Faculty of Engineering and Science
Aalborg University

Department of Computer Science

TITLE:
NoEsc

THEME:
AI Programming and User Experience

PROJECT PERIOD:
SP2,
February 4^{th} 2009 -
June 4^{th} 2009

PROJECT GROUP:
sp201a

GROUP MEMBERS:
Anders Ejlersen
Anders Tankred Holm
Rasmus Kristensen
Kim Jung Nissen
Mads Bøgeskov

SUPERVISOR:
Rene Rydhof Hansen

SYNOPSIS:

This report describes the process of using the Source SDK to develop NoEsc a stealth first person game with an appertaining Artificial Intelligence. Since the Source Engine was a required part of the project, the game had to made to fit this. The AI should work as the player opponent and was called Aslan, Automated Search and Locator ANdroid. Several AI techniques has been analysed, and a behaviour tree was chosen as the structure as the AI. The game was tested on two groups and it was found that the AI seemed intelligent and the game itself was found to be amusing, though hard.

NUMBER OF COPIES: 8

REPORT PAGES: 63

APPENDIX PAGES: 12

TOTAL PAGES: 75

Preface

This report has been written during the SP2-project period, by group sp201a at Aalborg University. The main theme is Artificial Intelligence (AI) Programming and User Experience. This report is addressed to other students, supervisors, and anyone else who might be interested in the subject. To read and understand the report correctly, it is necessary to have knowledge equalling a bachelor in computer science and basic knowledge concerning AI and user experince.

The entire report is written in English and no translation will be accessible. Abbreviations and acronyms will at first appearance be written in parenthesis, to avoid breaking the reading stream. Specification of gender in the report is not to be understood as suppression or any other form of political/religious position. The gender is only specified to simplify the process of writing for the authors.

References to sources is marked by [#], where # refers to the related literature in the bibliography at the end of the report.

The appendix to the report is found in the last chapter of the report, and on a DVD, located on the very last page of the report. The source code created under the project is also located on the disc.

The report is written in LaTeX and is accessible as a PDF-document, which can be read with Adobe Acrobat Reader.

Project group sp201a
Anders Ejlersen Anders Tankred Holm

Rasmus Kristensen Kim Jung Nissen

Mads Bøgeskov

Acknowledgements

Valve®, Source™, Steam™, Half-Life™, and Half-Life^{2}™ are either registered trademarks or trademarks of Valve in the United States and other countries. All other trademarks are the property of their respective owners.

For testing:

- SW802B
  - Mogens Kraus
  - Mikkel Søbye
  - Lars Kaastrup Vinther
- SW803A
  - Peter Heino Bøg
  - Allan Mørk Christensen
  - Morten Justesen
  - Martin Midtgaard
  - Anh Tuan Nguyen Dao
  - Luxshan Ratnaravi

For advice and inspiration:

- Nicolaj Søndberg-Jeppesen
- Yifeng Zeng

Contents

Chapter 1

Introduction

The main topics of the SP2 semester are AI Programming and User Experience, and the task has been to develop a game with an AI using the Source Engine. Developing a game includes analysis, design, implementation, and lastly testing of the game.

The goal of the group is to create a singleplayer game called NoEsc which contains computer controlled opponents, that works as the player resistance.

The vision is to create a game that takes place in an office building, where the mission is to hack different objectives. When all objectives has been hacked the player should run to the exit and complete the level. The computer opponents are authorities trying to stop the player from hacking the objectives. The authorities are not aware of the players position and must therefore work together in squads to try to locate the player by searching the building. The player should compete against the authorities by hiding from them, and distracting them. Lastly the game should provide different difficulty settings to suit the players skill level.

The vision entails technical challenges for the construction of a proper AI and also the use of the Source Engine. Another challenge in the game design is to make the game balanced and fun. Therefore it is necessary to set the proper difficulty values for the different difficulty settings.

Some screen-shots of the final product can be seen in Appendix C.1 on page 71 and a list of the content on the DVD can be seen in Appendix D.1 on page 75.

Chapter 2

Game Design

This chapter introduces the game concept for NoEsc. This covers the story, game rules and the visual style of the game, and lastly the capabilities of the Non-Player Character (NPC). First the game summary introduces the story line, giving a short introduction to the game concept. Next the game rules are introduced, which covers the player objectives and some basic requirements of the NPC. Then the visual style of the game is covered, which introduces the environment the story occurs in. Afterwards the job tasks are covered for the NPC, which are the actions the NPC has to be able to do, when interacting with the environment. Lastly the different difficulty levels and their differences are described.

2.1 Game Summary

NoEsc takes place in the present where the country has been taken over by a corrupt government. The protagonist is an underground terrorist whose speciality is to gather information by infiltrating government buildings.

The game starts at the point where the protagonist just has gained access into the building. Here she receives a message telling her that the building is being searched and that government forces will try to capture her alive, see figure below.

The authorities will attack the building from three points and will then search through the entire building to find her. The forces will split up into squads, but because of a technical error they can not use the radio. This means that they have to rely upon their own information and updates when meeting other squads.

The game focus is stealth, which must be utilised by the player throughout the game to get unnoticed to the objectives. The inspiration for a game like

this are games such as Tom Clancy's SplinterCell[1] and Thief[2]. However these games enable the player to incapacitate the opponents which is a feature NoEsc will not support to force stealth.

Dear Falcon,

This is it, our final chance. I will not kid you the prospect of survival on this mission is small, but based upon your previous achievements I gather that this is not a problem, and that you once again will show your excellence. As we learnt on your last mission the office building on 51st Grand Street is in fact a front for a government building and they have been sent several of the computing units that controls the infrastructure in the whole country. These units contains multiple encryptions keys that, if we can get our hands on them, potentially help us make enough chaos in the public for our forces to attack their fortifications and lay their corruption to dust. Your mission is to infiltrate this office building and get these units. They might be spread out over the entire building. Your wrist PC should show you an overview map with the locations of the units. The locations have been supplied by an insider so they might or might not be true.

Good luck and remember you have the thoughts of the entire organization with you,

Orion.

[CONFIDENTIAL]

DECLARATION (NON-ADHESIVE)

PP70 B

Impresión digital del dedo pulgar de la mano derecha

URGENT!

Our inside source has just admitted to have alerted the authorities. Your cover is blown.
Army forces are storming the building as we speak!
They will try to capture you alive, do not let them!

Get the units and run!

Orion.

[CONFIDENTIAL]

2.2 Game Rules

The goal of NoEsc is to hack seven computers and then escape through the front door located in the lobby. In order to accomplish this the player must use stealth to sneak around in the building and distract the NPCs with sounds. The player is able to pick and throw chairs and bottles to make sounds to attract the NPCs to the source of the sounds. This way the player can distract the NPCs while she is moving from one place to another. Besides sounds the player is able to hide in rooms and offices to keep out of view of the NPCs. An alarm is placed in the office building, and the player needs to avoid triggering it. If the alarm is triggered the alarm will attract NPCs and doors will seal the area where the alarm is placed to capture the player.

The player player has an cloaking ability, which can be used if the player gets into a situation that she cannot rescue herself from. The cloaking ability can however only be used one time.

The NPCs searches through the building with the sole goal of finding the protagonist. As mentioned it reacts to sounds such as footsteps, bottles and chairs. If an NPC sees the player it will run towards her and try to get into range to hit her with its stun stick. The NPCs will not wander around alone, but rather in squads of two or more. The general tactic when storming a room is to keep a guard at the door and then let the rest of the squad search the room.

2.3 Visual Style

The visual style must represent the environment described in the story. The office building is very boring and depressing in the areas where the people work in cubicles. These rooms can only have standard office lighting to further emphasise it.

The rooms that represent the building when there are visitors, such as the auditorium or the management offices, must be more interesting. This can be achieved with items such as fish tanks, but also with the sunlight through windows.

The Heads Up Display (HUD) of the game will only contain an overview map and game text. The overview display shows a floor plan with the objectives and the location of the NPCs storming the building. The player will not have access to any weapons so there will be no indicators for this. When the player interacts with an objective, a progress bar will be shown. Lastly when objectives has been hacked, the number of objectives remaining is displayed. In Figure 2.1 on the following page a illustration of the HUD can be seen.

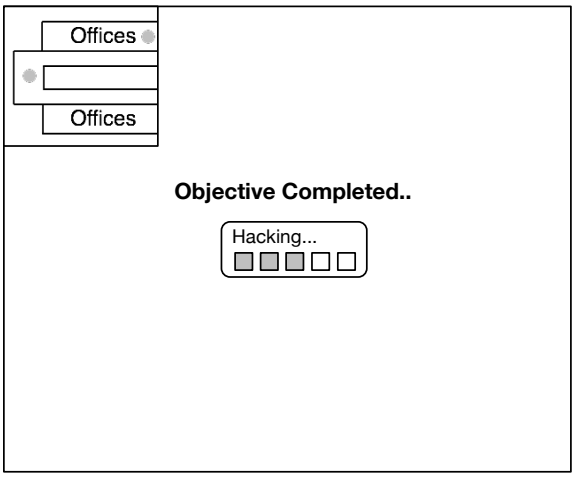

Figure 2.1: A mock-up of the HUD in NoEsc.

2.4 AI Specification

The NPC in NoEsc is called Automated Search and Locator ANdroid (Aslan). These tasks are specified in this section and together describe the overall behaviour of Aslan.

Search: The primary goal of Aslans is to search for the player in a building consisting of rooms, cubicles and hallways. Therefore Aslan has to be able to do the following:

- **Path:** Know where it has been and where to go.
- **Find:** Be able to prioritise rooms where the player might be hiding.

Squad: A squad based behaviour both as leader and as squad member:

- **Leader:** The leader needs to co-ordinate the members of the squad to do certain tasks, like guarding the entrance to a room, searching the room, and so on.
- **Member:** The member needs to follow the orders of the leader and report in if meeting other squads or seeing the player.

Communication: Communication between other Aslans could be the following:

- **Player:** If the player is spotted/heard Aslan needs to yell that it has seen the player or investigating a sound source.

Meeting Aslans: Either needs to exchange last searched locations or if an Aslan is not in a squad tell to form/join a squad with the other Aslans.

Orders: If the leader needs to give orders to the squad members, or if member needs to receive orders.

Attack: When the player is located Aslan needs to do the following:

Chase: Chase the player and try to pacify her.

Escape: If the player escapes, then Aslan needs to check the last known position of the player and afterwards return to its squad.

Distraction: Aslan should be able to be distracted from previous actions, except if Aslan is engaging the player, by the following:

Alarm: If an alarm goes off in the vicinity, then Aslan should investigate the source.

Running: If a running sound is heard, then Aslan should investigate the source.

Other: If distracted by another sound source, like a bottle, then Aslan should investigate.

When the game starts the Aslans will spawn at different locations in the map. They will then be split up into squads, each with a dedicated leader. These squads will then go and search the building. Whenever a squad meets another squad they will interchange information such as sightings, sounds, and recently visited locations. This will make sure that if an Aslan has been chasing the player then he will inform other Aslans about it, also the interchange of locations will ensure that the Aslans will not search the same area again and again.

2.5 Difficulty

The game should have different difficulty settings, which allows the player to choose the setting that fits the current skill level of the player. There are different parameters which can be altered to change the difficulty level, which are listed here:

- Senses like sight and hearing.
- Numbers of opposing players.

- Mini map visible to the player.
- Damage caused by opposing players.

These three elements can be adjusted to give the player different difficulty settings. The following is an example of the three difficulty settings that could be used:

Easy: Allowed a mini map where the Aslans are illustrated, low number of Aslans, low amount of damage caused and all Aslans has short sight and narrow view angle.

Medium: Allowed a mini map, where the Aslans are illustrated at close proximity, medium number of Aslans, medium damage caused and all Aslans has medium sight and medium size view angle.

Hard: Allowed a mini map, high number of Aslans, high amount of damage caused and they have long sight and a large view angle.

These three settings could also reflect upon the hearing sense of the Aslans, such that foot steps are prioritised lower at easy setting, then on medium or hard setting. Moreover the speed of the Aslans could also be reflected upon in the settings and the number of cloak available.

2.6 Summary

This chapter covered the foundation of the game: This includes the visual appearance, the overall game idea, the rules of the game, and the specification of how the Aslans has to work. The story and the visual style is bound together to form a modern office building game world, that reflects the corruptness of the world. The game rules tells how each party of the game, the Aslans and the player, tries to win. In addition to this the difficulty levels of NoEsc was defined to determine which differences easy, medium and hard difficulty should have. The next step to create the game is to get a closer look at the Source SDK.

Chapter 3

Source SDK

In 2004 Valve released the second chapter in the Half-Life series[3]. Like with the release of the first title Valve chose to include a Software Development Kit (SDK), to allow third party developers to create their own games based upon the same engine as Half-Life^2, the Source Engine. This type of game is usually known as a Modification (Mod), and can be made available to owners of Half-Life^2.

Half-Life^2 is a First Person Shooter (FPS), and the Source Engine it is built upon is therefore created to make a first person game. Therefore the Source Engine supports NoEsc as it is a first person game.

The Source SDK is a requirement of the project and it is therefore needed to have a basic knowledge of the possibilities of the tools available. The first part of the chapter will provide a basic overview of the Source SDK. After this the main parts of the Source SDK will be described.

The source for this chapter is the Valve Developer Community[4].

3.1 Overview

The Source SDK is available through Valve's Steam platform. Steam is a platform to distribute games digitally. The Source SDK is dependent upon Steam and can only be utilised if the user owns Half-Life^2.

The Source SDK contains two main parts, the code and the Hammer Editor. The code is split up into a client and a server, as the Source Engine is designed to create multiplayer as well as singleplayer games. The client is where e.g. the HUD is defined, and the server contains e.g. the game logic functionality. The other part namely the Hammer Editor is used to create the level and attach the game logic to this. The connection between the code and the Hammer Editor is done through a script file called Forge Game

Data (FGD) data. This means that entities can be made in the code, then added to the FGD file, and finally the entity can be used in the level via the Hammer Editor.

3.2 Existing AI

The Source SDK provides basic AI functionality that developers can use when making a Mod. This includes pathfinding and a customisable sensing system to enable the NPCs to see and hear. This means the developers can decide how far and how wide the NPC can see.

The Source SDK also provides a scheduling system to control AI. This schedule system will be filled with schedules containing tasks. the AI must then perform these tasks, and execute them in the appropriate order. Standard schedules are provided with the Source SDK such as attack. The last major part of the bundled AI is the squad handling system, which also is handled in the scheduling system.

3.3 The Code

When a Mod is to be created the basic building blocks for the Mod is written into the code. These building blocks along with the default functionality in the Source Engine can be used in the Hammer Editor. This section provides an overview of the different parts of the code that is provided. The HUD is the element to be discussed. Then the console with the variables and the commands within it will be discussed. Afterwards the topic to be explained is entities and their central role in the Source Engine. Furthermore the sound system of the Source Engine is discussed. Lastly the creation of a new NPC and nodes are explained.

3.3.1 HUD

The HUD system is placed in the client. The visual appearance of the HUD elements are defined in a script file, which is placed beside the compiled code. The script contains settings like placement on the screen, graphics, colour and so forth. However a HUD element has to be defined in the code, where it is an object which inherits from the `CHudElement` and `vgui::Panel` classes. An advantage of having the HUD defined in a script is that the code does not need to be compiled when the HUD is changed.

3.3.2 Console

The Source Engine provides an in-game console. The console is mainly for development purposes as a command can be executed or a variable can be changed. This can be used to tweak the game, e.g., change the running speed of the player, or to execute a special function. Therefore the console can be time saving, such that the code does not have to be recompiled each time a change has to be tested.

There are two types of entries to the console, a convar and a concommand. The convar is a variable that is defined in the code, which can be temporarily changed through the console. An example of how a convar is defined can be seen Listing 3.1.

```
1 ConVar in_code_name( "in_console_name","default_value");
```

Listing 3.1: Definition of a convar.

The concommand is also defined in the code, but however does not represent a variable, but executes a specified function in the code.

3.3.3 Entities

In Source SDK everything is an entity, NPCs, models, sound emitters and so on. Some entities can be seen, like models, and some cannot, like sound emitters. Because of this everything shares some common functionality, e.g. an identifier, so it is possible to differentiate between the different entities. Therefore to find an entity of a specific type, a global variable in the code is provided, namely `g_EntList`. This variable contains a list of all the instantiated entities and can be searched to find entities of a specific type.

A node is an entity in the game world that contains information, e.g. `ai_nodes` contains links to other `ai_nodes`, which are used for navigation. `Ai_nodes` are standard functionality and can be placed in-game using the Hammer Editor. These nodes however are designed for the already existing pathfinding system implemented in Source Engine.

If another type of node is needed it can be created in the code. Such a node is created by making a class that inherits from the `CPointEntity` class. Methods and variables to contain information can then be added to the node class, to enable it to handle custom tasks.

3.3.4 NPC Creation

When a new NPC has to be created, a template called `monster_dummy` can be used as the base. This template contains the basic functionality for a NPC

in the Source Engine. This however relies on a scheduling system to handle movement, attack, and general behaviour. This default behaviour is based upon the behaviour needed in Half-Life[2] and is therefore not desirable in all cases. If this scheduling system is disabled the functionality for movement and attack has to be re-implemented manually.

3.3.5 Think

In the code there are some entities that implements a `Think` function. The function is called from the lower parts of the engine with a given interval. This interval can be set from call to call with the function shown in Listing 3.2.

```
1   SetNextThink(gpGlobals->curtime + 1.0f),
```

Listing 3.2: Sets when to think next.

which in this case will create a delay of one second before the next call.

The `Think` function is used in, e.g., a NPC to enable it to handle the AI code with a given interval.

3.3.6 Sounds

The sound system of the Source SDK uses a script system to define the sound with name, volume, sound files, pitch and so on. When these sound scripts are specified the sound can be used on any entity from within the Hammer Editor or the code. An example of how a sound could be played from the code is shown in Listing 3.3.

```
1   entity->EmitSound( "Sound.Name.From.Script" );
```

Listing 3.3: Code to make entity emit sound.

The command plays the sound with the entity location as the source location of the sound.

3.4 Hammer Editor

The Hammer Editor is the tool used when designing levels for Source based games, and is a part of the Source SDK.

The Hammer Editorcan be used for the creation of level architecture: Geometry, texturing and lighting. When placing a model in the Hammer Editor there are three options for the entity type to which the model will be assigned: Static, dynamic and physic. A static entity is an object that

can not be affected by anything in the game world. A dynamic entity can be animated and/or be moved with another entity. The last is the physics entity which can be thrown around by other entities in the game world, e.g. by the player, this object is controlled by the physics system.

Models can be made in other 3D modeling applications such as Maya or 3D Studio Max, and later imported into the Hammer Editor by adding the model to a FGD file. The Hammer Editor needs its models in a special file format called mdl, which is created by taking the model from the 3D modeling application and converting it through studiomdl. Studiomdl is a model converter, and is also a part of the Source SDK.

The Hammer Editor can also be used to place various kinds of entities to control the gameplay, like entities for scripting input and output.

Source Engine provides its own AI as seen in Half-Life^2. This AI can be scripted in the Hammer Editor and then assigned to NPCs.

A picture of the Hammer Editor can be seen in Figure 3.1, where the player entity is visible.

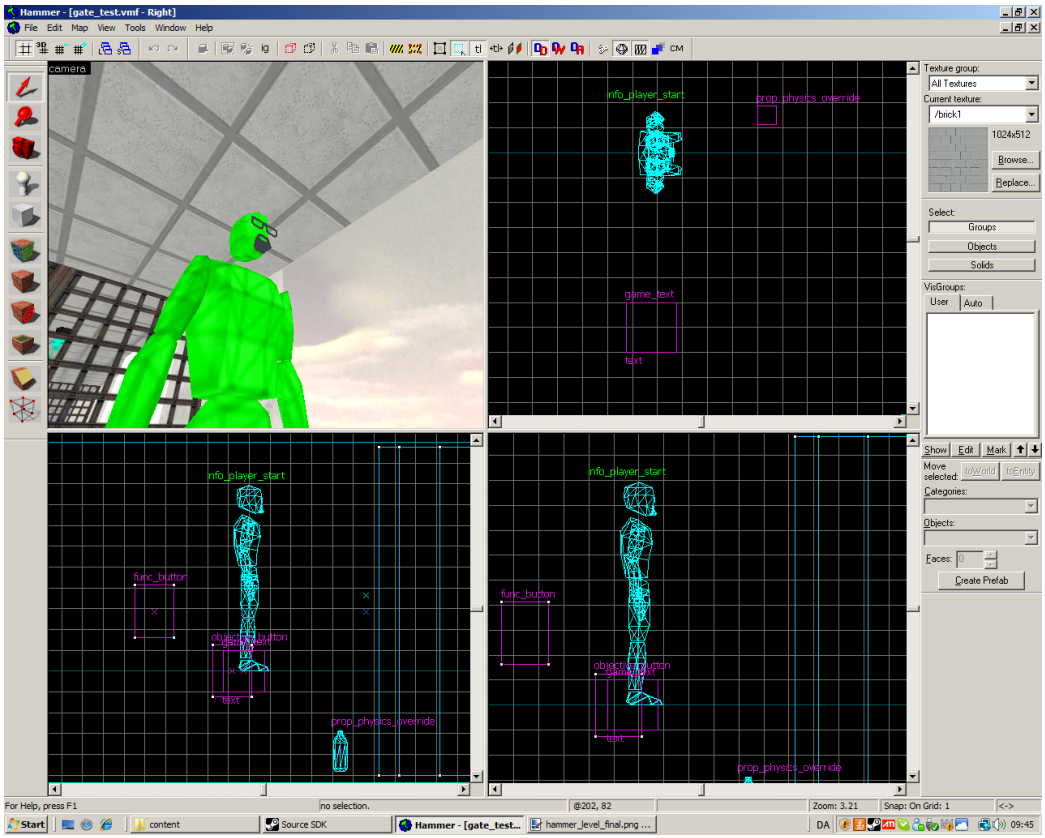

Figure 3.1: The Hammer Editor with a player entity.

Compiling

A level that is to be used in the Source Engine has to be compiled. This is because several things has to be prepared before a level is compatible with the Source Engine, e.g. the level must have a Binary Space Partition (BSP) tree, and all lightmaps must be created.

The Hammer Editor provides three compilation tools to create a level: VBSP, VVIS and VRAD.

1. **VBSP:** Generates the BSP, which in effect is the map. It uses the brushes and entities created in the Hammer Editor to construct the architecture of the level.

2. **VVIS:** Determines what is visible in the map and then ignores items that cannot be seen, such as lights and geometry.

3. **VRAD:** Computes the light maps, and High Dynamic Range Imaging (HDR) lighting.

The Hammer Editor is able to run all of these tools in the right order and then return a BSP file that can be used in the game.

3.5 Summary

The Source SDK has been covered in this chapter, where an overview of the Source SDK has been described. Furthermore the important parts of the code for this project of the Source SDK has been covered. Lastly the level and scripting tool, Hammer Editor, has been covered, including the compiling process of a level created in the Hammer Editor.

Chapter 4

Artificial Intelligence

This chapter first introduces AI used in games followed by different AI techniques, which can be used in creating a NPC for a game. The last section describes the choice of AI technique that will be used in the game.

4.1 Artificial Intelligence in Games

The source of this section is Artificial Intelligence for Games[5]. In a game the AI is used to enhance the player experience in such a way that it either can provide the player with resistance or try to help the player reach a goal. An AI can be a visible entity in the game world, or an godlike entity that controls entities in the game world. Such AIs is constructed out of parts such as Pathfinding, Scripting, and a way to make decisions regarding problems in the game world.

Most AIs in the game world needs to find a way between locations in order to move around the level. This is a recurring problem present in many types of games. Therefore optimised and general solutions exists such as the A* algorithm.

When developing an AI for a game some times the AI should act autonomously while at other times the behaviour should be strictly specified. Such a specified behaviour is e.g. in a cutscene where the behaviour must be specified to tell a story. The specified behaviour can be achieved by scripting, which combines several AI actions into a sequence which then can be executed whenever needed.

The autonomous behaviour requires a method for the AI to make decision based upon some information from the game world. Several techniques exist which enables the AI to make a decision. The problem of making a

good decision making system is, it has a large amount of parameters which needs to be specified. Since the definition of these parameters is done by the developers some values might still need further tweaking, which is an huge task. Therefore techniques has been devised able to tweak upon these values by learning from continuous evaluation of the AI.

4.2 AI Techniques

This section introduces different AI techniques that can be used in games for developing a NPC. These techniques are: Decision trees, behaviour trees, neural networks, fuzzy logic and bayesian networks. These techniques is described briefly, and the advantages and disadvantages is listed in order to provide an overview. The last part of the section chooses an AI technique that is to be used as the basis of NoEsc. The chosen AI technique is then compared to which possibilities the Source Engine provides.

4.2.1 Decision Trees

A decision tree[5] is made up of connected decision nodes, where the starting decision is called the root, and the leaves are actions. The decision nodes usually consists of easy boolean logic, that is: "Is treasure chest visible?", this question only has two answers: "Yes" and "No". The reason for this is boolean expressions can be created by the decision tree, e.g. AND of a and b is determined by building the decision sequential, that is: If a is true, then go to b and if b is true, then do action. Similary with OR, where the order is swapped: If a is true, then do action, else do b and if b is true, then do action. So a more complex boolean expression can be created by the decision tree.

An example of a decision tree is illustrated in Figure 4.1 on the next page.

The decision nodes are influenced by the knowledge base that is used to make the decision. If e.g. the treasure is visible, but locked, then the decision tree will end in the action: "Use key" in Figure 4.1 on the facing page. The reason decision trees often has a knowledge base consisting of the global game state is they are simple and fast decision makers.

A decision tree is usually build as a binary decision tree, because of easier binary testing and optimization. A non-binary decision tree would have the advantage of being a flatter and a less broad structure, because more than two choices from one decision node is possible, and thereby more efficient.

Operation and technique used for binary trees can be used on decision trees, like balancing the tree. Timing can be used to prevent that the same

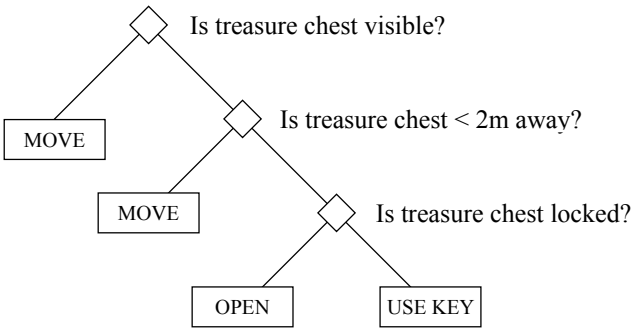

Figure 4.1: Decision tree about a treasure chest consisting of three decision points and four actions.

action is picked all the time. This often occur when random decision making is present. Therefore timing can be used to force a new action.

Advantages and Disadvantages

Decision trees have different advantage and disadvantage, which are listed below. First the advantages:

- Easy and fast decision making.
- It is possible to apply learning to the technique.

The disadvantages are:

- Binary trees are often flat, but broad.
- Deduced rules can be complex.

4.2.2 Behaviour Trees

This section is based upon the webpage AiGameDev[6]. A behaviour tree is a formal and graphical representation of a behaviour in tree form. The behaviour affects individual or networks of entities, which make decisions or exchange information by evaluating its tree.

A behaviour tree is a simple representation for decision making, which makes the notation expressive, simple, uniform and easy to use. E.g. a behaviour tree is able to use modularity such that there can be created reusable states to provide logic for different goals in their own trees, that is self-contained states.

Behaviour trees contains four building blocks:

Behaviour: The leaves of a behaviour tree, that composes the behaviour of the NPC, e.g. if it should run or walk and where to.

Sequence: A sequence of behaviours. If a child behaviour succeeds the sequence continues to the next child. If a child behaviour fails, then the sequence backtracks.

Selector: The selector selects a behaviour to execute. The selector can have different criterias for selecting the child behaviour, such as the probability selector and the priority selector. The selector has different termination criterias to deal with: If a child behaviour succeeds, the selector can terminate successfully. In case a child behaviour fails, the selector may backtrack and try the next child in order.

Decorator: A decorator adds functionality to a behaviour. It takes an existing behaviour and adds features. The decorator can be placed in the tree as an extension to a subtree, which creates more complex behaviour. Different kinds of decorators are present. Filters: Limit number of times a behaviour can be run; Prevent a behaviour from firing too often with a timer.

A behaviour is represented by a circle, sequence by a square, selector by a round-cornered square and a decorator by a diamond. A small example about a treasure chest is illustrated in Figure 4.2 on the next page, where it first checks if it can see the treasure. If the treasure chest is not visible, it looks around. If the treasure chest is visible it first moves towards the treasure and then checks if the treasure is locked. If the treasure chest is not locked it opens the chest, if it is locked it uses a key to open the treasure chest.

Advantages and Disadvantages

Behaviour trees have different advantages and disadvantages, which are listed below. First the advantages:

- A high level structure, which makes it easier for the designer to implement AI.
- Can use known AI techniques and algorithms like decision trees in the selector.

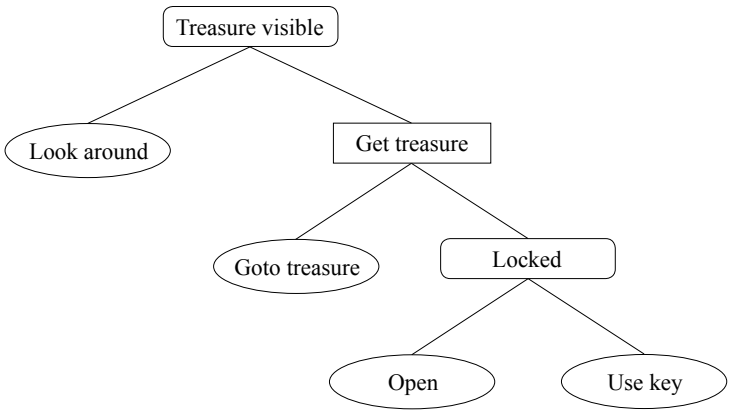

Figure 4.2: Behaviour tree about a treasure chest, where it either can look around or go to treasure and try to open it.

- Used in new games, like Spore and Halo 3.

The disadvantage is that it is a way to structure an AI and not a complete AI solution and therefore requires a decision system to function.

4.2.3 Fuzzy Logic

Fuzzy Logic [5] is a decision making technique that uses *fuzzy sets*. Fuzzy sets have values that define the membership of a certain set, e.g. a player is 0.3 hurt and 0.7 healthy. This means that it is possible to have memberships of multiple sets e.g. where the player is 0.2 hidden and 0.8 visible, but where mutual exclusive sets values will summarize to 1.

To be able to turn the original data, like boolean, integers, and so on into degrees of membership and back again, the following functions are used:

Fuzzyfication: Turns original data into degrees of membership.

Defuzzyfication: Turns membership values into data values, that may be usable other places in the code e.g. for setting the remaining life of the player.

There exist different kinds of defuzzyfication functions, which are blending based on membership, defuzzyfication to a boolean value, defuzzyfication to an enumerated value, centre of gravity and highest membership. The latter of them takes the highest membership value and is then defuzzyfied into the output value.

Following the fuzzy sets and functions, there also exist *fuzzy rules*, which relate certain membership values to generate a new membership value for other fuzzy sets. An example of this is if having a chest with four locks, locked is 0.6, and it was poorly hidden, hidden is 0.2, then a rule for treasure with the use of the operation AND:

$$m_{treasure} = min\left(m_{locked}, m_{hidden}\right),$$

then $m_{treasure}$ will have a value of 0.2. There are different operations, e.g. like OR and NOT, which can be used to create combined facts and thereby used as rules.

Advantages and Disadvantages

Fuzzy logic have different advantages and disadvantages, which are listed below. The advantage is that it is easy to fine tune by editing membership values.
The disadvantage is that it requires expert to apply the correct degrees of membership.

4.2.4 Bayesian Networks

Bayesian networks [7, 8] is a model representing a set of variables and their probabilistic independencies. This representation is a directed acyclic graph, where nodes represent the variables and edges represent relationships. The edges are directed and they point from the parent to the child. The Bayesian networks use probability theory to check the relevance of one or more conditions. These probabilities are known as Bayesian rules and consists of different operations, like:

$$P\left(A|B\right) = \frac{P\left(A \cap B\right)}{P\left(B\right)},$$

where A and B are events, $P\left(B\right) > 0$ and $P\left(B\right)$ is known as the probability of B, where $B \subseteq S$ and S is known as the sample space. $P\left(A|B\right)$ is the probability of A given event B. A small example given a fair die, what is the probability of that the die turns up three given that the die turns up a prime number:

$$
\begin{aligned}
P(A = \{3\} \,|B = \{2,3,5\}) &= \frac{P(\{3\} \cap \{2,3,5\})}{P(\{2,3,5\})} \\
&= \frac{P(\{3\})}{P(\{2,3,5\})} \\
&= \frac{\frac{1}{6}}{\frac{3}{6}} \qquad\qquad = \underline{\underline{\frac{1}{3}}}
\end{aligned}
$$

Given the probability calculus there are different rules that can be used to create Bayesian rules, like the fundamental rule and Bayes' rule, which are two important rules when using Bayesian networks.

The following is an example of a Bayesian network of whether a lootable chest is locked, trapped and/or containing treasure and is illustrated in Figure 4.3.

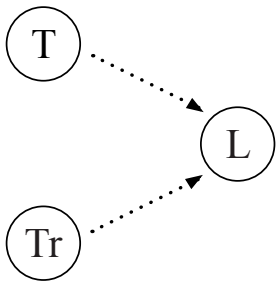

Figure 4.3: A Bayesian network illustrating treasure, where T is Trapped, Tr is Treasure and L is Locked.

A Conditional Probability Table is connected to each of the nodes, this table represents the probability of an event happening, e.g. the probability tables in Figure 4.4.

True	0.5
False	0.5

(a)

True	0.5
False	0.5

(b)

Treasure	True		False	
Trapped	True	False	True	False
True	0.8	0.5	0.0	0.4
False	0.2	0.5	1.0	0.6

(c)

Figure 4.4: (a) Trapped (T), (b) Treasure (Tr) and (c) $P(L|T,Tr)$ (L).

There are two states for the variable trapped, the chest is either true for trapped or false for not trapped. However if a node is a child node it has to take into account the parents, e.g. if the chest is trapped and containing

treasure there is a 80.0% probability of it to be locked. This is found by looking in Figure 4.4 on the preceding page under true for treasure and true for trapped, reaching the value of 0.8 in true and 0.2 in false. However if the likelihood of the chest being trapped is 0.7 and that the likelihood of treasure is 0.4, then the probability of the chest is locked is 35.6%. This is found by calculating the likelihoods of trapped and treasure, which both has to sum to 1, so its just taking the value given, g, and solve the equation: $1 - g = n$, where n is the not known value. Then for each cell in the table the states are found that corresponds to the values just calculated, that is if the cell is where trapped is true and treasure is true, then the following is calculated: $0.8 \cdot 0.7 \cdot 0.4$ and so on for all cells as illustrated in Figure 4.5. Lastly each row are summed, such that the row with the name "True" is: $0.224 + 0.060 + 0.000 + 0.073 = 0.356$ and also for "False": 0.644. Thereby given the probability of 35.6% for true and 64.4% for false.

Treasure	True		False	
Trapped	True	False	True	False
True	$0.8 \cdot 0.7 \cdot 0.4$	$0.5 \cdot 0.4 \cdot 0.3$	$0.0 \cdot 0.6 \cdot 0.7$	$0.4 \cdot 0.3 \cdot 0.6$
	0.224	**0.060**	**0.000**	**0.072**
False	$0.2 \cdot 0.7 \cdot 0.4$	$0.5 \cdot 0.4 \cdot 0.3$	$1.0 \cdot 0.6 \cdot 0.7$	$0.6 \cdot 0.3 \cdot 0.6$
	0.056	**0.060**	**0.420**	**0.108**

Figure 4.5: Table (c) from Figure 4.4 on the previous page, where the likelihoods have been inserted and calculated.

These tables has to be created beforehand for each of the nodes, this can be done by experts or using data mining.

Another approach is to use initial values, which are then updated by the NPC that has gathered new information, e.g. when the chest is opened and the outcome can be seen.

Advantages and Disadvantages

Bayesian networks have different advantages and disadvantages, which are listed below. First the advantages:

- Good at decision making.
- Probability of how likely a case is.
- Can learn if the probabilities are updated.

The disadvantages are:

- Computational expensive, so large networks are not recommended for realtime use.
- Probability tables has to be created beforehand.

4.2.5 Neural Networks

A neural network started out as an imitation of how the brain works. Further development has made this connection grow smaller, but neural networks are today used in many places such as Google Image Search [9]. The overall source for this text is a lecture made by Prof. P. Dasgupta from IIT Kharagpur [10]. The definition of a neural network is,

> *"Neural networks are complex non-linear functions that relate one or more input variables to an out variable."*

A neural network consists internally of non-linear processing elements, not unlike neurons in the brain, which are connected by a set of weights.

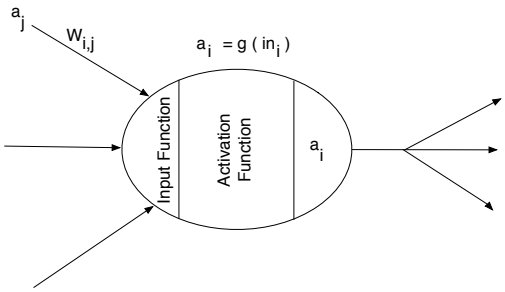

Figure 4.6: Illustration of the internals of a neuron in a neural network.

The neuron is a node in a neural network, and is illustrated in Figure 4.6. The arcs on the left side shows the input a_j into the neuron, this could come from either an input unit or from another neuron in a multi layered network. The input is then coupled with a weight $W_{i,j}$, which represents the weight from input unit i to node i. The neuron has several input links and the input function computes the total of the input:

$$in_i = \sum W_{j,i} \cdot a_j$$

The goal of the activation function is to transform the input computed by the input function into an output. The activation function $g(in_i)$ is a

function of the total input, and it returns the result a_i of the neuron, which is sent to the neuron connected to it. This could e.g. be a threshold function that returns 1 if input is greater than 0.7, and 0 if less than 0.7.

A neural network can consist of several layers of nodes, and the layers add more functionality to the network. A *perceptron network* is a single layered neural network. The perceptron network consists of perceptrons which can provide a boolean output. Therefore a perceptron network can e.g. be used when to attack an enemy based upon stimulations (input) to the perceptrons. This network is very limited in its functionality and can only be used to do simple functions. The problem with a multi layered network is that it is very expensive computationally.

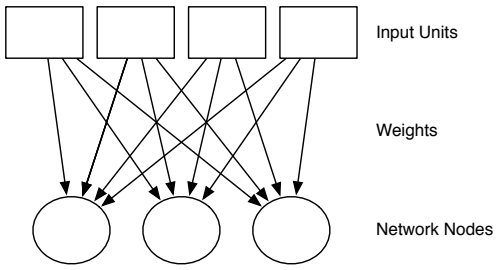

Figure 4.7: Illustration of a single layered neural network.

A single layered network consists of a set of input units and a layer of network nodes as illustrated in Figure 4.7. Each of these are connected by weights.

The functionality of a neural network is determined by a set of values on the weights between the nodes. So in order to train a neural network to perform a certain task, the network is provided with learning data. The learning data contains a set of input data and a set of the corresponding output data. The error correction of an output from a unit can be corrected by calculating $Err = T - O$, where O is current output and T is the correct output. Then a weight adjustment rule is used to adjust the weights such that the output node provides the corresponding output with the error differention.

Advantages and Disadvantages

Neural networks have different advantages and disadvantages, which are listed below. First the advantages:

- A Perceptron network is not as expensive as a multi layered network, and can be used in realtime applications.

- Neural network is known to be used in games for instance Black & White[11].

The disadvantages are:

- Computationally expensive; optimization and learning process.
- Large knowledge base for initial learning.

4.2.6 Choice

For this game the choice fell upon using a technique which could give a satisfying result in the behaviour of the NPC. Therefore the choice fell upon behaviour trees, since this technique gives an expressive, simple and easy to use technique for developing an AI for the NPC.

Behaviour trees provides the possibility of using other existing AI techniques in coherence with the selector. Therefore experiments will be carried out with Bayesian networks to compare this with a prioritisation selector, that is a regular if-else selector. This is done to see how bayesian networks can influence the selector in the way bayesian networks use probability theory to calculate a good choice of action.

It was chosen not to use the existing AI system from the Source Engine described in Section 3.2 on page 16 since it is not designed a game like NoEsc. The default schedules which Source Engine provides also conflicts with the wishes in NoEsc and can therefore not be used.

4.3 Summary

This chapter first covered an overview of how AI is used in games, whereas the AI techniques section covered some of the AI techniques that could be used to develop a NPC for a computer game. However the AI technique that was chosen for the game was a behaviour tree structure with a prioritisation sound selector, like choosing the highest volume of a given sound and follow it, but also a bayesian network will be created for experimentation with the plain method. These two methods will be compared to see which of the two gives the best player experience, but also to see what difference it does for the NPC. It was chosen not to use the existing AI from Source Engine.

Chapter 5

Design

This chapter covers the design of the behaviour tree used for the AI. How the constraints from the Source Engine affects the behaviour tree design and how the behaviour tree design is altered to fit these constraints. Afterwards a brief introduction to the functionality in Behaviour Tree Tool (BTTool), which is a tool for creating behaviour trees for NoEsc. Then the AI architecture is described, how the AI should search an area, how they should form squads together and how sound affects the AI. Lastly the level is described and then the types of nodes that are used in the level for the NPC.

5.1 Behaviour Tree

The behaviour tree design is covered in this section and starts with the constraints it had to accommodate. Afterwards a discussion of the the general design and the different nodes in the tree.

5.1.1 Constraints

The constraints the design of the behaviour tree has to accommodate are given from the choice of the Source Engine. In NoEsc the underlying architecture of the Source Engine dictates that the AI code is called with a given interval, e.g., with 10Hz as mentioned in Section 3.3.5 on page 18. In a behaviour tree there are the leaf nodes which are actions, these actions can take a long time to complete. A call to such an action must not be blocking, because this blocks the game as of the before mentioned constraint. Therefore the behaviour tree has to return within a short time, and will be called the next time the AI has CPU time. This means the state the behaviour tree is in has to be saved to continue executing the action.

5.1.2 Choices

Given the constraints there have been chosen a design for the behaviour tree that is different from the original design. Firstly it has been chosen to customize the nodes in the behaviour tree by writing code to specify what the different nodes should do, e.g. what actions a behaviour does and how the selector selects which subtree to execute. Because of this choice the decorator is not needed, since this can be done in code for the given node. An interrupt node has been added. This is because there is a need to be able to override a running behaviour, e.g. if the player is seen when an NPC is searching. The NPC should then stop searching and chase the player. The interrupt node is chosen to be symbolised by a diamond. These alternations accommodate with the constraints so it can be executed in the context of the Source Engine.

5.1.3 Node Types

There are different types of nodes in a behaviour tree, which were discussed in Section 4.2.2 on page 23. The modifications made to the nodes are:

Interrupt: A special node because each time the behaviour tree is updated this checks if the interrupt condition is set, if it is, it runs the interrupt subtree.

Status: The different nodes return status messages to inform if a node has completed successfully or not, or if it is still being executed.

In Figure 5.1 on the next page the sequence and the behaviour nodes are illustrated. This works by a test is executed to see if a child is already running, if this is the case the running child is called. Otherwise the running child is set to the first behaviour connected to the sequence, which then is called. Each time the behaviour is called a test is performed to determine if it is done, and if this is the case, it calls back to the parent node, with its ending status. The sequence can now determine if it should run the next child. If this is the case the child executed is stored as the running child and the selector will now call it each time the selector itself is called, and wait for it to callback. When all children are executed or one has failed, the selector make a callback to its parent with the status.

Figure 5.2 on page 35 the interrupt and the selector nodes are illustrated. The interrupt node is called each time the tree is called, when it is called it tests to see if its interrupt is set. When the interrupt is set, then it is currently running an interrupt, and it does not test for interrupts during this

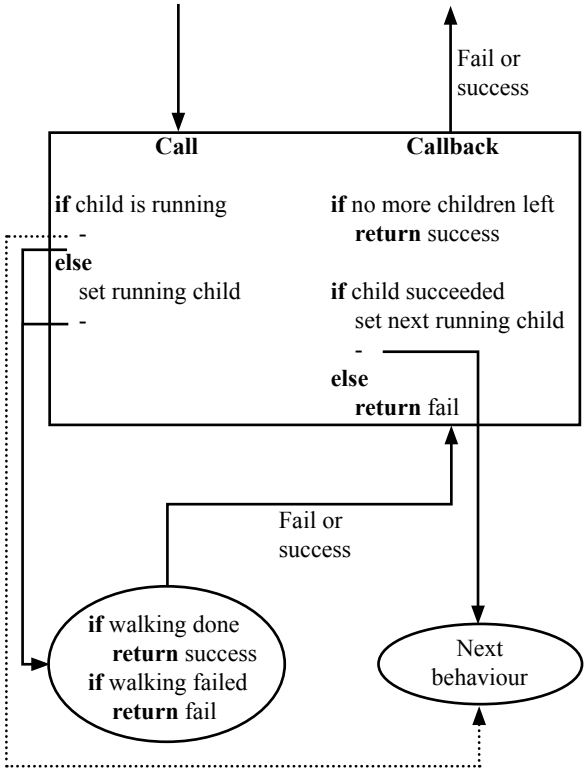

Figure 5.1: An example of how the constraints of the Source Engine could be overcome with respect to the sequence and behaviour in the behaviour tree.

time. The interrupt child is executed each time the interrupt node itself is called, until the child make a callback telling it is done. When this callback has occurred the node resumes to check for the interrupt and call the non-interrupt child if there is no interrupt. When the selector is called it executes an algorithm to determine which child should be executed, when this choice has been made, the chosen child is stored, and is called each time the selector itself is called, until the child make a callback telling it is done. When the callback occurs the select algorithm is executed again to determine if a new child should run or the selector should make a callback to its parent.

5.2 BTTool

To ease in the creation and alteration of the behaviour trees a tool has been created, called BTTool. The tool was created in C# and has the following functionality:

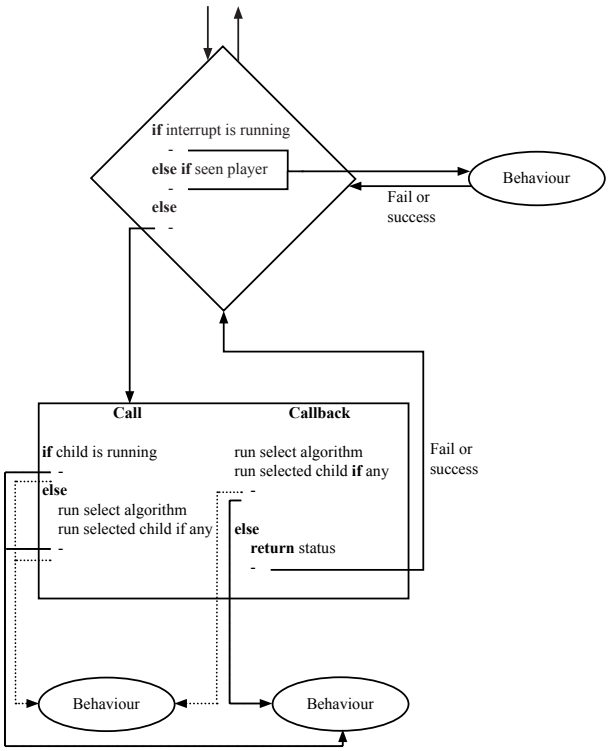

Figure 5.2: An example of how the constraints of the Source Engine could be overcome with respect to the interrupt and selector in the behaviour tree.

- Modelling GUI for creating a behaviour tree.
- Different types of components to be used in the behaviour tree: Selector, sequencer, behaviours, interrupts and edges.
- Property overview of each component, where functions, root and so on can be defined.
- Lastly a code generator, which traverses the behaviour tree and creates C++ code, which can be inserted directly in to the game.

BTTool incorporates roots and other behaviour trees by colour codes. The khaki colour is used for the root and the deep pink colour is used for other behaviour trees. This is done to give a separation from the nodes used in BTTool.

BTTool provides an overview of the behaviour tree, because its created visually as a model and not directly in code. Figure 5.3 on the next page illustrates the tool with a small behaviour tree, which makes the NPC patrol an area.

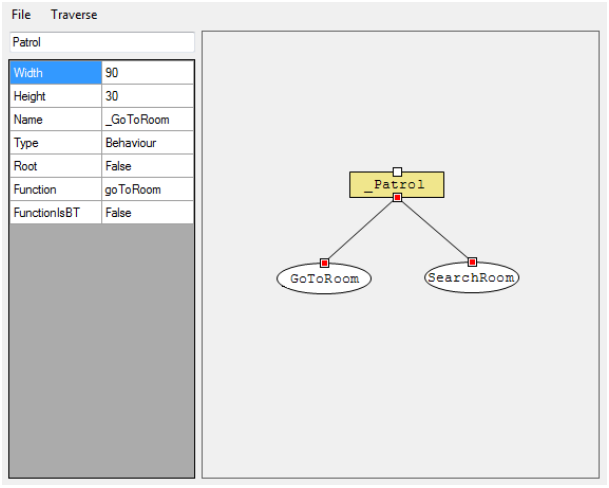

Figure 5.3: Illustrates BTTool with a small behaviour tree that makes the NPC go to a room and then searches the room.

5.3 AI Architecture

In this section the design of the AI architecture for Aslan is discussed. The first discussion concerns the transformation from the specification of the AI for Aslan into a usable behaviour tree. Afterwards the details of how the behaviour tree and squad behaviour is achieved are explained. Lastly it is explained how the communication between the Aslans making how each Aslan are taking decisions.

The design of the AI architecture is created from the specifications in Section 2.4 on page 12. In addition to this the design are created using the tool presented in Section 5.2 on page 34.

The first task is to determine an order in which the actions has to be performed, because the actions conflict in the sense they all specify a location Aslan should be in, and in NoEsc Aslan can be in only one location at the time. From the AI specification the following order is derived, where the lowest number has the highest priority:

1. If player is seen: Follow and attack.
2. Examine distraction sound.
3. Search for player.

To achieve the behaviour described, a behaviour tree has been created as illustrated in Figure 5.4 on the facing page. In the figure it can be seen that

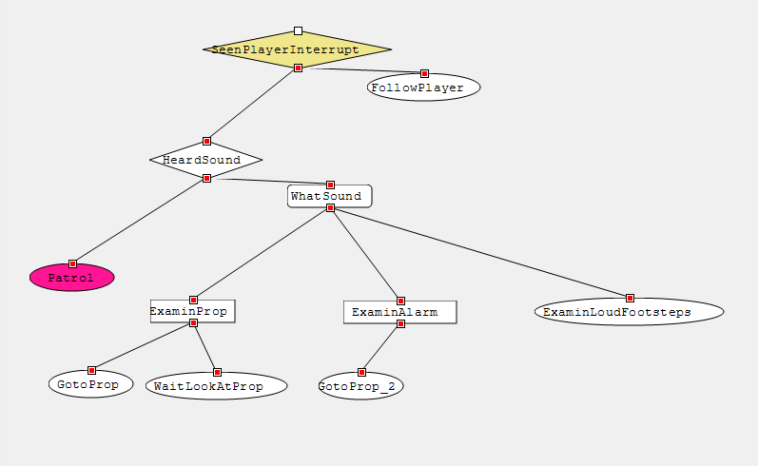

Figure 5.4: Behaviour tree with the base behaviour of Aslan in NoEsc.

the first node is an interrupt node, which enables Aslan to follow the player if the player is seen. The follow player behaviour is running to the location where player last was seen and does this as long as it has seen the player.

If the player is not seen it moves on to the next behaviour, which is to examine sounds if any. In the same manner as with the seen player, an interrupt node is testing if a sound is heard, if this is the case a selector node is executed. The selector determines which sound type should be investigated, this is explained further later in this section. If the selector selects a bottle or a prop in order to investigate a sequence is executed. The sequence first moves Aslan to the location of the sound, when this action is done the next part of the sequence is to wait a while to examine the sound. When the examination is done the sequence has no more children to run and will return. The running sound of footsteps and alarms are treated in a similar way, however without the waiting when the position is reached. There are shown two different ways of designing the same action, one where a sequence with only one child is added, this is done so waiting easily can be added as a second node if desired in a later development iteration.

If there was no sound heard, or no action taken on the sound, Aslan instead patrols for the player. The search for the player are done in squads and shown in Figure 5.5 on the next page.

5.3.1 Squad

Dividing the squads are done when the level loads, and the squads are created by a maximum amount of members. This is because there can not exist a

one man squad, so if there had to spawn 16 squad members with a maximum of five members per squad, the system must create four squads: two with five members, and two with three members. When this is organised and the system has chosen a squad leader for each squad it must divide the squads evenly among the different spawn points.

When the formation of squads is done and Aslans does not hear or see anything out of the ordinary, they will activate the search behaviour. On Figure 5.5 the search behaviour tree is illustrated.

The root node in the behaviour tree is a sequence, which handles the actions Aslan does. Aslan cannot take any new actions, after performed its task, before the rest of the squad is done as well. The action Aslan can take is based on whether or not it is the leader. If Aslan is not the leader it will wait to receive further orders, and when orders are received it will execute them and return to a defined meeting location. However if Aslan is the leader it will execute a selector, which decides where the squad should go, and if the room should be searched. This selector will be discussed in detail in the last part of this section.

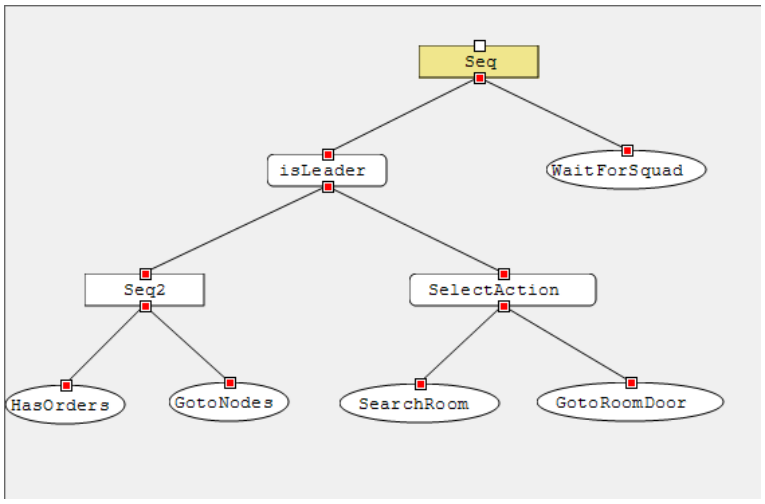

Figure 5.5: Behaviour tree with the squad patrol behaviour of Aslan in NoEsc.

Aslans searches an area by looking at the *ai_nodes* in the area. The nodes are found by looking at the door nodes and removing the *ai_nodes* around the doors. This separates the nodes going to and from the area. Then the remaining *ai_nodes* are limited to a handful as depicted in Figure 5.6 on the next page.

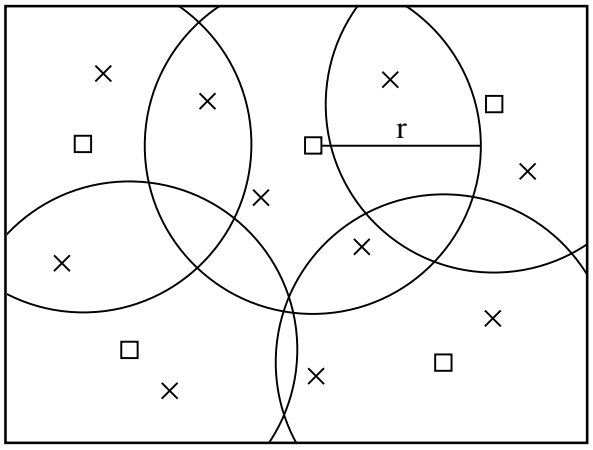

Figure 5.6: Illustration of the removal of *ai_nodes*, where crosses are removed nodes and square are nodes left behind.

To limit the amount of *ai_nodes* within the room, a node close to the center of the room is selected. All nodes within a radius r of the selected node are ignored, then a node outside the radius r is chosen. Then nodes within the radius r of the new selected node are ignored. This process is repeated until all nodes is either ignored or chosen. The selected nodes are visited in a greedy manner, by selecting the nearest unvisited node. This is done until all nodes are visited. When all squad members that are not standing guard outside the room have done this, the squad moves to the next area.

5.3.2 Communication

All communication is under the restriction that it only works within a given range. This range simulates the range Aslans can talk. The communication between Aslans is furthermore divided up into two levels, explicit and implicit. The explicit communication is when a decision is made and this is communicated to other Aslans. This kind of communication is initiated from the actions in the behaviour trees. The implicit communication is the sharing of information when Aslans meet, this information is where the other Aslans has been and when they have been there.

5.3.3 Decision

This section covers the decision making for moving around in an office building and for when a Aslan hears a sound. The first covered is the custom

nodes, how they work and how they are designed in contrast with Hammer Editor and Source SDK. Next is the decision making when hearing a sound, how the custom nodes weights are updated and how Aslans makes a decision given a variety of sounds.

Custom Node

In the game custom node has been created to identify rooms, doors and hallways. This has been done to help Aslans know which areas to search and what not to. The custom node consists of the following properties:

Type: Identifies the custom node as being a room or a hallway node. If it is a room node, then the area is searchable by Aslans, but if it is a hallway node, then Aslans only walk through the area without searching.

Weight: A weight that defines the need for Aslan to search the given area. The weight value can be updated by means of time and sound.

First door node: A pointer to the first door node in the area.

Every area has a custom node, but also door nodes. The door nodes consists of four properties:

Room node 1: Points to the custom node in area 1.

Room node 2: Points to the custom node in area 2.

Next door 1: Points to the next door node in area 1, if there are more doors to point at.

Next door 2: Points to the next door node in area 2, if there are more doors to point at.

The door nodes' next door pointers are set up as a chain of doors, as illustrated in Figure 5.7 on the facing page, where *R4* points to the first door node in the area, which is *D8*, so the area of *R4* consists of the following chain of doors: $D8 \rightarrow D7 \rightarrow D3 \rightarrow D6$. This is done because of the limitation of the Source Engine and the Hammer Editor, where the Hammer Editor can not use lists. Therefore this was a limitation that prevented the use of lists of door nodes for the custom node, but was solved by creating a chained list of doors.

Aslans use the custom nodes to search an area. This is done by using the weights of the custom nodes. Since a custom node can get all its door nodes and thereby get all neighbouring custom nodes, then it can find the custom node with the highest weight, which Aslan can go to.

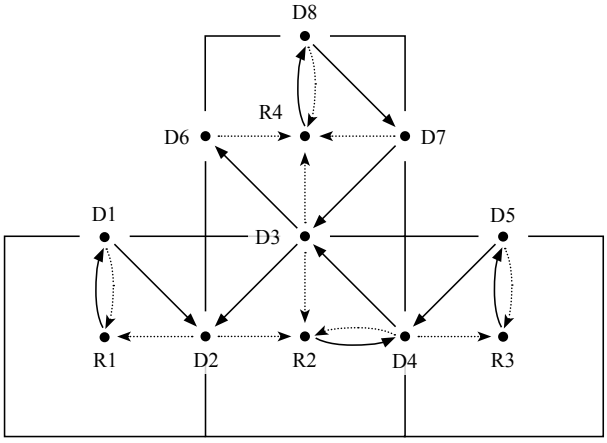

Figure 5.7: Illustration of four custom nodes: *R1 - R4*, and six door nodes *D1 - D8*.

Sound and Custom Nodes

The sound sense is used by Aslans to hear noises like bottles, footsteps and alarms. Sounds has an affect on the custom nodes weights, which are increased given distance between Aslan and the sound source, but also the distance from the sound source to the custom node location. This is illustrated in Figure 5.8, where d is the distance between Aslan and the sound source and s is the distance from the sound source to a custom node.

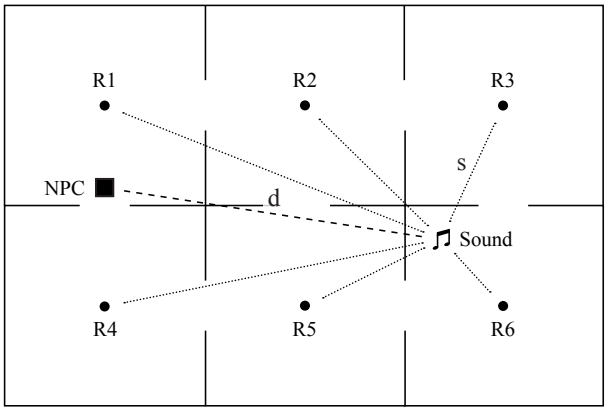

Figure 5.8: Illustration of Aslan hearing a sound at a distance of d.

The weight that needs to be added to the current weight of a custom node uses the following formula:

$$w = w + m,$$

where w is the weight of a custom node and m is:

$$m = \frac{s_v}{c} \cdot \frac{d - s}{d},$$

where s_v is the volume of the sound source, c is a constant and $m > 0$. The higher the value of c the smaller the value of m will be, which yields a smaller affect of the sound on the weights. The range of Aslans hearing and thereby the range of this method is controlled by Aslans hearing sensitivity.

Sound and Decision Making

Aslans has to make a decision when hearing a sound, whether to investigate the sound or ignore it. This decision making process can be done in a variety of ways. The first method is by prioritising the different sound types, which are: Alarm, bottle and footsteps. The prioritisation could be as the latter by making the alarms more important, than bottles and lastly footsteps. Thereby creating the following decision making process:

1. If Aslan hears an alarm sound then investigate the alarm with the highest volume.
2. If Aslan hears a bottle sound then investigate the bottle with the highest volume.
3. If Aslan hears a footstep sound then investigate the footstep sound that has the highest volume.
4. If neither of the three previous steps were chosen, then proceed with the search task.

The second method could be to calculate the probability of investigating a sound given a bayesian network as illustrated in Figure 5.9 on the facing page.

The four nodes are given probability tables, where alarm is $P(A)$, bottle is $P(B)$, footstep is $P(F)$ and lastly search is $P(S|A, B, F)$. By using this network the following states in the three sound nodes: Alarm, bottle and footsteps are:

- High
- Low
- Nothing

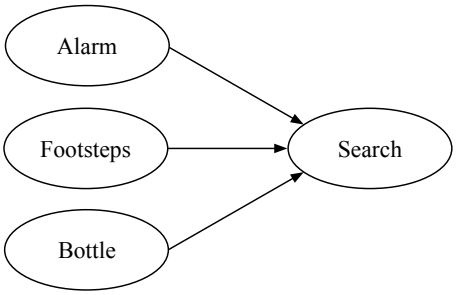

Figure 5.9: Bayesian Network illustrating the need to investigate a sound given alarm, footsteps and bottle sounds.

Where each indicates the sound intensity in three states. The search node has the following states:

- Normal
- Alarm
- Bottle
- Footstep

So instead of using the prioritised list from the former method, it is now possible to use the volume and the maximum volume to calculate a percentage of the three different sounds, which can be used to determine the highest probability of a sound that is interesting for Aslan. So if an Aslan hears an alarm with a value of 0.2 (low sound), bottle with a value of 0.7 (loud sound) and footsteps with a value of 0.0 (no sound). Then the values for high and low sound are calculated, but still need to be summed to 1, if there is no sound then the value of high and low sound sums to 0 and no sound will have a value of 1. The following probabilities could be calculated for the three sound types and for continue as normal:

Normal: 10.4%

Alarm: 63.6%

Bottle: 26.0%

Footsteps: 0.0%

Which were calculated by using a probability table for the search node. Choosing the task with the highest value assigns the NPC to investigate the alarm.

5.4 Level

As described in Section 2.1 on page 8 the player is located inside an office building, and Aslans are storming the building from three fronts. The level design is therefore constructed as seen in Figure 5.10.

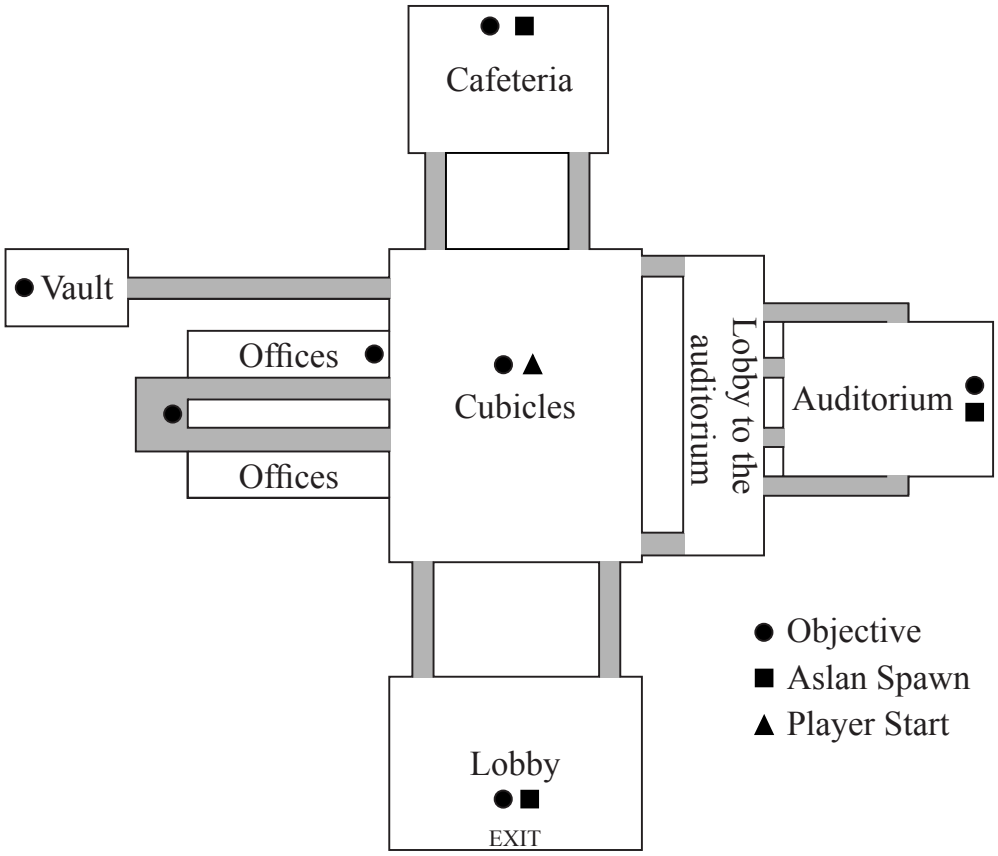

Figure 5.10: The level design for NoEsc.

The player starts in the big room in the center. This room contains multiple cubicles, which will give the player a feeling of being inside a maze. The cubicles should help the player hide from the Aslans, but also try to confuse the player. The room is fairly dark where only the light from the monitors are present. In one of the rooms an objective can be found.

The room with cubicles contains entrances to each of the rooms in the level. The first entrance as seen in Figure 5.10 leads to the auditorium, which is a big room where the player has a minimal chance of hiding while taking the objective. The auditorium is also one of the rooms in which the agents starts. All of the different rooms in the level has two or more exits, except the

vault. This is because the player should always have minimum two directions to take, if agents are approaching from one of them. The vault has only one exit, because it should be a hard objective to take.

The second room is the lobby, where an objective is placed. The room is bright and contains limited amount of hiding spots. The agents also spawns in the lobby, and the lobby contains the exit when all objectives have been taken.

The third room contains small offices, where one of them contains an objective, and a second objective at the end of the hallway. The rooms in the hallway are dark and small, but are also good hiding spots for the player.

The fourth room is the cafeteria, which is similar to the lobby, with a limited amount of hidings spots. The room is also a starting point for Aslans and an objective is present.

The last room is the vault. The entrance to the vault is a long dark hallway in which a trap is placed. The trap triggers if a player or Aslan walks into it. When the trap triggers an alarm sound occurs and fences entraps the player or Aslan.

As described earlier each room contains an objective and the difficulty in taking them varies from room to room. The different rooms are connected with small hallways, where the player has limited options to escape.

5.4.1 Nodes

In the level two types of nodes are placed. The first node type is a standard entity in the Source engine, which is called *ai_node*. This type of node is a representation of walkable areas in the level. These nodes a placed manually in the level to let the Aslan know where to walk.

The other type of nodes, is the custom node entity used for decision making as described in Section 5.3.3 on page 40. As described there exist three types of these nodes, room, hallway and door nodes. The nodes are placed on a print of the level as seen on Figure 5.11 on the next page, and later typed into the Hammer Editor, which has been a design aid.

The map is split up into two types of spaces, Hallways and Rooms, and every one of these spaces will contain a node with the information of what type of space it is. This way the AI will know how to behave, e.g. when dealing with a room the squad will leave one man at the door to keep guard and set the rest to search the room. In a hallway there is no need to have a guard so they will just run through.

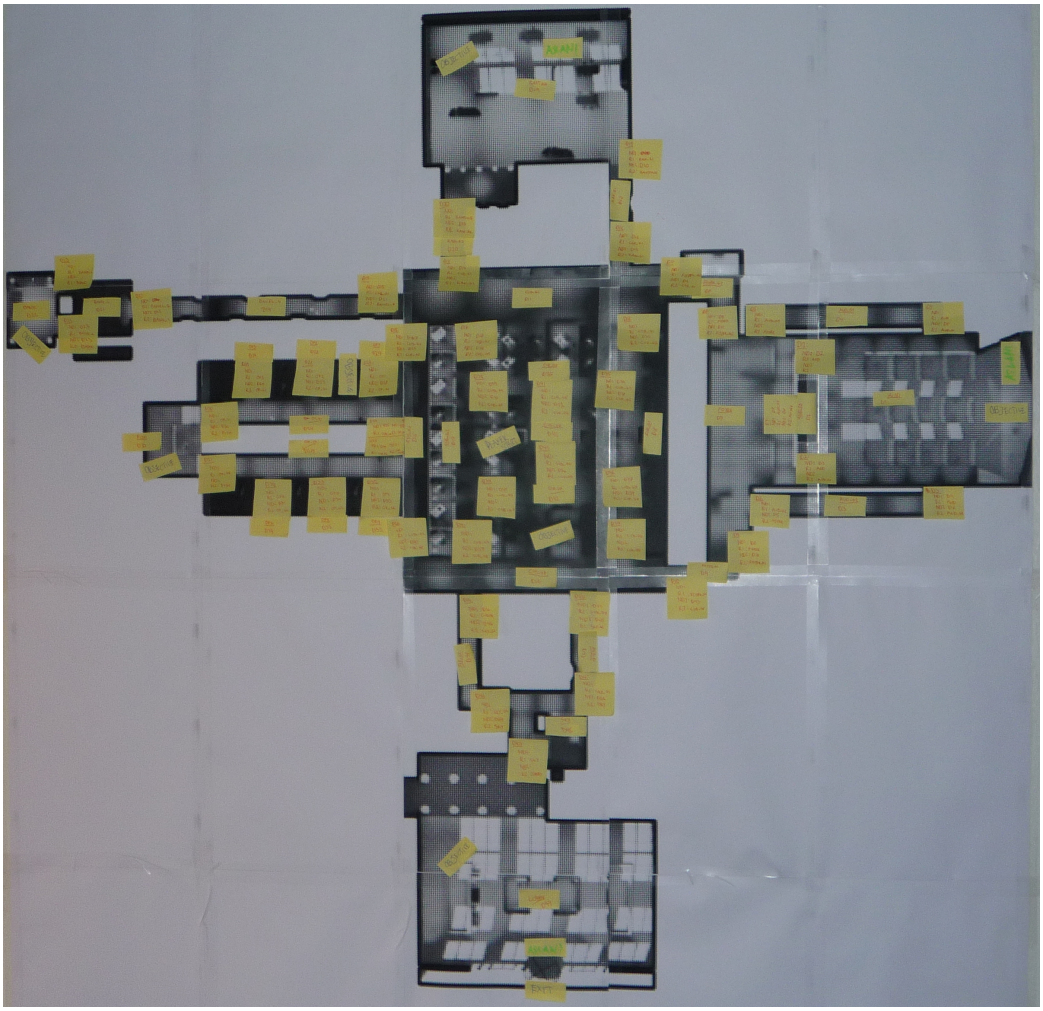

Figure 5.11: The custom nodes placed on a print of the level.

5.5 Summary

The AI technique used by the NPC is behaviour trees, but a modified version that does not consist of decorators, but interrupts instead. Decorators were removed, because each node now could have custom code, which meant the properties of decorators now could be coded instead. The constraints given by the Source Engine, meant that the AI needed to store the behaviour it is currently doing, since the AI in Source Engine only is allowed to work in given time intervals. The behaviour trees for the game was created in BTTool, which was a custom made piece of software, that can design the behaviour tree visually and generate code for use in the game.

Next the AI architecture was designed, which covered how the squad, communication and decision making should work. The squad creation is done when the level has loaded, where a squad leader is assigned to each squad. The communication is done explicitly and implicitly, where the first is orders given to Aslans squad members by the leader, and the latter is when Aslans meet and exchanges their searched areas, so they do not search the same areas.

Then the decision making by Aslans was discussed, that is how they search a building by the use of custom nodes. The custom nodes could either be rooms or hallways and was chained with door nodes, so all doors in an area could be accessed by the custom room node. The custom nodes also had a weight, which was used by Aslans to know which direction to take, that is the higher the weight the more likely Aslan will search that area. If a Aslan hears a sound the weights of the custom nodes are affected by this, so Aslans will search the area the sound is in.

Next was the decision making when hearing sounds. Two methods were described. The first method was a prioritised list, where alarm was prioritised higher than bottle and footsteps. The last method was the use of bayesian networks to make the decision. Both of which will be evaluated to see if there is any difference in Aslans actions.

Lastly the level was described and which types of nodes are present in the level. The nodes present in the level were *ai_nodes* and custom nodes.

Chapter 6

Implementation

This chapter presents the implementation of NoEsc. The primary focus of this chapter is how Aslan is created codewise using the the Source Engine. In addition to this the level and game logic for NoEsc is discussed.

The implementation of the AI design discussed in Chapter 5 on page 32 are not discussed in this chapter because the implementation of these parts follow the design, and therefore are no relevant topics in the implementation of these.

6.1 Programming in Source

When a Mod is created it must be decided what functionality to add in code and what to make in Hammer Editor. An example of such functionality is Aslan where the functionality is written in the source code, but it is inserted through the Hammer Editor where desired.

Therefore an API is available to access the functionality in the Source Engine. The API exposes features in the Source Engineand makes them available to the creator of a Mod. This section will explain the primary functionalities that was used while constructing NoEsc and also how they were used.

6.1.1 Creation of Aslan

Aslan is created as a regular NPC as described in Section 3.3.4 on page 17. Before a NPC in Source can move, shoot and so on, it has to have the proper capabilities. These capabilities can be run, jump, crawl and so on. So to make it possible for the NPC to e.g. run, then this capability has to be applied to the NPC. However the 3D model chosen for the NPC should

also support these capabilities in its animations. An example is the Barney model from Half-Life[2] that does not support the using of the stun-stick. The capabilities are added to a NPC as shown in Listing 6.1. This method has to be called when the NPC is spawned to ensure the capabilities are set properly.

```
1 CapabilitiesAdd( bits_CAP_MOVE_GROUND | bits_CAP_DUCK |
      bits_CAP_MOVE_CRAWL | bits_CAP_MOVE_CLIMB ... )
```

Listing 6.1: Shows how to add capabilities to Aslan.

Sensing

A NPC is able to receive inputs from the game world. This is done by emulating the idea of seeing and hearing. This emulation is done in the Source Engine and have API calls accessible to find which entities there have been seen and heard.

The API call for seeing only needs to be provided with view angle and a view-distance in order to return the list of entities which the NPC can see. Listing 6.2 shows how this is implemented in Aslan.

On line 1 an action calls `Look` to refresh the list of what the NPC can see. On line 7 a loop goes through all the entities seen, and on line 9 to 13 the found entities are checked if they are a player and if they are stored in the `_sightings` list. A wrapper class `NEEntity` is used to create easy accessors and attach additional data onto the entity.

```
 1 GetSenses()->Look( noesc_aslan_viewdistance.GetInt() );
 2
 3 AISightIter_t iter;
 4 CBaseEntity* foundEntity;
 5
 6 foundEntity = GetSenses()->GetFirstSeenEntity( &iter,
       SEEN_ALL );
 7 while( foundEntity )
 8 {
 9   if( foundEntity->Classify() == CLASS_PLAYER && !
         playerIsInvisible )
10   {
11     NEEntity entity(foundEntity, Player);
12     _sightings.push_front(entity);
13   }
14   foundEntity = GetSenses()->GetNextSeenEntity( &iter );
15 }
```

Listing 6.2: Code sample that shows how Aslan is able to see.

An interesting observation in the code is how to differentiate between the different types of seen entities in Source Engine. Each entity in Source Engine has a `Classify` method which holds an type identifier, that can be compared with the needed entity type identifier to get the entities needed. The only sightings that Aslan is interested in is when it sees the player, this way it may go into pursuit and try to capture the player.

To enable a NPC to hear, it must be specified what it is interested in hearing. The list of interesting sound types is created by using the set of standard sounds such as combat sounds, player made sounds, bullets and so on. Listing 6.3 shows how this is implemented in Aslan. The list of interesting sound types are defined in `GetSoundInterests()` which is used on line 1. If there are any sounds, these are all examined as shown on line 3 to 11 and 20. On line 14 and 15 the sounds are stored like the sightings, and on line 16 the nearby nodes, as explained in Section 5.3.3 on page 40, are updated to accommodate the sound. This updating will make the area where the sounds are heard more interesting for Aslan.

```
1  if( GetSoundInterests() )
2  {
3    int iSound = CSoundEnt::ActiveList();
4
5    while( iSound != SOUNDLIST_EMPTY )
6    {
7      CSound *pCurrentSound = CSoundEnt::SoundPointerForIndex(
           iSound );
8      Assert( pCurrentSound );
9
10     if ( (pCurrentSound->SoundType()) &&
11       GetSenses()->CanHearSound( pCurrentSound ))
12     {
13
14       NEEntity ent(pCurrentSound);
15       _sounds.push_front(ent);
16
17       data.roomNodes.soundExplosion(this->GetAbsOrigin(), ent
             .getSoundVolume(this), pCurrentSound->GetSoundOrigin
             ());
18     }
19
20     iSound = pCurrentSound->NextSound();
21   }
22 }
```

Listing 6.3: Code sample there shows how Aslan is able to hear.

Attack

Aslan also needs to be able to attack the player, which it does with a stun-stick. However Aslans goal is to get into a certain distance of the player to attack and damage the player. To determine if the player is within the attack vicinity, the player should be within a certain range of Aslan, without any obstacles between Aslan and the player. To check for obstacles a ray is created between Aslan and the player to check that it does not intersect with any blocking objects. If there is possibility for Aslan to attack it will attack.

6.1.2 Difficulty Levels

In NoEsc there are different difficulties as described in Section 2.5 on page 13. These difficulty levels are implemented by using the difficulty system already implemented within Source Engine. The difficulty is implemented via a value called skill, this value is an integer where 1 represent easy, 2 medium, and 3 hard difficulty. This is used throughout the code to alter the different difficulty parameters, e.g. in Listing 6.4 the view distance, damage and field of view of Aslan have, are defined for the easy skill level. This definition is done when Aslan is spawned.

```
1 ConVar const *skill = cvar->FindVar( "skill" );
2
3 switch( skill->GetInt() )
4 {
5 case 1:
6   {
7     noesc_aslan_viewdistance.SetValue(600);
8     noesc_aslan_damage.SetValue(40);
9     noesc_aslan_fov.SetValue(0.5f);
10  }
11  break;
12 ...
```

Listing 6.4: Code sample that shows how the difficulty level is set on Aslan.

Another example is when the Aslans are spawned and set into squads. This is done in the Concommand `noesc_spawn`, which is called every time the level loads. In this method the difficulty level decides the amount of Aslans, and how many that there is per squad. In 6.5 the squad size and total number of Aslans are shown for the easy skill level.

```
1 switch( skill->GetInt() )
2 {
3 case 1:
```

```
4   {
5     nAslans = 10;
6     nMaxSquad = 2;
7 ...
```

Listing 6.5: Code sample that shows how the difficulty level affects the amount of Aslans spawned and also how many there will be per squad.

6.1.3 Custom Nodes

As explained in Section 5.3.1 on page 37 each squad will when encountering a new room have a guard standing at the door and then let the rest of the squad search the room. To make this possible there needs to be a way to differentiate between rooms and hallways, and also a way to find the door where the guard will be positioned. Therefore custom nodes needed to be created as explained in Section 5.3.3 on page 40. In addition to the custom nodes, there also needs nodes to represent where Aslan can spawn. The implementation of these two kinds of nodes are similar so only one will be explained here, namely the spawn point node.

First the class `NESpawn` is created. It inherits from a standard node from the Source Engine. It is extended to contain an integer called location, that represents where it is located. Since this information has to be specified in the Hammer Editor it has to be added to the `DATADESC` as seen in Listing 6.6. The `DATADESC` is a table of data that defines the values which the Hammer Editor can edit. Line 1 tells the Source Engine which name the entity has at runtime, in this case `ne_spawn`. Then the `DATADESC` is defined and in it a field called location is defined, which is an integer.

```
1 LINK_ENTITY_TO_CLASS( ne_spawn, NESpawn );
2
3 BEGIN_DATADESC( NESpawn )
4   DEFINE_KEYFIELD(location, FIELD_INTEGER, "location"),
5 END_DATADESC()
```

Listing 6.6: Code sample that shows the field location being added to the `DATADESC`.

To enable the Hammer Editor to use this entity it has to be added to the FGD file. The addition of the `ne_spawn` can be seen in Listing 6.7 on the facing page. It shows that a new PointClass called `ne_spawn` is created and it has a the field location which has three choices in the Hammer Editor: Lobby, Cantine, and Aud. When an entity of type `ne_spawn` is created in the Hammer Editor it will show one field called Spawn Location with the three possible choices.

```
@PointClass base(Parentname) = ne_spawn :
  "An entity used to spawn Aslans."
[
  location(choices) : "Spawn Location" : 0 : "The location" =
  [
    0 : "Lobby"
    1 : "Cantine"
    2 : "Aud"
  ]
]
```

Listing 6.7: A sample from the FGD file that shows the definition of `NESpawn`.

6.2 Level

The level for NoEsc was developed using the Hammer Editor, and following the design criteria. All architecture was developed using the basic building blocks of Hammer Editor like blocks, wedges and cylinders. Afterwards materials were assigned to the architecture and props were placed in the levels to provide the right atmosphere.

Before the map could be compiled and run, different entities had to be placed in the level. The first entity was `info_player_start` which handles the spawning of the player. The Aslan has a custom created entity called `noesc_spawn` which has three different types depending on which room the Aslan spawns.

A picture of the level created with the Hammer Editor can be seen in Figure 6.1 on the next page.

6.2.1 Intro sequence

The intro screen is implemented using a `point_camera` entity and several `path_corner` entities. The camera is placed at a starting point and facing the player in the other room. Several `path_corner` entities are placed as a track for the camera to follow, where the Hammer Editor handles setting the next and previous track for each of the entities. On the camera entity the speed and first path target are specified. the Hammer Editor handles activation of the camera at the beginning of the level, and disabling the camera is achieved using a output function on the last `path_corner` entity telling the camera to disable.

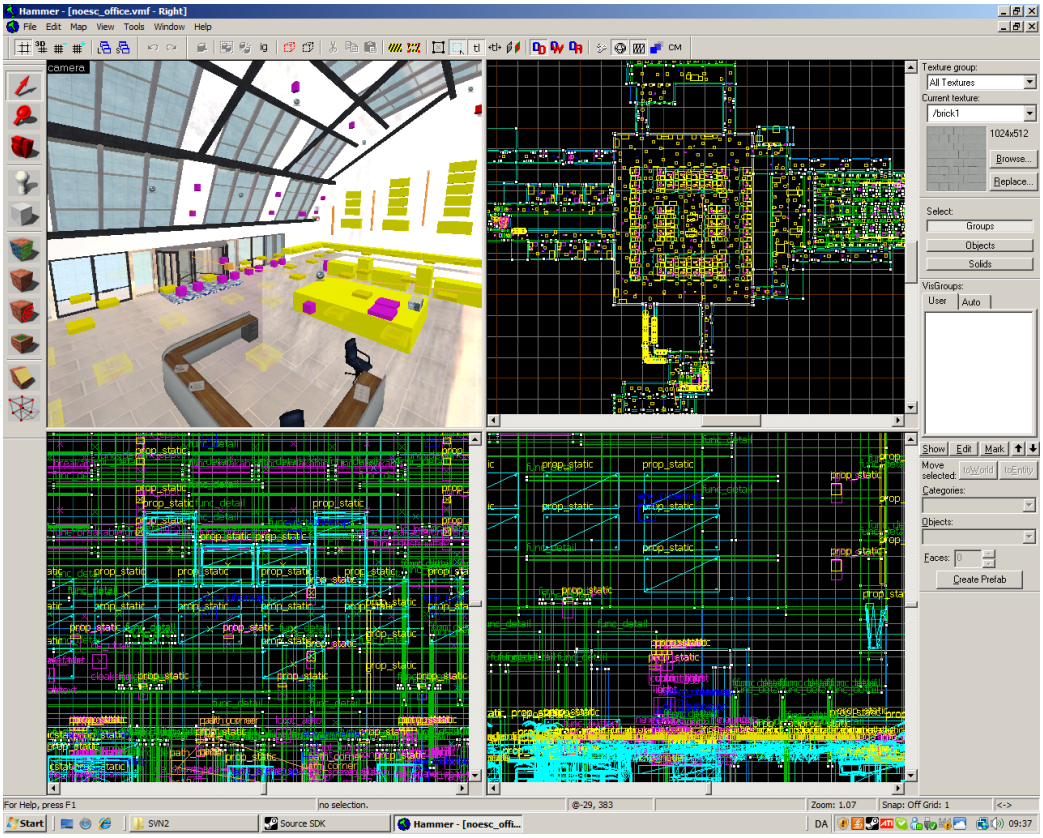

Figure 6.1: The final level for NoEsc created with Valve Hammer Editor.

6.2.2 Objectives

The objectives and win condition are created using custom and standard entities in the Hammer Editor. The objectives in NoEsc is a custom entity called `noesc_objective`, which is a button that has to be pressed a specific amount of time before it is fully pressed. If the player releases the button before it is fully pressed, the player has to start over. On each of the `noesc_objective` entities placed in the map there is defined an output function that handles the counting of objectives.

All output functions are occurring when the button is fully pressed. The first output function handles the removal of the objective, so the player can not see it on the overview map or in the level. This is achieved using the `Kill` method, which removes the entity from the world. The next output function calls another entity called `math_counter`, which is a basic counter. The output function adds one to the counter entity.

The entity `logic_case` handles when to display specific text on the screen,

telling the player how many objectives are left and when to run for the exit. The `math_counter` entity described above tells the `logic_case` entity when a new number is added and sends the sum of numbers to the entity. The `logic_case` entity works in the same way as a switch-case statement. If the number it gets as input is for instance two, it shows the entity `game_text` with the message: "Two objectives left". When the case is zero there are no more objectives left, and it fires an output function that displays the text on the screen, telling the player to run for the exit, and lastly it enables the entity at the exit. The entity at the exit is a `trigger_once` entity and when the player touches it, several events occurs. First the player is teleported to another location so there is no interference when showing the credits screen. Then all sounds are disabled and credit sound are played. Lastly the screen fades and credits are enabled.

6.3 Summary

In this chapter it was shown how the implementation of Aslan is done in the Source Engine. It is also shown how Aslan uses its senses to aid it in finding the player. Furthermore it was described how to cope with the difficulty in the Source Engine. In addition to this the implementation of game logic and intro sequence in the Hammer Editor were also explained. In addition with the parts not explicit explained in this chapter an working implementation of NoEsc is achieved.

Chapter 7

Testing

For a game to be a success it has to be playable. Therefore it is important to have the game tested to see if it has a balanced difficulty level and the gameplay is fun and challenging. It was therefore chosen to have NoEsc tested by people external to the developers of NoEsc, except when testing the sound selector, which is done by the developers. The testers should provide an idea of how far NoEsc is to be a playable game and thereby also provide a list of items that needs to be fixed or enhanced.

7.1 Sound Selector Test

The sound selector in NoEsc was experimented upon by creating two different types of selectors as mentioned in Section 5.3.3 on page 42. The first method was by prioritising the types of sounds, the second was by using bayesian networks. These two methods were tested by the developers to see if bayesian networks provided an enhanced game experience. The test consists of a test where a developer first plays where the prioritisation method is used and next where the bayesian network method is used. The following comments were given by the developers when using the bayesian network method with Aslan:

- Reacted more on the sound of footsteps.
- Better at finding and trapping the player.
- Stuck less times.
- The Aslans were better at spreading out.

The reason for Aslan being more reactive to the sound of footsteps is that the probability calculation given a footstep sound an no other sound types gives a higher probability of following the source of the sound, than doing as normal. This is also the reason of the AI being better to find and trap the player, since the player makes a lot of noise when trying to escape. This does however not explain why the testers thought that the NPCs got stuck less and were better at spreading out. Overall to get the NPCs to be less reactive to the sound of footsteps is to edit the values in the probability table, such that footstep has a lower initial value and normal has a higher value.

7.2 Play-testing

It was chosen to use play-testing as the method to test NoEsc. The test is based upon the Pluralistic method described in the article "The Pluralistic Usability Walk-Through Method"[12]. A major difference is that the testers tested a working version of the game and not a series of mock-ups, and that it was done late in the project period. Some natural limitations arose of the fact that only two test managers administrated the tests. Two groups were chosen to test the game: Group SW802B and SW803A, which were 8. semester software developers at Aalborg University, that already had experience playing computer games. None of the groups had seen or played the game before. The test was conducted on a full testing setup consisting of a laptop running Windows XP and the latest version of Valve Steam and NoEsc, and another laptop for notes. The testing was run in such a way that one tester at a time would be seated in front of the computer, and was then asked to play NoEsc.

One test manager sat together with the current tester and observed and asked some questions while the tester played. At first the tester should start the game, choose easy as the difficulty level, and then try to play the game. This should all be performed without any help from the test manager in order to see if the goal of the game was obvious enough. After some tries the test manager would answer the testers questions and try to help the tester in the right direction. When the tester had figured out how to play the game, he should try to come up with possible enhancements for the game. All of this while the second developer sat and took notes. After a while the tester would be sent over to the second test manager who would ask some pre-made questions, such as if the AI seemed intelligent enough or if the difficulty level was to hard or to easy. This was done with both groups.

7.2.1 The Test

The test was performed with groups SW802B and SW803A as explained in section 7.2 on the previous page. The questions and answers can be found in Appendix A.1 on page 66 and Appendix B.1 on page 69.

This section will evaluate the feedback from the playtests.

Game understanding

The members of the test groups agreed that the game was to hard to figure out. It was proposed to make a more fluent transition into the current level, such as tutorial levels where the player could learn that the objectives should be found and hacked. More so also to introduce the concepts of distraction, stealth movement, and the cloak.

Difficulty

The game was to hard even at the easy setting, and completely impossible for the testers at higher settings. This is a problem as the players eventually could give up on the game. The testers suggested health regeneration and an ability to run for a short while. Regeneration would, as one tester noticed, also fit better with the HUD in NoEsc as it requires no health indicator.

The AI

The testers agreed that the AI searched intelligently through the building, even though they at times seemed to get stuck at locations and moved along a couple of seconds later. Though it was noticed that the NPCs at times would focus to much on areas such as the central cubicle room or the vault hallway. The players also noticed that running away from the AI headlessly often would get them caught as the AI would capture through other ways. The testers also liked the way they could run into a room and hide behind the door, and then hear the NPCs run past. Also the distract feature was applaused, even though it was noticed by the developers that they seldom used it.

7.3 Summary

The test went well and it provided the group with a list of possible enhancements for the game. The fact that the testers thought the game was to hard

is understandable, and some extra tools provided to the player could enhance both playtime, but also the difficulty level. This could be tools such as a mirror to see around corners or the ability to run for a short period of time.

Generally the testers liked the AI, and found that it provided a high amount of resistance. They also liked the way it searched through the building.

Overall the testers thought the game was playable, and fun. Also several mentioned that the game introduced a new aspect into stealth game with a high amount of paranoia while playing the game.

The sound selector test confirmed that the bayesian network method enhanced the game experience by creating more intelligent behaviour in Aslan. However the bayesian network method made Aslan more aware of the present of the player, since the footstep sound was triggered more often, than the prioritisation method, which gave a more difficult Aslan to escape from.

Chapter 8

Epilogue

The epilogue is a collection of the reflection, conclusion, and future development. The reflection will reflect on the Source SDK, BTTool, and NoEsc. This is then followed by a conclusion of the project. The last section will present some suggestions for further development that if applied to the game could enhance gameplay or fix issues.

8.1 Reflection

The reflection is split into three main topics of discussion: Source SDK, BTTool, and NoEsc. The first section discusses the work experience got by working with the Source SDK. The next section discusses the development of the BTTool , and if it was worth developing. The last section will provide some thoughts about the game design and how to apply it to the game.

8.1.1 Source SDK

The group started the development by trying to implement everything in the code. This slowed down the development process, but it was however quickly discovered that the Hammer Editor was able to provide a lot of the needed functionality through its scripting interface. After this a developer was dedicated to try to make game functionality in the Hammer Editor, instead of implementing it in code. This decreased the time it took to implement features.

8.1.2 BTTool

At the beginning of the project it was decided to make a tool that could generate the code for the behaviour tree, based upon a modelling tool. A

major concern was if the time spent on the development of the tool was worth the time. Through the project the behaviour tree was changed several times to fit with new requirements or general changes to the tree. Without the tool this could have been very time consuming, but the tool made it visually easier to modify the existing trees. The tool was also able to provide an overview of the behaviour tree, which helped to explain and understand the behaviour of the AI.

8.1.3 NoEsc

After the game had been developed it had to be adjusted to be playable. While constructing the game all values relating to the game play, such as running speed, had been made into convars such that they could be changed during runtime. So the values had to be adjusted at the end, based upon gameplay requirements. It was found to be challenging to find the appropriate values, which later was supported by the testers that found the game to hard.

The AI turned out to be a challenge to develop as the behaviour was quite different from reguar AIs seen in e.g. Half-Life[2]. The greatest problem was that the AI was unable to interchange information and that a synchronisation scheme had to be applied to the Aslans. Besides this a lot of basic features of an AI had to be implemented in order to work. Some final work on the game included experiments with a bayesian rule based sound selector, which showed to improve the behaviour of the AI.

8.2 Conclusion

The goal of the project was to design and implement a single player game with an AI by using the Source Engine. The game idea developed was a stealth game called NoEsc, where the player was an agent trying to hack a number of objects, while avoiding the Aslans. Therefore some game rules was defined, both for the game content, but also for the AI of the NPCs.

Different AI techniques were researched and compared to find one that would fit the needs of NoEsc. The techniques chosen to be implemented were behaviour trees and Bayesian Networks. The latter was experimented upon in the sound selector of the behaviour tree structure against a prioritised list of sound types, which proved to enhance the behaviour of the AI. This made the game a bit harder, than the original prioritised list of sounds, since the footstep sound was triggered more often, than the original method. This made the Aslans more aware of the player position.

The behaviour tree technique was redesigned to remove decorators and replacing them with interrupts, since the Source Engine calls the AI in defined time intervals and since every node has custom code.

Different difficulty levels were created to fit the player skill level. This was done by editing the number of Aslans, and their ability to hear and see.

Custom nodes were designed for the level and were used by the AI for decision making and path finding. Two different kinds of nodes were created, one for doors and one for rooms, which then were linked together by a linked list, because of limitations in the Hammer Editor. The nodes contain different kinds of information such as the weight, a list of door nodes, and so on.

A play-testing test was performed with two test groups, which had not played the game before. The tests showed that even though there were different difficulty levels, the game was still to hard to complete. Furthermore it was hard for the testers to figure out what they had to do at the start, because there was no introduction to the game mechanics. Besides that, the test groups thought that the game AI seemed intelligent when searching through the level, and the sound of the Aslans running past the room they were hiding in, supported the game idea.

8.3 Further Development

In this section possible additions to NoEsc are discussed. First additional content to the game is described and lastly the additional behaviour.

8.3.1 Content

Only one level was developed for the game, but the testers wished for more levels. Where some should introduce the game elements.

Multiple power-ups could be added to the game such as a temporary sprint boost that could be used to escape from the NPCs more easily.

Several testers also pointed out that the minimap could turn with the player, or else contain an arrow showing in which direction the player was looking in order to improve navigation.

As it is the player has no way to defend herself against the Aslans, so a tool such as a speaker that could be dropped and then remotely activated could be an alternative way to escape.

A major problem is that the player mostly is running blindly around corners which could be fixed by a mirror that the player could use to see around obstacles. This could both be used to see around corners, but also to take a glimpse above a cubicle.

8.3.2 Additional Behaviour

The testers suggested the ability to open and close doors, which could create a more interesting AI and game play as the AI could react on doors that are open that should not be open and reversed.

Additionally it could be harder to see the player in dark spots, so the player could hide in the shadows and see the Aslans run past him.

At this point the squads are fixed and it is not possible to merge or split the existing squads. Additional behaviour could be that the squads would be able to perform the merge and split tasks, and depending on the choices for searching rooms, hallways and so on. This way a squad could be able to split into two to cover two different hallways, and merge together when they meet again.

Bibliography

[1] Wikipedia, Tom Clancy's Splinter Cell — Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/w/index.php?title=Tom_Clancy%27s_Splinter_Cell&oldid=282158843, 2009, [Online; accessed 19-April-2009].

[2] Wikipedia, Thief: The Dark Project — Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/w/index.php?title=Thief:_The_Dark_Project&oldid=281232499, 2009, [Online; accessed 19-April-2009].

[3] Wikipedia, Half-life 2 — Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/w/index.php?title=Half-Life_2&oldid=284310153, 2009, [Online; accessed 18-April-2009].

[4] T. V. D. Community, The Valve Developer Community, http://developer.valvesoftware.com/wiki/Main_Page.

[5] I. Millington, *Artificial Intelligence for Games*, First ed. (Morgan Kaufmann, 2006).

[6] A. Champandard, Aigamedev.com, http://aigamedev.com/premium/presentations/behavior-trees, 2009, [Online; accessed 19-April-2009].

[7] G. S. David M. Bourg, *AI for Game Developers*, First ed. (O'Reilly, 2004).

[8] T. D. N. Finn V. Jensen, *Bayesian Networks and Decision Graphs*, Second ed. (Springer, 2007).

[9] Google, The Next Generation of Neural Networks, http://www.youtube.com/watch?v=AyzOUbkUf3M.

[10] P. P. Dasgupta, Learning Neural Networks, http://www.youtube.com/watch?v=6ixqKw7uK6o.

[11] J. A. Farr, Neural Networks in Computer and Cognative Science, http://www.cubiclemuses.com/cm/blog/archives/000015.html/, 2003, [Online; accessed 19-April-2009].

[12] S. Riihiaho, The Pluralistic Usability Walk-Through Method, http://moodle.vrml.aau.dk/file.php/20/Artikel_-_Pluralistic.pdf, 2009.

Appendix A

Playtesting: SW802B

A.1 Group SW802B

The following are the comments and suggestions for further development received from group SW802B during the playtest.

The questions were asked to the different members in the group, and the answers can therefore be contradictory.

Was the difficulty suitable?

- The game is to hard on easy difficulty setting.
- The game seems a bit slow moving and therefore more difficult.
- More hiding spots would make the game easier.
- The difficulty is spot on.

Does the Aslans seem intelligent and varied?

- It works very well.
- The Aslans work great.
- It works but the Aslans spends to much time in the central cubicle room.
- Yes as the Aslans searches through the building and reacts on sounds.
- Yes they tricked and caught me.

Could you initially figure out the objective for the game?

- No idea what to do.
- After a brief introduction the game was easy to play.

What worked well?

- The level design was awesome.
- The sound effects from the Aslans, such as the footsteps.
- The intro video worked very well and gave the game a movielike effect.
- Has not seen a game like this before, and it worked.
- It was nice to see where agents was looking.
- Able to hide behind tables.
- Cool with the ability to distract the Aslans.
- The concept worked well and gave a pounding pulse.
- Able to throw chairs.
- The game works even though there is no killing involved.
- The fish tank was cool.

What did not work well?

- Maybe monotonous in the long run.
- It should be easier to escape the Aslans.
- Too difficult.
- The Aslans could see me if I raised just a bit in the cubicles.
- Higher resolution on the minimap.

Extensions to gameplay

- Cloak timer.
- Player indicator bigger on the minimap.
- The minimap should turn with the player.
- Speed boost ability.

Appendix B

Playtesting: SW803A

B.1 Group SW803A

The following are the comments and suggestions for further development received from group SW803A during the playtest. The questions were asked to the different members in the group, and the answers can therefore be contradictory.

Was the difficulty suitable?

- The game is to hard even on the lowest difficulty.
- A smaller and simpler level to start with.
- A brief introduction or tutorial to the game would ease the learning curve.

Does the Aslans seem intelligent and varied?

- They seem very intelligent, at times to intelligent.
- They look intelligent in the way they are searching the building.
- Yes they seem very intelligent.
- They appear intelligent as individuals and as part of a squad.

Could you initially figure out the objective for the game?

- It took some time, but it is possible to figure out what to do as a player.

- It was difficult, a briefing screen would help.
- No I had no idea what to do.

What worked well?

- The cloak feature.
- Aslans ability to find the player.
- Distract Aslans with props.
- Hide in a room and see Aslans running past you.
- The paranoia effect the game gives you.
- The Aslans patrolling in squads.

What did not work well?

- The player could walk through chairs.
- The intro video could not be skipped.
- The agents can still see in dark spots.
- The minimap should have a indicator on players direction.

Extensions to gameplay

- An arrow pointing in the players facing direction would help.
- A power to run in a amount of time, like the cloak feature.
- Use shadows to hide in.
- Ability to make some sort of resistance to the Aslans.
- Open and close doors.
- Walk button
- Counter to know when cloak is done.
- Regeneration of health.

Appendix C

Screenshots

C.1 NoEsc Pictures

The following is four screen-shots from the game NoEsc.

Figure C.1: Intro sequence from NoEsc, Aslans running in the building.

Figure C.2: The cubicles room with the main character in focus.

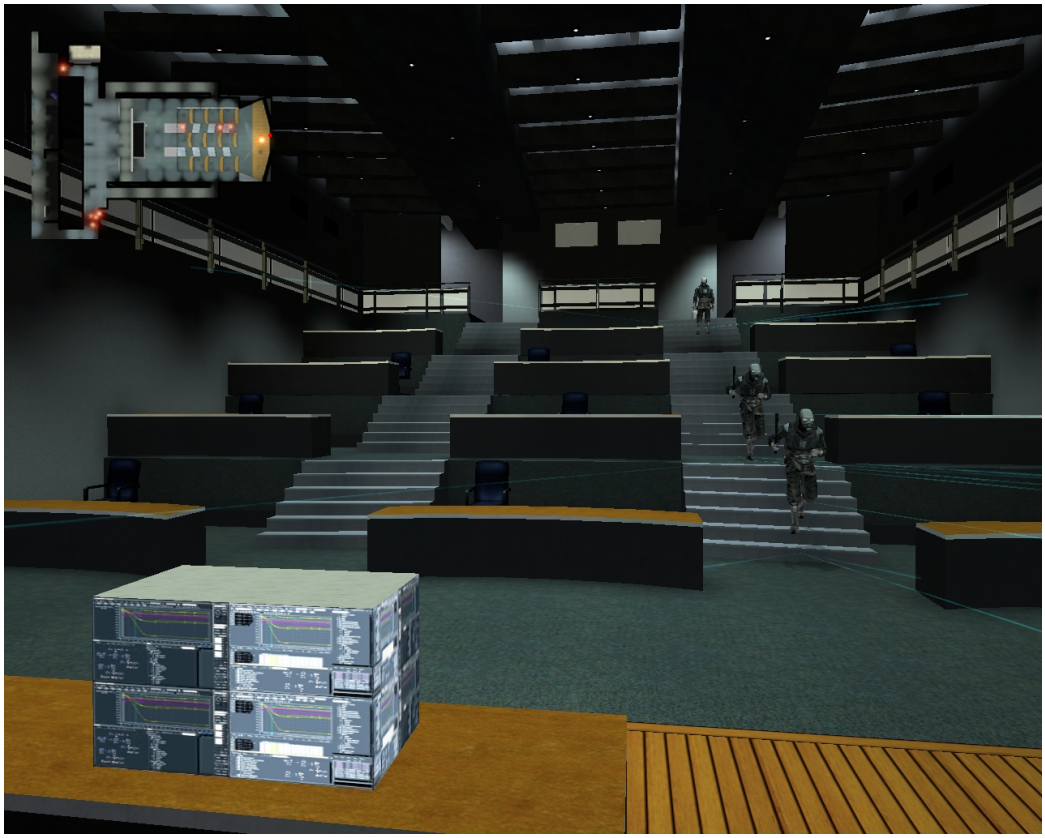

Figure C.3: Player facing an objective, while Aslans are running towards her.

Figure C.4: An objective close to the fish tank.

Appendix D

DVD

D.1 DVD Content

The DVD contains the following parts:

- Code:
  - NoEsc written in C++ with Visual C++ solution.
  - BTTool written in C# with Visual C# solution.
- NoEsc game folder and installation instructions.
- NoEsc Report:
  - Report as a PDF file.
  - Folder with figures from the report.
- Subversion repositories.