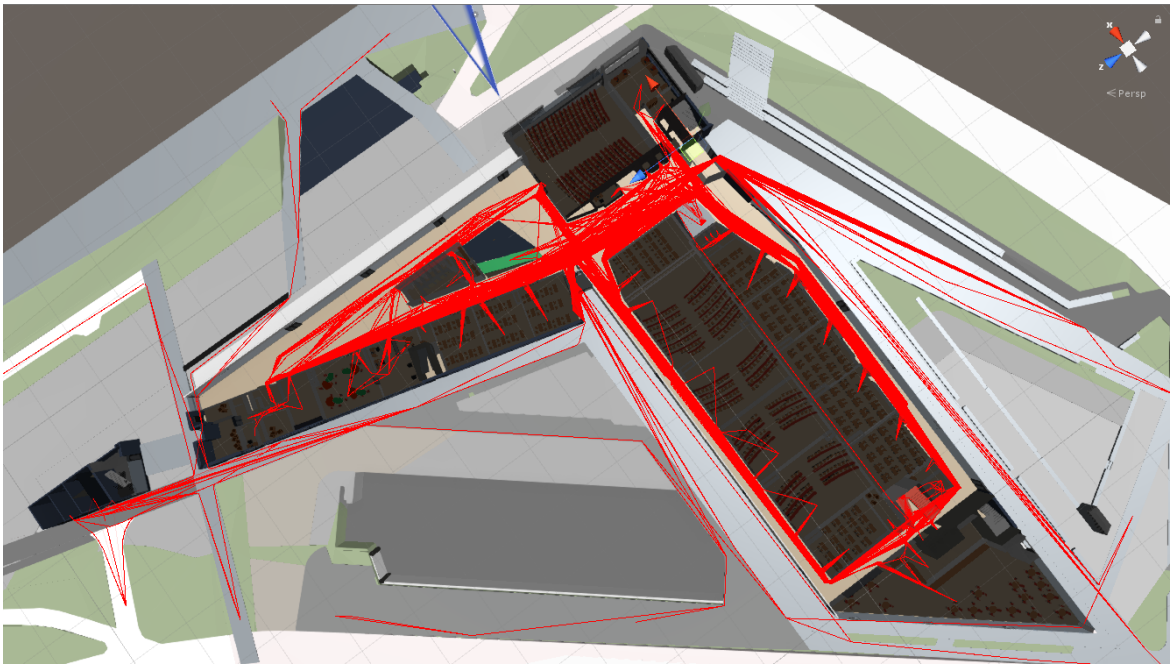

Figure 16. Precalculated paths on the first floor after adding the influencers.

There are more than 100 rooms in the visualized Delta building. Making a direct path between all of them leads to a lot of paths. The next subchapter covers different waypoints that were made during this thesis to reduce the number of precalculated paths.

3.2 Waypoints

The pathfinding system in DBV has two layers. A higher level, that consists of a graph of waypoints and a lower level, which is the navmesh. To get from one location to another a queue of waypoints is first put together in the *RoomManager* and the *Room* classes. A component called *ActorPath* was created to store this queue and to give the next path segments to the agents. A path segment is a precalculated path on the navmesh from one waypoint to the next one. Figure 17 illustrates how all the waypoints are connected, where these can be found and in which classes are the waypoints stored.

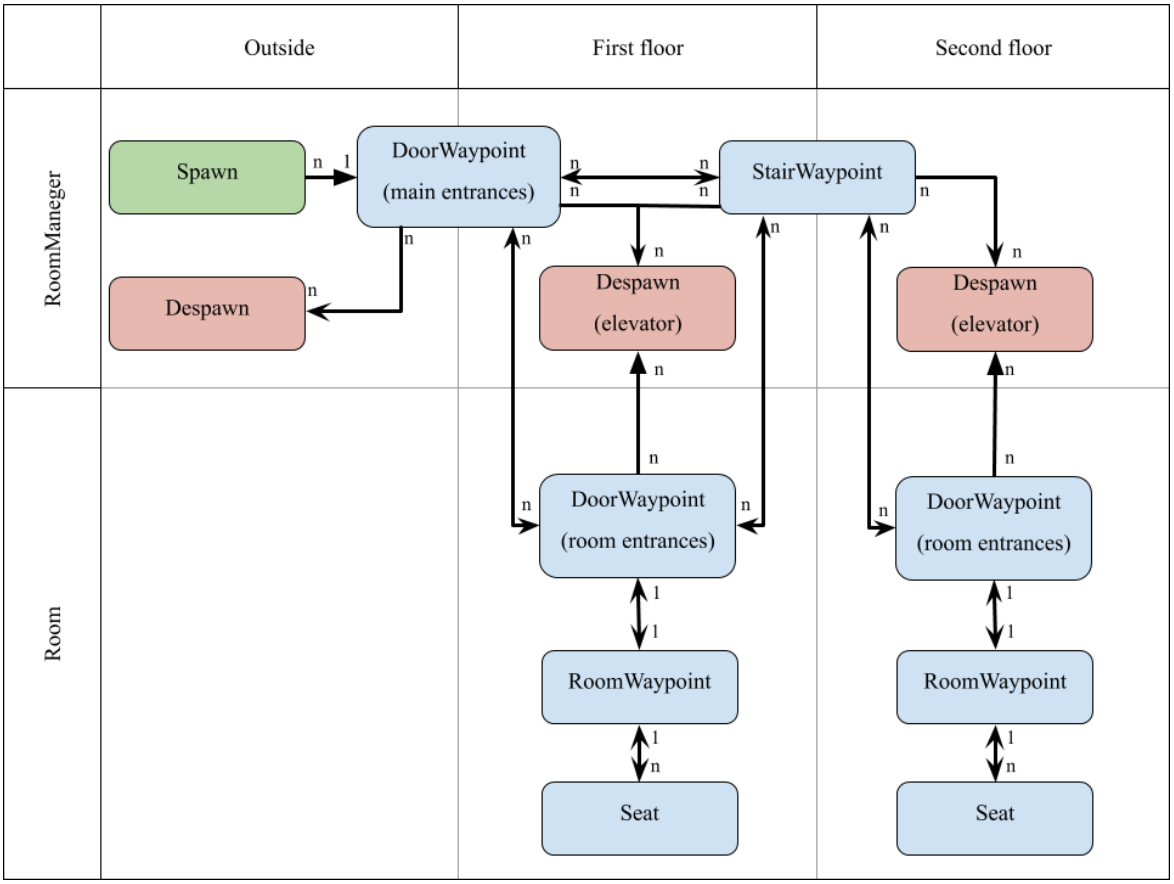

Figure 17. A graph of waypoints, with their locations and the classes which are used to put together a higher-level path.

All the waypoints are subclasses of the class *Destination*. The hierarchy of the waypoint classes is shown in Figure 18. All the waypoints have a `onReached()` method that is called

once an actor reaches the waypoint it was moving towards. This method is mostly used to make agents continue their path, but some waypoints have a special use for this method.

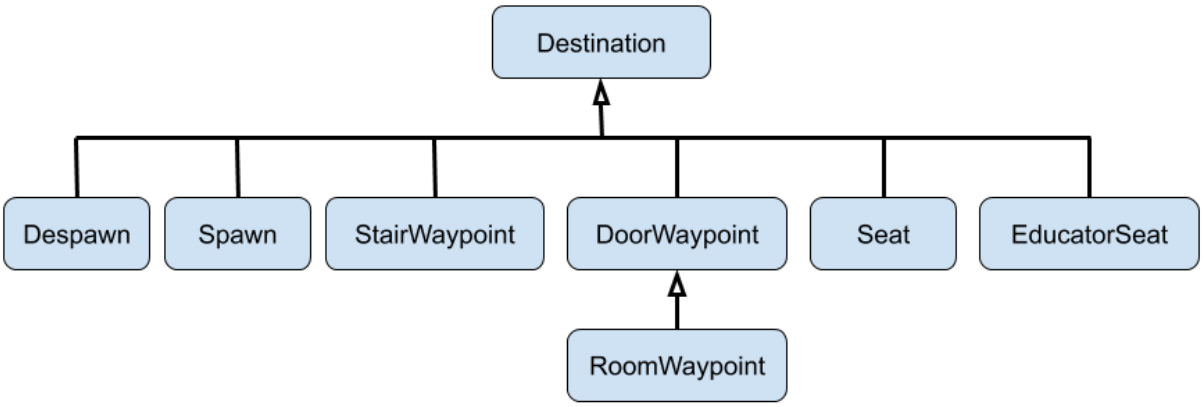

Figure 18. The hierarchy of the waypoint classes.

The following subchapters describe each waypoint more in depth.

3.2.1 Spawn Location

The spawn locations are the beginning of every agent's path. If a room has too few agents and none of the other rooms have a surplus of agents, then this waypoint is used to spawn a new agent. The spawning process begins by taking an agent out of the agent pool, then it is moved to the spawn location, made active, given a higher-level path to its seat and finally the agent is set to move. The next waypoint that the agents reach is the *DoorWaypoint*, which is covered next.

3.2.2 DoorWaypoint

The old solution for opening the doors was to assign every agent a door, that the agent was going to pass through. The door was opened once an agent got close to it [1]. It meant that the distance between the agent and its door were calculated almost every frame. In this thesis the `onReached()` method was used to optimize it. These waypoints were put near the doors, outside the rooms. So once an agent reaches this waypoint while not exiting the room, the door is updated. If the door is closed, then the agent waits for the door to open and then continues its path. If the door is already opened, then the agent goes right through and the door closing timer is reset. Once 3 seconds have passed without any agents going into the room, the door is closed.

These waypoints at the main entrances are also used to switch agent's layer[8] from outside to first floor and vice versa. This could be used to add visual effects to agents that are outside. Furthermore, the current solution allows to render the agents that are on the second floor and outside at the same time. Previously the agents outside were on the *first floor* layer and the cameras on the second floor do not render agents that have *first floor* as their layer.

The next waypoint for an agent that is moving towards its seat is called *RoomWaypoint*, which is covered next.

3.2.3 RoomWaypoint

The *RoomWaypoint* is a waypoint that is in every room. Even if it is not manually placed there, it is created at the room's coordinates. It is also a subclass of the *DoorWaypoint*, because the doors also need to be opened when an agent leaves the room.

Previously the paths were not precalculated between a room and its seats but they are now. The change was made because if too many agents calculated paths from room entrances to their seats or vice versa at the same time, then some of those agents were delayed. This caused some agents to gather near the room entrances or wait at their seats. From here the next waypoint would be *Seat* or *EducatorSeat*, but these were already covered in chapter 2.2.2.

3.2.4 StairWaypoint

If an agent's destination is not on the same floor as its starting location, then the agent needs to use stairs. In this case a *StairWaypoint*, which results in minimal travel distance, is added to the agent's path.

The main purpose of this waypoint is to reduce the number of paths between the rooms. The number of edges in a complete graph is given by:

$$Edges = \frac{n(n-1)}{2}, \tag{1}$$

where n is the number of vertices. Thus, only calculating the paths between the rooms that are on the same floor can reduce the total number of paths by down to 50%. Initially there were 15006 directed paths between the rooms. After splitting the floors there were 8456 directed paths in total between all the rooms and stairs. This means that there are now 44.4%

[8] https://docs.unity3d.com/ScriptReference/GameObject-layer.html

fewer paths between the rooms. The new number of paths also contains paths to stairs, because paths also need to be created between the rooms and the stairs.

Depending on which floor the agent is going to, the `onReached()` method of this waypoint is used to switch the agent's layer to the second or the first floor.

3.2.5 Despawn location

The last waypoint in an agent's journey is a *Despawn*. Once an agent reaches this waypoint the agent is made inactive and put back into the agent pool.

During this thesis the despawn locations were also added to outside, since real people usually tend to leave the building if they do not need to be there anymore. Previously the despawns were only in the elevators.

3.3 Agent groups

In real life some students move in groups of friends. In this chapter it is described how the agents in DBV were made to mimic this behavior. First, a new class called *ActorGroup* was created. This is a subclass of the *Actor* class, because agent groups should move similarly as single agents. Basically, an agent group is just one big agent, that contains smaller agents. Since the group acts as one agent, the groups should also improve the performance.

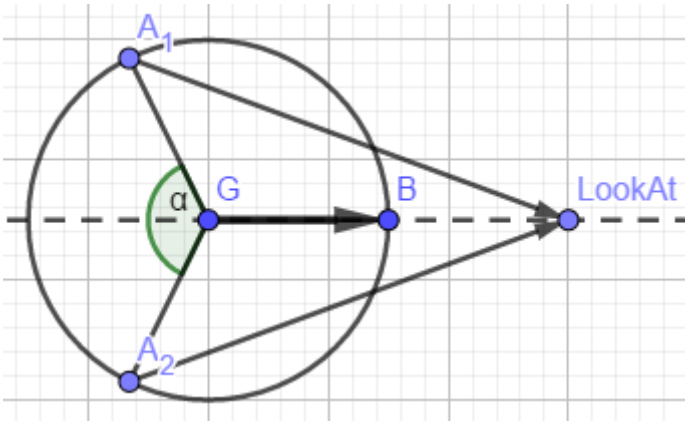

Figure 19. Agent group structure

Figure 19 illustrates how the agents are situated in a group of 2. Vertex `G` is the center of the group. All the agents in the group are placed on the circle around it. The angle α is the configurable angle between the two outermost agents. Upon adding or removing an agent from the group, the formation of the group is changed. Additional agents would be placed on the arc A_1A_2 which is divided into `n-1` equal parts, where `n` is the number of agents. The vector `GB` shows the forward direction of the group. The agents look at the position of

LookAt. The location of LookAt, the angle α and the radius of the circle can be modified to chance how the agents are situated in the group.

The parameters are tuned for a group of two, three and four agents. The parameters that the agent groups in DBV have are shown in Table 1. With those parameters the agent groups looked like in Figure 20.

Table 1. The parameters of every possible group size.

Group size	Angle α	||LookAt – G||	Radius
2	180	1	0.3
3	170	0.9	0.4
4	160	0.8	0.5

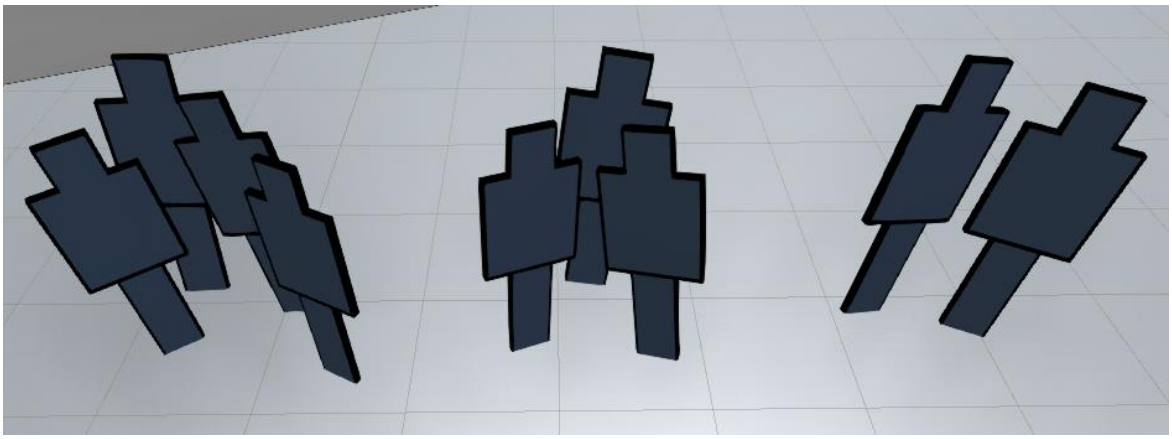

Figure 20. Agent groups in DBV

There are two different ways for the groups to form. Agents can form a group while spawning or upon reaching a *DoorWaypoint* while exiting a room. In both cases a number from 1 to 4 is randomly chosen from a list. To give some group sizes higher chance of appearing, the lists can contain the same value multiple times. If the number 1 is picked, then a group will not be formed. Otherwise a group of the picked size will be formed. Sometimes a higher number than the number of available agents is rolled. In such a case the group size will be reduced to match the number of available agents.

At the spawn locations the agent groups are spawned so that the agents in the group already have a common next waypoint. At the *DoorWaypoints* it is different. Only the agents that have the same next waypoint, can be grouped together. Thus, multiple groups can form at *DoorWaypoints* at the same time. Once an agent starts a group, there is a set amount of time

before the group is finalized and begins to move. If no agents with the same next waypoint joined the group, then the group is dissolved and the one agent there continues its path alone. Otherwise the group is finalized and set to move.

Agent groups follow the path of the first agent i.e. the agent that was first added to the group. Once a group reaches its next waypoint, all agents in the group update their path segments. If some of the agents do not share the next waypoint with the first agent anymore, then they are removed from the group and continue their paths alone. There cannot be groups of size 1, i.e. if all other members leave the group, the first agent is also removed.

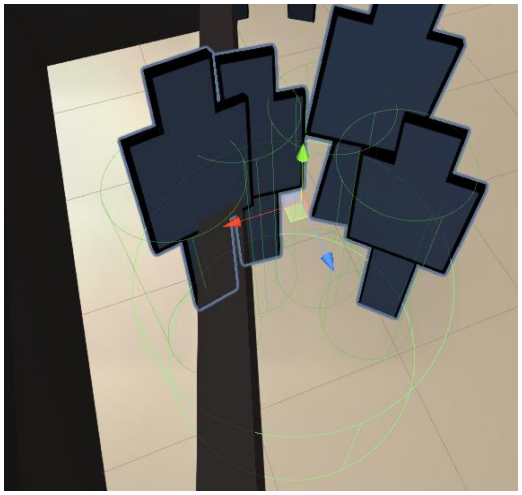

Figure 21. Agents inside a wall

Agent groups have around double the radius of the normal agents. Meaning that if they used the same paths as the agents, they could sometimes go through walls as seen on Figure 21. The fix for this issue was to create a separate layer for the agent groups. The creation of this new layer is covered in the next subchapter.

3.3.1 Layered Navmesh

Unity's own NavMesh system only supports creating a single layer of navmesh. Fortunately, there is an additional official asset called High Level API Components for Runtime NavMesh Building[9]. This asset was used in this thesis to create a second layer of navmesh for the agent groups.

There are some differences to the Unity's default system for building a NavMesh. For instance, the object's navigation area[10] that would usually be set in the Navigation window is not used. Instead the object's layers and the High Level API components are used.

In the DBV the object layers were mostly used for mapping the building again. That's because a lot of objects already had a suitable layer. A NavMeshModifier[11] component was added to the objects instead if the layer was not suitable. For example, if the floor and walls

[9] https://github.com/Unity-Technologies/NavMeshComponents

[10] https://docs.unity3d.com/Manual/nav-AreasAndCosts.html

[11] https://docs.unity3d.com/Manual/class-NavMeshModifier.html

had the same layer, then the NavMeshModifier component was added to the floor and the layer was used to mark the walls as unwalkable. That component allowed to change the object's navigation area.

Once every object was properly mapped once again, the second layer was precalculated. An example of the double layered navmesh can be seen in Figure 22. The darker blue is the area where both layers are at the same time.

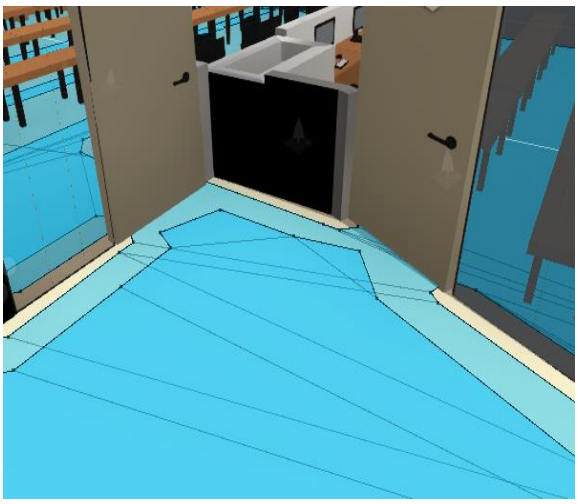

Figure 22. Double layered navmesh

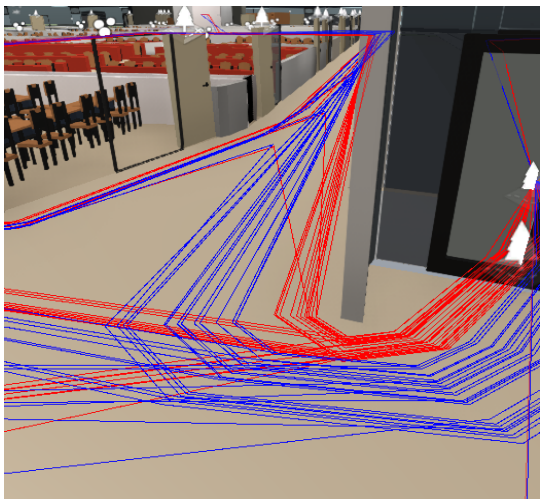

Figure 23. Normal and group paths side by side.

Separate paths for the groups were also precalculated. These are displayed as blue lines in Figure 23. The group paths are not calculated within the rooms, because the groups would not fit through the doorways and there cannot be any groups that go from the *RoomWaypoint* to a *Seat*. That is also the reason why the groups can form at the *DoorWaypoints* instead of the *RoomWaypoints*.

3.4 Additional Fixes

While improving the agents, some additional issues were discovered, that required fixing. To begin with, in some cases the agents teleported to their seats or could not reach it at all. This issue was caused by the location difference of seat's mesh and its real position (see Figure 24). A good way to fix it would be to move these locations together in the prefab editor, but unfortunately these seat prefabs are unlinked. Which means that all seats would

need to be replaced. A simpler approach was to modify the *Seat* component to use its mesh's location instead.

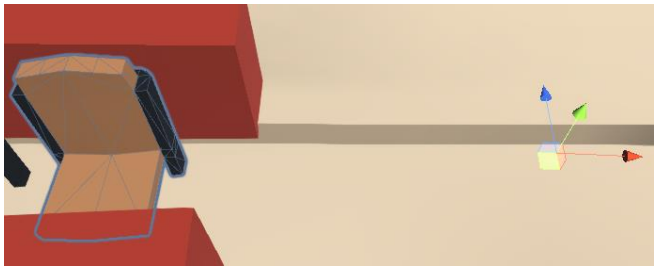

Figure 24. Location of the seat's mesh (left) and seats real location (right).

The seats also had issues with the *ChanceToEmote* component, which is covered in the next subchapter.

3.4.1 ChanceToEmote

During this thesis the *ChanceToEmote* component was also fixed. Initially only the seats had this component, but in this thesis, it was also added to the agent groups. That component used to have only one chance to do an emote. After that it did not do anything. Simply adding an infinite loop to its coroutine[12] fixed the issue. A reference of the coroutine was also added to the component, so it would be possible to break the infinite loop by stopping the coroutine. This coroutine also used to check every 15 seconds if an agent has sat down. To fix this issue, the coroutine was set to start in the `OnEnable()`[13] method and to stop in `OnDisable()`[14] method. Using these methods is better, because otherwise all seats check every 15 seconds if an agent has sat down.

Another issue with this component was that the speech bubbles it created were not oriented towards the camera. This happened because the speech bubbles were only rotated when they were made visible. There are six different views, that display the visualized building. The views are displayed one by one and the one currently displayed moves on a linear trajectory. After some time, the view is changed to the next one, making some speech bubbles oriented towards the camera's old position. To fix this, the coroutine was set to update the rotation of the speech bubble every second while the bubble was active.

[12] https://docs.unity3d.com/Manual/Coroutines.html

[13] https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnEnable.html

[14] https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnDisable.html

4 Testing

To ensure that the visualization performs well, it was tested. The testing was done on a computer with the following hardware:

OS: Microsoft Windows 10 Enterprise x64 bit

Processor: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz

GPU: NVIDIA GeForce GTX 980 Ti, 4 GB RAM

RAM: 8 GB

The hardware is the same as Nikolajev used for testing in his thesis [1]. Although he left out some important details from his thesis on how he tested the application, the results are still compared. Also, instead of using frames per second (FPS) for measuring performance, frame time[15] in milliseconds is used. Frame time is better because it is linear regarding to the performance [12]. Frame time is just the inverse value of FPS and for reference 30 FPS would be 33.(3) ms per frame

The performance difference of agents before and after optimizing them is not covered here, because it is difficult to measure their performance correctly. Comparing the frame times would also be off, because in addition to improving the agents, Linde also added new features to the visualization. Those new features could also affect the performance. The following chapters cover the testing of the DBV and the testing of the agent groups.

4.1 Performance of DBV

The performance of DBV is not only affected by the agents, but also the new visual effects that Linde created in his thesis [3]. Since the Unity Profiler has a large overhead, Fraps[16] was used to measure the frame time instead. With low a overhead it takes less time to take measurements than with a large overhead. Thus, the lower the overhead is, the more accurately the frame time is measured.

For taking the measurements, a build of the project was first created, because the build of the project almost always has a better performance than running it in the editor. Resolution also has a huge impact on the performance. The resolution while taking the measurements

[15] https://cgvr.cs.ut.ee/wp/index.php/frame-rate-vs-frame-time/

[16] http://www.fraps.com/

was 1920×1080, which was the highest that could be selected. The graphics quality setting was set to *Fastest* (see Appendix II).

Each view of the visualization was tested separately, because it was also done so by Nikolajev [1]. The automatic camera switching was turned off, so the performance of each view could be measured separately.

The performance was tested with 2010 agents. Every 10 seconds the agents that had despawned, were spawned again. The agents that reached their seats, were made to stand up and set to move towards a despawn location. Each view's frame times were recorded for 5 minutes. The results are displayed on Figure 25 as blue boxes.

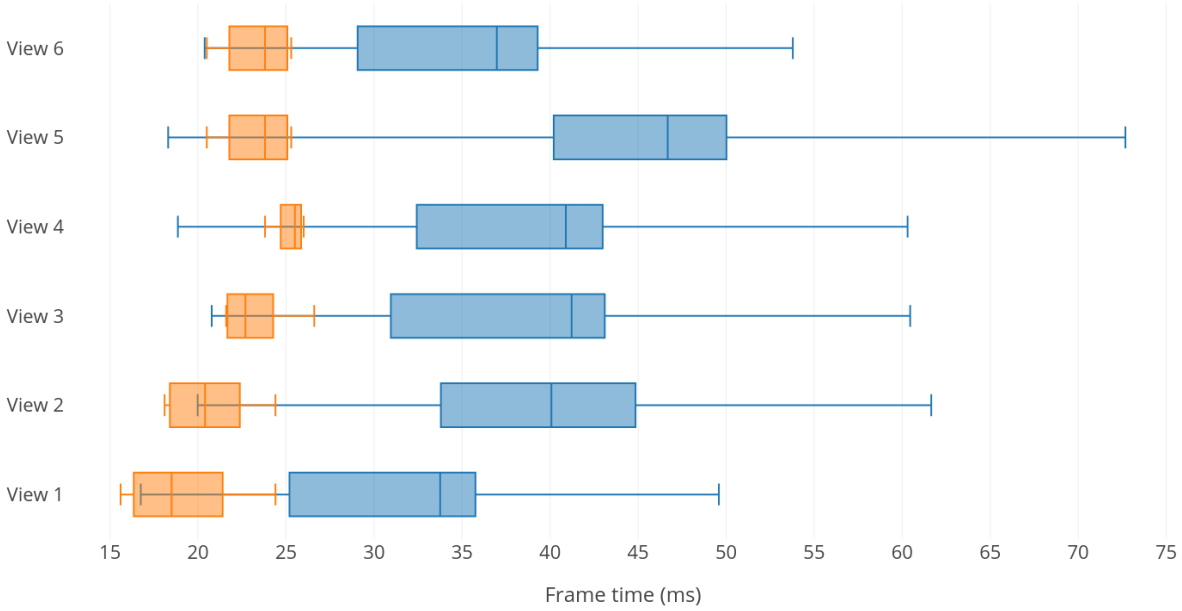

Figure 25. Nikolajev's frame time (converted from FPS) measurements (orange) compared to the measurements taken in this thesis (blue).

There were some anomalies in this graph. Less than 0.1% of the frame times have a value up to 130 ms. Those outliers were removed from the graph to better show the results but were not removed from Figure 26, to show that they exist. These anomalies happen when a lot of agents start to move at once. The delaying of agents was not done in this thesis due to time constraints.

The previous developers had a goal to have an FPS of at least 30 i.e. frame time of less than 33.(3) ms. Unfortunately, this was not achieved in this thesis, since most of the frames took more than 33. (3) ms to render. Again, the frame times were also affected by the features that Linde implemented. Based on Figure 25, it cannot be said how the features implemented during this thesis affected the performance.

Figure 25 also contains the measurements that Nikolajev took during his thesis [1]. As can be seen, the frame times have gotten considerably worse. Because it is unsure exactly how Nikolajev took his measurements, the procedure used to measure frame time in this thesis can differ from his.

The optimization of agents is not represented well here. Thus, the next chapter covers the performance difference that having the agent groups gives.

4.2 Performance with Groups and No Groups

One of the reasons the agent's groups were implemented in this thesis is that they could improve the performance. Two different builds were created to test it. One where the agent

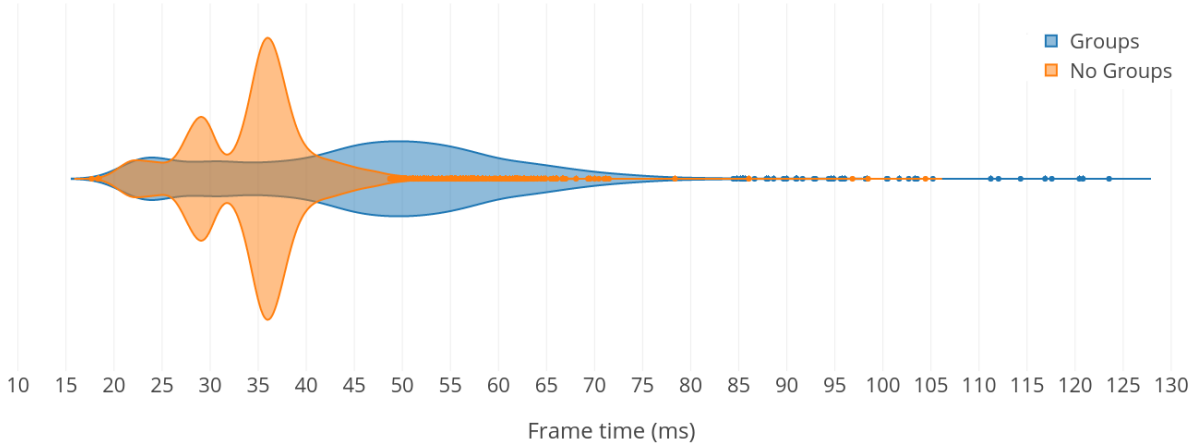

Figure 26. Frame times with mostly groups (blue) and no groups (orange).

groups cannot be spawned or formed. In the other build, the agents always tried to form groups and spawned as groups whenever possible. This was done for 10 minutes for each build. Everything else remained the same as the test described in the previous subchapter. The results are shown in Figure 26.

The difference is quite substantial. The decrease in performance could be caused by the size of the groups. Due to their size, they sometimes can cause blockades, which causes the agents to pile up. Thus, agents must recalculate their paths more often. The number and the size of the blockades was especially high because most of the agents moved in groups. Figure 9 also supports this claim by showings that increasing the obstacle avoidance radius worsens the performance.

4.3 RAM and Stability Test

The visualization will have to run continuously once the Delta building is ready. To ensure that the random-access memory (RAM) will not run out or something else does not break,

it had to be tested. A custom scenario was created to aid in this. This scenario changes the number of agents in certain rooms to a set amount every minute. The scenario covered rooms on both floors, so the agents would constantly move between the floors as well.

The scenario was set on a loop for 5.5 hours. Ram usage was measured every 4 seconds using the MSI Afterburner[17]. The result of the test is shown on Figure 27.

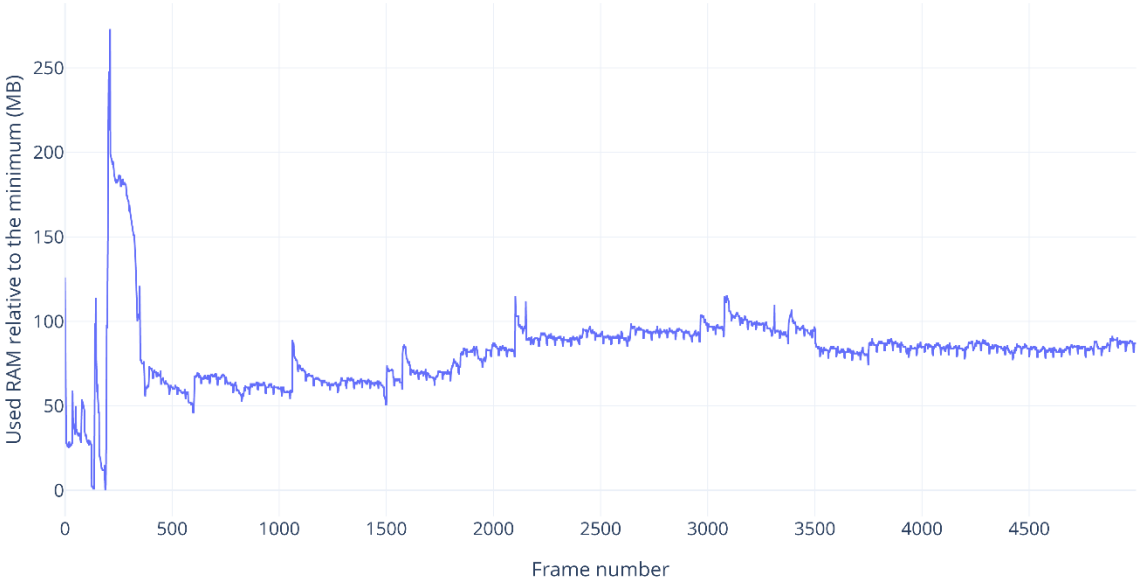

Figure 27. Used RAM over the period of 5.5 hours

To better show how the RAM consumption changed over the course of 5.5 hours, the measurements were subtracted by the minimum recorded RAM usage value. Otherwise the RAM values would depend on other irrelevant programs as well. As can be seen on Figure 27, the RAM usage is quite stable. Only at the beginning there were some bigger fluctuations. After the 5.5 hours the agents still moved like they did in the beginning of the test.

4.4 User Testing

To evaluate the features implemented in this thesis visually, the visualization was displayed on a Video Wall in the University of Tartu Library (see Figure 28). 6 people were asked to view the visualization and then answer a questionnaire (see Appendix II) about it.

Unfortunately, the admin tool, that Kütt developed as his thesis [4], did not work as it should have during the testing. Trying to get it to work at the last-minute cost 4 viewers, who had

[17] https://www.msi.com/page/afterburner

to leave prematurely. Only the custom built-in scenarios, that were made for debugging could be shown. The last remaining viewers were only shown the scenario where the agents fill the rooms and then just go to the despawn locations.

There were 2 types of questions. The ones where the viewers had to rate something in an inclusive range from 1 to 6. The second question type required the viewer to write an answer.

One of the goals of this thesis was to make the agent pathfinding more realistic. The viewers rated the realism of the paths 4,5 out of 6. The agent groups were also asked about. One of

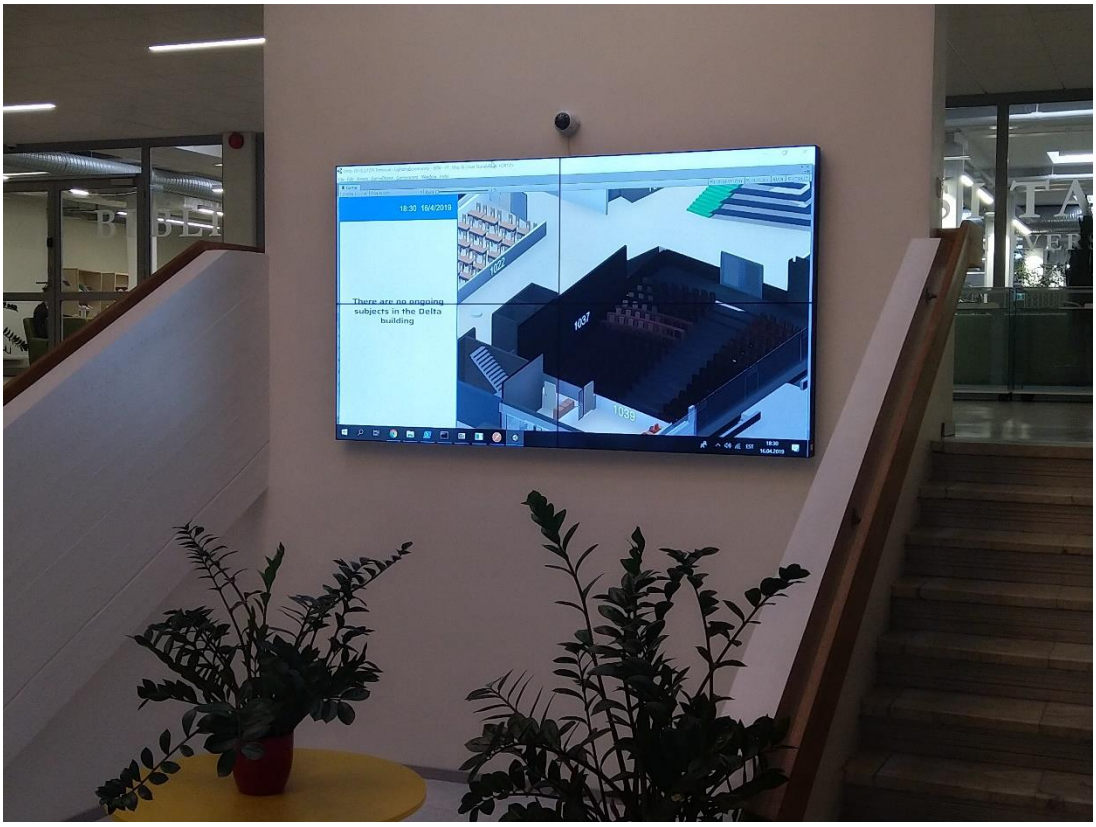

Figure 28. Video Wall in the University of Tartu Library

the viewers noticed them and the other did not. Thus, also suggesting that they should be made more noticeable. The viewer who noticed them, also liked them by rating the groups by likeness of 4.

The overall appearance and understandability of the visualization was rated 5.

5 Future Improvements

Before the visualization is displayed in the Delta building, there is still time for the future developers to improve this visualization. The author of this thesis put together the following list of suggestions for future developers:

1. Currently the agents get seat positions from a queue. This often results in the seated agents forming clusters. For example, one half of the auditorium could get filled and the other half does not have any agents (see Figure 29). The agents at minimum could be distributed more evenly by randomizing the queue. Even better would be to make them take more seats where real people would more often sit. For example, more people sit on the back and few on the front rows.

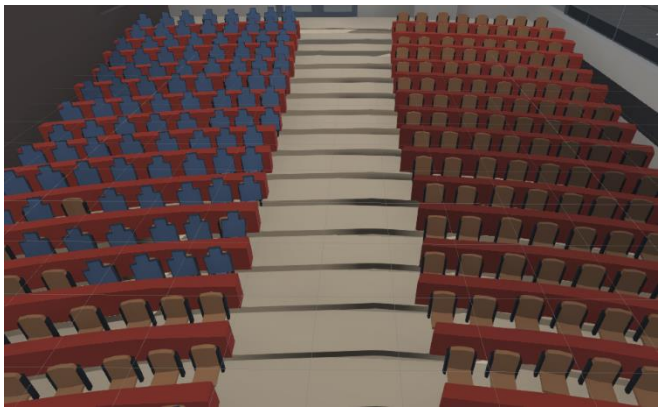

Figure 29. Agents sitting in a cluster.

2. The agents should be made to leave the classrooms slower. Making them move all at once, is the reason for the frame time outliers, that were discovered in chapter 4.1. Also, it does not look realistic when all the agents stand up at the same time.
3. Optimize the agent animations. The animations were found to take a lot of resources, so improving the animations should improve the overall performance of the visualization.
4. Currently on the corridors there are fixed locations where the agents can perform emotes. This could be made more random.

The viewers in chapter 4.4 suggested to give agents more activities to do. Currently they only move to their destinations and do emotes there. For example, some agents could be made to wander around the building or made to stand with their backs against the walls.

6 Conclusion

In this thesis the Delta building visualization project was extended. The main goals of this thesis were to improve agent behavior and to improve the performance of the visualization. Before that was done, the project was refactored.

First, the project's file tree structure was improved. Then the scripts, which were relevant to this thesis, were refactored. The scripts were refactored so that it would be easier to achieve the main goals. Thus, the agent pooling and a base for the higher-level navigation system were created during the refactor.

Although other pathfinding algorithms were investigated during this thesis, the existing Unity's NavMesh pathfinding solution was improved and used for lower-level navigation. To make the paths more random, objects called path influencers were created. In short, the *path influencers* move the precalculated path's vertices closer to the influencer or further away from the influencer. The distance depends on the influencer's parameters and a random value.

For higher-level navigation a graph of waypoints was created. The waypoints reduce the number of the precalculated paths, make them shorter and further improve the performance of the visualization. The graph was simple enough for the waypoints to construct a path for the agents, without having to search for the optimal path between the waypoints. If new types of waypoints are added to the graph in the future, it is advised to switch to the A* algorithm.

Agent groups were also created during this thesis. High numbers groups were found to decrease the performance of the visualization, but the groups did improve some visual aspects of the visualization.

The overall performance is affected by the features implemented during this thesis and in Linde's thesis. Unfortunately, the overall performance of this visualization got worse after implementing all the new features. The stability of the visualization was also tested and concluded that the agents acted like they did in the beginning.

Reflecting on the work, implementing the agent's pathfinding required a lot of experimentation. Often the agents started to take weird paths, not move at all or even just teleport to the end destination. A huge help was the debugging tool, that was created during this thesis. Extending the project was not easy. Quite a bit of time went into understanding and getting used to the already written code. Overall the project was fun to work on.

References

[1] A. Nikolajev, Delta õppehoone visualiseerimine ja optimeerimine, Bachelor's thesis, University of Tartu, 2018.

[2] A. Voitenko, Delta Building Environment Visualization, Bachelor's thesis, University of Tartu, 2018.

[3] E. Linde, Delta Building Visualization - Visual Effects, Bachelor's thesis, University of Tartu, 2019.

[4] D. Kütt, Delta Building Visualization - Admin Tool, Bachelor's thesis, University of Tartu, 2019.

[5] M. Fowler, K. Beck, J. Brant, Refactoring: Improving the Design of Existing Code, pages 46–72, 2002.

[6] Optimizing Garbage collection in Unity games,https://unity3d.com/learn/tutorials/topics/performance-optimization/optimizing-garbage-collection-unity-games (21.02.2019)

[7] R. Nystrom, Game Programming Patterns, genever benning, pages 305–320.

[8] Z. Abd Algfoor, M. S. Sunar, H. Kolivand, A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games, International Journal of Computer Games Technology, 2015. https://www.hindawi.com/journals/ijcgt/2015/736138/ (08-Mar-2019).

[9] W. G. van Toll, A. F. Cook, R. Geraerts, Real-time density-based crowd simulation, Comput. Animat. Virtual Worlds, volume. 23, pages 59–69, 2012.

[10] S. Chenney, Flow Tiles, in Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pages 233–242, Goslar Germany, Germany, 2004.

[11] L. E. Kavraki, P. Svestka, J.- Latombe, M. H. Overmars, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, IEEE Trans. Robot. Autom., volume 12, issue 4, pages. 566–580, Aug. 1996.

[12] Mali GPU Application Optimization Guide. ARM, 2011. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0555a/BEIGDEGC.html (02.05.2019)

Appendices

I Glossary

Abstract data structure - A data structure for objects whose behavior is defined by a set of value and a set of operations.

Bloated class - A class with too many responsibilities.

Code rot - A slow deterioration of software performance over time or its diminishing responsiveness that will eventually lead to software becoming faulty, unusable, or otherwise called “legacy” and in need of upgrade.

Code smell - Any characteristic in the source code of a program that possibly indicates a deeper problem.

Complete graph - A fully connected graph i.e. every node of the graph is connected with every other node of the same graph.

Despawn location - A location, where an object is destroyed or made inactive.

Frame time - The time it takes for the frame to be rendered.

Frames per second - The number of frames that were rendered in 1 second.

Game engine - a software-development environment designed for people to build video games.

Garbage collection - A process, that frees unused allocated memory.

Multi-agent system - A computerized system composed of multiple interacting intelligent agents.

Overhead - Any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task.

Pathfinding - The plotting, by a computer application, of the shortest route between two locations.

Spawn location - A location, where an object is instantiated or made active.

Static object - An object that’s position, rotation and size is fixed.

Trajectory - The path that an object follows through space as a function of time.

II Source Code Build and Other files

The source code is available as an attachment and from a Gitlab git repository[18] and the build is available as an attachment.

Measurements, Python programs that were used to convert data are available as attachments in the folder called *measurements*. Additional figures, graphics setting details used for testing, and a video of the application can be found in the *extras* folder.

[18] https://gitlab.com/UT-CGVR-Projects/DeltaBuildingVisualization

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Meelis Perli**,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Delta Building Visualization – Agent Logic,

   (title of thesis)

   supervised by Raimond-Hendrik Tunnel.

   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I know the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Tartu, **09.05.2019**