Vanja Falck
Ludvig Lilleberg
Kristoffer Madsen

Development of Dynamic Difficulty Adaptation Using Hidden Markov Models in Learning Games

Bachelor's project in Programming
Supervisor: Mariusz Nowostawski
May 2021

Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Vanja Falck
Ludvig Lilleberg
Kristoffer Madsen

Development of Dynamic Difficulty Adaptation Using Hidden Markov Models in Learning Games

Bachelor's project in Programming
Supervisor: Mariusz Nowostawski
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

An educational game is successful if the player achieves the accompanied learning goals. However, to make an educational game that is exciting enough for the player to keep playing until the learning goals are reached depends on both challenges in the gameplay and in the subject matter of the learning tasks. The learning process is perceived as a player being in a proximal zone of development and flow. One strategy for keeping a game exiting enough while also getting progress in learning is to keep a load of difficulty that is difficult enough, but not too difficult. We experiment with Hidden Markov Models to test different ways of implementing dynamic difficulty adaptation in the development of a learning game. As a proof-of-concept, we propose an outline of a framework for implementing dynamic difficulty adaptation in learning games.

Sammendrag

Et læringsspill er vellykket når spilleren oppnår de medfølgende læringsmålene. Det å lage et læringsspill som er spennende nok til at spilleren fortsetter å spille fram til læringsmålene er oppnådd avhenger av både utfordringene i spillet og innholdet av oppgavene. Læringsprosessen blir oppfattet som en spiller som befinner seg i den optimale sonen for utvikling og flyt. En strategi for å holde et spill underholdende, men samtidig oppnå fremgang i læring, er å opprettholde et vanskelighetsnivå som er vanskelig nok, men ikke for vanskelig. Vi eksperimenterer med Hidden Markov-modeller for å teste ulike måter å implementere dynamisk vanskelighetstilpasning i utviklingen av et læringsspill. Som konseptbevis legger vi frem en skisse av et rammeverk for å implementere dynamisk vanskelighetstilpasning i læringsspill.

Contents

Figures

Code Listings

Acronyms

AI Artificial intelligence. 21, 45

GUI graphical user interface. ix, x, 17, 19, 27, 32, 33, 44, 45, 71, 72

IIK Department of Information Security and Communication Technology. xi, 2

NPC non-player character. 2, 13, 18, 19, 21–23, 25, 26, 33, 36, 37, 41–46, 51–55, 63, 71, 72

NTNU Norwegian University of Science and Technology. xi, 2, 71

UDIR Norwegian Directorate for Education and Training. 2

Glossary

Blender A free and open-source 3D graphics software tool set with features such as 3D modelling, texturing, and animation[1]. 32, 71

Canvas The area in Unity where all GUI elements should be, with screen space (overlay or camera) and world space render modes[2]. 32, 45

Firebase A Google platform for creating mobile and web applications[3], providing products such as databases, authentication, hosting, and machine learning features[4]. 24, 31, 33, 71

Firestore A NoSQL cloud database used to store and synchronise data, supporting hierarchical data structures consisting of documents and document collections, provided by Firebase and Google Cloud[5]. 2, 3, 10, 16, 18–20, 22–26, 31, 33, 34, 36, 41, 43, 44, 46, 48–53, 55, 60, 61, 63, 64, 71, 72

Fungus A Unity extension optimal for making visual novels and interactive fiction[6]. 26

Hierarchy window A window in Unity where all Game Objects in a scene are displayed in a parent-child hierarchy[7], similar to a scene graph data structure. 32, 59

Inspector A window in Unity where one can view and edit properties and settings for elements that are in the Unity editor[8]. 32, 34, 59, 71

Material A Unity asset which describes the appearance of surfaces, and which can contain colour and texture data[9]. 32, 34, 59

Prefab A Game Object as a reusable asset, acting as a template from which Prefab instances can be created[10]. 32, 47, 48, 59

ProBuilder A Unity package which allows for the creation of 3D models with simple geometry, used for level design[11]. 32

Scroll View A Unity list object which can contain other objects, and which allows for vertical and horizontal scrolling when the content exceeds the Scroll View size. 27, 29, 30, 47, 48, 53–57, 59, 64

TextMeshPro A Unity package which can be used as a replacement for Unity's native GUI text system, offering visual quality improvements and more[12]. 27, 29, 30

Twinery An online tool for making interactive texts[13]. 10, 21, 22, 26, 44, 45, 63, 72

Unity A cross-platform game engine developed by Unity Technologies which can be used to create 3D, 2D, virtual reality, and augmented reality games, simulations, and other experiences[14]. x, 2, 3, 13, 22, 24, 26, 27, 30–33, 37, 38, 43, 58, 59, 61, 62, 66, 71, 72

Visual Studio A code editor and compiler owned by Microsoft, the default editor of Unity. 31

Preface

Our client is Department of Information Security and Communication Technology (IIK) at NTNU. This task is born from questions about using entertainment as a means for education, and in particular the idea that challenges in learning and gameplay can enhance learning and give joy. If the challenge level of learning games can successfully be set and added to the curriculum, they could potentially help people to learn in an enjoyable and efficient way.
We chose this task because we learning games is a genuinely interesting topic.

Chapter 1

Introduction

Our bachelor task is focused on the field of educational technology, specifically educational software. Educational games may provide both theoretical and practical knowledge due to their sensory and interactive nature, while also providing an enjoyable experience. The combination of the sensory, interactive, and entertaining aspects may be an effective and motivating way for students to learn [15].

Learning games are developed to improve learning beyond what is achieved by more traditional methods. A successful learning game must help the player reach learning goals and continue playing until these goals are reached. Challenges tuned in with the player's current performance level is believed to enhance learning [16]. Similar strategies have support in the learning theory of Lev Vygotskij and his concept of proximal zone of development and in Mihaly Csikszentmihalyi's concept of flow [17]. These theories claim that a sufficiently high load of difficulty gives the student or player the most efficient learning and the experience of excitement [17].

Different approaches to difficulty adaptation in games and learning games have been proven to be successful [17]. However, getting appropriate knowledge about a player's performance levels and using the right techniques to enhance learning is not at all trivial [15]. How do you start out with a new player? How do you choose what elements to adjust to ensure learning? Such challenges are linked to find the components in a game and particularly the components in the learning parts and method to tweak them. Methods using functions to adjust difficulty level are called *dynamic*. Techniques applying support vector machines and neural nets have proved useful. Even if neural nets can be useful, it is hard to understand what the nets are doing. Other approaches running better with sparse data and which are better explainable, like dynamic Bayesian nets. Even when these probabilistic models are less scalable than neural nets, they can prove useful in the development of dynamic difficulty adaptation in learning games.

1.1 Delimitation

According to our client, the Department of Information Security and Communication Technology (IIK) at NTNU; the Norwegian Directorate for Education and Training (UDIR) has developed a new curriculum in digital citizenship, which applies to Norwegian students in all stages of primary and secondary education. Our client wants to develop games which will aid in the education of the digital security part of the curriculum.

In agreement with our client, we decided to make a game mock-up in Unity to investigate how learning sniffing and dynamic difficulty adjustment can be applied in game development. We demonstrate these ideas and solutions with a proof-of-concept game, a database solution, and a system of functions for dynamic difficulty adaptation.

The proof-of-concept game will work for both PC and mobile, and will be playable with a mouse and a touch screen, respectively. It will contain a simple avatar creator, and functionality for reading, and being a part of, conversations with NPCs (non-player characters). Text in NPC-to-NPC conversations, and data for the in-game users, will be stored in a Firestore database (Section 9.2). This is a solution which allows the game's educational content to be changed whenever changes are made to the curriculum.

1.2 Target Groups

This report will detail the game design ideas and implementation ideas that were utilised in our proof-of-concept game, which will help our client develop a real application. Our client is therefore the primary target group of this report.

We limited the target group of our proof-of-concept game to be high school students only, which would remove some complexity from the application, and which would not limit us when it comes to digital security topics and content, due to the students' relatively high maturity and knowledge. Our prototype game will show our client possible game design ideas and how they would work in practice, meaning our client is also a target group for the proof-of-concept game itself.

1.3 Report Organisation

This report is a hybrid combining a first part covering the bachelor thesis research question and a second part elaborating on the software development of the mock-up game. Because a mix of the answering of the research question and a detailed description of the game development would overshadow each other, we decided to write them separately. Dividing the blocks best ensure that our project owner gets a detailed guide to our Unity mock-up game.

Chapter 1 is a general introduction to both the research part and the game mock-up. Chapter 2 gives the general outline for the research part, followed up by methods (Chapter 3), results (Chapter 4), discussion (Chapter 5) and conclusion (Chapter 6). The last part is in detail describing the development of the game mock-up (Chapter 7 to 14). Some technical aspects of the dynamic difficulty adaptations and a brief introduction to Hidden Markov Model as implemented is given, but most of the difficulty adaptation is dealt with in the first part (Chapter 2-6).

Information about the authors is placed in Appendix A. The source code for the project can be requested using the contact information in Appendix A.3. A glossary and a list of acronyms are included. All instances of these words and acronyms are clickable links in the PDF which lead to their explanations. URL links and references to chapters and sections are also clickable. Names of elements in scripts, names of collections, documents, and values in Firestore, and similar names, are written in the `typewriter font`. Other notable names and words are *emphasised*.

Chapter 2

Background

Gamification of learning faces the challenge of making the gameplay engaging enough for students to keep playing until the learning goals are reached. Educational games aim at supporting players in achieving learning goals [16]. Dynamic difficulty adaptation in learning games consists of methods balancing the player's perceived difficulty regarding both gameplay and educational demands [18] [19] [16] [20]. Difficulty adaptation in games is generally used to keep players engaged. Adjustments can be added to game elements like content [19], user interfaces or game mechanics to personalise the experience of difficulty [16][21]. Dynamic adjustment of difficulty in games can also be dealt with by using automated assistance and feedback systems [22].

Current research focuses mainly on dynamic difficult adaptation with respect to how the human player perceives the challenges in a game [16][21]. Automated tracking of difficulty requires measurables and functions that reflect a player's actual learning and mastering progress. A mixture of measures for both learning of the subject matter and mastering of the gameplay are needed, as well as methods to properly evaluate learning outcomes.

Difficulty adaptation in games has been explored in the context of design, simulation, and self-learning and self-adapting systems [16]. Machine learning approaches like support vector machines and neural nets show potential for useful data driven implementations [23]. However, neural nets run into problems of explaining what is actually done [24], resulting in additional need of human user testing to ensure validity. Integrating the player and game world models of dynamic adaptations, can theoretically reduce the number of suboptimal situations during gameplay [16]. Revealing and reducing suboptimal situations regarding learning and flow in the development phase seems necessary to ensure a high quality learning game.

2.1 Problem Description

Our main focus is not to develop a learning game, but **to explore how to implement dynamic difficulty adaptation in learning games, using Hidden Markov Models and social media security for youths aged 16 to 19 as a dummy-topic**.

We are not underestimating the challenges regarding design when forming learning games [25][21]. Neither are we underestimating the challenges of evaluating learning in games. Our proposed game examples are not claimed to actually be suitable in an educational setting. The theories of learning are also just suggestive, and useful to outline some general understanding of the concept of learning. Our game examples and learning suggestions are primarily for demonstrating dynamic difficulty adaptation and how it can be used in early prototyping of learning games.

Chapter 3

Methods

3.1 Learning in Games

Successful learning games can be defined as a process driven by the gameplay leading to anticipated learning goals. Measuring learning processes and goals however are not trivial. In this context it suffices to define the learning process in a broad sense by applying Lev Vygotskijs social constructivist concept of proximal zone of development [26], adding Mihaly Csikszentmihalyi's concept of float to describe a process both of learning and excitement that keep the player engaged enough to reach the learning goals [17]. Several forms of knowledge and skills can be embraced by these broad models [17]. Flow in a game is according to Chen (in Streicher and Smeddinck [16]) created by actions:

- Lead to or present clear goals
- Immediate feedback merging with awareness (you will instantly know the result)
- Trigger concentration on the task (no interruptions)
- Create a sense of power and control
- Make the player lose track of time and place (absorbed)
- Make the player wanting to play (not forced to play)

3.2 Difficulty Adaptation in Learning Games

The two major reasons for exploring dynamic difficulty adaptation in learning games are to ensure learning goals are met and ensure that the game is engaging enough for the player to keep playing until reaching the learning goal. But what then is difficulty and difficulty adaptation in a setting like this? Current research on difficulty adaptation in games often focus on the balance between the player (human aspects) and the gameplay (game mechanics) [16]. What is difficult to the player (learning goals and gameplay), and what challenges or other aspects in the gameplay are perceived by the player as difficult? In the case of dynamic difficulty adaptation we should ask how we can measure how the player perceives difficulties and how game mechanics and content can be manipulated to keep the player in their proximal zone of development while still remaining in a state of flow. A successful attempt to quantify the level of difficulty on the player's hand is to calculate the probability of the player failing a particular task in the game [18]. The general idea is then to combine these calculations to design an overall game logic that keeps the level of difficulty within the limits suggested by proximal development zone and float.

3.3 Approaching Dynamic Difficulty Adaptation

To improve designing learning games, Gallego-Durán *et al.* [25] suggests focusing on different characteristics instead of only game elements. These are open decision space, challenge, learning by trial and error, progress assessment, feedback, randomness, discovery, emotional entailment, playfulness and automation. Four of these, the challenge, progress assessment, randomness, and automation, are considered particularly relevant for the learning and difficulty adaptation. Feedback, emotional entailment, and playfulness are important for learning and engagement. These two elements could also be automated (Conati and Manske in Streicher and Smeddinck [16]), [22]. Open decision space, learning by trial and error, and discovery are more related to the choice of type of game and will not be dealt with in this project.

Educational content used in a learning game can change over time. Even if a subject matter like mathematics is relatively stable, the knowledge about didactics may change. What type of content that teaches different students, depending on age, gender, mental and physical disabilities, cultural background, and life experiences, might also change [27] [21]. Even if these considerations go beyond what is strictly relevant for the dynamic difficulty adaptation, they are relevant in discussions about flexibility in the development and maintenance phases of learning games used in public schools.

Different methods have been applied to measure player performance and skill-influencing aspects of the gameplay [16]. In this project the aim is to explore dynamic difficulty adaptation during the early development and prototyping of a game. We demonstrate a method for developing components using Hidden Markov Models to set up the dynamic difficulty adaptation and prepare for testing and simulation of the implementations.

3.4 Hidden Markov Models

Hidden Markov Models are dynamic Bayesian nets [24]. A Hidden Markov Model is made up of nodes in a directed acyclic graph. Nodes represent random variables carrying conditional probabilities given the values on their parents' nodes. A Hidden Markov Model represents world states as a discrete variable, but the values on each node could be either discrete or continuous. The dynamic in Bayesian nets is a time series model, meaning they are useful for a probabilistic understanding of phenomena changing over time. The main components are a transition model and a sensor model, which do not change over time. As a consequence, dynamic Bayesian nets are well suited for near to causal modelling of real life situations in a more or less stable environment. Stable environment is understood as a context that does not change the parameters of the stationary transition and sensor models. Given a well designed model, every causal random variable is taken care of, and therefore the stationary model handles all transitions in world states well.

Because Hidden Markov Models are well studied and make good causal models, when well designed, they are useful in testing out dynamic difficulty adaptation in learning games. The parameters of a Hidden Markov Model can be learnt from data by i. e. the expectation-maximisation algorithm [24]. Existing data on users or learning issues can be used to quickly get well tuned models up and running in game development. Another feature of Hidden Markov Models is that they are excellent as a driver for simulation. Weighted randomness sampling like Markov Chain Monte Carlo [24], can be used to simulate highly realistic scenarios to test how different dynamic difficulty adaptation implementations turn out for different player profiles, game or learning components.

3.5 Bloom's Taxonomy

Bloom's taxonomy is a systematic overview of learning skills based on cumulative levels of learning that are supposed to build on each other [28]. There are six levels. The first is to memorise knowledge, the second to be able to explain it, and the third to use and adapt knowledge in real life settings. The fourth is to be able to analyse complex situations using the knowledge. The fifth is to be able to discuss and criticise knowledge from different points of view. The sixth level is to be able to use knowledge to create something new. This taxonomy is often used as a basis in both summative and formative evaluation of learning [29]. Bloom's taxonomy differentiates types of learning that can be implemented in learning games. It is also believed to represent learning progress, and could be regarded as a scale of difficulty with regard to learning tasks.

To operationalise the concept of learning goals we add the five elements in Bloom's taxonomy (in Pierce *et al.* [19]) to our concept of learning. Bloom's taxonomy is often used by domain experts to set up more explicit summative and formative evaluations of learning outcomes.

3.6 Set Up

The dynamic difficulty adaptation component is made up of Bayesian networks as Hidden Markov Models. Our system is built on adding and tweaking Hidden Markov Models to adjust separate game components like feeding content and guiding non-player characters' behaviour based on the player's actions.

Three types of gameplay scenarios are implemented. The first is a 3D modelled school yard where different sequences of digital messages between pupils and external actors are exchanged. The player watches different digital conversations and should identify various digital threats or bullying. In this scenario, the player's ability to identify occurrences, measured by points as rewards, adjusts the level of difficulty on the upcoming next conversations by picking out a probability adjusted level.

In the second scenario, the player engages in interactive digital and live conversations with one or more non-player characters. Two mock-up scenarios were suggested, the first being new at school getting new friends while meeting peers engaged in live and digital conversations (messages and image exchanges). The second being a pupil or teacher as an investigator watching other pupils' digital conversations to identify potential digital security issues like grooming, fraud and digital harassment. Dialogues are structured as dialogue-trees, using Hidden Markov Models to guide an emotion and a difficulty component. The difficulty adaptation rests on tracking the player's performance and modulating the non-player characters' responses. The emotion component is added both to the player and non-player characters to guide the automated responses in the dialogue. The emotion component switches on three levels; angry, sad or friendly. Simulations based on probability distributed sampling, or a Monte Carlo Markov Chain Gibb's sampling [24] can be used offline or online using the Hidden Markov Models (without any players) can be run for on the dialogue-trees to monitor and evaluate the difficulty adaptation models.

In the third scenario, the player manages a spawned public social media page. The task is to identify potential issues related to revealing personal information that could be exploited or could harm themselves in general or in a special case of job seeking. The difficulty adaptation in this scenario can be linked to domain experts' pre-evaluation of vulnerabilities in the game's construction of a profile. Based on the defined levels of threat for each component, the adaptation model can automate the picking of components at the time of spawning based on the player's current level. A second approach could be applied while the user makes up their own social media profile, having an identifier of threat level, done by domain experts, to monitor profile development and setting up the player's gain or loss.

Chapter 4

Results

Three small scale game components, one a quest, the second a mystery, and the third a socio-quiz like game setting, are equipped with Hidden Markov Models to do the difficulty adaptation. The mystery game has a game component modulating emotions using a Hidden Markov Model. Simulation options are implemented in the mystery game. Simulations can be used by experts' options for testing models against i. e. learning outcomes and gameplay.

All difficulty adaptation implementations on game components feed into the player's current skill level and vise-versa, responds to gradually better game performance. When playing the first game you can either start with skill level at zero, or set up for a random level. No trained prior to i. e. overcome cold start problems for the player's performance level is provided.

When setting up the testing environment, we try to make it relatively easy to change the text sources used in the quest and mystery gameplays. These methods can be used for the sosio-quiz as well, but the current input here is hard coded. Two types of setups were used for feeding dialogues, one using Firestore database importing directly from a google sheet, the other parsing complete dialogue trees from text outputs from the online dialogue generator Twinery. All dialogues in the quest and mystery part can therefore be exchanged by experts in the subject matter of the learning game. A generator to make a random Hidden Markov Model is set up. Functions for doing basic procedures like filtering, prediction, smoothing and finding the most likely sequence are provided.

This project delivers a proof-of-concept of a flexible framework, using Bayesian nets in the form of Hidden Markov Models in game components and the player's profile to do component wise prototyping of dynamic difficulty adaptation aimed at high demands on learning and experience of flow in gameplay. We have Bloom's taxonomy to conceptualise different aspects of learning with the aim of further developing game components that are useful for summative and formative evaluations.

Chapter 5

Research Discussion

Hidden Markov Models are proving easy to use as modulators for prototyping dynamic difficulty adaptation on components and players in a learning game. However, these models are not particularly scalable as the complexity quickly increases when components are combined. For simpler games or in games where the dynamic adaptation can stay isolated, these models can be applicable also in production. The advantage of using Hidden Markov Models as a tool in iterative development of a game, is that the different models' parameters can be set by using other methods like neural nets and even recommender systems if information of the component types or target group player is available. If the parameters in the Hidden Markov Models are close to a real life or game world life, a simulation of the game can reveal its potential for learning and intended effects even before it is tested on real users.

A second advantage of using Hidden Markov Models in prototyping dynamic difficulty adaptation is that it is easy to port the human understandable models to better performing machine learning techniques like neural nets or advanced decision networks (Belahbib *et al.* [21]) in production. By starting out with Hidden Markov Models, it is easier to keep track of what the neural net is actually doing without extensive user testing to validate the model [24]. Learning sniffing on players in all new games has a cold start problem [30]. That is, we do not have a prior on the skill level of the player. There are several ways to deal with this, and if the game is applied in a school setting with many users, recommender systems like collaborative filtering and similarity filtering [30] can be used to probe a prior on a new player for a particular game item. A learning game with difficulty adaptation must keep and actively use stored players' performance measures (Belahbib *et al.* [21]).

Difficulty as a concept in this context has at least two edges. One is linked to the learning part, the other to the gameplay. Vygotskij's and Csikszentmihalyi's proximal zone and float graphs are both good illustrations of the ideas behind designing learning games. However, it is hard to actually translate and measure these abstract concepts as difficulty. For gameplay, performance as points and time used until accomplishment can be used as a measure. In a learning game, such game related performance is also a part of the learning component, although not exactly the same (Wouters *et al.* [31]). Is the game performance then synonymous with what is learned? If so, game performance could replace summative or formative evaluation of players' skills (Wouters *et al.* [31]). We would nevertheless separate the gameplay from the learning when analyzing a learning game (Belahbib *et al.* [21]). However, we suggest that a well designed learning game should pass a test where player performance qualifies as equal to the result of a summative or formative evaluation on learning goals (Wouters *et al.* [31]).

Digital security for youths as a subject matter is diverse and not easily divided into clear levels of difficulty that could mirror the actually perceived difficulty in a game. A level based construction of increasing difficulty, if the subject matter allowed it, would nevertheless only take into account the subject matter (as perceived by the creator), and not the actual competency of the player. Methods taking into account subject matter, gameplay, and players' competence are needed if we want to keep both flow and learning in the proximal zone. Adjusting the level of difficulty, related to both subject matter and gameplay, according to the player's existing and gradually developing skills, using Hidden Markov Models or other machine learning approaches can help accomplish this [24].

In this proof-of-concept case we let the "experts" set the difficulty level related to spotting i. e. predator in a chat-dialogue. The expert rates different chats that are fed into the game, and these rates are used when feeding the player according to their level of mastering. If the quest for the player is to reveal a predator among several chats, we have no guarantee that the player and the expert agree on the level of difficulty to accomplish this task. Using a data oriented approach, we can search for a difficulty adaptation function taking care of different players' learning profiles. Earlier research has mainly relied on user testing, but this approach is both resource demanding, not easily generalizable, and slow. Using a Hidden Markov Model to investigate impact on difficulty can also be extended to find the parameters for Hidden Markov Models that are tuned to different target groups of youths. This is one way to deal with the cold start problems our approach has.

We have not implemented feedback, non-character tutoring, or other mechanisms for directly helping the player in the game. Neither have we put emphasis on the time component, which could be used to increase difficulty as well as used as when learning goals consist of some kind of drill. We have avoided the planning based systems [16] because of their more rigid structure, but also because we are not experts on the subject matter of digital security in this project. Planning based difficulty adaptations requires a better area of knowledge than we have. The next step can be to extend the Bayesian networks with decision networks ([24] p. 528). A decision network has the same nodes as in a Bayesian network with the addition of nodes for action called decision nodes and nodes representing the utility ([24] p. 545). This model can, like Hidden Markov Models, be represented as a directed acyclic graph. In the Unity game mock-up this can be combined with using a GOAP (Goal Action Planner) to carry out the actions activated by the utility functions of a decision network.

Vygotskij's theory of learning has a strong link to the concept of scaffolding that was further elaborated by Bruner. In Bruner's words, scaffolding "…refers to the steps taken to reduce the degrees of freedom in carrying out some task so that the child can concentrate on the difficult skill she is in the process of acquiring" (Bruner 1978 in Slussareff *et al.* [17]). Even if we have not dealt with supporting NPCs and feedback in the game, games can be designed to support the player even without such explicit elements. In fact, we agree with Melero (2011 in Slussareff *et al.* [17]), that a general approach to dynamic difficulty adaptation in itself can be treated as a means of scaffolding because this approach actively assists towards the learning goal.

We implement elements of generative programs ([24] p.520-522), defined by linking random choices to associated probability models (Hidden Markov Model). When doing complete simulation over a game structure made up of probability models, we can get a Gaussian representation of the game that later can be exploited for further analysis that can help tweaking the difficulty adaptation as to accomplish the final learning goals. We have only explored the surface of these possibilities by giving a proof-of-concept on how to use separate clusters of Hidden Markov Models on game components in a learning game mock-up.

Chapter 6

Research Conclusion

We have set up a mock-up game for teaching digital security using three different gameplay ideas. Each of the gameplays are equipped with Hidden Markov Models to guide content generation (quest), emotion, and dynamic difficulty adaptation on non-players' responses in interactive dialogues (mystery), and suggestions for score and penalty sensors input when analysing own and others' social media profiles (socio-quiz).

We suggest graph models for Hidden Markov Models based on Bloom's five taxonomy levels, and starting out with two simple models either choosing players' skills or game items as a starting point for setting up models. These models can be generated randomly to make it easier for users not so familiar with Hidden Markov Models to start experimenting.

Our contribution is only a small demonstration of how Hidden Markov Models can be used in game mock-ups. There is further need to investigate how to nest game components together to analyse and develop the complete game. The toolbox of dynamic Bayesian networks is well equipped to use, and has a great advantage over other machine learning approaches like neural nets when it comes to explain what happens. Other researches have explored how advanced machine learning can actually be used to improve dynamic difficulty adaptation in a production setting.

We used Vygotskij and Bloom's taxonomy as the basic idea to understand learning, however, we did not deep dive into the learning aspects. Nevertheless, we can see the potential in further exploring linking methods for dynamic difficulty adaptation to i. e. Bloom's taxonomy to expand it first in a direction of treating game components as tutors in learning games in line with Vygotskij's scaffolding, and second, in a direction of making mastering of a learning game equal to traditional methods of summative and formative evaluations.

Chapter 7

Development Process

7.1 Choice of Software Development Model

The open nature of our bachelor task allow us to decide what to focus on and how the game should work. We understand that priorities and game ideas could change throughout the project, meaning that a flexible agile development model is more appropriate than a rigid waterfall model.

We first considered Scrum, which is a relatively structured agile model. It requires a product backlog, and each sprint has a fixed length. Daily scrum meetings are held to provide development updates, and a sprint review and a sprint retrospective are held at the end of each sprint.[32] The existence of the product backlog, and the fact that each sprint has a fixed length, imply that priorities should be relatively stable when using the Scrum model. One sprint focuses on one part of the product backlog, meaning that there is a clear goal, which implies that making drastic changes during a sprint could cause problems.

The bachelor task does not have a complete list of requirements, so we have to make them ourselves. We did not have a clear idea about how the game should work from the beginning, so we decided we would experiment with game ideas and make requirements throughout development. This means that we do not have a product backlog. As mentioned, we also know that priorities could change, which means that the sprint process in Scrum will not be the best option for us. The fixed length of the sprints also feels limiting to us in the way we want to work, and our lack of experience makes it difficult to predict how long functionality will take to develop in a sprint.

We realise that the Kanban model does not have these limitations. Kanban does not have strict time limits, the lack of sprints will allow us to change our plans whenever it is appropriate, and the model does not require a backlog of requirements. The Kanban board will allow us to come up with ideas at any time during development, place them on the board, and choose which of the ideas to prioritise next. We still like the idea of frequent meetings from Scrum, which will allow us to keep each other updated and discuss development problems. It also add more structure where Kanban may be somewhat lacking, and it will encourage our group members to work consistently.

For these reasons, the software development model we ultimately choose is a modified Kanban model with frequent meetings.

7.2 Usage of the Model

Our group use GitLab's issue board functionality as a Kanban board. GitLab's issue functionality allow users to create text entries which may represent issues such as tasks, problems, or announcements, which can be categorised by using custom labels. Active issues are in an *open* state, while issues that are no longer relevant can be set to the *closed* state. The issue board functionality features sections where issues can be placed, including an *open* section and a *closed* section, which change the issue's *open* and *closed* states. Additional label sections can be added, and placing an issue in one of these sections will give the issue that label. Our group made an *under development* label, so that tasks actively being worked on can be put in the *under development* section. Pending tasks not being worked on are in the *open* section, and finished tasks are put in the *closed* section.

Our group does not have a strict meeting schedule. Throughout development, our group members will give one another tasks during meetings, which we will have to work on until the next meeting. During each meeting we also schedule the next meeting. The time between each meeting is determined based on how long we think it would take to make meaningful progress to discuss during the next meeting, and what will fit our schedules. Our meetings are usually held two to four days apart.

7.3 Documentation

During development, we comment our code well. The comments help each group member understand another group member's code. Comments are our main form of documentation. Our GitLab repository has a file called `README.md` which is used during development to provide more detailed descriptions about, and instructions for, our individual implementations, when we consider it useful. Our repository also has some additional instruction documents, one of them being a description of our Firestore database structure using JSON.

Chapter 8

Requirements Specification

Our task description does not contain a detailed requirements specification, so we have to make the requirements ourselves. Our group has been using a Kanban-like development model (Chapter 7), so the requirements are made throughout the project. This chapter contains the summary of our software requirements, in order of conception.

8.1 Setting

The game will take place in a three-dimensional representation of a fictional social media platform. The player controls a user on the platform who tries to help other users. Helping others is accomplished through reporting malicious or suspicious users, and talking to and improving relationships with users.

8.2 GUI Interaction

All GUI menu options/buttons and other interactive GUI elements will be clickable using a mouse on PC, and touchable using a mobile smart phone's touch screen. All GUI elements will be in 2D.

8.3 Avatar Creation

All users on the social media platform are visually represented by 3D model avatars. They will have a human-looking form, and all avatars will wear a sweater, trousers (called *pants* in-game), and shoes.

The game will have an avatar creation screen for the player, which will be shown when the game is first started, and when the player chooses to change their avatar's appearance during gameplay, as detailed in Section 8.6 and Section 8.12. There will be options for changing skin tone, hair/headgear type, hair colour, and clothing colours. There will be 5 skin tone options representing a diverse range of realistic skin tones. There will be 4 hair/headgear types; bald, short hair, long hair, and hijab. The *short hair* and *long hair* options will have 5 hair colour options; black, grey, brown, red, and blond. There will be colouring options for each individual clothing item (hijab, sweater, trousers, and shoes), where each colouring option has the same 12 colour palette.

There will be a button which will randomise the player avatar's appearance using the aforementioned options when clicked. Finally, there will be a button which will display a *finish avatar creation* confirmation text box when clicked. The text box will contain a *back* button and a *finish* button. The *back* button will close the text box, and the *finish* button will complete the avatar creation.

8.4 Player Controls

The player will control their character by clicking or tapping on objects in the game. The place on the object that is clicked will be marked, and the player character will smoothly rotate and move towards the point's X and Z position values. The game camera will follow the player's position.

8.5 Progression Structure

The game will consist of several levels. Each level will have a different 3D environment, and will contain NPC-to-NPC and player-to-NPC conversations. The player will be able to report malicious or suspicious users at a reporting terminal, and go through the player's personal user profile. The player will be able to earn points in each level, and the player's score will determine the difficulty of the next level, as an example of dynamic difficulty adjustment.

The door/exit to the next level is opened up when the player sends all of their reports by clicking on a button on the terminal main screen. When the player gets close to the door, a popup text box will appear, asking whether the player wants to go to the next area. The text box will have a *cancel* button and a *confirm* button. Clicking on the *confirm* button will make the level feedback screen appear, which will be detailed in Section 8.9. Clicking on the *cancel* button will close the text box and position the player away from the exit so that the player will not be stuck by constantly getting new *exit level* confirmation popup messages.

The appearances of the player and NPC avatars in all levels will be saved in Firestore, so that when the player leaves a level scene and comes back, all avatar appearances will be the same. The player and all NPCs have their own `user` document in a `users` collection, which will contain appearance data, in addition to other data which will be detailed in Section 8.12.

8.6 Pause Menu

There will be a *pause* button on screen at all times in level scenes, which will open a pause menu window when clicked. The window will have a *resume game* button, a button which leads to the avatar creation scene and allows the player to change their avatar's appearance, and a *quit game* button.

8.7 NPC-to-NPC Conversations

There is conversations between NPCs that the player can observe. A conversation is represented by an object containing two NPC objects facing each other. When the player is close to the conversation object, a *start conversation* button will appear. Clicking on the button will play the conversation. The dialogues from the NPCs will be represented as GUI text messages, and the player goes from one text message to another by clicking on the screen. The text messages will be able to display images.

The conversation text will be stored in Firestore. Each conversation document will have a value for whether the conversation has appeared before, so that it will not appear again in another level; a difficulty value, to ensure only conversations suitable to the current difficulty level will appear; the names of the NPC users in the conversation and the name of the problematic user, so that the correct user will be reported; and values for the number of the level the conversation appears in and the conversation's number in the level, so that the conversation's location data is saved between scenes. Each conversation document will also have a collection of issues from the problematic user, where each `issue` document has a `points` value that is used when evaluating the player's score, and a collection containing the conversation's text box data.

8.8 Reporting Functionality

The game will feature a reporting terminal object where the player can report malicious or suspicious users. The GUI terminal screen hierarchy consists of a main screen, user reporting sub screens, and a sub screen detailing the reports on an individual user. When the player is close to the terminal, a *start terminal* button will appear, which opens the main terminal screen. Each terminal sub screen will have a button which leads back to the main screen. All *delete* buttons in the terminal screens will lead to a deletion confirmation popup message with a *cancel* button and a *confirm* button.

The main screen will have a section containing buttons with the names of all NPC users in the current scene that can be reported, and a section containing a list of reported users. There will be a button which closes the terminal, and a *send* *reports* button which will be detailed later in this section. Clicking on one of the username buttons will start the reporting process for that user.

The first user reporting sub screen will feature the main report issue categories. Each main category has its own sub category screen. Clicking on a sub category option in the sub category screen will lead to a screen asking who to report the issue to. Choosing an option will lead to the final confirmation screen, which contains a summary of the report and a report confirmation button. The summary includes the name of the user that is being reported, the issue they are reported for, and who the issue is reported to. Confirming the report will add a user to the *reported users* list if they are not already there, and the report will be added to the user's list of reports. The user is taken back to the main terminal screen, and the visual list of reported users on the screen is updated. Each username button in the list of reported users is accompanied by a button which will delete the user and all of their reports from the list when clicked.

Clicking on a username in the *reported users* list leads to the user's report data screen, which contains a list of issues the user was reported for and who the issues were reported to. This means that each user can be reported for multiple issues. Each issue must be reported individually. Each reported issue in the list is accompanied by a *delete* button like in the main screen list of reported users, which will delete the reported issue from the user's list, and refresh the report data screen. The report data screen also has a *delete* button outside of the list, which serves the same function as the user's *delete* button in the main screen list, in addition to closing the report data screen and showing the main screen.

When clicked, the *send reports* button on the terminal main screen will calculate the maximum amount of points that can be earned by reporting malicious or suspicious users in the current level, and the amount of points the player has earned. The button will also add feedback messages to a *positive* list and a *negative* list. The exit to the next level will then be activated, and all terminal screens will close.

All sub category issues are saved as documents in a `report issues` collection in Firestore. Each document will contain a collection called `who to report to` with documents of all entities the issue can be reported to. Each entity document has a `points` value containing the amount of points the player can gain by reporting the issue to that entity.

8.9 Result Screen

As mentioned in Section 8.5, when the player exits a level, the result screen, featuring the player's score and feedback messages, will appear. It will display the number of points the player earned in the level and the level's maximum number of obtainable points. A general feedback message is also shown, which is based on the quotient obtained by dividing the player's earned points by the maximum number of points. The negative and positive feedback messages mentioned in Section 8.8 will be displayed in separate lists on the screen. Finally, the feedback screen has an *OK* button, which will take the player to the next level.

8.10 Game Scenarios

As mentioned in Section 3.6, there are three different scenarios implemented in the game. In the first scenario, the player can walk up to any number of the character pairs in the scene and go through the conversation presented by them. Each conversation consists of multiple parts that are displayed sequentially as the player clicks or touches the screen (depending on platform). After going through a conversation, the player can go to the terminal to add a report of who they think is a threat. When the player decides that they have reported everything correctly, they can go to the terminal and move on to the next level by sending the reports and passing through the gate that opens.

In the second scenario, which takes place in the Twinery scene, the player will participate in an interactive conversation with one or more NPCs. The player and NPCs will take turns picking one of a number of options displayed on screen. To make the player able to keep up with the progression of the conversation, the player has to click to progress it both when they and when the AI is choosing. The *MultiSpeechLayout* screen displays all the information necessary for the player to understand what is happening by displaying who is currently picking and what the last pick of the AI was. The choices of the AI are chosen and affected by the choices of the play using the Hidden Markov Model.
The player is also able to run the conversation through simulation by activating the simulation box located on the *DialogueController* script, in this case all choices are made by the AI. The player is able to go through and see the choices the AI made by clicking anywhere on the screen.
Which conversation that is active is manually chosen by dragging an appropriate *.txt* file of their choice into the *DialogueController* script on the *MultiSpeechLayout* object in the *Canvas* object.

In the third scenario (the job scenario), the player will take a look at their own social media profile. By selecting the profile logo, the player arrives at their profile page. Here they will have to go through their posts and remove any that might be damaging to themselves. They will also go through their list of accounts that they follow, and unfollow problematic accounts.

8.11 Player-to-NPC Conversations

The player-to-NPC conversations make use of the external web program Twinery so domain experts can create dialogues that can be imported and run in the Unity Twinery scene.

When the conversation starts, one of two different variations will play.
The first operates like the usual multiple choice dialogue in a game where the player will select one of multiple options and be shown the standard answer to that option before continuing to traverse the dialogue tree. In the second variation, the player and the NPC take turns choosing where to lead the dialogue.

8.12 User Profiles

In addition to the *pause* button that was detailed in Section 8.6, there will be a *profile* button on the screen at all times in level scenes. The button takes the player to their in-game profile pages, where other NPC users' profiles can be accessed as well. All profile sub pages will have a *back* button which takes the player up a level in the page hierarchy (either to the main page or to a page closer to the main page in the hierarchy). All profile pages that display information from Firestore will update the information every time the pages are entered.

The player's user profile is part of the *job scenario*, which is described in Section 8.10. All user profile data will be stored in the `users` collection in Firestore, which was mentioned in Section 8.5.

There will be 8 available profile pictures in the game, and each user's profile picture will be saved in that user's document in Firestore. The correct profile picture for a user will be displayed in all places where profile pictures are shown. NPC users' profile pictures will be randomly chosen.

Each user profile main page will show the user's profile picture and username, and buttons which lead to the user's *following*, *followers*, and *posts* pages. There will also be a *back* button which closes all profile pages, and an *all users* button. On all NPC profile main pages, a follow/unfollow button will be displayed. The player's main page will have a *user settings* button, where the player can change their own settings.

The *all users* page will show all users in the `users` collection, including the player, in a list. Each user in the list will be clickable, and will lead to that user's main profile page.

The *followers* and *following* pages will display the users that the player is followed by or follows, respectively. Each user will have a `followers` collection and a `fol-`

`lowing` collection in their Firestore user document, which will save this information. All users will have a `problematic` boolean value in their user document, and the player will receive negative points to their score (Section 8.9) if they follow a problematic user. The follow/unfollow button on an NPC profile page, as mentioned earlier in this section, will add or remove that NPC user to or from the player's `following` collection. The `followers` collection in the NPC's `user` document will also add or remove the player depending on whether the player follows the NPC. The users that each user follows or is followed by will vary depending on which level the player is currently in, so the documents in those collections have a list of numbers of the levels that they apply to, called `level numbers`.

The *posts* page will show all of the user's posts in the Firestore `posts` collection in the user's document, which belong to the level the player is currently in. NPC users' posts will let the player know which users are problematic and should be unfollowed. The player's *posts* page will have additional functionality and will serve a different purpose. The `post` documents in the player's `posts` collection will have additional `problematic` and `deleted` boolean values. Each of the player's posts will have a *delete/restore* button accompanying it, which will change the post's `deleted` boolean, meaning individual posts can be deleted or restored. The player's *posts* page will also have a *restore all deleted posts* button. The player will receive a negative point for each problematic post that is not deleted (Section 8.9).

The player's *user settings* page will include three options; *change profile picture*, *change username*, and *edit avatar*. The *edit avatar* option will lead the player to the avatar creation scene, like the option in the pause menu (Section 8.6). The *change profile picture* page displays all available profile pictures. Clicking on one will change the player's profile picture to that image, and will take the player to their main page. Finally, the *change username* page will feature an input field where the player can type their new username, and a *save changes* button which changes the username and takes the player to the main screen.

Chapter 9

Technical Design

9.1 Choice of Game Engine and Game Style

We choose the Unity game engine as a development platform as we all have previous experience with it, and because it supports many different devices and video game platforms. We also have decided that the game would be in 3D, as two of our members are more comfortable with, and more interested in, 3D game development.

9.2 Choice of Database

We choose Firebase's Firestore database service to hold all of the game's educational text content. Two of our group members have prior experience with Firestore, and Firebase is primarily used for mobile and web applications, which fits with our primary focus on mobile and PC. Firestore's hierarchical system of documents and document collections is also intuitive to us because of its resemblance to JSON, and will allow for flexibility with the educational text content.

9.3 Firestore Integration in Unity

To integrate Firestore into Unity, the Unity project must be configured to work with either Android or iOS. A project needs to be created on the Firebase Console web site, and the Unity application needs to be registered in the Firebase project. Firestore is integrated into Unity by installing the Firebase package into the Unity project and by using the package in the C# scripts. Firebase and Firestore package API is described in Section 11.1.

9.4 Firestore Database Structure

This section describes our Firestore database structure, but will not explain the usage of the database elements in the game. Explanations of the usage of the

Firestore collections, documents, and document values are in Chapter 11.

Our Firestore database contains three main collections that we use in the game; `conversations`, `report issues`, and `users`.

The `conversations` collection contains documents for every conversation that the player may encounter in the game. There are two conversation types; NPC-to-NPC conversations and player-to-NPC conversations. The NPC-to-NPC documents have the naming scheme `conversationX`, where *X* is the order number, and the player-to-NPC documents have the naming scheme `player-NPC-conversationX`. We use the player-to-NPC documents to store conversation location data and the name of the NPC in the conversation, which we use to retrieve the NPC's avatar appearance data. However, the NPC-to-NPC documents store everything related to their conversations. Both document types have the following values: `already used`, `difficulty`, `level conversation number`, and `level number`. The NPC-to-NPC documents also have the following values: `issue user` (problematic user), `person 1`, and `person 2`. Additionally, the NPC-to-NPC documents have the `issues` and `text boxes` collections. In the `issues` collection, the ID of each document is an issue that exists in the `report issues`, and each `issue` document contains a `points` value. The `text boxes` collection contains documents ordered by their ID numbers, and each document has a `name` value and a `text` value.

The `report issues` collection contains documents for all issues that users can be reported for. Each document has a `who to report to` collection, which contains a document for each entity that the issue can be reported to. Each `entity` document has a `points` value.

The `users` collection contains documents for all users in the game. Each `user` document has the following appearance values: `hair colour`, `hair/headgear`, `hijab colour`, `pants colour`, `shoe colour`, `skin tone`, and `sweater colour`. They also have a `problematic` value and a `profile picture` value. Each document has `followers`, `following`, and `posts` collections. The `followers` and `following` collections have documents for every user that follows the user with the collection, or every user that is followed by the current user. Each `user` document in those collections has a `level numbers` list, containing the numbers of all of the levels where the user follows or is followed by the current `users` collection document. The `posts` collection contains a document for every post that the user has posted. Each document has a `level number` value and a `text` value.

Our project repository contains a document with our database structure represented in JSON. Access to this document may be granted upon request. Contact information is in Appendix A.3.

9.5 Decision to Use Twine

When we decided that we wanted to create a couple of different examples to show our difficulty adjustment model, making a dialogue tree seemed like a pretty logical decision as our project is mainly focused around dialogue. However, just forcing through and making one or more dialogue trees from scratch is not really an option as making normal 1 dimensional dialogues already require more than enough work. Especially so since one of the focuses for our task is to make the learning content changeable, finding an external and easily accessible solution that we can use to construct the dialogue trees is therefore optimal.

We searched for and found a couple of different options like Twinery and Fungus, both great tools for making interactive dialogue. Fungus has the advantage of being a Unity extension making it very easy to incorporate into the game. Twinery on the other hand holds the advantage in accessibility, only requiring internet connection as opposed to Fungus which requires you to have unity installed. Also unlike Twinery, Fungus will not have the same ability to just format out the dialogue to a text file without us having to make some additional functionality. The main advantage of Fungus being native to unity is not too great either as we had found a finished solution for importing Twinery to Unity.

9.6 Twine Integration in Unity

For incorporating Twinery in Unity, we found a guide by Ventures [33]. By converting Twinery to *txt* files and parsing the Twinery tree directly into Unity, we have an optional text based database for our dialogues. However, integrating this text database with our already existing system for NPC-to-NPC conversations in Firestore has proven to be challenging. One of the issues lies in the functionality of clicking *response* buttons in the Twinery implementation. Because Twinery comprises of tuples of answer text and response text, it is harder to fully disconnect them into two separately modulated (by Hidden Markov Models) dialogues. We have a trade-off by using the simple text database represented by the *txt* files and a much more flexible storage in Firestore. Nevertheless, we parsed Twinery into graph nodes and could have split the tuple by storing in Firestore. We decided not to do that, but kept both systems separated, mainly to keep the dialogue system provided by Twinery as close to its original as possible. The reasons for this is that Twinery has a very good web interface for testing dialogues as you write them.

Another issue that has surfaced a couple of times in the development of the Twinery scenario is where to store data and what data to store. The solution here has been to store information that only needs to show up once, like the number of NPCs, in the tags of the first node. Information that is not the same for each node, like whether this node is controlled by an NPC or the player, is stored in the tags of each node.

Chapter 10

Graphic Design

Graphic design has not been a primary focus in our bachelor project, but some of our graphic design decisions may be helpful to our client's development of an educational application.

10.1 Art Style and General Design Decisions

None of our group members are skilled artists, so the graphic design and art style in our proof-of-concept game is simple. 3D objects and environments have simple designs and colourful textures, giving the game a cartoon-like appearance. The game uses Unity's default lighting and shadows for mobile games, which adds some realism to the cartoon-like style.

All of the text in our game uses the Liberation Sans font, which is the default font in the TextMeshPro package. Most of the clickable GUI menu elements use TextMeshPro's default button template. All list elements, such as usernames, report issues, posts, and feedback messages, are displayed in a Unity Scroll View, so that all of the necessary information can fit on one screen, which is particularly important for devices with small screens. All `delete` buttons are red to alert the player that the action may have significant consequences, which conforms with the common understanding that the colour red often symbolises the words *danger*, *stop*, and *warning*. A confirmation text box appears whenever the player is about to delete something, to prevent accidental deletions.

10.2 Avatars and the Avatar Creator

We want the avatars to be inclusive in their design. The user avatars have humanoid designs that are simple enough to appear gender neutral. As opposed to an advanced character creator for realistic looking human characters, this is a relatively easy solution to make the avatars more inclusive. Our avatar creator allows for customisation of skin tone [1] , hair type (bald, short, or long), hijab, hair colour [2] , and clothing colours.

The background of the avatar creator features two shades of blue to fit with the theme colour of our fictional social media platform. Options for skin tone and hair/headgear are on page 1 of the creator, and options for the sweater, trousers, and shoes are on page 2. Separating these options makes the interface simpler for the player, and will prevent the screen from appearing cluttered, which will be a problem for small devices in particular.

10.3 Level Environments

The game takes place in a three-dimensional representation of a social media platform. Borders and walls in each level consist of rectangular prisms of various shapes and sizes, which may represent pixels. This gives the environment a more "digital" appearance. Level environment objects are coloured in various shades of blue, which is the theme colour of our social media platform. The environment background is white, and may represent the background of text messages or the areas between different windows or containers in a social media graphical layout. These design choices will help the player understand that they are in a social media platform, which could make the player's gameplay experience more immersive.

Interactions between users on the social media platform are represented as in-person interactions between the users' avatars in the level environment. This makes relatively impersonal text conversations seem more personal, and may represent the real, personal effects conversations on the Internet may have on people. The presence of what appears to be other people also makes the environment feel more alive and immersive.

Another example of graphic design in the level environments are the level exits. The exits light up when they are open (Section 11.3), which helps the player understand that they need to go there. The terminal in each level (Section 8.8) is represented by a blue 3D model of a computer monitor.

[1]Skin tones retrieved from: `https://www.color-hex.com/color-palette/36175`
[2]Hair colours retrieved from:
`https://www.color-hex.com/color-palette/22939`
`https://www.color-hex.com/color-palette/4614`
`https://www.color-hex.com/color-palette/85503`
`https://www.color-hex.com/color-palette/102849`
`https://www.color-hex.com/color-palette/91001`

10.4 Social Media Conversations

The first scenario is designed to mimic some sort of social media or messaging service, and the *start conversation* button is designed to look like a speech bubble. The *MultiSpeechLayout* is designed to look like a mobile version of a social media messaging page. The sending of images reinforces this look.

For the conversation images, we found images in the public domain or with similar licences online [3] , and altered them so that the player will not see them clearly, which makes the images appear to be less appropriate than they really are.

10.5 Reporting Terminal

The *start terminal* button (Section 8.8) uses a custom image of a white monitor symbol with an exclamation mark on its screen, which is on a blue background. The symbol looks like the terminal 3D model (Section 10.3), and the exclamation mark symbolises that actions can be taken by using the terminal.

The reporting terminal pages use a custom light blue background, which fits with the blue theme of the social media platform, meaning that the terminal is considered to be a part of the platform. The main screen shows the names of the users that can be reported as buttons, and the list of reported users in a Scroll View (Section 10.1).

When reporting users, the player navigates through the pages by clicking on default TextMeshPro buttons (Section 10.1). Scroll Views are used on main category pages that have many sub categories. The report data screen for an individual user features a Scroll View of the individual reports on the user.

List elements that can be deleted individually, such as the users in the list of reported users, or the reports on an individual user, have a *delete* button accompanying them (Section 8.8). The buttons use a custom *delete* image featuring a white trash can symbol on top of a red background, which should make it obvious to the player what it does.

[3]Conversation image sources:
`https://pixabay.com/no/photos/dame-dans-nattklubb-disco-jente-3492751/`
(Pixabay License)
`https://stock.adobe.com/no/images/cropped-view-of-sexy-man-taking-off-jeans-while-lying-on-bed/301208447` (Adobe Stock Standard License)

10.6 Profile Pages

The button which takes the player to their profile pages (Section 8.12) uses a "default profile picture" image [4] , which is inspired by default profile pictures in social media applications. Default profile pictures should be recognisable to most people, and they would most likely attribute it to a profile.

All profile pages have the same background as the terminal pages (Section 10.5), and the pages are navigated by using default TextMeshPro buttons (Section 10.1). All Scroll View list element have a darker blue colour compared to the background to make them stand out.

The main profile page features the user's profile picture, which can be one out of 8 options (Section 8.12). One of the options is the default profile picture, and the others are simple custom recoloured versions of it. The *all users*, *following*, and *followers* pages (Section 8.12) all use Scroll Views with username list elements, each featuring the users' correct profile pictures. The *posts* page contains a Scroll View with *post* elements. As mentioned in Section 8.12, the *post* elements on the player's *posts* page have a *delete* or *restore* button accompanying them. The *delete* button uses the same trash can symbol image as on the terminal's main page and user report information page (Section 10.5). The *restore* button is green and features an arrow pointing away from a trash can symbol, the green indicating that it is a safe action, and the arrow indicating that something is taken out of the trash can.

10.7 Pause Menu, Popup Text Boxes, and Result Screen

The pause menu and all popup text boxes have a default light grey *panel* Unity object as their background. The result screen has a grey background that is a recolouring of the background we use in the terminal and profile pages. The grey backgrounds of the popup text boxes make them stand out from the blue social media theme, which lets the player know that they are important and require attention. The grey backgrounds of the pause menu, popup text boxes, and result screen also differentiate them from the theme of the terminal and profile pages, and lets the player know that they should not be considered as parts of the social media platform.

[4]Retrieved from: `https://pixabay.com/vectors/blank-profile-picture-mystery-man-973460/` (Pixabay License)

Chapter 11

Implementation

This chapter describes everything that our group implemented, which follows the requirements specification in Chapter 8. The descriptions of the implementations are quite detailed, as it is primarily meant for our client, so it is not necessary for the grader of this report to read everything here.

11.1 Programming and Implementation Tools

In our Unity project scripts, we use the C# programming language, which is Unity's scripting language[1]. We use the Visual Studio development environment to write the scripts. We use the UnityEngine C# package to access Unity specific APIs. Retrieving, adding, and updating data in Firestore is accomplished by using the Firebase C# package. Firestore collections and documents are accessed by using database references and dot notations where e.g. accessing a collection can be written as demonstrated by the following code listing.

Code listing 11.1: Getting collection and document references

```
FirebaseFirestore db = FirebaseFirestore.DefaultInstance;
CollectionReference collectionRef =
    db.Collection("collection_ID")
    .Document("document_ID")
    .Collection("collection_ID");
```

Adding a document is accomplished by using the `SetAsync` function, and `UpdateAsync` is used to update a document. Looping through documents in a collection can be accomplished by getting a collection/document snapshot asynchronously, and creating a task which uses the snapshot and runs on the main thread, as demonstrated in the following code listing.

Code listing 11.2: Looping through Firestore documents

```
FirebaseFirestore db = FirebaseFirestore.DefaultInstance;
CollectionReference usersRef = db.Collection("users");
usersRef.GetSnapshotAsync().ContinueWithOnMainThread(
```

[1]https://unity3d.com/learning-c-sharp-in-unity-for-beginners

```
    task =>
    {
        QuerySnapshot snapshot = task.Result;
        foreach (DocumentSnapshot document in snapshot.Documents)
        {
            DocumentReference userRef = usersRef.Document(document.Id);
            userRef.UpdateAsync("hair_color", "");
        }
    }
);
```

11.2 General Implementation Practices

Almost all variables are set in the scripts, and all functions that are activated by buttons are set to the button click listeners in scripts, rather than in the Inspector. This prevents us from having to set all variables and functions in the Inspector every time we are creating a new level. The only variables that are not set in scripts are objects that cannot be found in the Hierarchy window, such as Prefabs (not Prefab instances), Materials, and sprites.

11.3 3D Models and Environments

All of our Unity scenes have default directional light, and each level exit has an added white point light that is activated when the exit opens (Section 10.3). Level environments in our game mostly consist of simple three-dimensional geometric shapes. The simplest and most basic shapes used for the environments are rectangular prisms and two-dimensional planes (in 3D space), which are default Unity shapes. We used the ProBuilder Unity package to create simple shapes that are not default shapes in Unity, such as a ramp. More advanced 3D models, such as the in-game avatars and the reporting terminal, were created in Blender. Avatars consist of several parts in a hierarchy, which means that each individual part can have its own properties.

11.4 GUI Menus and Interaction

All GUI elements in each scene in our game are children of a Unity Canvas object with the *Screen Space - Overlay* render mode. The *overlay* mode places GUI elements on the screen, rendered on top of the game scene[2], in a completely two-dimensional space. The draw order of the elements is in the same order as the objects in the Unity Hierarchy window[2].

GUI menu navigation in our game utilises the draw order, so that elements higher up in the menu hierarchy are drawn after (on top of) elements lower in the hierarchy. As an example, this means that when the player is on page 1 and clicks on a button showing page 2, the button only shows page 2 without needing to hide page 1, and when the player clicks on the *back* button on page 2, the button only needs to hide page 2.

All Unity GUI buttons are by default clickable with both a computer mouse and a touch screen, which complies with the requirement stated in Section 8.2.

11.5 Static Variables

There is a script with a static class called `StaticVariables`, which applies to all scenes and holds static variables that are persistent throughout a gameplay session. It contains data such as a `gameRunning` boolean variable, which is `true` when the player is in a level scene and the game is not paused, a variable containing the number of the level that the player is currently in (`currentLevelNumber`), a variable containing the current difficulty level (`currentDifficultyLevel`), and a variable containing the player's score percentage from the previous level (`last-PlayerScorePercentage`) (Sections 11.13 and 11.19).

11.6 Game Manager and Pause Menu

There is a Game Manager object and script in all scenes in the game. It sets click listeners for the pause button and the *to profile* button which are always on the screen in level scenes, and it handles the pause menu, loading level scenes by calling the script's `LoadLevel` function, and game states such as whether the game is running. The Game Manager handles the *game running* state by updating the `gameRunning` variable in `StaticVariables` (Section 11.5).

The pause menu has options for resuming the game by setting `gameRunning` to `true`, going to the avatar creation scene (by calling the `GoToAvatarCreation` function), and quitting the game.

11.7 Firebase and Firestore Initialisation

There is an object and a script in the avatar creation scene called `FirebaseInit`, which only runs at the start of a game session. This object shows an error message if not all Firebase dependencies could be resolved. It also resets Firestore information that is exclusive to one game session, such as the conversations' `already used` value and their location data (Section 11.13), and the avatar appearance data. It also randomises the profile pictures of all NPC users in the game, by choosing a random *colour* string from a selection, and setting it in the `user` document's `profile picture` value. (Section 11.18).

11.8 Avatar Creator

The avatar creation scene's most important object and script is the Avatar Creation Controller. This object and script is also in level scenes so that some of its functions can be accessed there. The script has Material variables for skin tones, hair colours, and clothing colours that are set in the Inspector, which are used to colour the avatar. The script assigns all avatar parts (Section 11.3) to variables, in part by using a function called `SetObjectsToBeColored` which is also used elsewhere (Section 11.11). The avatar's hair and headgear objects are in separate variables. Parts containing skin tones, and parts that make up the sweater, trouser, and shoe clothing items, are in four separate lists, so that all skin parts get the same skin tone, and all parts of one clothing item get the same colour.

The avatar creator has buttons for which hair type or headgear the avatar should have, and clicking on a button will enable the object associated with the button, and disable all other hair/headgear objects. The creator also has buttons which change skin tone, hair colour, and clothing colours. Skin, hair, and the four clothing items including headgear, all have separate sets of colour buttons, which are added to separate lists. Each tone/colour button in the avatar creation scene has a name indicating the parts that the button will colour, and which colour the parts will have. Clicking on a button will then colour the correct parts with the correct material.

The function which generates random avatars, called `CreateRandomAvatar`, finds random colours by generating random indices in the tone/colour arrays, and it finds a random hair/headgear type by generating an integer between 1 and 4 (there are four hair/headgear options). It then uses the Avatar Creation Controller's appearance setting functions to create the avatar.

When the player presses the *finished creating* button, the avatar appearance is saved in the player's `user` document in Firestore, and the first or current level scene is loaded, depending on when in the play session the player is in the avatar creation scene. The function for saving avatar appearances, called `SaveAvatarAppearance`, is also used outside of the avatar creator script (Section 11.11), and it saves values for skin tone level, hair colour, and clothing colours in Firestore (mentioned in Section 8.5). There is also a function which does the inverse and sets the avatar appearance based on the values in Firestore, by using the Avatar Creation Controller's appearance setting functions. This function is called `SetAvatarAppearance`, and is used exclusively outside of the avatar creation script, which is described in Section 11.11.

11.9 Camera

The in-game camera has a script which updates the camera's position depending on the player object's position, making it follow the player at all times.

11.10 Player Movement

The player object has a script called `PlayerController`, which, among other things, controls the player avatar's movement. The player can click on an object in the game, and the player avatar will smoothly rotate and move towards the point on the object that was clicked. It detects both mouse and touch clicks, and both methods call the movement function and sets the point on the screen that was clicked as a parameter. The movement function uses the `ScreenPointToRay` function, which casts a ray from a point on the screen in the direction the in-game camera is facing. When the ray hits an object, the point on the object that was hit is saved in a variable called `targetPoint`. The point's X and Z position values are then used in two `MoveTowards` function calls, and the final movement vector is used in the `MovePosition` function, which smoothly moves the player towards the point at a constant specified speed. The Y value is not used, so that the player will not be able to fly. The player movement functionality is shown in the following code listing.

Code listing 11.3: Smooth movement towards a point

```
// Move player rigidbody toward the point on the ground
// that is tapped/clicked on, in the X and Z directions
Vector3 movePosition = transform.position;
movePosition.x =
    Mathf.MoveTowards(
        transform.position.x, targetPoint.x, walkSpeed * Time.deltaTime
    );
movePosition.z =
    Mathf.MoveTowards(
        transform.position.z, targetPoint.z, walkSpeed * Time.deltaTime
    );
playerRb.MovePosition(movePosition);
```

For rotation, the vector between the player's position and the target position is found, and the player avatar's forward direction is gradually rotated towards the point at a specified speed using the `RotateTowards` and `LookRotation` functions, as shown in the following code listing.

Code listing 11.4: Smooth rotation towards a point

```
// Get target point with the player's Y position, so that the player
// always stands vertically when rotating towards the target
Vector3 targetPointAtPlayerY =
    new Vector3(targetPoint.x, transform.position.y, targetPoint.z);

// Find direction to the target point from the player's current position
Vector3 targetDirection = targetPointAtPlayerY - transform.position;
```

```
// Rotate the player's forward vector towards the
// target direction by one step per frame
float rotationStep = rotationSpeed * Time.deltaTime;
Vector3 newDirection =
    Vector3.RotateTowards(transform.forward, targetDirection, rotationStep, 0.0f);

// Rotate the player one step closer to the target direction per frame
transform.rotation = Quaternion.LookRotation(newDirection);
```

The point on the object that is clicked is marked by a symbol object called `Marker` which has its own script. Its position is always set to the `targetPoint` variable.

11.11 Setting Player and NPC Avatar Appearances

As in the Avatar Creation Controller, the scripts that set the avatar appearance for the player avatar and the NPC avatars, `PlayerController` and `SpawnConversationAvatars` respectively, have variables for avatar parts. They both use functions from the Avatar Creation Controller (Section 11.8). The scripts use the `SetObjectsToBeColored` function to add avatar parts to their respective lists. In the Player Controller, the avatar appearance is then set using the `SetAvatarAppearance` function with the player's appearance data from Firestore.

The `SpawnConversationAvatars` script is used for both NPC-to-NPC and player-to-NPC conversation objects. The user documents which store the avatars' appearance data (Section 11.8) are found by using the usernames in the Firestore `conversation` document that is allocated to each conversation object by the Conversation Allocator (Section 11.13). Each of the two conversation types has its own appearance setting function. In both functions, if the avatars have appearance data saved in Firestore, the `SetAvatarAppearance` function is used, otherwise random avatars are created by using the `CreateRandomAvatar` function, and the appearance data is then saved to Firestore with the `SaveAvatarAppearance` function.

11.12 Hidden Markov Model

11.12.1 Hidden Markov Models Equations

Operations on a Hidden Markov Model comprise filtering, prediction, smoothing, most likely explanation and learning [24]. Filtering is the process where a new evidence is provided and a new estimated prior is calculated. Prediction is based on previous evidence, but at the moment a prediction is given, no evidence is provided contrary to filtering. Both filtering and prediction are made by equation 11.1 outlined below [24]. Smoothing provides a posterior distribution over a past state given all evidence up to present. Smoothing is done by combining equation 11.1 and 11.2, the forward and backward passes [24]. The most likely explanation is the sequence of events that reach the maximum probability. The Viterbi algorithm can be used to find this sequence. The recursive step of the Viterbi algorithm is outlined in equation 11.3. We have used the matrix implementation of forward and backward explicitly outlined in equation 11.4 and 11.5. They are equal to 11.1 and 11.2.

Learning of a dynamic Bayesian network as Hidden Markov Model is a by-product of inference providing estimates of what transitions actually occurred and what states generated the readings. These estimates can be used to learn the model i.e. through the expectation-maximisation algorithm [24].

Hidden Markov Equations [24] (14.5,14.9,14.11,14.12,14.13)

$$P(X_{t+1}|e_{1:t+1}) = \alpha P(e_{t+1}|X_{t+1})\Sigma_{x_t} P(X_{t+1}|x_t)P(x_t|e_{1:t}) \tag{11.1}$$

$$P(e_{k+1:t}|X_k) = \Sigma_{x_{k+1}} P(e_{k+1}|x_{k+1})P(e_{k+2}|x_{k+1})P(x_{k+1}|X_k) \tag{11.2}$$

$$m_{1:t+1} = P(e_{t+1}|X_{t+1}) max_{x_t} P(X_{t+1}|x_t) max_{x_{1:t-1}} P(x_{1:t-1}, x_t, e_{1:t}) \tag{11.3}$$

$$f_{1:t+1} = \alpha O_{t+1} T^T f_{1:t} \tag{11.4}$$

$$b_{k+1:t} = T O_{k+1} b_{k+2:t} \tag{11.5}$$

We have included online functions building on updating next based on previous (recursion) that does not depend on inverse matrices. There is a recursive smoothing algorithm, but the online version include inversions, so our implementation does not fully take advantage of combining forward and backward passes. To find the best models and train them to get relevant parameters, we suggest using an external python environment or use Bayesian network software, and then include found parameters in the Hidden Markov Models in the Unity implementation.

11.12.2 Implementation of Hidden Markov Models

We start with a small Hidden Markov Model class placed on the player and NPCs and/or game components. We can use Bloom's taxonomy to operationalise learning. Bloom's theory assumes a sequential order from the lowest to the highest level [28]. This is not necessarily the case in real life learning, as i.e. Vygotskij has suggested with his scaffolding theories [17]. As a starting point this is not important, as the Hidden Markov Model should be tweaked to fit game components and learning goal. The outline for the transition model (row oriented) for the network in figure 11.1 will look like this:

$P(X_t|x_{t-1}) =$

state	*recite*	*combine*	*analyse*	*critique*	*create*	*sum*
recite						1
combine						1
analyse						1
critique						1
create						1

The columns represent the state at time t. The rows represent the prior state at time t-1. The conditional variable, the prior, is always summing up to 1. .

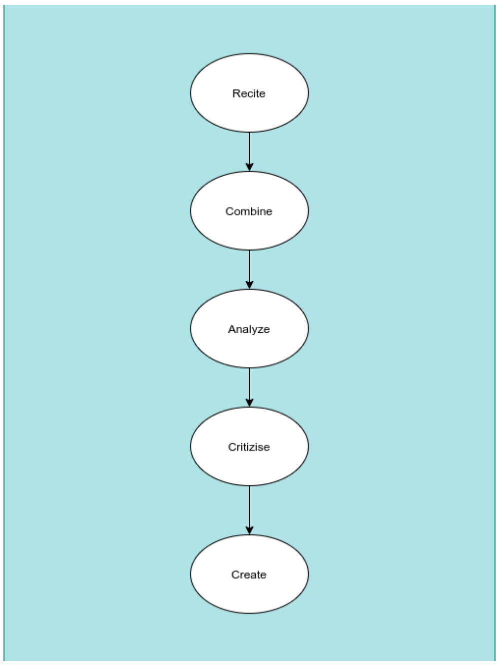

Figure 11.1: A Bayesian net representation of Bloom's taxonomy[34]

The code for making and manipulating Hidden Markov Models is ported from python development making all functions without any other libraries than Numpy. Functions are first tested in python and then ported to C-sharp. To handle odd shaped matrices, simple functions for matrix multiplication and transposing is provided. This is definitely not a viable long term solution. Unity have an interface for running python 2 code, however, that seemed a bit waste as python 3 now is the standard for all modern machine learning python libraries. In the future, if Unity makes python 3 interface available, the users should decide to change to this. Currently the Hidden Markov Models in the Unity game mock-up can be randomly generated and run online either with random settings or user defined parameter settings. This suffice for both running online dynamic difficulty adaptations and for setting up simulations. Currently the dialogue mystery gameplay is set up to run completely simulated. The mechanism used is probability weighting, calculated in real time. To run large simulations we suggest doing a pre-sampling and using this to feed the next actions.

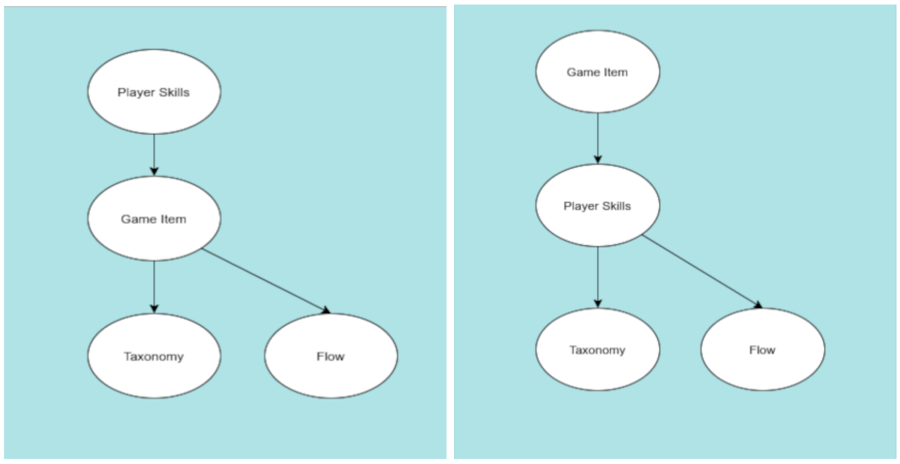

Figure 11.2: Bayesian network graph model for difficulty adaptation with taxonomy and flow as targets. Figure a) starts from player and figure b) starts from game item. The choice of model depends on purpose and real life causal dependencies.

11.12.3 Overview of Mock-ups and HMM

Component	Quest	Mystery	Socio-Quiz
Player's Profile	y	y	y
Content Generation	y	n	y
Emotions	n	y	n
Points	y	n	y
Punishment	n	n	y
Randomness	y	y	y
NPC Manipulations	n	y	n
Simulation	n	y	n

11.12.4 Setting up the Hidden Markov Model - an example

In order to set up a Hidden Markov Model on any game component, you need to create the model. Three components are needed. A prior, a transition model and a sensor model. A prior is just your guess that the starting state is at a certain state or has the possibility of being in a certain state. If you do not know, you can start out with equal probability on all states. If you have two states, that is 0.5 on each. If you have 4, 0.25 on each.

Setting up the transition table i.e. for a player´s learning using the taxonomy you need a 5 by 5 matrix. Either you fill in you data or use the random generation function.

$P(X_t|x_{t-1}) =$

Transition	recite	combine	analyse	critique	create
recite					
combine					
analyse					
critique					
create					

The sensor model comprise applicable measures to confirm a particular state or transition to or from a state. Even if states have to be discrete, the sensor variables values can be both discrete or continuous. The sensor model i a K by N shaped matrix where N is number of states (here 5) and K is the number of sensor variables (flexible).

$P(e_{k+1}|x_{k+1}) =$

Sensor	recite	combine	analyse	critique	create
sensor1					
sensor2					
sensor3					
sensor4					

Set the prior as a 5 by 1 array, with values either at your choice, random or evenly distributed. You are now good to go with running your model.

11.13 Conversation Allocator

The Conversation Allocator object and script is in all level scenes, and allocates conversation data from Firestore to conversation objects in the scene. It finds NPC-to-NPC and player-to-NPC conversation objects in the scene and places them in two separate lists. All conversation objects and the terminal (Section 11.17) are set to inactive, and will be activated once all conversations have been allocated.

The script has a boolean variable called `levelHasSavedConversations`, which is `true` if conversation data associated with the current level exists in Firestore. The script tries to find existing conversations by using a function called `FindAndAllocateSavedConversations`, which loops through all Firestore `conversation` documents, and uses the `already used` value and the location data in each document. If the conversation has been used and the location is in the current level, `levelHasSavedConversations` is set to true. The function then checks the conversation type, and for each of the two types, the same function but with different parameters is called. The function loops through the conversation object list, and then in the loop checks the Firestore conversation number. If the number matches with the current index in the list, the conversation data will be allocated to the object. `FindAndAllocateSavedConversations` allocates conversation data to all objects of both types, by assigning the Firestore conversations' document IDs to variables in scripts belonging to the conversation objects. The variable will be assigned in the `SpawnConversationAvatars` script for both conversation types (Section 11.11), and it will also be assigned in the `ConversationController` script belonging to the NPC-to-NPC conversation objects (Section 11.15).

If the `levelHasSavedConversations` boolean is `false`, new conversations will be found and allocated with the `FindAndAllocateNewConversations` function. The function creates two lists of eligible `conversation` documents, one for each conversation type. The function calls another function which uses a Hidden Markov Model (Section 11.12) to determine the difficulty of each new conversation that will appear in the level, if the player has a score value from the previous level (Section 11.19). The score value and the current difficulty level are saved in `StaticVariables` (Section 11.5). The player score is used to determine the difficulty of each conversation in the next level, as a method of dynamic difficulty adjustment. The function loops through all Firestore `conversation` documents, and if the document's `already used` value is `false` and its difficulty matches with the current difficulty level in `StaticVariables`, the document will be added to the correct list of eligible conversations. After the loop is done, two lists containing all indices in each `eligible conversations` list are created, as shown in the following code listing.

Code listing 11.5: Adding list indices to new lists

```
// All indices in the lists of eligible conversations are added to separate
// index lists, used to guarantee different conversation data for all
// conversation game objects
List<int> availablePlayerToNPCIndices = new List<int>();
List<int> availableNPCToNPCIndices = new List<int>();
for (int i = 0; i < newEligiblePlayerToNPCConversations.Count; i++)
{
    availablePlayerToNPCIndices.Add(i);
}
for (int i = 0; i < newEligibleNPCToNPCConversations.Count; i++)
{
    availableNPCToNPCIndices.Add(i);
}
```

Both conversation object lists are then looped through in two separate calls to the same function, with different parameters. For NPC-to-NPC conversation objects, the document ID is also added to their Conversation Controllers as with `FindAndAllocateSavedConversations`. In each call to the function, an eligible conversation is randomly chosen by finding a random index in the correct index list, removing that index from the list so that it cannot be used again, and using the index to get the document ID from the correct list of eligible conversations. This is shown in the following code listing.

Code listing 11.6: Retrieving a random index from the list

```
// Finds a random index from the list of available indices,
// and then removes it from the list, so that it
// cannot be used again
int uniqueRandEligibleConvIndex =
    availableIndicesList[Random.Range(0, availableNPCToNPCIndices.Count)];
availableIndicesList.Remove(uniqueRandEligibleConvIndex);

// Gets the document ID from the eligible
// conversation with the random index
string conversationDocumentID =
    eligibleConversations[uniqueRandEligibleConvIndex].Id;
```

The document ID is then allocated to the conversation object scripts as mentioned. The `conversation` document's `already used` value is set to `true`, and the document's `level number` and `conversation number` values are set.

11.14 Conversation Objects' Detection of the Player

The `SpawnConversationAvatars` script, which is attached to all conversation objects, also detects the player object and displays the *start conversation* button, according to the requirement in Section 8.7. Each conversation object has a *start conversation* button associated with it. In `SpawnConversationAvatars`, the button is found by using the conversation object's name, which is a part of the button's name. When clicked, if the object is an NPC-to-NPC conversation object, the button calls the `StartConversation` function in the object's Conversation Controller, which is detailed in Section 11.15. The *start conversation* button will also appear when the player is close to the player-to-NPC conversation objects, but at the time of writing, the player-to-NPC conversations do not start when clicking on the button. An example of a player-to-NPC conversations can be played separately in the `Twinery` scene, as mentioned in Section 8.11.

The script's `Update` function calls a function called `DetectPlayer` every frame, which uses Unity's `Physics.OverlapSphere` function. The function detects objects which enter a sphere with a certain radius around the conversation object's position. If the object is the player, a `playerDetected` boolean is set to `true`. An `if` statement activates the *start conversation* button if the boolean is `true` and the Conversation Controller's `conversationPlaying` boolean (Section 11.15) is `false`, and disables the button otherwise.

11.15 NPC-to-NPC Conversations

Once the Conversation Allocator allocates the conversation IDs to the conversation objects, the objects are activated (Section 11.13). The script called `ConversationController` is attached to all NPC-to-NPC conversation objects, and handles the playing and display of the conversation in the object the script is attached to. It has a `GetTextBoxData` function, where the conversation ID allocated to the conversation object is used to retrieve the conversation's text box data from Firestore. The retrieved text box data is saved in a list of `TextBox` struct objects, where each object has values for the name of the user who is talking, and the text in the user's text box.

To display the conversation in an understandable manner that gives the player the possibility of looking at more than just one entry at the time, we make use of three object collections. They are identical in all aspects except their location, where their y value is different hence the name top, middle and bottom. They each consist of a background(Image), which is the parent, two texts, a name and content texts and an image. When a conversation is started you will only see the top object and the two lower ones are disabled to simulate the visuals of some messaging software. In that box the name of the character speaking will be shown at the top and either an text or an image will be shown. When the player continues the dialogue, the middle box will be added next and lastly the bottom box. After all boxes are shown new content will be shown in the bottom box while the previous content of the bottom and middle box will be moved up. The operations take place in a series of if elses in the IEnumerator function `PlayConversation`. the ifs check whether or not a given box has been activated or not to determine which step to do.

The *start conversation* button, which appears when the player gets close to the conversation object (Section 11.14), calls the Conversation Controller's `StartConversation` function. The function starts a coroutine called `PlayConversation`. In the coroutine, the boolean `conversationPlaying` is set to `true`, the game is paused so the player character will not be able to move, and the conversation starts. The coroutine loops through all text boxes in the `TextBox` list. In each iteration of the loop, there is a series of `if/else` statements which display the correct top, middle, and bottom text box GUI elements, which are updated with the text from the `TextBox` struct object with the current index in the loop. After all of the `if/else` statements, the coroutine starts another coroutine called `WaitForPlayerClick`, which waits for the player to click or tap on the screen before stopping. After the player has clicked on the screen, the loop in `PlayConversation` starts the next iteration. Once the loop is done, the game is resumed, the `conversationPlaying` boolean is set to `false`, and a `conversationPlayed` boolean is set to `true` for the rest of the game session.

In each `if/else` statement in the loop in `PlayConversation`, there is a call to a function called `ShowTextOrImage`. The function uses the text in the current Firestore text box document to determine whether to display the text box text or an image. Text boxes displaying images complies with the requirement in Section 8.7. If the text does not start with the `<image>` tag, the text is displayed in the GUI text box. Otherwise, the image's file name is taken from the text box text, a list of images from the Profile Controller (Section 11.18) is looped through, and the correct image is found and set to the text box image's sprite. The image is then enabled.

In the `ShowTextOrImage` function, if the conversation's `conversationPlayed` boolean is `false`, a function called `AddImageToList` is called. The function adds the image that was found in `ShowTextOrImage` to a list of users and the images they have sent in the Profile Controller script (Section 11.18). The list consists of `UserImages` struct objects, each of which contains the name of the user that sent the images, and a list of image strings for the images that the user has sent. `AddImageToList` checks to see whether the user that sent the image exists in the list, and if not, creates a new `UserImages` struct object for the user and adds the image string, and then adds the struct object to the `UserImages` list. If the user exists in the `UserImages` list, the function checks whether the image from `ShowTextOrImage` is in the user's image list. If not, the image string is added to the user's list.

11.16 Player-to-NPC Conversations

The player-to-NPC conversations mainly take place in three script files; `DialogueObject`, `DialogueController` and `DialogueViewer`. `DialogueObject` is practically a container class. It contains the meta data about the dialogue tree and the dialogue tree itself in the form of a dictionary that stores nodes, the class that stores the data about each individual dialogue step. The class has a constructor where it takes in a *TextAsset*, the Twinery dialogue tree, and runs it through a parse function that extracts all data. The `DialogueController` is the script that works with the `DialogueObject`. It is this script that holds the node pointer that points to the currently active node and that updates it whenever `OnNodeSelected` is called.

the `DialogueController` also is where the Hidden Markov Model functionality for the Twinery scenario is located. Finally, there is the `DialogueViewer` which controls when and how things are shown.

Before running the scene, you can choose between two versions, the normal version and the simulation version. You can choose this by using the *Simulation On* checkbox located on the `DialogueController` script in *MultiSpeechLayout* in the Canvas.

The simulation runs on a different track than the normal player-to-NPC conversation. It does not disable some GUI elements and runs recursively, calling `OnNodeSelected` which calls `DialogueController`'s `ChooseResponse` which calls `OnNodeEntered`. This process loops through all the nodes in the path the AI chooses until an end node is reached, the variable `pathLength` is incremented each time.

After having finished the recursive run, the update function starts. It checks if the player has clicked each frame and if so, it goes backwards through the node tree using `DialogueController`'s `curNode` and the `pathLength` to backtrack through the path chosen by the AI and to show the path at the player's tempo. If the Player has clicked the GUI elements update to show the relevant information.

The normal player-to-NPC conversations operate very similarly to Section 11.15 in the way that the player has to approach the character to enable the `StartConversation` button and making use of asynchronous functions, though the asynchronous function is not responsible for controlling the GUI elements. Through the use of the tags 'Player' or 'NPC'(and a number from 1 through the number of NPCs) when making the Twinery document, the creator of the document can decide which nodes are controlled by NPCs and which are controlled by the player. The GUI tells the player whether it is their turn to choose a response, though the player will also have to click a response even when it is the AI's turn. When the player clicks a node during the AI's turn, the player's choice does not affect the choice of the AI; this is purely so that the player controls the progression of the game.

11.17 Reporting Terminal

Once all conversation IDs have been allocated and all conversation objects have been activated, the Conversation Allocator activates the terminal object (Section 11.13). The object has a script called `TerminalController`, which controls terminal menu navigation and functionality, and handles reporting users, finding information about reported users, calculating the maximum points the player can get in the game level and the points they have earned in the level, and adding feedback messages to the player in lists. The `Start` function assigns the menu objects to their variables, puts certain buttons in separate lists, and sets click listeners for all buttons. There is a list for the main page username buttons (Section 8.8), a list for all sub category buttons, and a list of *report to* buttons. Each button in one list calls the function associated with the list, using the button's text as a parameter. After assigning variables and setting click listeners, `Start` calls a function called `CalculateMaxLevelPoints`, which will be detailed later in this section.

In the script's `Update` function, the player is detected when they get close to the terminal, and the *start terminal* button is activated, just like with the conversation objects (Section 11.14). Clicking the button calls the `StartTerminal` function, which pauses the game so that the player character will not move, shows the main terminal screen, and closes all other terminal screens.

11.17.1 Reporting a User

The names of all users in the current level each have their own button on the terminal main screen. The function `FindAndAddAllUsernamesAndClickListenersToButtons` in `Start` sets the buttons' username text. It finds all NPC-to-NPC Firestore conversations that belong to the current game level, and adds all usernames in the `conversation` documents to a list called `allUsernames`. The function then loops through the username buttons and sets the usernames as their text, and sets their click listeners. Clicking on a username button will call the `ToMainCategoryScreen` function, using the username as a parameter. `ToMainCategoryScreen` saves the username parameter to the `tempUsername` variable, which is used in the reporting screen headers and when adding the report to the report document list (Section 8.8 and later in this section). The function then displays the main category screen.

The main category screen has four main category buttons, where each button leads to its own sub category screen. Each sub category screen has sub category buttons, which, when clicked, call the `ToReportToScreen` function using the button's sub category string as a parameter. `ToReportToScreen` saves the sub category parameter to the `tempSubCategory` variable, which is used when adding the report to the report list, as with the `tempUsername` variable. The function has a boolean variable called `alreadyReported`, which is found with the `CheckAlreadyReported` function.

`CheckAlreadyReported` loops through all documents in the report document list to find the document belonging to the user that is being reported (by using `tempUsername`), then loops through all of the user's reports to find the issue that the user is being reported for (by using `tempSubCategory`). If the issue is found, `alreadyReported` is `true`. The `alreadyReported` boolean is then returned.

In `ToReportToScreen`, if the user already has been reported for this issue (`alreadyReported`), a popup message alerting the player appears, and when it is dismissed by the player, the reporting process is cancelled. If the user has not been reported for this issue, the *report to* screen is displayed.

The *report to* screen contains three buttons for each of the entities that issues can be reported to. Clicking on any of the buttons will call the `ToReportConfirmationScreen` function, using the string for the entity that was chosen as a parameter. In the function, the parameter is saved in the `tempReportedTo` variable, which is used when adding the report to the report document list as with the other two `temp` variables. The confirmation screen shows a summary text using all three `temp` variables, and options for confirming or cancelling the report. Clicking on the confirmation button calls the `AddReportToList` function.

The report document list contains `ReportDocument` struct objects. Each `ReportDocument` struct object contains the name of the user that the report document belongs to, and a list of `Report` struct objects. Each `Report` struct object contains a string for the issue that is reported (`issue`), and a string for who the issue is reported to (`reportedTo`).

The `AddReportToList` function first checks whether the user exists in the `ReportDocument` list by looping through it and comparing the `ReportDocument` objects' `username` variable with `tempUsername`. If the user exists, the `userExists` boolean variable is set to `true`, and a new `Report` struct object with the `tempSubCategory` and `tempReportedTo` values is created and added to the list of `Report` objects in the user's report document. After the loop, if `userExists` is `false`, a new `ReportDocument` with the `tempUsername` value is created, and a new `Report` struct object is added to the document's list of reports. The list of reported users on the main page (Section 8.8) is then updated, and the main terminal screen is displayed, completing the reporting process for the user.

11.17.2 Checking Reported Users

The list of reported users on the main page is a Scroll View which is updated by the `UpdateReportDocumentScrollView` function. First, the function destroys all Scroll View list elements, and then it loops through all objects in the `ReportDocument` list. For each object, the loop instantiates a new list element Prefab instance into the Scroll View and sets the `ReportDocument` `username` string as the text in the list element's username button. It also sets click listeners for the username button and the *delete* button in the list element.

The list element *delete* button calls the `ShowDeleteDocumentPopup` function using the loop's current `ReportDocument` username as a parameter. In `ShowDeleteDocumentPopup`, the username parameter is set to `tempUsername`, and a popup message asking whether the player wants to confirm the deletion is displayed. Clicking on *no* cancels the deletion, while clicking on *yes* calls the `DeleteReportDocu-`

mentFromList function. DeleteReportDocumentFromList loops through the report documents and deletes the document where the username string is the same as tempUsername. The Scroll View is then updated with UpdateReportDocumentScrollView.

The list element username button calls the ToReportDocumentScreen function using the loop's current ReportDocument username as a parameter. In ToReportDocumentScreen, the username parameter is set to tempUsername, the report document screen's *reports* Scroll View is updated by the UpdateReportsScrollView function, and the report document screen is displayed. The *delete* button outside of the Scroll View (Section 8.8) does the same as the *delete* buttons in the main page Scroll View.

The report document screen's report Scroll View shows all of the issues that the user has been reported for, and who the issues were reported to (Section 8.8). The UpdateReportsScrollView function deletes all Scroll View elements and adds all of the reports from the Report object list in the user's (tempUsername) ReportDocument object as Prefab instances, similarly to the UpdateReportDocumentScrollView.

The *delete* button in each Report list element calls the ShowDeleteReportPopup function using the issue string from the Report object as a parameter. The function sets the parameter issue string to tempSubCategory, and displays a popup message similar to the *delete report document* popup message. Confirming the deletion calls the DeleteReportFromList function, which finds the correct report from the correct user report document, using tempSubCategory and tempUsername, and deletes the report. The Report list is then updated with UpdateReportsScrollView.

11.17.3 Sending Reports and Calculating Score

At the start of this section (Section 11.17), a function in Start called CalculateMaxLevelPoints was mentioned. It adds all possible points the player can get in the current game level to a list called maxLevelPointsList. The function finds all conversation documents in Firestore that have been allocated to conversation objects in the current game level, and that have a problematic user. For each of these conversations, a loop will go through all documents in the conversation document's issues collection. Each issue document has a points value (Section 8.7), which is the amount of points the player receives when reporting the problematic user for that issue. The points value in each issue document, and the maximum amount of points the player can earn by reporting the issue to the most ideal entity/entities, are added to maxLevelPointsList. After looping through all conversation documents, the function loops through all of the player's post documents in Firestore (Sections 8.12 and 11.18), and for each player post that belongs to the current game level and is problematic, 1 point is added to the list.

The terminal main page has a *send reports* button, which calls the SendReports function when clicked. It adds all of the points from maxLevelPointsList to a maxLevelPoints variable in TerminalController, creates a new list called playerPointsList which contains all of the points the player has earned, and creates a *positive* list and a *negative* list of feedback messages (Section 8.8). It then calls a function called GetPointsAndFeedbackMessages, activates the level exit by calling ExitLevel's ActivateExit function (Section 11.19), and closes the terminal.

GetPointsAndFeedbackMessages finds all Firestore conversation documents belonging to the current game level, and for each document, four functions related to the reports and which add points to playerPointsList and messages to the feedback lists are called. After looping through the conversation documents, two functions which find points and feedback messages related to player posts and who the player follows (Section 11.18) respectively are called.

The first of the four functions related to reporting tries to find non-problematic users who the player has reported. It finds the objects in the ReportDocument list that belong to the current conversation in the GetPointsAndFeedbackMessages loop, and if one of the ReportDocument objects is for a non-problematic user, a negative feedback message is added.

The second of the four functions tries to find problematic users that have not been reported. The function loops through all usernames in the allUsernames list mentioned earlier in this section (Section 11.17), and tries to find whether a user with one of the usernames is a problematic user in the current conversation in the GetPointsAndFeedbackMessages loop. If so, a boolean variable called issueUserReported is set to false, and the function loops through all objects in the ReportDocument list. If one of the report documents belongs to the problematic user, issueUserReported is set to true. After the ReportDocument loop, if issueUserReported is false, a negative feedback message is added.

The third of the four functions tries to find problematic users that have been reported incorrectly. It loops through all ReportDocument list objects, and finds the document object which belongs to the problematic user in the current conversation in the GetPointsAndFeedbackMessages loop. Then, all Report objects in the list of reports in the ReportDocument object that was found are looped through. In the Report loop, a boolean variable called correctReportIssue is set to false, and all documents in the current conversation document's issues collection are looped through. In the issue document loop, if the current Report object's issue variable matches the current issue in the loop, correctReportIssue is set to true. After the issue document loop, but still in the Report object loop, if correctReportIssue is false, a negative feedback message is added.

The fourth and final of the four functions tries to find correct reports of, and unreported issues from, problematic users. It loops through all documents in the issues collection in the document for the current conversation in the GetPointsAndFeedbackMessages loop. In each iteration of the issue loop, all objects in the ReportDocument list are looped through. The ReportDocument loop tries to find the report document for the problematic user in the current conversation. If found, an issueReported boolean variable is set to false, and all objects in the report document's Report list are looped through. In the Report loop, if the Report object's issue variable is the same as the current issue in the issue loop, issueReported is set to true, a positive feedback message is added, the issue document's points value is added to playerPointsList, and a function called FindReportToChoices is called. In the ReportDocument loop, after the Report loop, if issueReported is false, a negative feedback message is added.

In FindReportToChoices, all documents in the report issues Firestore collection (Section 8.8) are looped through. The loop tries to find the report issue document which corresponds to the current issue document in the issue loop described in the previous paragraph. If the report issue document is found, all of the documents in the report issue document's who to report to collection (Section 8.8) are looped through. The report to loop tries to find a report to document which corresponds to the reportedTo variable in the current Report document in the Report loop described in the previous paragraph. If found, the report to document's points value (Section 8.8) is added to playerPointsList, and feedback messages are added depending on the points value. If the points value is 1 or higher, a positive feedback message is added, and if the points value is 0, a negative feedback message is added.

The function called in GetPointsAndFeedbackMessages which finds points and feedback messages related to the player's posts, tries to find posts that the player has deleted. It loops through all of the post documents in the player's posts collection in Firestore, and finds the post documents which belong to the current game level. If the post document's problematic value is true and its deleted value is true (Section 8.12), 1 point is added to playerPointsList and a positive feedback message is added. If the post is problematic and is not deleted, a negative feedback message is added. If the post is not problematic but has been deleted, negative 1 (-1) point is added to playerPointsList, and a negative feedback message is added.

In the function called in GetPointsAndFeedbackMessages which finds points and feedback messages related to who the player follows, all documents in the player's following collection (Section 8.12) are looped through. All level numbers values in each following document (Section 11.18) are looped through. If the number of the current game level is found in the level numbers list, the user document for the user that the player follows (from the player's following document) is found in the users collection (Sections 8.5 and 8.12). If the user document's `problematic` value is `true`, it means that the player is following a problematic user, so negative 1 (-1) point is added to `playerPointsList`, and a negative feedback message is added.

Context for the player posts and who the player follows is provided in Section 11.18. The lists containing the maximum game level score, the player's game level score, the positive feedback messages, and the negative feedback messages, are all used in the result screen, which is detailed in Section 11.19.

11.18 User Profiles

The Profile Controller handles everything related to user profile pages. A list of all sprites of images that can be shown in NPC-to-NPC conversations, the list of `UserImages` struct objects (Section 11.15), and a list of all profile picture sprites, are saved in this object and script. The `ProfileController` script's `Start` function sets variables and button click listeners, and it adds all profile picture buttons in the *change profile picture* page, which will be described later in this section, to a list. It also sets the `ProfileController` script's `tempUsername` variable to be the string *player*, as it is the player's profile main page that is displayed first when clicking on the `to profile` button (Section 11.6). The `tempUsername` variable is used in a similar way to the `tempUsername` variable in the Terminal Controller (Section 11.17), and it determines which profile page is displayed.

11.18.1 Profile Main Page

The Game Manager sets the click listener for the `to profile` button, making the button call the `ShowProfileMainPage` function in the `ProfileController` script. `ShowProfileMainPage` pauses the game, and uses `tempUsername` to determine whether to show the player's main page or an NPC user's main page. When showing an NPC user's main page, `tempUsername` is used as the username text on the page. Finally, `ShowProfileMainPage` starts a coroutine called `RefreshMainProfilePageDelay`.

`RefreshMainProfilePageDelay` delays the display of certain elements in the profile main page by a split second, to allow for the main page's asynchronous Firestore data updates to complete before the page is displayed. After the time delay, the user's profile picture in their main page is set by the `SetProfilePicture` function, the `follow/unfollow` button is set by the `SetFollowUnfollowButtons` function if it is the main page for a non-player user (`tempUsername` is not *player*), and the player's username text is set by the `SetPlayerUsernameText` function if it is the main page for the player (`tempUsername` is *player*).

In `SetProfilePicture`, the Firestore `user` document (Sections 8.5 and 8.12) for the current `tempUsername` user is found, the `user` document's profile picture colour value (Section 8.12) is saved in a `tempProfilePicture` variable, which is used to retrieve the correct profile picture sprite from the Profile Controller's profile picture sprite list using the `GetProfilePictureSprite` function. The `tempProfilePicture` variable is also used to set the user's profile picture in the post elements on their *posts* page, which is detailed later in this section. After retrieving the correct profile picture sprite, the player's *main page* object or the NPC's *main page* object is set to the `mainPage` variable depending on whether `tempUsername` is *player* or not, and the main page's profile picture object is found and the sprite that was found is set to the object.

`GetProfilePictureSprite` is used when setting profile pictures in all places where they appear in profile pages, so the function is called throughout `ProfileController`. It uses a *profile picture colour* string, which is what is saved in the `profile picture` value in each Firestore `user` document, as a parameter, and loops through all sprites in the profile picture sprite list. In each iteration of the loop, the profile picture sprite's colour is extracted from the name of the sprite, and if the sprite name's *colour* string matches the profile picture colour parameter, the sprite is set to a variable that the function returns.

The `SetFollowUnfollowButtons` function, which is called in the `RefreshMainProfilePageDelay` coroutine, first disables both the *follow* button and the *unfollow* button. It has a `playerFollowsUser` boolean variable which is initialised to `false`, and it loops through the player's Firestore `following` collection (Section 8.12). If the user in the player's `following` collection matches `tempUsername` (the name of the user whose profile page is displayed for the player), the `following` document's `level numbers` list (Section 8.12) is looped through. If the level number that matches the number of the current game level is found, `playerFollowsUser` is set to `true`. After the `following` document loop, if `playerFollowsUser` is `false`, the *follow* button is enabled and the *unfollow* button is disabled, otherwise the *unfollow* button is enabled and the *follow* button is disabled.

The `SetPlayerUsernameText` function, which is called in the `RefreshMainProfilePageDelay` coroutine, finds the player's `user` document. The string value for the player's custom username (detailed later in this section) in the document is found and set to the `playerUsername` variable, which is set to the *username* text object in the player's profile main page. The `playerUsername` variable is used in other places in the script.

When the player is on an NPC user's main profile page, they will either see a *follow* button or an *unfollow* button depending on whether the player's user follows the NPC user. When clicked, the *follow* button calls the `FollowUser` function. In `FollowUser`, the player's `following` collection is found. A boolean variable called `userExistsInFollowingCollection` is initialised to `false`, and the `following` documents are looped through. In each iteration of the loop, if the `following` document username matches `tempUsername` (the user that the player will follow), `userExistsInFollowingCollection` is set to `true`. After the `following` document loop, if `userExistsInFollowingCollection` is `true`, the number of the current game level is added to the `level numbers` list in the `tempUsername` document in the player's `following` collection, and the number is also added to the `level numbers` list in the `player` document in the `tempUsername` user's `followers` collection. If `userExistsInFollowingCollection` is `false`, the `following` and `follower` documents mentioned in the previous sentence are created, and the `level numbers` lists are initialised with the number of the current game level. At the end of the `FollowUser` function, the profile main page is refreshed by calling the `ShowProfileMainPage` function.

The *unfollow* button calls the `UnfollowUser` function which is similar to `FollowUser`, except the number of the current game level is removed from `level numbers` in the `tempUsername` user document in the player's `following` collection, and from `level numbers` in the player document in the `tempUsername` user's `followers` collection.

11.18.2 *Following* and *Followers* Pages

The functionality in the *following* and *followers* pages are identical for the player user and NPC users. When the pages are displayed, a function called `UpdateFollowingFollowersScrollView` updates the username list Scroll Views in both pages. The function has the collection type string (*following* or *followers*) as a parameter, and the correct Firestore collection is found using the parameter and `tempUsername` (the name of the user whose profile pages are displayed for the player). All elements in the page's Scroll View are destroyed, and all documents in the `following` or `followers` collection are looped through. For each document in the loop, the document's `level numbers` list is looped through. If the number of the current game level is found in the list, the process of adding the username list element to the Scroll View starts. In the `level numbers` loop, the string variable containing the *following/follower* user's name is set depending on whether the user is an NPC user or the player (the player has a custom username, saved in the `playerUsername` variable, that is different from the player's user ID, which is *player*). The *following/follower* user's document ID is used to find the user's document in the `users` collection. Using the `profile picture` value in the `user` document, the `GetProfilePictureSprite` function finds the correct profile picture sprite. The current user list element is instantiated into the Scroll View, and the profile picture and the username text is set in the element. The click listener for the element's `Button` component is set, so that, when clicked, the button changes `tempUsername` to be the document ID of the user in the list element and calls the `ShowProfileMainPage` function, which will show that user's main profile page.

11.18.3 *Posts* Pages

When the *posts* buttons in the player and NPC users' main profile pages are clicked, the `ShowPostsPage` function is called. The function calls the `UpdatePostsScrollView` function, and the correct *posts* page is displayed, depending on whether `tempUsername` is *player* or not.

`UpdatePostsScrollView` finds the `tempUsername` user's `posts` collection (Section 8.12). A Scroll View variable is set depending on whether `tempUsername` is *player* or not, as the player user has a different Scroll View than NPC users. All elements in the *posts* Scroll View are destroyed, and all `post` documents in the collection are looped through. If the post document's `level number` variable is the same as the number of the current game level, the post is added to the Scroll View. The post is added to the Scroll View differently depending on whether `tempUsername` is *player* or not, either by the `AddPlayerPostsToScrollView` function or the `AddNPCPostsToScrollView` function. The *post* elements in the player's *posts* page can be deleted or restored, while the elements in an NPC user's *posts* page cannot be interacted with (Section 8.12).

In `AddPlayerPostsToScrollView`, if the `deleted` value in the current `post` document from `UpdatePostsScrollView` is `false`, a non-deleted version of the *post* list element is instantiated into the player user's *posts* Scroll View, and the element's *delete* button is set to call the `DeletePost` function. If the `deleted` value is `true`, a *deleted* version of the *post* list element is instantiated into the Scroll View, and the element's *restore* button calls the `RestoreDeletedPost` function. Outside of the `deleted` `if/else` statements, the `playerUsername` string is set in the *username* text object in the list element. Then, the profile picture sprite is retrieved with `GetProfilePictureSprite` and `tempProfilePicture`, and is set in the element's *profile picture* object. Finally, the post text in the current document in the `posts` loop in `UpdatePostsScrollView` is set in the *post* text object in the element.

In `DeletePost`, the player `post` document's `deleted` value is set to `true`, and the Scroll View is updated by calling `UpdatePostsScrollView`. `RestoreDeletedPost` does the same, but sets `deleted` to `false`.

In `AddNPCPostsToScrollView`, an NPC *post* list element is instantiated into the NPC user's *posts* Scroll View, `tempUsername` is set in the element's *username* text object, the profile picture is retrieved by `GetProfilePictureSprite` and `tempProfilePicture` and is set in the element's *profile picture* object, and the post text in the current document in the `posts` loop in `UpdatePostsScrollView` is set in the element's *post* text object.

The *restore all deleted posts* button on the player's *posts* page (Section 8.12) calls the `RestoreAllDeletedPosts` function. The function loops through all documents in the player's `posts` collection, and sets the `deleted` value for each document to `false`. After the loop, the Scroll View is updated by calling `UpdatePostsScrollView`.

11.18.4 *User Settings* Pages

Clicking on the *user settings* button on the player's main profile page opens the *user settings* page. The *user settings* page has two options; *profile settings* and *privacy settings*. *Privacy settings* functionality was an idea our group had that was not implemented, but the *profile settings* option leads to the *profile settings* page. The *profile settings* page has three options; *change profile picture*, *change username*, and *edit avatar*.

The *change profile picture* option leads to the *change profile picture* page. The page contains 8 profile picture options. Clicking on one calls the `SavePlayerProfilePictureChanges` function, which extracts the *colour* string from the profile picture button's name, and updates the `profile picture` value in the player's document in the `users` collection with the *colour* string. The player is then taken to the main profile page by the `ShowProfileMainPage` function, where the profile picture is updated.

The *change username* option leads to the *change username* page. The page contains an input field where the player can write their new username. Clicking on the *save changes* button on the page calls the `SavePlayerUsernameChanges` function, which retrieves the text in the input field and updates the `username` value in the player's `user` document with it. The player is then taken to the main profile page by the `ShowProfileMainPage` function, where the username is updated.

The *edit avatar* option calls the Game Manager's `GoToAvatarCreation` function (Section 11.6), just like the *edit avatar* option in the pause menu, which takes the player to the avatar creation scene.

11.18.5 *All Users* Page

On the main profile page for both the player and NPC users, there is an *all users* button, which takes the player to the *all users* page. The *all users* page contains a Scroll View with all users in the `users` collection in Firestore. When the player enters the page, the `UpdateAllUsersScrollView` function is called.

`UpdateAllUsersScrollView` finds the `users` collection, destroys all list elements in the Scroll View, and loops through all `user` document. In the `users` collection loop, a `username` variable is set to either the player's custom username (`playerUsername`) or an NPC user's name, depending on whether the current `user` document ID is *player* or not, and then the user's profile picture is found by `GetProfilePictureSprite` with the `user` document's `profile picture` colour value as a parameter. Then, a *user* list element is instantiated into the Scroll View, the username and profile picture is set in the element, and the element's click listener is set so that when the list element is clicked, the `user` document ID is set to `tempUsername` and `ShowProfileMainPage` is called, so that the user's profile main page is displayed.

11.19 Level Exit and Result Screen

The `ExitLevel` script is attached to the game level exit plane, and handles the level feedback/result screen (Section 8.5) and loading the next level. In the script's `Start` function, variables and button click listeners are set, and the number of the current level is extracted from the name of the current level scene and set in `StaticVariables`'s `currentLevelNumber` variable.

When the player sends all of the reports in the report terminal, the exit is activated with the `ActivateExit` function (Section 11.17), which sets the `exitActive` boolean variable to `true` and opens the *door* object in the scene. When the player object collides with the activated exit plane (`exitActive` is `true`), the function which displays a confirmation popup message is called. Pressing the *OK* button in the popup message calls the `ShowResultScreen` function. Pressing the *cancel* button closes the popup message and positions the player away from the exit, to comply with the requirement in Section 8.5.

`ShowResultScreen` loops through the Terminal Controller's `playerPointsList` (Section 11.17) and sums all of the points, and if the sum is a negative number, the player's score is set to be 0. The score header in the result screen, which includes the player point sum and the Terminal Controller's `maxLevelPoints` value, is set, and the `SetScoreFeedbackText` and `AddFeedbackMessagesToScrollViews` functions are called. The result screen is then displayed. The screen has a score header, a score feedback text object, and two Scroll Views; one for positive feedback messages and one for negative feedback messages. The screen also has an *OK* button, which loads the next level scene.

`SetScoreFeedbackText` uses the player points sum as a parameter and calculates the player's score percentage compared to the maximum score value from `TerminalController`, which is set in `StaticVariables`'s `lastPlayerScorePercentage` variable, which is used in the Conversation Allocator (Section 11.13). Depending on the percentage score, one of several feedback messages will be set in the score feedback text object.

`AddFeedbackMessagesToScrollViews` first loops through the list of positive feedback messages from `TerminalController` (Section 11.17). In each loop iteration, a *feedback* list element is instantiated into the *positive* Scroll View, and the current feedback message in the list is added to the list element's text object. The function then loops through the list of negative feedback messages, and adds list elements to the *negative* Scroll View in the same way as with the positive messages.

When the player clicks on the *OK* button on the result screen, the `LoadNextLevel` function is called. The function finds the number of the next level by adding 1 to the number of the current level, updates `currentLevelNumber` in `StaticVari-ables`, and calls the Game Manager's `LoadLevel` function (Section 11.6), which loads the next level.

Chapter 12

Deployment

This chapter provides instructions for how to set up and test the proof-of-concept game, and is primarily meant for our client. At the time of writing, the game can only be tested in Unity, and not in a game build. Testing the game requires access to our group's repository, which may be granted upon request. Contact information can be found in Appendix A.3.

12.1 Download and Install Unity

Download and install Unity Hub, and use it to install Unity. The Unity version that is used in the project is `2020.2.2f1`, so that may be the safest Unity version to install, but later versions should also work.

12.2 Clone the Repository

Once access to the repository has been granted, clone the repository by clicking on the "clone" button on the repository's main page, and by using Git.

12.3 Apply Project and Game Settings

In the open Unity project, from the menu at the top of the screen, go to *File -> Build Settings*. Set the Platform to be *Android*.

If the *Scenes In Build* window is empty, go to the following scenes in *Assets -> Scenes* in the *Project* window in Unity: `AvatarCreation`, `Level1`, `Level2`, and `Level3`. For each scene you visit, go to the *Build Settings* page, and click the *Add Open Scenes* button.

In the Unity *Game* window, in the second option from the left, at the top of the *Game* window, choose the *16:9 Landscape* option.

12.4 Set Variables in the Inspector

For the game to work, certain variables have to be set in the Unity Inspector.

In all four of the scenes mentioned in Section 12.3, click on the Avatar Creation Controller object in the Hierarchy window. The Inspector window will show that object's properties, including the public arrays in the `Avatar Creation Controller`. Drag and drop all Materials in the *Assets -> Materials -> Avatar -> Skin Tones* folder into the *Skin Tone Materials* array in the Inspector. Add all of the Materials in the *Assets -> Materials -> Avatar -> Hair Colors* folder to the *Hair Color Materials* array, and add all of the Materials in the *Assets -> Materials -> Avatar -> Clothing Colors* folder to the *Clothing Color Materials* array.

In every scene with a name starting with *level*, click on the *Profile Controller* object in the Hierarchy window. For all Scroll View element variables in the object's Inspector window, place each Prefab from the *Assets -> Prefabs* folder with the same name as a variable in the Inspector, into that variable. Place all images in the *Assets -> Sprites -> Profile Pictures* folder into the *Profile Picture Sprites* array in the *Profile Controller* Inspector. Also place all images in the *Assets -> Sprites -> Conversation Images* folder into the *Conversation Image Sprites* array in the *Profile Controller* Inspector.

In every scene with a name starting with *level*, click on the *Terminal* object in the Hierarchy window. In both Scroll View element variables in the object's Inspector window, place the Prefabs in the *Assets -> Prefabs* folder with the same name as a variable in the Inspector, into that variable.

In every scene with a name starting with *level*, click on the *Exit -> Exit Load Plane* object in the Hierarchy window. Place the *Choices Scroll View Element* Prefab in the *Assets -> Prefabs* folder, into the *Choices Scroll View Element* variable in the object's Inspector.

12.5 Play the Game

When in the `AvatarCreation` scene, press the button with the *play* symbol, located in the middle of the horizontal menu bar just below the top menu bar, to start the game. Certain messages which may be useful to the tester will appear in the *Console* window.

Chapter 13

Development Discussion

13.1 Development Process

As mentioned in Chapter 7, we used a Kanban-like development process in the project, using GitLab's issue board functionality as a Kanban board. This functionality was mostly used at the start of the project, and some of our group members used it more than others. The somewhat lacking use of the functionality did not seem to affect our efficiency, so we eventually stopped using it. During development, our group members communicated over Discord, and wrote our tasks in a *to-do list* section there. In the Kanban board, we only had one *under development* section in addition to the *open* and *closed* sections (Section 7.2). We realised that the tasks in our *to-do list* already implied that the tasks were under development, which defeated the purpose of our Kanban board.

13.2 Deployment

Having to apply the settings and set the variables specified in Chapter 12 is inconvenient for the tester. This would not be an issue with a finished game, but it is necessary with our proof-of-concept game.

13.3 Firestore Database Structure

Our Firestore database structure worked well for us overall. However, documents in our Firestore database will sometimes contain numbers to order them correctly, but due to Firestore sorting documents by their IDs completely alphabetically with no regard for the size of multiple-digit numbers, this is not an ideal solution. Currently, if documents that are ordered by numbers exceed 9, the document order will become incorrect, as "document10" will be placed before "document2", etc. If we are to add new documents, we will exclusively order them with alphabetic letters to avoid this problem.

13.4 Firestore Asynchronous Functions

Retrieving and updating data from Firestore is done in asynchronous functions, which means that functionality which uses the data must wait for the asynchronous functions to finish. To make this work, we made certain objects activate other objects to make sure that they run after the activating object (Section 11.13), and we added a delay in other cases (Section 11.18). The most notable consequence of the asynchronous functions may be the fact that the player sometimes must wait for avatar and conversation data in Firestore to reset before finishing the avatar creation at the start of a game session. In the Unity project, we added a *console* window message which explains to the tester that they must wait for two other messages before finishing the avatar creation. The data is usually reset quite quickly, but this is still not an ideal solution. Ideally, there should be loading screens in such cases, which would let the player know that they have to wait.

13.5 Avatar Creator

As mentioned in Section 11.8, colour and skin tone is set depending on the name of the button that was clicked, specifically the most recently clicked button. The solution works well for us, but we do not think this is a common method in character creators. It depends on the buttons having the correct names with the correct format, and we find our button colouring solution to be lacking in structure.

We wanted our avatar creation solution to be inclusive, but there are only five skin tone options, four hair/headgear types, and five hair colour options to choose from, which is not fully representative of all humans. The fact that all avatars have the same sweater, trousers, and shoes, and the fact that there are only 12 clothing colour options for each piece of clothing, also limit customisation. There is a large enough number of possible avatar appearances to make them feel somewhat personalised, but it can be improved by adding more options.

13.6 Player Controls

We thought that the player controls, where the player object smoothly rotates and moves towards a point on an object that is clicked (Section 11.10), was implemented well. However, the fact that the camera, and by extension the screen, always follows the player (Section 11.9), means that the point that is clicked always follows the player as well, and moves in relation to the objects that are clicked. This makes it almost impossible to walk in a straight line. In future work, this could be fixed by finding a point on an invisible, circular plane which always follows the player, instead of points on objects which do not move with the player, and finding the angle between the player and the point. The angle could then be used to find the player's movement direction. This has not been tested, so we are not sure about whether this would work.

13.7 Hidden Markov Model

We have suggested and shown how small independent Hidden Markov Models can be used on the player and different game components to modulate dynamic difficulty adaptation. The Hidden Markov Model has also been used to modulate emotion. Unity has some limitations regarding matrix computations other than those tuned to motions. Nevertheless, we have applied online recursive Hidden Markov Models without any major challenges except for having to make some basic functions to treat C# arrays as matrices. If Unity had support for using python3, there would have been options for more extensive use of other online functions like fixed lag smoothing [24] to accomplish more precise adaptation and simulations. Hidden Markov Models have a broad area stochastic modulations for game components, even if they are not particularly scalable to large models. However, by using Hidden Markov Models in the development of game logic that can help improving dynamic difficulty adaptation, much is done to extend the models to other types of more scalable dynamic Bayesian networks. Using Hidden Markov Models and Bayesian methods can also help to better understand what is happening in the game when later applying neural networks to perhaps take over in production. Another advantage is that Hidden Markov Models are tightly linked to the Markov Decision Process approaches that can add stochastic decision making support to the game [24].

13.8 Conversation Allocator

We consider our way of finding random eligible conversations to be implemented well. Previously, the two conversation type lists in the Conversation Allocator were looped through separately in two instances, doing largely the same actions. To prevent redundancy, we made two new functions, each of which combines two loops into one, looping through the conversation object list that is used as a parameter.

For each new conversation that is allocated, if the player has a score value from a previous level, the Hidden Markov Model is used to find the difficulty of the conversation (Section 11.13). We could have only used the model once to determine one difficulty value for all new conversations, but we wanted the new conversations to have variable difficulties. This is to increase the overall difficulty more gradually, as the performance of a player in one level does not necessarily accurately reflect their skill level. We did not make enough test conversations or game level scenes to demonstrate the use of the Hidden Markov Model well.

13.9 NPC-to-NPC Conversations

The conversations we have added to Firestore are very simple, and may not be good examples of how deep and potentially educational the conversations could be. If a more skilled author were to write the conversations, with assistance from someone knowledgeable with the digital security curriculum, the conversations could be much more interesting and educational.

We believe our decision to use coroutines for pausing the game and going through the NPC-to-NPC conversations (Section 11.15) was good. Coroutines allow for execution to be suspended or resumed, which works well for functionality which waits for player input.

Our decision to use an *<image>* tag to show images in conversations worked well, and prevented us from having to add an `image` value to the `text box` document. However, our solution did not allow for showing text in addition to the image in one text box, which can be implemented in future work.

We believe the decision to make the player able to see the three last entries is a good choice as it gives the player a better overview of the conversation as well as providing a look that more closely resembles messaging apps. The way of implementing it with the use of four if/else clauses might not be the most elegant but is a simple and functional solution.

13.10 Player-to-NPC Conversations

The Twinery tree dialogues we have created so far are essentially just examples to test if the framework works. With the use of professionals, creating good dialogues will not be an issue.
While Twinery might not be the best end solution for a task like this due to reasons described in Section 9.6, it is still a workable solution for demonstrative purposes, which fits us fine.

The decision to integrate the model from [33] with our existing model created some friction and it might have worked better to throw away the `DialogueViewer` and to base the solution completely on our existing model.

As for our decision on how to structure the Twinery dialogue, there might be some room for improvement, but there is not a whole lot of space to work with without making an overly complicated system that destroys the purpose of using Twinery in the first place.

13.11 Reporting Terminal

Ideally, the username buttons on the main terminal page should have been replaced by another Scroll View with clickable username list objects, so that the names can be added dynamically as opposed to requiring the buttons to be manually placed beforehand. This may be changed in the future.
Something similar could have been done with the *sub category* buttons on the *sub category* pages, where the script could have used the names of the main categories to find sub categories from the `report issues` collection in Firestore, and then add them to a Scroll View on the *sub category* page. This way, only one sub category page would be necessary, as at the time of writing, there is one *sub category* screen for each main category.

The maximum points and the points that the player has earned are in lists instead of single-value variables with the sum of the points, because the variables that were originally used did not get updated properly because of the Firestore asynchronous functions used to calculate the points. This could probably be fixed with a loading screen which makes sure that the asynchronous functions finish before new actions are executed, so that the `points` variables get updated properly.

13.12 Profile Pages

The coroutine which loads the main profile pages after a delay to ensure Firestore data is updated has always worked for us, but it is not an ideal solution. In certain cases, the data may take a long time to update, and the delay may not be long enough. As with the other issues with the asynchronous functions, a loading screen could be used to solve this.

The "privacy settings" functionality that was not implemented (Section 11.18) can be used to teach players about recommended privacy settings. This functionality can be explored in future work.

The `UserImages` list in the Profile Controller (Section 11.15) was intended to be a part of the *detective* scenario (Section 8.10). The player would look through the images to determine if any of them break the social media rules. This idea can be developed further in the future.

13.13 Result Screen

The feedback message which appears below the score (Section 11.19) may not be necessary, as the player will already know how well they did from the score. This feedback message may serve as a motivator to improve, but the score alone should serve this function. It also does not teach the player anything. For these reasons, this feedback message may be removed in the future.

The feedback messages on the result screen do not provide reasoning for why the player's actions were positive or negative. The player would likely learn more if the reasoning was there. This can easily be fixed by writing more in the feedback messages that are added.

Chapter 14

Development Conclusion

Unity does not have out-of-the-box support for using machine learning techniques like dynamic Bayesian networks and Hidden Markov Models. To run online filtering and prediction on Hidden Markov Models placed to monitor the player or guide different types of game components, only a few custom matrix functions and the forward and backwards algorithms can be implemented as a small script. We avoided the online smoothing algorithm (fixed-lag-smoothing) [24] because they require inversing matrices. These kinds of refinements of the more plain forward and backward algorithms implemented, to suit an online smoothing function, is not even necessary to run the basic explorations of how to implement dynamic difficulty adaptations on player and game components.

Using Hidden Markov Models to prepare for dynamic difficulty adaptation is a constructive starting point for preparing more extensive machine learning techniques that in general are less causally explained than the Bayesian stochastic techniques. More knowledge is needed to explore how the different components Hidden Markov Models can be orchestrated together to give a broader view and comprehension of the dynamic difficulty adaptation in a complete learning game.

Future projects aiming at a dynamic difficulty inspired approach to development are advised to connect to domain experts to clarify common or future measurement of learning within the domain. Even though we had domain experts available, the subject matter is not clearly defined in terms of learning goals. The domain experts were not able to give clear ranking of what youths could perceive as difficult. Our choice was therefore to let the mock-ups be open for domain experts to fill difficulty ranked content later. Measuring difficulty both in learning and in gameplay is not straight forward. Future work should elaborate on this within and across the expert domains, as this is crucial to succeed with proper dynamic difficulty adaptation in learning games.

Dialogues in games are particularly challenging to modulate. This was our most difficult task, as no one had any previous experience with it. Nevertheless, we managed to modulate dialogues both by dynamic content management and inside the dialogue tree. Using more or less ready made tools turned out to be challenging, but less so than making such systems from scratch. Future support systems for modulating dialogues with the purpose of dynamic difficulty adjustment in learning games would be helpful.

Our project explored isolated game components. To develop a complete learning game, we need a way to link the components. We suggest finishing our framework with a Goal Oriented Action Planner (GOAP) that can run both animations and actions in a coordinated way. The GOAP can later be combined with Markov Decision Process algorithms [24] that are related to Hidden Markov Models and dynamic Bayesian networks.

In early learning game development, the lack of knowledge about the players makes it more challenging to apply dynamic difficulty adaptation in the game. We have earlier talked about dealing with this as a cold start problem in recommender systems. If the domain of the subject matter have available data, these can be used to probe both currently unknown players regarding several aspects of difficulty in learning and gameplay [30].

Our proof-of-concept game provides several suggestions for how a game can teach players a curriculum in entertaining ways. We have suggested the use of Hidden Markov Models, and ways of changing the difficulty of the game depending on the player's performance. The avatar creator is relatively inclusive, and the avatar creator and the player's profile add some personalised elements to the game. The conversations that the player can observe or take part in have an opportunity to teach the player important digital citizenship concepts in a way that can be engaging and immersive. Reporting problematic users may provide a sense of satisfaction in the player, and by reading the feedback messages after each level, the player may achieve a deeper understanding of the topics. Our work provides a good starting point for further development, and we believe our findings and suggestions will be useful for our client in their development of a finished game.

Bibliography

[1] (13th Apr. 2021). 'Blender (software),' [Online]. Available: https://en.wikipedia.org/wiki/Blender_(software) (visited on 14/04/2021).

[2] Unity. (24th Feb. 2021). 'Canvas,' [Online]. Available: https://docs.unity3d.com/2020.1/Documentation/Manual/UICanvas.html (visited on 03/05/2021).

[3] (7th Apr. 2021). 'Firebase,' [Online]. Available: https://en.wikipedia.org/wiki/Firebase (visited on 13/04/2021).

[4] Firebase. (2021). 'Accelerate and scale app development without managing infrastructure,' [Online]. Available: https://firebase.google.com/products-build (visited on 13/04/2021).

[5] Firebase. (16th Dec. 2020). 'Cloud Firestore,' [Online]. Available: https://firebase.google.com/docs/firestore (visited on 13/04/2021).

[6] F. Games. (24th Aug. 2020). 'Fungus,' [Online]. Available: https://assetstore.unity.com/packages/tools/game-toolkits/fungus-34184 (visited on 19/05/2021).

[7] Unity. (25th Apr. 2021). 'The Hierarchy window,' [Online]. Available: https://docs.unity3d.com/Manual/Hierarchy.html (visited on 03/05/2021).

[8] Unity. (11th Apr. 2021). 'The Inspector window,' [Online]. Available: https://docs.unity3d.com/Manual/UsingTheInspector.html (visited on 13/04/2021).

[9] Unity. (30th Apr. 2021). 'Materials introduction,' [Online]. Available: https://docs.unity3d.com/Manual/materials-introduction.html (visited on 04/05/2021).

[10] Unity. (25th Apr. 2021). 'Prefabs,' [Online]. Available: https://docs.unity3d.com/Manual/Prefabs.html (visited on 03/05/2021).

[11] Unity. (2021). 'ProBuilder,' [Online]. Available: https://unity3d.com/unity/features/worldbuilding/probuilder (visited on 03/05/2021).

[12] Unity. (25th Apr. 2021). 'TextMeshPro,' [Online]. Available: https://docs.unity3d.com/Manual/com.unity.textmeshpro.html (visited on 27/04/2021).

[13] Twinery. (19th May 2021). 'Twinery,' [Online]. Available: https://twinery.org/ (visited on 19/05/2021).

[14] (5th Apr. 2021). 'Unity (game engine),' [Online]. Available: https://en.wikipedia.org/wiki/Unity_(game_engine) (visited on 13/04/2021).

[15] N. Hamdaoui, K. KIdriss and S. Bennani, 'Modeling learners in educational games: Relationship between playing and learning styles,' *Simulation and Gaming*, vol. 49, pp. 675–699, 2018. DOI: DOI:10.1177/1046878118783804.

[16] A. Streicher and J. Smeddinck, 'Personalized and Adaptive Serious Games,' in *Entertainment Computing and Serious Games*, R. Dörner, S. Göbel, M. Kickmeier-Rust, M. Masuch and K. Zweig, Eds. Springer, Cham, 2016, pp. 332–377. DOI: 10.1007/978-3-319-46152-6_14. [Online]. Available: https://doi.org/10.1007/978-3-319-46152-6_14.

[17] M. Slussareff, E. Braad, P. Wilkinson and B. Strååt, *Games for Learning*, ser. Entertainment Computing and Serious Games. 2016.

[18] M.-V. Aponte, G. Levieux and S. Natkin, 'Difficulty in Videogames: An Experimental Validation of a Formal Definition,' Paper presented at the ACE, Lisbon, Portugal, 2011. DOI: 10.1145/2071423.2071484.

[19] N. Pierce, O. Conlan and V. Wade, 'Adaptive Educational Games: Providing Non-invasive Personalised Learning Experiences,' Paper presented at the Second IEEE International Conference on Digital Game and Intelligent Toy Enhanced Learning, 2008. DOI: 10.1109/DIGITEL.2008.30.

[20] M. Kickmeier-Rust and D. Albert, 'Educationally adaptive: Balancing serious games,' *Int. J. Comput. Sci. Sport*, vol. 11, pp. 15–28, 2012.

[21] A. Belahbib, L. El Aachak, M. Bouhorma, O. Yedri, S. Abdelali and E. Fatiha, 'Serious Games Adaptation According to the Learner's performances,' *International Journal of Electrical and Computer Engineering*, vol. 7, no. 1, pp. 451–459, 2017. DOI: 10.11591/ijece.v7i1.pp451-459.

[22] C. S. González González, A. Mora and P. Toledo, 'Gamification in Intelligent Tutoring Systems,' Paper presented at the Second International Conference on Technological Ecosystems for Enhancing Multiculturality, Salamanca, Spain, 2014. DOI: 10.1145/2669711.2669903.

[23] D. Hooshyar, M. Yousefi, M. Wang and H. Lim, 'A data-driven procedural-content-generation approach for educational games,' *Journal for Computer Assisted Learning*, vol. 34, pp. 731–739, 2018. DOI: https://doi.org/10.1111/jcal.12280.

[24] S. Russel and P. Norvig, *Artificial Intelligence. A Modern Approach.* 4th ed. Pearson, 2020, ISBN: 0-13-461099-7.

[25] F. Gallego-Durán, C. Villagrá-Arnedo, R. Satorre-Cuerda, P. Compañ-Rosique, R. Rafael Molina-Carmona and F. Llorens-Largo, 'A Guide for Game-Design-Based Gamification,' *Informatics*, vol. 6, no. 4, 2019. DOI: 10.3390/informatics6040049.

[26] Conference Paper, 2014. DOI: 10.1145/2669711.2669903. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2669711.2669903.

[27] D. Hutchison, T. Kanade, J. Kittler, J. Kleinberg, F. Mattern, J. Mitchell, M. Naor, C. Rangan, B. Steffen, D. Terzopoulos, D. Tygar and G. Weikum, *Entertainment Computing and Serious Games*. Cham, Switzerland: Springer Nature, 2016. DOI: `10.1007/978-3-319-46152-6`.

[28] R. C. Pettersen, *Oppgaveskrivingens abc.* 2nd ed. Oslo: Universitetsforlaget, 2016.

[29] R. C. Pettersen, *Problembasert læring for studenter og lærere. Introduksjon til PBL og studentaktive læringsformer.* 3rd ed. Oslo: Universitetsforlaget, 2017.

[30] C. C. Aggarwal, *Recommender Systems: The Textbook*. London: Springer, 2016. DOI: `DOI10.1007/978-3-319-29659-3`.

[31] P. Wouters, E. Spek and H. Oostendorp, 'Measuring learning in serious games: A case study with structural assessment,' *Education Tech Research Development*, no. 59, pp. 741–763, 2011. DOI: `10.1007/s11423-010-9183-0`.

[32] Scrum.org. (2021). 'What is Scrum?' [Online]. Available: `https://www.scrum.org/resources/what-is-scrum` (visited on 29/04/2021).

[33] M. Ventures. (17th Apr. 2020). 'Converting-a-twine-story-to-unity,' [Online]. Available: `http://www.mrventures.net/all-tutorials/converting-a-twine-story-to-unity` (visited on 19/05/2021).

[34] W. Wu, H. Hsiao, P. Wu, C. Lin and S. Huang, 'Investigating the learning-theory foundations of game-based learning: A meta-analysis,' *Journal of Computer Assisted Learning*, vol. 28, no. 3, pp. 265–279, 2012. DOI: `10.1111/j.1365-2729.2011.00437.x`.

Appendix A

Group Members and Project Roles

A.1 Group Members' Backgrounds and Competencies

Ludvig Lilleberg started using Unity during the autumn 2020 semester at NTNU, and used it throughout the semester in the Game Programming and Rapid Prototyping courses. All four games he developed and co-developed during the semester explored different game mechanics, giving him experience in various features of the game engine. Using Unity, he gained experience in C# programming, 3D Unity animation, Unity physics, player controls, non-player character (NPC) movement and actions, terrain and game environment creation, graphical user interface (GUI) elements and functionality, and more. He also learned how to create simple 3D models and how to animate them in Blender. He also has some experience with Firebase and Firestore, from the Mobile/Wearable Programming course.
Among the skills Ludvig had to learn during the development of the bachelor game was how to create mobile games in Unity and how to test them using a mobile phone, simple mobile touch and computer mouse controls, smooth player movement and rotation towards a point on the screen that is touched or clicked, using Firebase and Firestore functionality in Unity, using asynchronous functions and how to make sure they finish before the game needs their results, making sure functions run in the correct order, the importance of setting variables and button functions in the script rather than in the Inspector, and GUI menu navigation and functionality.

Vanja Falck started using Unity during the autumn 2020 semester at NTNU, and used it throughout the semester in the Game Programming and Rapid Prototyping courses. Her main focus is on artificial intelligence in games. She has used ML-agents and reinforcement learning, implemented graph based searches and finite state machines in Unity3D games. During this project she acquired skills in using dynamic Bayesian networks (Hidden Markov Models) by first developing the framework on a python platform and then porting the online part of it to C-Sharp and the Unity3D game.

Kristoffer Madsen used Unity first in 2017 during a year of folk high school, but did not spend much time on it during the first couple years of the university. His main focus has been coordinating the conversations in the game with the GUI and creating the GUI for the conversations. Has also worked with Twinery and implementing Twinery into the project. Kristoffer has acquired skills in usage of asynchronous functions, working with two different code sets, making sure functions run in correct order, setting up GUI and the importance of setting variables through script.

A.2 Project Roles

During development, Ludvig Lilleberg has been a game designer, programmer, 3D modelling artist, 2D GUI artist, animator, and database modeller. He has been responsible for 3D models and animation, 3D environments, avatar creation functionality, player controls, 2D menu graphics and functionality, part of the NPC-to-NPC conversation functionality (Section 11.15), and part of the Firestore database structure.

During development, Vanja Falck has been responsible for implementing the Hidden Markov Model as a tool for dynamic difficulty adaptation and emotion tuning. She has contributed to the database infrastructure by setting up the google sheet pipeline and researching dialogue systems finding the online service twinery.org. She used first a python and later a Unity3D framework set up with a combination of a GOAP and twinery based dialogues to develop and test the main parts of Hidden Markov Models that later was transferred to the Unity3D game mock-up.

During development Kristoffer Madsen has been doing game designer, programming and 2D GUI art. He has the responsibility of contacting and organising meetings with our client and supervisor. He has had responsibility over the Twinery scenario and conversation to screen.

Our client's contact person, Espen Torseth, and our supervisor, Mariusz Nowostawski, have participated in meetings with our group. They have answered our questions and provided feedback on our ideas. Mariusz has also provided general tips for writing the bachelor report.

A.3 Contact Information

For any questions about our Unity project or our project repository, our group can be contacted at: *vanja.falck@ntnu.no*.