- Shai -
Scenario authoring for simulation games

Master Thesis
June 2010
Casper van Est

TU Delft
Delft University of Technology
THESIS

Submitted in partial fulfilment of the requirements of the degree of

Master of Science

in

Media and Knowledge Engineering

by

Casper van Est
Born in Geldrop, the Netherlands

TU Delft University of Technology
Computer Graphics and CAD/CAM Group
Department of Mediamatics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology Delft
the Netherlands

http://graphics.tudelft.nl/Game_Technology
Abstract

With the advent of affordable and powerful game engines, simulation games have become an increasingly effective solution for organising virtual training sessions in areas such as incident response, safety supervision and medical services. These simulation games are replacing real-life training sessions, which are often expensive and time-consuming. The downside with using simulation games for training purposes is that current simulation games lack the flexibility of adjustable training programs that real-life trainings have, due to simulation games being a finalised, and thus static, product.

This research focuses on how to increase the flexibility of training programs offered by simulation games. This is done by abstracting the scenario of simulation games, so it can be re-arranged by the training instructor using a standalone editing application. A prototype of such an application, called Shai, is developed to demonstrate and validate this approach.
Preface

This is my master thesis, which reports about my Computer Science graduation project at the faculty of Electrical Engineering, Mathematics and Computer Science at TU Delft. All related work was performed at the Gaming Street, a group within the Section Systems Engineering at the faculty of Technology, Policy and Management, also at TU Delft.

I would like to thank Dr. Ir. R. Bidarra, my supervisor at the TU Delft, and Ir. R. Poelman, my supervisor at the Section Systems Engineering, for their help, guidance and constructive criticism during this endeavour. I would also like to thank my colleagues at the Gaming Street and my friends and family for their support during the past year.
- Casper van Est, June 2010
# Table of Contents

1. Introduction 1
   .: 1.1 Problem Definition 1
   .: 1.2 Research Questions 4
   .: 1.3 Methodology and Structure 5
2. Domain Analysis 7
   .: 2.1 Simulation Games 7
   .: 2.2 Examples of Simulation Games 8
   .: 2.3 Definitions 12
   .: 2.4 User Analysis 13
3. Literature Review 15
   .: 3.1 Environment Authoring 15
   .: 3.2 Causality Authoring 18
   .: 3.3 Discussion and Recommendations 22
4. Approach 25
   .: 4.1 Scenario Abstraction 25
   .: 4.2 Building Blocks 29
   .: 4.3 Communication with simulation games 30
5. Prototype Design 33
   .: 5.1 Goals 33
   .: 5.2 Requirements 34
   .: 5.3 Features 35
   .: 5.4 Simulation Game 43
6. Implementation 45
   .: 6.1 Shai 45
   .: 6.3 Supervisor 52
7. Prototype Evaluation 55
   .: 7.1 Evaluation Plan 55
   .: 7.2 Evaluation Results 57
8. Conclusions and Recommendations 63
   .: 8.1 Conclusions 63
   .: 8.2 Recommendations 64

A. Original design & example scenario 67
B. Evaluation Tutorial and Questionnaire 69
C. Filled in questionnaires 79
D. References 91
1. Introduction

1.1 Problem Definition

When video games originally came into existence, their purpose was solely to entertain. Nowadays, with video games becoming more accepted by the mainstream, and with more scientific research being done in this area, a subset of video games called serious games (or more precisely simulation games, as we will see later) is being used for business and educational purposes as well (Smith 2007). One such application of simulation games is as a replacement of professional training sessions, where they are being used to educate safety supervisors, medical professionals, police officers, and other professionals.

The advantage with using a simulation game to replace real-life training sessions are numerous; simulation games are cheaper, safer, less time consuming and potentially more powerful (Susi, Johannesson et al. 2007). Regular training sessions can often be quite expensive, as they require real locations, real personnel and real equipment, not to mention the costs of recreating dangerous situations, such as explosions. Virtual training sessions can be carried out within virtual environments and as such require a lower budget, for instance when training fire fighters (Backlund, Engström et al. 2007). Also, the possibilities for recreating any type of training exercise are endless, even when exercises have to take place in huge, complex or even dangerous environments.

Using the advantages a computer offers, these games also have the potential to be more effective learning tools, considering improved assessment, higher adaptability, improved learning material, instant replays of training sessions, higher trainee motivation and so on. However, simulation games are not being used to their full potential yet.

However, one important issue that simulation games are currently facing is that the flexibility of adjusting the virtual training session to an individual trainee’s needs is relatively low, compared to current real-life training sessions. The reason for this issue is made apparent in the following Section, which globally describes the development and application phases of a simulation game.

When scenario authoring is better supported, it will be easier for a larger audience to be creative in the game development domain. This ultimately leads to better and more varied games, with improved learning processes. This research is aimed at providing an initial step in improving that learning process, while it can be used as a starting point for future research on scenario authoring.

1.1.1 The current development and application of a simulation game

To properly analyse the problems facing current simulation games, we take a look at how they are being developed and used. A visualization of the current process of developing and using a simulation game can be found in Figure 1, on the following page.
Figure 1: Current development and application of a Simulation Game

The development of a game is instantiated by a Commissioner. He or she recognizes a need for a simulation game and orders a Game Developer to develop the game. Also, the commissioner orders the instructor to develop a Training program with learning objectives and a training schedule. Then, the game developer combines the game world with the training program to create the final simulation game. This simulation game is then handed back to the instructor to use in a training session.

The fundamental problem with this approach is that the simulation game is handed to the instructor as a finalised product. In optimal conditions, the instructor is indirectly involved in the process of making the game, by defining the training program, but once the development of the game has finished, no additional changes can be made to the game, or to its training program. The feedback received from a training session needs to return into the development area, where a new version of the game needs to be developed for it to have any effect. Please note that there are exceptions to this scheme, but as we will see in the literature study in Chapter 3, these exceptions also have their issues and downsides.

One such alternative to this scheme is that some serious games are delivered as a finalised product, but with some options for adjustability. With these games, the game developer has (perhaps in consultation with an instructor) prepared a few options for the instructor, which the instructor can use to alter the gameplay. While this allows to instructor to exercise some control over the game's scenario, these options are very
limited and still restrictive in nature, since the instructor can only adjust these predefined settings. Thus, the serious game is still delivered as a finalised, static product.

Please note that this scheme makes some assumptions about the various roles. For instance, the Game Developer is usually a team of artists, programmers, game developers and musicians. Secondly, the instructor may not himself actually be involved in the development process, but in ideal circumstances, he is. Also, in some cases the instructor is referred to as a facilitator. Throughout this thesis, however, he will be referred to as an instructor.

1.1.2 Improvements to the development and application of a simulation game

The suggestion on which this thesis is based, is to make the training session more adaptable to the individual trainee’s needs, by providing the instructor with a Scenario Editor. Figure 2 illustrates this improvement, which can be found on the following page.

In this illustration, the game developer delivers not one, but two products; the simulation game and a collection of scenario building blocks. These are both supplied to the instructor. There, a separate application called the scenario editor can be used to combine the training program with the supplied scenario building blocks, to create an individual training experience, with which a training session can be organized, specific to a certain trainee. Using the feedback from the training session, adjustments can then be made to the training program, which can then be used to create another individual training experience, using the same scenario building blocks and simulation game. So, in fact, as opposed to the current situation, the feedback loop stays within the training area. Thus, the flexibility of adjusting a training scenario is returned to the instructor.

This approach is discussed in more detail in Chapter 4.
Figure 2: Improved development and application situation of a simulation game

1.2 Research Questions

This Section introduces our research questions, arising from the problems identified in the previous Section. These problems can be roughly summarized as:

Although simulation games are a powerful and cost-effective alternative to real-life training sessions, the flexibility of adjusting such training sessions to fit the needs of individual trainees is relatively low in simulation games. The reason for this is that simulation games are often delivered as a static product with predefined scenarios that mostly cannot be edited by an instructor.

This leads us to the research question, which is formulated as:

How can an instructor be given more control over the scenario of a simulation game?

Let us briefly discuss this research question in more detail. First, it references an instructor. As will be defined in Chapter 2, this is the person who leads training sessions, by executing a scenario, aimed at teaching something to trainees. Then, a scenario was defined by the global progression of a level in a game, defined by its initial settings and the logical flow of events and actions that follow. Specifically, this research question deals with scenarios in simulation games, i.e. games that have been designed for training
purposes. So, the domain under which this research question falls is that of training games. Examples of such games will be discussed in Chapter 2 and include Shell Supervisor, which will be used as an example and case study throughout this thesis. Finally, the expression ‘more control’ refers to enabling instructors to be free in adjusting the content of a training session to fit individual trainees’ needs, or to adjust the training program entirely in order to be more effective, according to feedback gathered in training sessions.

Finding an answer to the research question involves finding answers to smaller, more detailed questions. Therefore, the research question is subdivided into three more specific research questions. First, the description of a scenario needs to be elevated to a higher level of abstraction, so that it is understandable to an instructor. This leads us to sub-question no. 1:

**Q1: How can the scenario of a simulation game be represented at a more abstract level of scripting?**

Then, when the instructor can grasp the concepts used in a scenario, he needs to be able to use these concepts and edit the game’s scenarios. This leads us to the second sub-question:

**Q2: How can an abstracted scenario be made editable to an instructor?**

Finally, once the instructor has edited the scenario, these adjustments need to be communicated to the game. This leads us to the final sub-question:

**Q3: How can the adjustments that have been made to a scenario be communicated to a simulation game?**

When all these questions have been answered, we will know how to abstract a scenario, let the instructor make adjustments to it, and apply these adjustments in a game. This will give the instructor more control over the scenario, thus answering the main research question.

.: 1.3 Methodology and Structure

This Section will discuss the methodology used in this project. Also, it will propose the structure of this report. First, the requirements for the project are defined.

The following requirements have been set up for this project:
- A prototype standalone application should be developed which can be used by instructors to control scenarios in a simulation game.
- This prototype should be tested with regard to functionality and usability.
- The research should provide a starting point for future research.
- The thesis report should be a cohesive document that presents a clear overview of the research work. It should contain conclusions that are well founded and address all research questions.
- All products should follow scientific standards (source quotations, repeatability, reproducibility, etc.)
The methodology used follows the following steps:

- First, a literature study has been performed, with the purpose of exploring current scenario authoring solutions, identifying their approaches and issues. The contents of this literature study will be revisited in Chapter 3.
- At the end of the literature study, a research question was posed for this thesis, which was discussed in the previous Section. The main research question was subdivided into three smaller, more specific questions.
- To answer the research question, an approach has been formalized that discusses how an instructor can be given more control over a scenario, and how to do this by circumventing the issues found in currently existing scenario authoring solutions.
- To test whether or not this approach can be successful, a prototype has been designed and developed. During the development of this prototype, some issues were identified, to improve upon the approach. Secondary parties, such as the serious game development company e-Semble, were approached to shed their thoughts on scenario authoring as well.
- Then, when the prototype was finished, it will be evaluated to see if it fulfils the goals it was set out to achieve. The evaluation led to the final conclusion and recommendations for this project.

The prototype was developed in an iterative process. First, an initial design was formulated and implemented. This basic version was used as a visual support to the initial approach discussed in this thesis. Using this prototype, the main issues in the approach were identified. Feedback came from the project supervisor, the supervising professor and e-Semble, who also have interesting ideas on the subject. Improvements were then be made to the approach and the design, which resulted in a better and more complete approach and an improved prototype. This process continued until a satisfactory approach was developed and implemented. The final prototype was then evaluated by all involved parties, upon which a conclusion and recommendations were resolved.

This thesis report starts by giving an introduction on simulation games. Some examples are discussed, to highlight current issues. Then, the literature study, which has been performed prior, is briefly revisited in Chapter 3. The approach that has been used to find a solution to the research question is discussed in Section 4. Then, in Chapter 5, the design of the prototype is discussed, using a goal tree, requirements and features. Chapter 6 discusses the implementation details, while Chapter 7 discusses a case study, in which the prototype is used in cooperation with a simulation game. Finally, Chapter 8 discusses the evaluation of the prototype, concluded by Chapter 9, in which a conclusion and future recommendations are discussed.
2. Domain Analysis

This Chapter presents the domain of this research. It starts by discussing simulation games and provides some examples of them, with the goal of illustrating current issues. Also, some definitions of terms that will be used throughout this report are presented for the sake of clarity. Finally, a Section is dedicated to an analysis of users targeted in this research.

2.1 Simulation Games

The scope of this research work is simulation games, a subcategory of the category of digital games known as Serious Games. This term has been in use since the 1970’s, when it was coined by Clark Abt in his book Serious Games (Abt 1970). The definition he used, “we are concerned with serious games in the sense that these games have an explicit and carefully thought-out educational purpose and are not intended to be played primarily for amusement”, is still applicable today. It was not until 2002, however, when the Serious Game Initiative was launched, that serious games were taken seriously.

Today, there is no single, strict, definition of serious games, though they are generally held to be games used for training, simulation, education, and arguably advertising. Serious games can be of any genre, use any game technology, and be developed for any platform. So, a serious game is not limited to digital games, it can also be for instance a board game, or a combination of board and digital game. This research, though, is focused on serious games that use computer game technology.

Currently, serious games are classified in various categories (although this classification is not by any means official, nor unanimous), including:
- Advergames
- Art games
- Edumarket games
- Edutainment games
- Game-based learning
- Games for health
- Militainment
- News games
- Organizational-dynamic games
- Persuasive games
- Simulation games

On a personal note, I would not categorise advergames, art games or edumarket games as serious games, as they do not have an explicit educational purpose. Other categories can also be argued to be included in this list, but since there is no explicit definition of serious games that is widely used, we'll leave this discussion for another day.

The research discussed in this thesis will focus primarily on the subcategory of simulation games. A simulation game is usually defined as a simulation or re-enactment of various activities of real life in the form of a game for training, analysis or prediction purposes. So, basically, the difference with ‘regular’ serious games is that simulation games focus more on training or prediction purposes.
2.2 Examples of Simulation Games

This Section shows some examples of simulation games, and discusses their details. The first two examples, Shell Supervisor and Hugo Grotius, have been developed by the Gaming Street, and are useful examples, as the instructor has cooperated in the development of these games, and can therefore give extensive details. Other examples include America’s Army and SimPort, to give a varied view of the types and applications of simulation games. As will be clear by the end of this Section, simulation games exist in widely different genres, goals and applications.

2.2.1 Shell Supervisor

The Supervisor simulation game was created in 2009 by the Gaming Street Section of TU Delft, commissioned by Shell. It is designed to be used as a virtual alternative to real-life training sessions. In this game, the player assumes the role of a safety supervisor at an oil drilling site. The player is expected to handle hazardous situations, watch personnel and take care of health, safety and environment requirements.

The game is played through a 'first person perspective', which is also used in games such as Halo, Unreal Tournament and Half-life. While playing the game, a scenario unfolds in which the player must:
- correct any safety issues, such as clogged drainpipes or broken fire extinguishers;
- deal with unexpected situations such as heavy rain;
- correctly and safely handle typical events such as the unloading of a truck.

The scenarios used in this game have been developed by researchers at TU Delft, in cooperation with employees from Shell who have experience with such drilling facilities. These scenarios are in accordance with scenarios used in the real life training sessions which the game is meant to replace.

In Supervisor, the instructor is responsible for teaching trainees how to become a competent safety supervisor. This is done by developing training scenarios and guiding trainees through them.

 Supervisor is a prime example of the issues that can be found in simulation games, as discussed in Section 1.1. The instructors, or other employees at Shell, did have some influence on the development of the game; they provided information on what type of scenarios should be developed. However, once the game was finished, its use was limited to whatever the game developers had implemented. The instructor who then used the game to perform training sessions had only the choice of a handful of different scenarios. Therefore, the use of the game was very limited.
2.2.2 Hugo Grotius

Hugo Grotius was another serious game developed by the Gaming Street at TU Delft in 2009. The game was created using the same game engine as in Supervisor. However, the game is played quite differently. Instead of a first person viewpoint, the viewing perspective is similar to point-and-click adventures such as is used in the Monkey Island series by LucasArts.

The game has been commissioned by the Natural History Museum (NHM). In Hugo Grotius, the player assumes the role of Hugo’s chamber maid Elsje. The player has the task of preparing the escape of Hugo Grotius from his imprisonment in Castle Loevenstein.

While playing the game, a scenario unfolds in which the player must:
- Unload books from a book chest and place them in Hugo’s bed, mimicking his body;
- Assist Hugo in getting in the book chest;
- Distract the guards to ensure Hugo’s escape.
The scenario used in this game has been developed by researchers at TU Delft, in cooperation with historians from the NHM, to ensure historical correctness. The scenario is in accordance with how the actual escape was performed.

Although providing an interesting example of a different kind of serious game, there were no direct issues with respect to scenario authoring in this particular game. This was because the game was meant to be a finalised, static product which would only be played once (in a museum). However, the game could still benefit from standalone scenario authoring, as this could potentially expand upon the concept and give more freedom to the players.

2.2.3 SimPort

SimPort-MV2 is a computer-supported multi-player simulation game that mimics the real processes involved in planning, equipping and exploiting the Second Maasvlakte (MV2) in the Port of Rotterdam, the Netherlands. The game is owned by a consortium of TU Delft/CPS, The Port of Rotterdam and Tygron Serious Gaming.

When compared to well-known entertainment games, its closest resemblance would be to SimCity, although this game is different in a lot of respects. First of all, the game uses a lot of real objects besides the game, such as a planning board, maps of the port and a whiteboard or flipover. Secondly, the duration of the game is usually between 5 and 8 hours and it is played by a team of people (3-6 players per team).
The goal of the game is to:
“Individually and collectively make the appropriate planning and implementation decisions to lead to satisfactory design and sustainable exploitation of the Second Maasvlakte (1,000 ha) over a 30-year period.”

SimPort-MV2 is a game in which participants gain insight in the consequences and effects of choices within long-term strategies. Participants tackle different aspects of managing a complex project as they are confronted by time pressure and seemingly conflicting interests. Whether the participants will be able to stick to their chosen strategies throughout the game is a decisive factor for eventual success.

This simulation game is, in the first place, meant for staff of the Port of Rotterdam. But its emphasis on strategizing, project management and teamwork make it appropriate for others as well, such as trainees and (young) professionals as part of their education or training courses.

2.2.5 America’s Army

America’s Army is probably one of the best known simulation games. It was developed by the US Army in 2002 using the Unreal Engine and was released (as freeware) as a global PR initiative to promote recruitment. The vision of the game was to use computer game technology to provide the public a virtual Soldier experience that was engaging,
America’s Army is a multiplayer, round based team tactical shooter game with the player acting as a soldier in the US Army, with combat at squad-level with three fire teams. It is considered to be one of the most realistic games in its portrayal of weapons and combat. In the game, players are bound by Rules of Engagement and grow in experience as they navigate challenges in teamwork-based, multiplayer, force versus force operations. As in the real Army, accomplishing missions requires a team effort and adherence to the seven Army Core Values.

America’s Army is a prime example of militainment, defined as *entertainment featuring and celebrating the military*. Besides the various awards the game has won, the game has also received some controversy, with it being a propaganda tool for the US military. It has also been the focus of various academic studies.

### 2.3 Definitions

For the sake of clarity, this Section defines the terms that will be used throughout this report, including the definitions of serious game and simulation game that have been given earlier.

- A *Scenario* is the global progression of a level in a game, defined by its initial settings and the logical flow of events and actions that follow.
- A *Scenario Editor* is an application allowing the creation, editing and deployment of a scenario.
- A *Scenario Instructor* is the user that creates and edits a scenario using the Scenario Editor.
- An *Instructor* is the person who leads training sessions, by deploying a scenario, aimed at teaching something to trainees. Typically, the instructor is the same person as the scenario author, but this is not necessarily so. In some situations, the term *Facilitator* is used to indicate the instructor.
- A *Game Developer* is any person who creates the game and defines the content that can be utilised in the Scenario Editor. This is usually a team, but for the sake of argument, this will be referred to as a person.
- A *Player* is the person who plays the serious game.
- A *Trainee* is a Player who is using the game to be trained or educated.
- A *Serious Game* is a game that has an explicit and carefully thought-out educational purpose and is not intended to be played primarily for amusement.
- A *Simulation Game* is a Serious Game that has been developed for training, analysis or prediction purposes.
2.4 User Analysis

This Section will discuss two actors involved in the design, development and use of simulation games; the instructor and the game developer.

2.4.1 User Characteristics

First, we identify the characteristics of the instructor and the game developer. The instructor is considered to be an expert at education, and an expert in the subject domain. The game developer is considered to be an expert in domains related to game development, such as game and level design, programming, art, user interface design and human-computer interaction. While the instructor is an expert in educating trainees on the subject domain, he is typically not acquainted with virtual 3D environments. Similarly, the game developer is highly skilled with respect to the various aspects involved in the development of virtual 3D environments, but typically has little knowledge of the subject domain.

2.4.2 User Responsibilities

Connected with each user’s characteristics are their responsibilities. Naturally, the responsibilities of instructors should be restricted to education and the subject domain, while the responsibilities of the game developer should lie within the area of game development. However, in the current situation, as described in Fig. 1, the responsibilities of these two users tend to overlap.

So, it is important to define exactly what the responsibilities are of instructors, with respect to the development of a game. Sometimes this is not so easily defined, as at some points in the development of a simulation game, the responsibilities of the instructor and the game developer overlap. An example of this is the placement of gameplay objects in the virtual environment:

In a game such as Supervisor, discussed in Section 2.2.1, the location of gameplay objects is irrelevant to the learning process. For example, when the goal stated by the learning process is that the player needs to learn the proper use of, say, a fire extinguisher, it is irrelevant \textit{where} that extinguisher is placed. However, in other games, the location of certain objects \textit{can} be very relevant to the learning process.

So, in one case, the location of a gameplay object does not fall under the responsibilities of the instructor, while in another case it does. One could now say that since the location of such an object is at least \textit{sometimes} the responsibility of the instructor, he should therefore \textit{always} be burdened with the responsibility of placing such objects. However, this would also imply that the instructor needs to place \textit{all} gameplay objects himself, even some objects that are irrelevant to the learning process and thus do not fall under the responsibility of the instructor. A solution is to allow the game developer to \textit{tag} certain gameplay objects that he deems relevant to the learning process. The selection of which objects should be tagged as such can be influenced by the instructor, in the design phase of the game.
Placing objects can happen in two ways. Either the instructor has total freedom to place objects, or he is restricted to several options. Although it would be preferred to give the instructor as much freedom as possible, too much freedom can be overwhelming. For instance, this would typically require the use of some kind of 3D environment editor, which is considered to be quite technical to operate, and it has been stated earlier that this is one of the weaknesses of instructors. Instead, we will offer the instructor the option to select the location of certain objects from a list of predefined choices. This list is defined by the game developer, because he is considered to have the best insight into which locations are appropriate, both from a technical as well as from an artistic standpoint.
3. Literature Review

This Chapter presents a review of a literature study that has been performed in 2009 in preparation of this research work, with the goal of answering the following research question:

What affordances are currently being offered by scenario authoring applications for users with little technical experience, and how is this limiting the use of these tools?

The study discusses current authoring methods. For each method, one or more tools are discussed, as they provide an example of the use of these authoring methods. The methods can be divided into three categories; textual authoring, environment authoring and causality authoring. In environment authoring, the editing environment is similar to the game world; it provides the user with a view of the environment which is comparable to the world as it is presented to the player. In causality authoring, the instructor operates in a distinctly different environment than the game world, such as an abstract graph-based editor. Finally, in textual authoring, the scenario is defined by some kind of written script, be it through XML or programming.

This last type of scenario authoring is not discussed in the research report, as it requires programming experience and is very user-unfriendly, and therefore falls outside of the scope of the research. The strengths and weaknesses of the two other authoring methods will now be discussed in the following Sections.

.: 3.1 Environment Authoring

In environment authoring, the instructor can directly influence the game’s environment. The instructor can place assets (objects, characters, triggers, markers, etc) and move them around. By placing a number of these assets, and assigning certain properties to them, the instructor can influence the course of the scenario. A real world example of environment authoring is a child playing in a sandbox: he builds an environment with perhaps buildings or foliage, places a handful of characters and then 'runs' the scenario.

Environment authoring offers the instructor direct control over specific assets in a game environment. Thus, it offers the instructor great and precise power. However, it also requires the instructor to directly influence the game world, thus requiring the instructor to have a decent amount of knowledge about the game world. The instructor is required to know about game development concepts such as placing and moving objects around in a virtual 3D world, using triggers, materials, etc.

Two game development tools using Environment Authoring were discussed; the Unreal Environment Editor and e-Adventure. The former tool is a prime example of current generation top-notch commercial game development tools, while the latter tool has been developed at the Universidad Complutense de Madrid and has been extensively written about by its developers.
3.1.1 Unreal 3 Engine: Environment Editor

![Screenshot of the Unreal 3 Environment Editor](image)

Figure 6: Screenshot of the Unreal 3 Environment Editor

This editor lets the user place all game related objects such as buildings, foliage, particle, characters, objects, triggers, cameras and much more in the game environment. Its interface is comparable to that of a 3D modelling tool such as Maya or 3DMax and usually offers 4 viewports; a front- side- and top-view plus a 3D view of the scene. The editor offers a lot of functionality, available from all the different buttons scattered around the screen. To fully make use of all the features in the Unreal Editor, the user has to use several other screens as well, such as the Generic Browser, the Scene Manager and the Actor Classes Tab, on which he find the classes, object and materials he wishes to use in the environment.

As one can imagine, all the features offered by the Unreal Editor make it a powerful, but also very complex program. The requirements this tools asks of its user are quite high. For instance, the user needs to be comfortable with moving around and placing 3D objects in a 3D environment, he needs to understand what classes, objects, particle systems, materials and textures are, how to correctly apply them, how to create a camera animation, and so on.

The Unreal environment editor is perfectly suited for professional game developers; it offers a great amount of options and freedom to fully create a beautiful and complex 3D game environment. However, the abundance and complexity of all its options make it too powerful and too detailed for anybody without game developing experience.
3.1.2 e-Adventure

For the instructor, creating a game in e-Adventure (Moreno-Ger, Martinez-Ortiz et al. 2005) (Moreno-Ger, Blesius et al. 2007) is made easier by allowing him to author and execute a game without any background in programming. The instructor can author game scenarios and add content to them, such as objects, characters and conversations. The authoring application focuses on supporting those tasks that are specific to the educational domain. Among these are assessment and adaptation: the need to track and evaluate the activity of the trainee and the need to adapt the behaviour of the game to fit different ranges of trainees, respectively. A noteworthy feature of e-Adventure is the possibility to link to other sources of information, to be accessible during the game. This is of course also an important educational aspect.

The functionality offered by e-Adventure is too limited to be suited for professional game developers; only one type of game can be created, that game has to follow certain specific guidelines and there are little options for customizing the game. At the same time, the actions required to create a game using e-Adventure are too detailed to be suited for non-professional game developers. The user has to concern himself with technical issues such as foreground masks, layers, inventory item icons, state transitions, conditional logic, etc. Aside from causing the creation process of a game to take an unnecessarily long time, these options are overwhelming to a didactic expert with no game development experience.

The main objection to e-Adventure is that although it is easier to use than commercial engines, it still requires the instructor to create a game, instead of creating or adapting a scenario. It is up to the instructor to create the environment and place assets in it. This
work could be done by game developers, but then there would still be no possibility for flexibly adapting the scenario.

3.2 Causality Authoring

This Section will discuss the second authoring method; causality authoring. This method lets the instructor edit the causality processes of a scenario, usually by presenting a graph metaphor. Using this authoring method, authors can specify causalities such as ‘when the user opens that box, he will receive this object’.

As opposed to environment authoring applications, a lot has been written about tools that use causality authoring. This is particularly due to authoring applications created for authoring interactive drama. These tools such as Scribe, Façade, Scenejo, Art-e-fact and SAVEace, will be discussed in this Section. The commercial Unreal 3 Engine also offers a tool for causality authoring, called Kismet.

3.2.1 Unreal 3 Engine: Kismet Editor

Accompanying the Environment Editor discussed earlier, the Unreal 3 Engine also offers a graphical, node-based, flow graph-type editing system, called Kismet. This tool allows game developers to set up events, triggers, and other game logic by connecting various logic boxes to each other with lines between their input and output gates, and defining the logic boxes’ properties and state changes. This allows designers to build complex level behaviour without needing to write C++ code or any type of scripts. An example screenshot of a small piece of Kismet is shown below.

![Screenshot of Unreal's Kismet](image.png)

However, even for this more easy-to-use editing system, a lot of knowledge about games and developing games is necessary from the designer. Even though he doesn’t have to
write the code himself, in order to use the node effectively, the designer does have to have knowledge about complex virtual world concepts such as collision, physics, artificial intelligence, user interfaces and other aspects of games.

Therefore, although the Unreal Engine is one of the best available commercial engines when it comes to usability and design tools, it is still too powerful and complex to be used by training facilitators without a background in game development or programming.

3.2.2 Scribe

Scribe (Medler and Magerko 2006) is an authoring application for event driven interactive drama, developed by Medler and Magerko in 2006. Scribe offers three different authoring modes: element placement, story creation and debugging. In element placement mode, the instructor can edit the environment, place element pieces on the level map, which is an example of environment authoring, discussed in the previous Section. In story creation mode, the instructor can create the storyline structure across varying levels of story detail, using the element pieces from the level map. In debugging mode, the instructor can query the director agent inside Scribe as to how it would handle various storyline situations, checking if the behaviour is as desired.

![Figure 9: Story Creation window for Scribe](image)

When the instructor is in story creation mode, a graph area to lay out plot points is shown. Plots points are used to describe major parts of the overall storyline. Each plot point consists of three parts: preconditions, events and actions. These three parts help with the generality of plot points because the statements in each of these parts can be relative to whatever environment Scribe is used for.
The plot point graph area allows for the visual organization of plot points and events. Plot points can be created or selected by the instructor and annotated (i.e. adding preconditions, events, and actions). Scribe is also capable of making use of a planning algorithm, instead of relying on the instructor to causally connect each plot point to form a story plan. When the instructor sets a plot point’s preconditions and actions for use in the story plan, ordering and causal links are created to connect plot points.

Finally, the instructor can use a skill model. Performing skills correctly means that the trainee has a higher score in the corresponding skills. Story statements can take this skill information into account, such as setting how high a trainee’s skill has to be before a certain plot point can begin.

### 3.2.3 Façade

In 2000, Mateas et al. started work on Façade (Mateas and Stern 2000) (Mateas and Stern 2003), with the goal of improving interactive drama worlds by integrating the local behaviour of agents with the global story plot. A drama manager is used to guide the characters and story progression, making sure the story is unified, efficient and properly paced.

The drama manager continuously monitors the progression of the story and regularly updates the behaviour of the characters by using story beats. A beat is defined as the smallest unit of dramatic action that moves the story forward. Beats are annotated by the story’s instructor with preconditions and effects on the story state, which the drama manager can use to solve a partial ordering plan of beat sequences, creating a dramatic narrative. In the Beat Sequencing Language developed for Façade, the instructor can annotate each beat with preconditions, weights, priorities and effects.

The instructor can also define actions that are performed at various stages in the beat selection process, and can define beat variables that are accessible by all tests and actions within a beat. So basically, the instructor can define the little elements that together form a story. He can annotate each element, a beat, with pre- and post conditions. It is then up to the drama manager and beat sequencer to use the beats to create a single storyline, using input from the player and the artificial characters.

So, the instructor has no direct (real-time) control over how the current story will evolve; it is up to the drama manager to select the most appropriate beats for the current story state, which in Interactive Storytelling is usually measured according to how well a beat corresponds with an ideal story arch.

### 3.2.4 Scenejo

Scenejo (Weiss, Muller et al. 2005) (Spierling, Weiss et al. 2006) uses a graph metaphor as a means for structuring content, on several levels of abstraction. A plot in Scenejo is divided into scenes, where a scene element is a means to separate potential dialogues and actions according to a timed order. Scenes have transition conditions for moving from one scene to the next, for example some elapsed time, certain user utterances or state changes. The authors of Scenejo have written an evaluation of their prototype, which provided some interesting points on their use of a graph metaphor.
First, the visual plot line of scenes appears to be the best fit for the highest level of abstraction. Although the directed graph approach is a useful tool, it does seem to result in a linear structure very often. At the high level of a story, this is not so much a problem. It can be a problem, however, at lower levels, since those levels require more emergence. Immediate feedback to the authors is essential when emergence is involved, since it makes tuning and testing more important aspects.

Second, the authors note that the graph tools offer accessibility to first time users. However, frequent users experience usability problems due to low efficiency. In future iterations of Scenejo, text interfaces will be mixed with graphical interfaces. Another positive quality of the graph representation is the recognizable visual patterns, allowing easy identification.

Finally, a main drawback with graphs is noted to be their size; in complex stories, they can easily expand to a larger-than-screen size. Also, manually creating an entire story can become cumbersome and time-consuming. Proposed solutions include a fisheye view and graph abstraction.

3.2.5 Art-E-fact

Iurgel et al. proposed Art-E-Fact in 2004 (Iurgel 2004), which is similar to the previously discussed Façade project as it offers the player an experience in which he can interact with artificial characters, guiding him past artworks in a museum. The user can interact using a keyboard, which makes the dialogue management very complex, as the characters need to be able to process a wide variety of inputs.

A dialogue management module is presented, which uses a directed graph approach. This approach is considered by Iurgel et al. to be appropriate, if certain measures are taken to ensure its usability. The authors note that visual representations of authoring applications for a directed graph are the most intuitive interfaces for non-programmers. In an attempt to avoid an exploding number of connections, which is often a problem in graph editing, composites states and rule based state changes without explicit transitions were used.

3.2.6 SAVEace

Holm et al. discussed their work on SAVE in 2002 (Holm, Stauder et al. 2002). In their paper, the desktop authoring application SAVEace is proposed, which is an element of the SAVE Safety Virtual Environment. SAVEace allows users to construct a complete scenario for SAVE without scripting or programming, where scenarios are assembled from a collection of prebuilt components. SAVEace consists of a GUI editor, a 3D editor as well as a dependency editor, between which the user can switch effortlessly. Consequently, it is reminiscent of the Unreal 3 Engine. As such, it offers the same problems found in the Unreal Environment Editor and Kismet Editor, in that the user has to have some knowledge of games to operate the dependency editor.
3.3 Discussion and Recommendations

In the preceding Sections, two authoring methods have been discussed; Environment Authoring and Causality Authoring. With environment authoring, the instructor can directly influence the game environment. The instructor can place assets (objects, characters, markers, etc) and move them around. It gives the instructor direct control over content in the game. As such, it is very powerful. However, it also requires a lot of work and technical knowledge of the author, since he is responsible for creating the entire environment.

Causality authoring is featured in most scenario authoring applications. Causality authoring, in which the instructor edits the causality processes of a scenario, often uses a graph metaphor. Using this authoring method, authors can specify causalities such as 'when the user opens that box, he will receive this object'. Editing a graph is easier than editing a game environment, since it requires less technical knowledge of the author.

For both methods, some example game development tools have been discussed. Some tools actually offer the use of more than one of these methods, as each method has a specific application it is best suited for. Such is the case with the Unreal 3 Engine, which has been discussed with respect to Environment Authoring as well as Causality Authoring.

These tools have been discussed with a focus on their features, in an attempt to provide an overview of which features are useful and which issues concerning usability still exist. A summary of the results can be found in table 1.

<table>
<thead>
<tr>
<th>Authoring type</th>
<th>Unreal Editor</th>
<th>e-Adventure</th>
<th>Unreal Kismet</th>
<th>Scribe</th>
<th>Scenejo</th>
<th>Facade</th>
<th>Art-E-Fact</th>
<th>SAVEace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of a graph metaphor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requires knowledge of gaming concepts</td>
<td></td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Complex User Interface</td>
<td>v</td>
<td></td>
<td>v</td>
<td>v</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requires a lot of work</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Generic design</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Aimed at (serious) gaming</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Aimed at interactive storytelling</td>
<td></td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>

Table 1: An overview of features found in current scenario authoring applications

In this table, one can see that some issues are prevalent across most tools. Tools often require a lot of work, offer a complex user interface and some require knowledge of gaming concepts. It can also be noted from the table that all causality authoring applications make use of a graph metaphor. Apparently, this metaphor is considered very suitable to this application.
Interestingly, only one tool (Scribe) has built a design which is focused on generality, although the current implementation is only suitable with one environment.

Some interesting features, which could be useful in any future work, include the thoughtful user-friendly interface of Scribe, which provides a timeline as is used in video-editing software. This could be particularly helpful to keep track of the timing of a scenario, and is something that feels missing in other causality authoring applications, such as Unreal's Kismet. Another interesting feature is implemented in Scenejo, which offers multiple layers of abstraction. This can help provide an overview to the player, and potentially be a solution to the problem of graphs getting overly complex in larger scenarios.

One of the issues involved when using graphs for scenario authoring is the tendency of graphs to become extremely complex when scenarios increase in size. Another issue, noted by the developers of Scenejo, is the low efficiency of exclusively using graphs. In fact, they suggest using a combination of text and graphs for advanced users. Interestingly, they also note that using graphs has proved to be very user-friendly for first time users.

In summary, limitations in current authoring methods provided by game development tools are found to be:

- Authoring requires knowledge of gaming concepts
- Authoring requires too much work
- Authoring systems are designed non-generically
- Authoring systems offer unfriendly user interfaces
- Graphs can become too complex

One disadvantage of all current authoring methods is that none of them offers a generic solution; no standalone tool exists that allows scenario authoring to function with any other game development tool. This would improve the effectiveness of such a tool, as it can then be applied in multiple projects.

The graph metaphor used in several high level scenario authoring applications seems to be a good fit, since it corresponds well with the user's concept of a scenario. However, the interface that is usually provided can be very complex, especially for non technical users. An authoring application would benefit from using more graphical metaphors, such as icons for pre- and post conditions, as can be found in tools aimed at children.

The main issue with current authoring methods for the use in simulation games is that they have not been explicitly designed for use by a field expert or instructor. These experts typically have little knowledge of the technical concepts required in current authoring methods. For maximum usability, a scenario authoring application should offer easy functionality that allows anyone with zero experience to create or edit a scenario within a few minutes. Of course, since the scenario editor also needs to be powerful and support creativity, the tool should offer considerable depth, allowing the user to create complex scenarios as well. Perhaps a separation between basic and advanced features could offer some improvements to the usability.
4. Approach

This Chapter presents the approach used to tackle the problem stated in Section 1.1. The main research question posed in that Section was

How can an instructor be given more control over the scenario of a simulation game?

The approach taken in this project is to give an instructor more control over the scenario of a simulation game by letting him write a script at the abstracted level of scenario scripting, using the language of scenario building blocks. More control is given to the instructor by allowing him to make real-time adjustments to the scenario as well.

The following Sections discuss the various aspects of this approach, each by answering one of the three sub questions posed in Section 1.2. First, we will discuss how the scenario of a simulation game can be scripted at a more abstracted level. Then, we will discuss how such an abstracted scenario can be made easily editable to the instructor. Finally, we will discuss how the scenario script can be communicated to a simulation game.

4.1 Scenario Abstraction

In this Section, we identify three levels of scripting, which are (in order of abstraction level) programming, gameplay scripting and scenario scripting. The first two levels are commonly used in game development, while the last level will be introduced by this approach. This Section also discusses how this third, more abstract, level of scripting can be achieved, by taking the same abstraction step which is used to go from the programming level to the gameplay scripting level. Thus, it provides an answer to the first sub question posed in Section 1.2.

4.1.1 Levels of scripting

The first, lowest, level is called the programming level. The scripts in this level deal with low level concepts, such as objects, vectors, math functions, etc. The language used to write a script in this programming level could be any programming language such as C++ or UnrealScript. An example of a script at the programming level would for instance be:

1. If key == “up”
   2. Move Forward by 1 step
3. Else if key == “down”
   4. Move Backward by 1 step

In this example, the script deals with handling player input. The programming skills required at this level are high, so only those users with a background in programming can write scripts at this level.

The second level of scripting is called the gameplay scripting level, in which the scripts deal with the objects in a game level. The language used in the gameplay scripting level
is easier to use than programming languages and could for instance be UnrealEd's visual scripting language Kismet, which was discussed in Section 3.2.1. An example of a script at the gameplay scripting level could read:

1. Create a truck T at location XYZ
2. If the Player clicks on button B
   3. Play animation: opening gate G
   4. Play animation: truck T drive through gate G

In this example, the script deals with specific objects and specific locations in a game, for instance 'truck T' and 'location XYZ'. Considering interaction with the player, it operates on a level of atomic actions that the player's avatar performs, such as 'click on button', as opposed to the physical interaction of pressing a key in the lower programming level.

These two scripting levels, programming and gameplay scripting, are commonly used to develop games. In current game development teams, programmers operate in the lowest scripting level and write all kinds of scripts on how the engine should simulate the playing world. Then, gameplay scripters (or level designers) define what can be called the game's behaviour, e.g. how it interacts with the player, in the gameplay scripting level.

What we propose in this approach is to add a third level of scripting above the previously mentioned levels, in which the global scenario of a game is scripted. We call this third level the scenario scripting level, and it deals with the scenario of a game, as defined in Section 2.3: the global progression of a level in a game, defined by its initial settings and the logical flow of events and actions that follow. The language used in this level is even easier to use than the visual scripting language used in the gameplay scripting language, thus making it understandable to user with little technical background. An example of a script in this most abstracted scripting level could be:

1. Drive the truck to the unloading area
2. If the player has unloaded the truck
   3. Drive the truck away

In this example, the script is more concerned with the global logical progression in the game, as in 'what events follow other events'. An event could here be a process that takes several minutes to finish, as opposed to the few seconds of an atomic action in the gameplay scripting level. As can be seen, this script also has no references to specific objects or locations in the form of for instance 'location XYZ', but instead references them as 'the unloading area'.

This high level of scripting is useful because it allows the instructor to not worry about programming or gameplay issues, and instead focus on the scenario. The instructor doesn't want to be bothered by issues such as which truck is used, how it moves or what colour it is. All these issues are dealt with in lower scripting levels, and are defined by one of the game developers. Since the scenario is the most important didactic tool an instructor has, this approach is useful to the instructor because it lets him focus on his area of expertise, which is creating an educational scenario, fitting his didactic goals, as we have seen in Section 2.4.
4.1.2 Achieving a more abstract level of scripting

Achieving this higher abstracted scripting level can be done by taking the same abstraction step that is taken from the programming level to the gameplay level, but by now applying it from the gameplay level to the scenario level.

To illustrate the abstraction step that is taken from the programming level to the gameplay scripting level, we will use Supervisor as an example. As noted before, Supervisor was developed using the Unreal 3 Engine, which features UnrealScript as the programming language and Kismet as the gameplay scripting language. Fig. 10 shows these two scripting levels. The bottom part shows examples of UnrealScript, while the top part is an example of Kismet. Notice how a single node in Kismet is linked to an UnrealScript file.

![Diagram showing the relationship between programming and gameplay scripting levels](image)

**Figure 10: Scripts at the programming level are the implementations of nodes at the gameplay scripting level**

At the programming level of the Unreal 3 Engine, lines of UnrealScript can be written that together represent a node in the more abstracted level of Kismet. In Kismet, these
nodes can then be combined to define a gameplay script. In this way, a step of abstraction is taken from the programming level to the gameplay scripting level.

Now, if we take that same abstraction step again, and combine several nodes in Kismet, we get a more abstracted element, which we will call a *scenario building block*. These building blocks then operate at a more abstracted level; that of scenario scripting. Fig. 11 illustrates this concept.

![Figure 11: Scripts at the gameplay scripting level are the implementations of nodes at the scenario scripting level](image)

At this point it should be noted that problems can occur when there is some kind of conflict of interests, such as a use of common assets, between two scenario events. For instance, if the instructor has no knowledge at all about the game world, and orders a truck to move to point A and without knowing where or what point A is; orders another truck to move to point A, or orders the same truck to move to point B, the result may be unpredictable, and potentially problematic.

So, the abstraction level of scenario scripting needs to be chosen in such a manner that it allows the instructor to have as little knowledge about virtual worlds as possible, but at the same time still provides enough information to allow the instructor to accurately predict the results of his actions. The instructor should therefore be limited in his actions, making it impossible for him to define any conflicts in his scenario. The instructor can be helped in this issue, by providing him with quick and accurate feedback on the scenario. That way, he can easily check if the scenario will play out according to his prediction.
In this approach, we've chosen to let the game developer decide on how to abstract the contents of the gameplay scripting level to the level of scenario scripting, just as the programmer decides how to abstract his code into the gameplay scripting level. With this approach, the game developer is responsible for creating the content that can be used by the instructor. In that sense, it is up to the game developer to choose what information, or meta-data, is supplied to the instructor, and thus at which level of abstraction the instructor operates. Therefore, there is no exact, strict, definition of the boundary between the scenario scripting level and the gameplay scripting level, so it can be decided on a case-by-case basis.

4.1.3 Answering sub question Q1

In Section 1.2, the following sub question was posed, as a smaller, more detailed question that will help in finding an answer to the main research question.

How can the scenario of a simulation game be represented at a more abstract level of scripting?

The answer to this question, resulting from what has been discussed in this Section, is that we can represent the scenario of a simulation game at the more abstract level of scenario scripting, by combining scripts at the gameplay scripting level into scenario building blocks, which can then be used as verbs in the language of scenario scripting. The following Section will discuss what exactly the composition and content of these building blocks should be, and how the game developer can decide how to abstract scripts from the gameplay scripting level to the scenario scripting level.

.: 4.2 Building Blocks

In this Section, we identify the verbs of the scenario scripting language, called building blocks, which the instructor should have at his disposal at the scenario scripting level. These building blocks together are thought to be a complete toolset with which the instructor has the means to fully express his didactic desires. Thus, it provides an answer to the second sub question posed in Section 1.2.

The previous Section showed that sections of gameplay script can together form scenario building blocks. But what forms can such a building block take? We start by identifying two verbs that are the most basic building blocks of a scenario script; actions and events. An action is something that is performed by the player, while an event is something that is performed by the game. Examples of actions are ‘player opened the door’, ‘player killed the monster’, etc. Examples of events are ‘close all doors’, ‘monster enters the room’, etc. Together, these two building blocks allow an instructor to script a basic scenario, for instance in the form of ‘event A follows event B, which is executed when action C has been performed, which in turn can be performed when event D has finished’.

However, with just these two verbs, the instructor is fairly limited in his expression of a scenario, as he can only create linear scenarios. In order to create non-linear scenarios, the instructor also needs to be able to apply some logic at the level of scenario scripting,
in the sense of 'if action A is performed, then execute event B, otherwise execute event C'. Therefore, logical verbs are introduced to the vocabulary as well.

Additionally, in a session with serious game development company e-semble (of which the details can be found in Appendix B), it was identified that in some types of games, the instructor might want more detailed control over a scenario by using states. An example of basing the logic of scenarios on states in a simulation game is in the domain of surgery. Here, the state of the patient (i.e. his blood pressure, heart rate, etc) is an important factor in making logical decisions at the level of the scenario.

In the same session with e-semble, the use of variables was discussed. On the level of scenario scripting, a variable is a verb that can be used for logical decisions. The use of variables widens the options an instructor has for creating non-linearity in his scripts. Whereas without using variables, the instructor can only base the logic in his script on whether or not an action has or has not been performed, with the use of variables the instructor can also write scripts that base their logic on how well an action was performed.

Looking back at the definition of a scenario proposed in Section 2.3, we see that a scenario is the global progression of a level in a game, defined by its initial settings and the logical flow of events and actions that follow. Up until now, we have discussed how an instructor has control over the logical flow of action and events, but with the current set of verbs, he does not yet have control over a scenario's initial settings.

For this, the final verb settings is introduced. These settings include the scenario relevant properties of a game, such as the number of pedestrians in a driving simulation game, or the health status of a patient in a surgical simulation game. In some games, such as in Supervisor, these settings do not play an important role, but in other games they are very relevant to the instructor and are as such included in the vocabulary of scenario scripting.

In Section 1.2, the following sub question was posed, as a second small, detailed question that will build on the first sub question and help in finding an answer to the main research question.

How can an abstracted scenario be made editable to an instructor?

The answer to this question, resulting from what has been discussed in this Section, is to let the instructor (re-)arrange events and actions, with the help of logic, states and variables, and to allow him to edit certain game settings. These verbs together are called building blocks. The following Section will discuss how the arrangement and properties of these building blocks can be communicated to a game engine, so the scenario scripted by the instructor can be executed within a game.

.: 4.3 Communication with simulation games

In this Section, we discuss how the arrangement and properties of the building blocks defined in the previous Section can be applied in a simulation game. As we will see, this is done by utilising an event-based communication protocol between a standalone
scenario authoring application and the simulation game. This protocol allows real-time adjustments of the scenario, i.e. while the game is being played, which maximises the control the instructor has over the scenario of the simulation game. Thus, it provides an answer to the third sub question posed in Section 1.2.

4.3.1 Game-independent communication

The actual process of scripting a scenario can potentially happen in two locations; within a game development environment (as is the case in gameplay scripting), or outside of the game development environment, by utilising a standalone scenario authoring application. This approach is based on offering a standalone application. That way, scripting a scenario is independent of the specific game in which it will be used, so the instructor can learn how to write scripts for one game and apply his knowledge to other games as well.

The communication part of the scenario application will be separated from the rest of the application to fully realise this goal of game-independence. Please note that this makes the entire system only partially independent of the specific game, as of course the specific building blocks used in any scenario arise from specific games. However, the knowledge on how to operate the authoring application will be independent of the type of game. The next Section will discuss how the communication between game and authoring application will take place.

4.3.2 Event-based communication

Considering that scenario building blocks are created from sections of gameplay scripts, we have chosen to apply an event-based communication system between the scenario authoring application and the simulation game. This system sends messages back and forth when a building block in the form of an action of event needs to be executed. The game is then responsible for handling all the scripts at the gameplay scripting and the programming level, while the scenario authoring application is responsible for handling the scripts at the scenario scripting level, including the handling of scenario logic, states and variables.

Using this system of sending messages between the editor and the game, the communication happens in real-time. This allows the instructor to make adjustments to the scenario while the game is being played, as long as this does not corrupt the scenario. Let's say for instance that the instructor has prepared an elaborate scenario in which the trainee will deal with a variety of situations. However, the instructor notices during play that the trainee is having difficulties with one of the situations and is taking longer than expected. The instructor can now decide to re-route the causality sequence somewhere ahead in the scenario, to skip one of the situations. Or, he can decide to change a specific variable, which will make the following sequence a bit less stressful for the trainee.

Alternatively, if the instructor doesn’t wish to alter the course of a scenario during a play session, he is also free to alter the scenario at another time. This could for instance be very appropriate when the instructor has spotted a defect in the scenario, or has seen that multiple trainees are having difficulty with a specific section. The instructor can
then easily alter and save the scenario, thus improving the learning experience in future play sessions. Also, the instructor can prepare multiple scenarios, all tweaked to provide an interesting experience to trainees with varying levels of skill.

4.3.3 Answering sub question Q3

In Section 1.2, the following sub question was posed, as a third question that would build on the first two sub questions and help in finding an answer to the main research question.

**How can the adjustments that have been made to the scenario be communicated to a simulation game?**

The answer to this question, resulting from what has been discussed in this Section, is by using an event-based communication system. The result is that the communication can happen real-time, while the game is being played, allowing more flexibility for the instructor to make adjustments during a training session. Also, by making the communication independent from the game, authoring a scenario is made independent of the type of game in which the scenario will be played. This has the advantage that instructors can adapt to different types of games, without having to re-learn how to author a scenario for that game.

Now that all three sub questions are answered, we can answer the main research question:

**How can an instructor be given more control over the scenario of a simulation game?**

The answer discussed in this Chapter is to let the instructor exercise control over the scenario by writing a script at the abstracted level of scenario scripting, using the language of scenario building blocks. More control is given to the instructor by allowing him to make real-time adjustments to the scenario as well.

To evaluate this approach, a prototype scenario authoring application will be developed. This prototype will then be evaluated to assess how well this approach actually works in practice. The following Chapters will discuss the design, implementation and evaluation of that prototype.
5. Prototype Design

This Chapter discusses the key elements for the design of the prototype. The approach taken in designing the prototype is a top-down approach that starts with defining a goal tree, based on the project’s main goal. The goal tree is then used as a basis to define the requirements and features that will be implemented in the prototype. The requirements are discussed in Section 5.2, while the features are discussed in detail in Section 5.3.

: 5.1 Goals

The overarching goal of this project, which is to give instructors more control over scenarios, can be broken down into sub-goals, which subsequently can also be broken down into more sub-goals, eventually leading to concrete requirements and desired features. Fig. 12 depicts the top part of the goal-tree that has been developed for this project.

The goal tree was developed on the basis of the approach discussed in Chapter 4, which discusses the decision to make the application in which a scenario can be scripted independent from any specific game engine. Thus, the main goal is separated into two sub-goals: to create a standalone scenario scripting application (a scenario editor) and to implement support for this editor in a simulation game.

For the scenario editor, the approach proposed a solution that lets the instructor exercise control over the scenario by writing a script at the abstracted level of scenario scripting, using the language of scenario building blocks. More control is given to the instructor by allowing him to make real-time adjustments to the scenario as well, using an event-based communication.

Thus, the approach is translated into the following three requirements for a scenario editor:

- Provide scenario scripting functionality, using building blocks
- Allow for real-time adjustments of a scenario
- Implement an event-based communication with a simulation game

These requirements will be discussed in more detail in the next Section. To fulfil these requirements, a list of features has been defined. These can be found as sub-goals under the requirements, and will be discussed in more detail in the third Section of this Chapter.
5.2 Requirements

The previous Section presented a goal tree, shown in Fig. 12, from which a list of requirements is distilled. These requirements are functional requirements, in the sense that they discuss what the prototype must do. This Section will discuss these requirements in more detail, exploring how they result into a list of features. Also, a new, non-functional requirement will be added to the list, more related to what the prototype must have.

R1. The prototype must provide the user with scenario scripting functionality, using building blocks

The scenario scripting functionality that the prototype will provide is the functionality discussed in the approach in Chapter 4, which is to let the instructor (re-)arrange events and actions, with the help of logic, states and variables, and to allow him to edit certain game settings. As was discussed in the literature survey in Chapter 3, a graph representation is a suitable metaphor for scripting a scenario. Thus, the previously mentioned building blocks will be featured as nodes in a graph. Subsection 5.3.1 discusses the various features that are developed to fulfil this requirement.

R2. The prototype must allow the instructor to make in-game adjustments to a running scenario

Allowing the instructor to make adjustments to a scenario while it is being executed by a game gives the instructor more control over the scenario, as was discussed in the approach in Chapter 4. As the application is a standalone application (thus running separately from the game), allowing scenario editing independently of whether or not a game is running, is fairly straightforward. Using the event-based communication, the adjustments can also easily be replicated to the game. An important condition when adjusting a scenario during play is that the scenario must remain consistent. This is facilitated by implementing a check for the validity of a scenario every time an adjustment is made. The implementation features of this requirement are discussed in Subsection 5.3.2.
R3. The prototype must implement event-based communication with a game engine

The approach is designed to create a generic editor, which can be used in a wide variety of games and game types. Therefore, it is important that the editor can communicate with any type of game engine, be it a highly advanced 3D engine such as the Unreal Engine or a 2D engine written in Java, such as being used by SimPort. For the purpose of this research, the prototype will only support the communication with a single game engine (specifically the Unreal3 Engine), because the goal of this research is to provide a proof of concept, not to provide a fully functional application. The prototype will, however, have to be written in such a manner that extending it to other game engines is as easy as possible.

R4. The prototype must be usable by an instructor

Finally, as the scenario editor will be used primarily by instructors, and these users typically do not have a lot of experience working with complex applications on computers, it is imperative that the scenario editor has an easy to understand interface, allowing the instructors to focus on developing scenarios, as opposed to struggling with the application itself. This is an important requirement, and although it does not result into a specific list of features, it is tested for in the evaluation in Chapter 7.

5.3 Features

This Section discusses the various features of the prototype, as they are designed to support the requirements mentioned in the previous Section. They have been distilled from the goal tree in Section 5.1 and will be discussed using that framework.

5.3.1 Scenario scripting functionality

Fig. 13, on the following page, shows the section of the goal tree beneath the first requirement. The features required for scripting a scenario are quite extensive, and are each discussed in this Section.

Create New, Save & Load a Scenario
Naturally, the prototype needs functionality to support the creation of a new scenario, loading an existing one and saving the current scenario, to a file on the user’s storage device.

Run a Scenario
The user can easily initiate a game session from the editor. Doing so causes a communications server and the specified game to be started up. The prototype then starts the scripted scenario. A so called ‘control flow’ then passes through the nodes in the scenario graph, triggering each node as it passes. How the control flow is handled by each type of node will be discussed further on in this Section.
Scenario building blocks

The sequence of actions and events that partly define the scenario will be represented by a graph. This has been identified as a suitable visual representation of a scenario in the literature survey in Chapter 3. Now, we shall discuss the several components of the scenario graph.

Node Graph

The graph editor is the main component of the application. Here, the user can place nodes and connect them, creating a graph-like structure. Each node in this graph can be connected to multiple other nodes. During play, the graph is traversed in a thread-like manner.

A node in the graph is triggered when a thread reaches that node. From there, the outcome of the node determines which path in the graph the thread will follow, and thus which node will be triggered next. A thread can also be split by a node, to go on and
simultaneously trigger two or more nodes at once. There are several types of nodes the user has to his disposal:

An **event node** is an abstracted representation of a predefined, scripted, gameplay event. An example from the supervisor game would be 'The painter runs for shelter'. An event is triggered when it is reached by a thread. When the event has finished executing (in the previous example, this would be when the painter has reached the shelter), the thread continues to trigger the next node in the graph.

Technically, all that an event does is send a signal to the game that a specified set of actions needs to be executed. The thread then waits until the actions have finished executing before it goes on to trigger the next node.

![Event Node](image)

*Figure 14: An event node*

In Fig. 14, the thread reaches the node on the left side. When this happens, the editor sends a message to the game saying the game should perform the specified event. Then, when the game has finished performing the event, a message is sent back and the thread continues to the next node in the graph, which is connected to the right side of this node.

An **action node** is a node that waits for input from the player. In the supervisor game, an example of an action would be 'User repairs scaffolding'. An action becomes available when it is reached by a thread. The thread then waits until the user has performed the requested action, upon which the next connected node is triggered.

Technically, when the scenario reaches an action node, execution of further events pause until the player has performed the defined action. A signal is then sent from the game to the scenario script to continue the thread. When an action node is triggered by the game before the script is ready for it, i.e. before it is reached by a thread, the next connected node will not be triggered.

Also, when an action is performed, the game can choose to append a value, which can be an indication of how well it was performed, or how long it took the player to perform it, etc. This value can then be used later on in the scenario script to determine the scenario’s path.

![Action Node](image)

*Figure 15: An action node*

In Fig. 15, the thread reaches the node on the left side. When this happens, the scenario goes into waiting mode, waiting for a message from the game, saying the action was
performed. When the scenario receives this message, the thread continues and the next connected node is triggered. Also, if an additional parameter was provided in the message, the value of a variable node that might be connected to the bottom connection is set to the parameter's value. Variable nodes will be discussed further on in this Section.

A timer event is triggered when the thread has reached the node. A timer event is relative to a time label and is defined by the time it waits until it continues the thread. So, for instance, when the user wants an event to be triggered 5 minutes after another event has been triggered, he can use a timer event in between these events to implement a 5 minute delay.

![Figure 16: A time event node](image)

A time label is set by a thread and can be used to mark a certain point in time. Using this node, other events can be triggered based on global and relative time.

![Figure 17: A time label node](image)

For the scenario scripter to be able to write some branching scenarios, logic nodes will be implemented as well. These nodes will function as 'if-else' statements, logic gates and variable comparison nodes. Details on the implementation of these nodes and how they will work can be found in the next Chapter.

![Figure 18: A logical if node](image)  ![Figure 19: A logical gate node](image)  ![Figure 20: A logical comparison node](image)

Preconditions & Precondition Checking
A node can also have preconditions. These are events or actions that need to have been triggered before the current node can be triggered. The scenario will check the preconditions of a node when it is placed by the user, thus avoiding any possible inconsistencies early on, during the scenario design process. Preconditions can be in the
form of nodes that need to have been triggered, nodes that should not have been triggered or a combination (either via AND or OR) of both. Consequently, the preconditions for a node need to be defined by the game developer, as he is the one with the detailed technical and gameplay knowledge required for setting such a precondition.

When a node is placed in a scenario, the editor should check the scenario for inconsistencies. This includes checking the preconditions of nodes, but also if one and only one Start Game node exists and one and only one End Game node exists.

**Node Properties**

Nodes can have specific properties, which can be edited in the property window. An example of a property is the 'time' property of the timer event node.

**States & State Transitions**

Some games require the use of states at a high level of the game logic. Examples are an operation simulation game, or a SimCity-like game. In the case of an operation game, a scenario might be defined as ‘if the patient’s heart rate falls below a certain threshold, different events and actions will occur than when the heart rate stays above that threshold’. In that case, using state logic for the scenario can be very helpful in giving the user an intuitive overview for defining a scenario.

To implement state transitions, two additional nodes are featured; one that handles the switching between states, and one that can check in which state the game currently is.

![Figure 21: A switch state node](image)

![Figure 22: A check state node](image)

**Variables**

As mentioned earlier with regard to action nodes, variables can also be used in scripting a scenario. These variables can be related to the game world (such as the heart rate in the aforementioned example), or they can be more abstract. Suppose for example that the player needs to place a safety ribbon in a certain situation. The variable that will be used to apply some logic to this action can be for instance the length of the placed ribbon, or its location. However, it could also be a meta-value indicating how well the player placed the safety ribbon. The precise value of this variable can be determined by the game, using rules defined by the game developer, but the scenario instructor doesn’t need to know about this. This way, the scenario instructor can still operate at a high level of abstraction, if this is deemed necessary. We devised two ways of using variables; as part of nodes, or as standalone variables.

When variables are used as part of nodes, they help decide the outcome of a node. The decision which path should be taken by the thread is made by evaluating the return variable. For example; an action ‘BarrelMoved’ is triggered when the trainee has moved an (incorrectly placed) barrel full of chemicals. It could be interesting to the instructor to
know how long it took before the trainee moved the barrel, as this could be used in the trainee's evaluation. So, along with the message that the action was performed, a variable is sent by the game, indicating how long it took. Say for instance that the action Node 'BarrelMoved' is has several connections to other nodes, with the connections being labelled as 'less than 1 minute', 'less than 5 minutes' and 'more than 10 minutes'. Then, when the message from the game is received, the node evaluates the variable that has been sent along and triggers the node connected to the correct connection. Of course, the variable does not necessarily have to be a numerical value. Other values, such as strings or vectors could also be evaluated this way.

Alternatively, variables could be used outside of nodes. Here, the variable that is returned by the game is stored in a variable node. The variable can then be used or evaluated by other nodes. The situation described above would then require an extra node that handles the evaluation of the variable.

The approach that uses standalone variables result in a more complex scenario, as extra nodes need to be placed to perform the evaluation, as well as for the variables themselves that will take up space in the scenario. However, that approach also allows more flexibility, since the evaluation of a variable does not explicitly have to be performed at the node that received the value from the game. Thus, the variable can be used at another point in the scenario as well, which allows the instructor to create richer scenarios, and thus gives him more control. Therefore, the decision has been made to implement the second method; standalone variables.

![Figure 23: A variable node](image)

**Game Settings**

Game Settings can be used to set some initial values which can be important for certain types of games. Initial value can be values such as the number of pedestrians in an area, or certain threshold values, or any other relevant value. These settings can also be adjusted during play. As such, settings work similarly to variables, as both represent values that are communicated between the editor and the game. However, the difference between settings and variables are that settings represent actual values in the game world (such as the number of pedestrians) while variables are only used in the scenario scripting level, and as such only represent a meta-value (such as 'how well an action was performed').
Import Building Blocks Data

Ultimately, it should be possible for the editor and the game to exchange other information, besides events. For example, the introduction text should be made editable to the instructor, so he can adjust the text according to the scenario he has defined. The game should then render the text on the screen. For this to happen, the editor and the game must be able to exchange this textual information.

The list of event and nodes that are available to be used in a game is defined by that game. Therefore, it is the game developer’s responsibility to create this list. Of course, such a list can be generated automatically, as opposed to manually, by the game engine. The list is then exported and imported into the editor, creating its library.

The game developer can export the scenario building blocks to an intermediate file, such as an XML file, which can be read by the authoring application, as can be seen in Fig. 24.

![Diagram](image.png)

*Figure 24: Scenario Building Blocks are exported by the game engine and can be read by the instructor.*

The instructor then imports the scenario building blocks into the authoring application. The library will be filled with blocks that can be used to construct the scenario sequence, and the options for game settings are filled in.

Scenario Error Checking

When a node is placed, a link is made or any other changes are made to a scenario, the system will automatically check the validity of the resulting scenario, using the preconditions specified for each node. This will ensure the scenario instructor doesn’t build any inconsistencies in the scenario, such as ordering two objects to move to the same location or ordering one object to move to two different locations, in other words; giving inconsistent orders. When an error in the scenario’s consistency is detected, visual feedback will be given. This method ensures that the scenario is not set up in a way that is not intended by the game developer. However, it should be noted that it is the game developer’s responsibility to properly define each node’s preconditions and ensure that the set of preconditions provided to the instructor is sufficient to prevent scenario inconsistencies.
Zooming, Linking and Other Graph Operations
Finally, some graph operations will also be available, such as zooming in and out, linking nodes to each other to actually form a graph and perhaps managing a graph into subgraphs or collapsing Sections of a graph. Also, of course, basic functionality will be available such as moving nodes and the scenario around, placing new nodes and deleting them.

Feedback
An important aspect of running a scenario to teach a trainee something is getting feedback on how the trainee is performing. Therefore, the system should support some kind of option that allows the instructor to monitor the actions the trainees are performing real-time, preferably in the same environment as the player is operating in. Most likely, however, high quality feedback will not be featured in the current prototype, due to this being mostly beyond the scope of the research question. Perhaps it can be featured in a future, more extensive, version of the application.

The goal of providing feedback is to let the instructor have an indication of the actions a trainee is performing. Feedback can be incorporated in various ways.

I. The instructor can be shown the exact viewport the player is seeing. This way, the instructor can see everything the player is seeing, and can thus directly monitor the player's thought and movement processes. This is probably the most optimal form of monitoring. However, it becomes increasingly more difficult to keep an overview when multiple players are involved.

II. The instructor can move around and operate in the same world as the player. This way, the instructor can view the movement of the player and at the same time see parts of the environment that the player may not see. This is a better option when multiple players are involved.

III. The instructor does not see the game world, but he gets some form of status report on the player’s actions. For instance, he gets a message on his screen detailing when a player has reached a certain goal. This form of monitoring is weaker than the other two methods, since the instructor receives minimal feedback on the player’s progress.

Although the third method of monitor is considered to be the weakest, the scenario editor will feature this method. This is mostly due to the technical implications of enabling one of the other options. Implementing method I or II requires implementing some form of 3D viewport, or perhaps streaming the player's viewport to the instructors monitor. The implementation of this type of monitoring is considered to be beyond the scope of this research. Therefore, it is opted to solely provide the third method of monitoring, while it is recommended for future extensions of the application to implement a better form of feedback.
5.3.2 Real-time Adjustments

Fig. 26, pictured below, shows the section of the goal tree beneath the second and third requirements. The features required for implementing these requirements are discussed in this Section and Section 5.3.4.

![Goal tree; requirements R2 and R3 translated into features](image)

**Triggering Nodes on-the-fly**
Nodes can also be triggered manually during a play session. This gives the instructor the freedom to make last minute decisions, altering the course of a scenario, based on the current performance of a trainee.

**Editing a Scenario during play**
Also, to give complete freedom to the instructor, he can edit a scenario while it is being played. This gives him the ability to make any changes when he seems fit.

5.3.3 Communication with Game Engine

The editor application needs to be able to communicate with an external game. This is facilitated through the communicator application, which acts as a two way communications channel.

Both the editor and the game engine that is selected for this prototype implement functionality to communicate with each other. The editor can handle input and send output, as well as the game engine.

5.4 Simulation Game

Aside from developing a prototype scenario editor, a simulation game needs to be developed that can communicate with the scenario editor. This game can then be used to test the editor. It would be beyond the scope of this research however, to develop a full game from scratch. Luckily, the environment in which this research is performed provided some games which can be re-designed to cooperate with the scenario editor. The partial goal tree showed in Fig. 26 provides some goals for the re-development of such a simulation game.
**Scenario editor support in a simulation game**

The simulation game that has been chosen to work as a test subject for the scenario editor will be the Shell Supervisor game. The second version of this game is being developed while the scenario editor is being developed. It provides an excellent setup for the conversion of gameplay sections to scenario elements.

**Scenario script support**

For a scenario editor to work, the game must be split up into building blocks, which can be exported to the scenario editor. Thus, splitting up an existing game into scenario building blocks is an important and challenging subject.

**Communication with scenario editor**

Finally, the game must successfully communicate with the scenario editor, as also stated in the requirements Section.
6. Implementation

This Chapter discusses the implementation details of three systems, as the prototype application has been divided into two subsystems; an editor and a communicator. The third system is the test simulation game, which is a variation of Supervisor. The editor, named Shai after the Egyptian god of fate (pronounced as “shy”), is the instructor’s application, in which the creation and scripting of a scenario is performed. The communicator is the application that handles the communication between Shai and the simulation game. This communicator was developed separately from the editor application, as this will improve the possibility to extend the prototype to also be able to communicate with other game engines, without the need to alter the instructor’s application. In the case of this prototype, the game is implemented in the Unreal 3 Engine (UE3), as was the original Supervisor.

This Chapter starts by discussing the implementation of the various features of the editor application, how to set up and run a scenario, and provides a quick insight in the data model that is used to represent a scenario. The implementation of the communicator application is briefly discussed in Section 6.2, while Section 6.3 discusses the implementation of the Supervisor test game.

.: 6.1 Shai

Shai has been developed using Visual Studio .NET in C#. This language was chosen at it is easy to develop windows applications with. Shai is not a technically complicated application; therefore it would be wise to choose a language that is easy to pick up, in order for Shai to be easily extended by future developers. This Section discusses the implementation of the features discussed in Chapter 5.

6.1.1 GUI

As can be seen in Fig. 27 on the following page, the editor is divided into five panels. This Section discusses each of these panels separately.
Library
The library panel on the left consists of a tree-view depicting a list of available nodes. The nodes in the library can be dragged and dropped onto the large Scenario panel on the right. Nodes in the tree are ordered by their type; event nodes, action nodes, logic nodes, time-based nodes and miscellaneous nodes. Event and Action nodes are specific to a certain game, while the other nodes can be applied in any game.

Game Settings
In this panel on the bottom left, some game specific settings can be set, such as how many pedestrians there are in a certain area (in the case of a traffic game), or what the weather is like. These settings can be used as initial settings, but can also be adjusted during play.

States
In this panel, game states can be defined. A state is used to describe the current situation in the game, and can be used for logic by using a ‘switch state’ and a ‘check state’ node. An example of using states is defining scenario logic based on whether or not the game is in the ‘begin’ state, or in the ‘middle’ state. States have been implemented as they were described in Chapter 5.

Scenario
This is the main panel of the editor. Nodes can be dragged and dropped in here from the library. Also, nodes can be linked together to form a scenario. Notice how the panel is split into two sections. The upper section, with a ruler, is used for time-based nodes, while the lower section is used for all the other nodes. The separation of these two sections is made to draw the instructor’s attention to the distinction between time-based nodes and other nodes. Also, the nodes in the ruler section are static in location, while the other nodes can be moved around freely. This allows the instructor to always...
have an overview of the timeline. How to properly use the scenario panel will be described in the next Section.

**Properties**
This panel allows the adjustment the properties of the currently selected node or state. For instance, in the properties of an event node ‘Assistant: move to gas tank’, a property that can be edited is the walking speed of that assistant. Similarly, the properties of a ‘switch state’ node include the name of the state to which to switch.

**6.1.2 Setting up a scenario**
This Section describes the various options available with which to build a scenario. The instructor has the ability to add nodes, move them around and link them together to form a graph.

**Adding nodes**
Nodes can be added by dragging them from the library onto the scenario.

**Linking nodes**
Nodes can be linked together by dragging a connection from the plug on one node to the plug on another. There are two types of plugs; input plugs and output plugs. Input plugs can only be linked to output plugs, and vice versa.

**Moving around**
A node can be moved around by clicking on it with the left mouse button, holding it and moving it around. Similarly, using the right mouse button moves the entire scenario around. Zooming in and out can be done by scrolling the mouse wheel.

**Node types**
There are several types of nodes at the instructor’s disposal. These nodes have been implemented as they were discussed in the design in Chapter 5. This Section specifically discusses the several logic nodes, time-based nodes, and the use of the variable node.

There are several logic nodes that can be used to influence the progression of a scenario. First, there are the logical gate and the logical if nodes. These nodes fire one of their outputs, depending on which inputs has reached them. The gate node fires the top output (“All”) when all nodes connected to its input have been fired. The bottom output (“Any”) is already fired when only one connected node has been fired. The if node fires its top output (“If”) when the bottom input has been reached before the left input. Otherwise, it will fire the bottom (“Else”) output. The following illustrations will clarify these concepts.
In the example in Fig. 28, when one of the actions (for instance ‘action A’) has finished, but one of the others hasn’t (either ‘action B’ or ‘action C’), event Y will be triggered. However, if all connected actions (A, B and C) have finished, event X will trigger as well.

Fig. 29, above, illustrates the logical if node. In this example, when event A has finished executing, the if node checks whether or not action B has also finished. If so, event X is triggered. Otherwise, event Y is triggered.

Then, there are two logic nodes that deal with states. These are the check state and switch state nodes, responsible for checking if the game is in a certain state, and for putting it in a state, respectively. The current state a game is in is highlighted by a thick white line. Finally, the logical compare node compares two variables to see if they are equal, larger or smaller.
In some scenarios, it would be nice to have some control over the timing. For this, **time-based nodes** can be used. A **label node** can be used to mark a specific moment in time. A label node can be combined with a **time event node**, to create time-based scenarios. Fig. 30 illustrates this concept with an example.

**Figure 30: Using time-based nodes**

In the example of Fig. 30, a label node is used to mark the moment at which the player fixed the scaffold. Then, two time event nodes are used. First, one minute after the scaffold was fixed, the painter is ordered to walk to it. Then, three minutes after the scaffold was fixed, the game ends.

Finally, a **variable node** is used to store a value so it can be used later on in the scenario. The variable's value can be set by the instructor (in the properties panel) or it can be set by the game. In the latter case, the value of the variable is determined by the action node to which it is connected. See Fig. 31 for an example.

**Figure 31: Using variables and a compare node**

In Fig. 31, two variables are used. The value of the first variable is determined by the action node, and could for instance be ‘the time it took the player to perform the action’ or any other numerical value, such as ‘the distance between two objects placed by the action’. Then, a logical compare node is used to see if this value is above or below a certain threshold, defined by another variable. In this case, the instructor has manually set the other variable to 3.
6.1.3 Running a scenario

Starting the playback of a scenario is straightforward; by simply pressing the play button, the application boots up the game and when the booting up process has finished, the scenario begins at the ‘start game’ node. Alternatively, the game can be booted manually, and the scenario can be set into play by right-clicking on any node and selecting the option ‘trigger this node now’.

While a scenario is being played, the instructor is free to move nodes around, add or remove nodes, rearrange nodes and perform all other operations mentioned in Section 6.1.2, as long as this does not invalidate the scenario. The command ‘trigger this node now’ also works, to force a node to trigger in real-time.

6.1.4 Scenario Data Model

This section discusses the data model that is used to represent scenarios in Shai, shown in Fig. 32.

The content of a scenario, which basically is a collection of nodes, is stored in the Scenario class. A ScenarioNode itself has a collection of input- and output Plugs, which
are used as an implicit definition of the link between nodes, since each plug has a reference to other plugs it is connected to. So, the scenario graph and the links within it are defined by a list of nodes, and the connections are monitored by the node’s plugs. Naturally, the ScenarioNode class has many children, such as the ActionNode and EventNode, but these were omitted from this illustration due to the space these nodes would take up. A ScenarioNode also has a Description property, that is used to provide the user with extra info on the node’s functionality.

Besides the scenario class, there is also the StateDiagram class that works as a container for the collection of States. Both states and nodes are children of the GenericNode class, which is defined by an identification number, and a name. The identification number is used for the communication with the game engine, as well as for saving and loading form a save file, while the name of a GenericNode is only used for the convenience of the user. The GenericNode in turn, along with the Scenario and StateDiagram classes, are children of the ShaiElement class, which provides functionality for user interactions, such as selecting and dragging. With this setup, not only nodes within a scenario can be dragged, but the entire scenario can be moved around as well.

.: 6.2 Communicator

Fig. 33 shows the communication between Shai and the Supervisor game. A message sent from Shai to the game is first handled by the communicator. The communicator can read this message and decide what to do with it. For instance, he can simply forward the message to any or all connected game engines, or he can write a debug message, forward it to a database, etc. The communicator can also receive messages from the game engine, which he can forward to Shai. The communicator can handle multiple connections, both from Shai and game engines. Also, if necessary, multiple types of game engines can be connected, at the same time, to the communicator, i.e. not just the UE3 engine, but also any other type of game engine.

By separating the communicator and the instructor’s application, the instructor’s application is shielded off from the game engine. This makes the functionality of Shai independent from the type of game engine used. Therefore, when future iterations of the prototype are developed, the developer doesn’t have to alter Shai’s code. He only needs to write specific communication for the new game engine, and Shai will be ready to go.

![Figure 33: Shai - Game Communication](image-url)
The communicator is a simple single class application, written in Visual Studio and C#, just as Shai. The communicator acts as a TCP/IP server, listening to clients in the form of Shai and game engine(s). The communicator can currently receive two types of messages, one designated to be forwarded to the other clients, and one designated for the server itself.

The protocol used for these messages is one that was custom designed for this system, using packets of data. A packet is constructed using an indicator of its length, an indicator of the packet’s type and an array of data. Additionally, a packet is prefixed with a timestamp indicating when a packet was send. Five types of packets are defined; debug, event, action, property and setting. Together, these packet types cover the various types of messages being sent between the editor and the game. The data array can hold several values; the first value is always the id of the node to or from which the message is sent. The other values are optional and can hold information such as the properties of an event node. This protocol does not follow any official guidelines on protocol definitions, since it was not considered necessary for a prototype that focuses on evaluating a theory to do so.

Potentially, the communicator could offer features such as logging to a database, using other types of messages as well. However, such features are beyond the scope of this research, as they are not relevant to providing a proof of concept for answering the research question.

6.3 Supervisor

Before Shai can be used to exercise influence over the scenario of a game, the game needs to be set up properly by the game developer. This Section will discuss the steps taken while adjusting Supervisor for use by Shai, thus working as a guideline for setting up games in UE3 for use with Shai. Setting up a simulation game in another game engine will of course be different, but the general approach will be the same. This Section is therefore intended to be used as reference for future developers.

Implementing scenario building blocks in UE3 requires two steps. First, a programmer needs to write code for Kismet nodes, so they can be used in the second step by the gameplay scripter.

6.3.1 UnrealScript

To provide communication between the game and Shai, four nodes were written for UE3’s gameplay scripting language Kismet; two SequenceEvents and two SequenceActions. These communication nodes need to be available to the game developer at the gameplay scripting level, so he define the moments in time at which communication needs to occur, at his level of operation.

In Kismet, a SequenceEvent is used when an action is performed at the programming level, that needs to be communicated to the gameplay scripting level. An example is when the player has touched a trigger. A SequenceAction is used when an action is performed at the gameplay scripting level, that needs to be communicated to the programming level, such as setting a certain variable.
The first SequenceEvent that was implemented is used to indicate that Shai has sent a message containing an event and the second SequenceEvent is used to indicate that Shai has sent a message containing a game setting. The SequenceActions are used to send messages to Shai detailing that either an action was performed, or that an event was finished.

Besides the Kismet nodes, a class was written at the programming level, handling the communication with the communicator. This class is responsible for:

- Setting up a TCP Link
- Connecting to the communicator using that TCP Link
- Sending and receiving messages, using the protocol discussed in Section 6.2
- Handling received messages and forwarding them to the Kismet nodes (SequenceEvents)

6.3.2 Kismet

When the programmer has finished writing the code for the Kismet nodes, the nodes can be placed in Kismet. This is done by the gameplay scripter. He will decide at which points in the game's logic messages will be sent back and forth between Shai and the game. Following are examples of where these nodes were placed in Supervisor.

![Figure 34: Example of the use of 'Trigger Shai Action' Kismet node](image)

In the example in Fig. 34, an action is connected to an Unreal event which is triggered when the player talks to a certain NPC. The 'Trigger Shai Action' node sends a message to Shai.
In Fig. 35, the sequence starts when a message is received from Shai, conveying that the event ‘Painter move to garage’ needs to be executed. Additionally, a parameter is provided defining the walking speed of the painter. The two following nodes represent the implementation of the event in UE3. Finally, when the painter has reached the garage, the game sends a message back to Shai using the ‘Finish Shai Event’ node, saying it has finished executing the event.

The final type of node available to the Kismet designer is the ‘Shai setting’ node. This node is triggered when a message is sent from Shai to the game holding a setting value. This value is then interpreted by the game and handled according to the wishes of the Kismet designer. An example of its use is shown in Fig. 36.

All in all, the extra work required to make UE3 function with Shai is relatively little. On the programming side, the handful of classes that need to be written, and their content, is trivial. On the Kismet side however, placing the extra nodes can be a bit tiresome. However, when this is done while the level is being created (as opposed to afterwards, as was the case in developing this prototype), the extra work is little. Plus, it helps the designer in keeping the Kismet sequences organized, and guides him into using modular design, which is always beneficial.
7. Prototype Evaluation

One of the requirements defined in Chapter 5 is that the application needs to be usable by instructors. To fulfil this requirement, the prototype needs to be evaluated by these intended users. For this purpose, an evaluation plan was developed and executed featuring a tutorial for users to follow and a questionnaire to fill in. This Chapter discusses the evaluation plan and the results of the evaluation.

.: 7.1 Evaluation Plan

The evaluation was performed using a combination of several evaluation methods. Mainly, the user was asked to fulfil a tutorial, in which he was asked to perform several tasks using the prototype. This method had the advantage of discussing all the features of the application, while it didn’t require a completely functional prototype (i.e. it was okay for some features to not work 100%). However, the disadvantage of this method was that the user may not behave as he would in real life. Secondly, the user was interviewed, using a group discussion as well as a questionnaire, to find out about his experience with the prototype. The advantage with this method was that it got more detailed results and produced useful ideas for improvement. However, the disadvantage with this method was that the results tended to be somewhat unstructured (depending on what was and was not discussed in the interview) and thus had low validity.

The combination of these two methods was chosen as the advantages of these methods are well suited to the state in which the project was at the time of evaluation; a prototype was developed, but not feature-complete, and ideas for improvement were welcome. However, it should be noted that the results of this evaluation should be considered with respect to the disadvantages embedded in the evaluation methods, mentioned above.

7.1.1 Scope

The prototype, implemented as discussed in Chapter 6, was tested by several field experts, including representatives from Shell, who are familiar with the Supervisor game, and serious game experts from e-Semble, who are familiar with scenario editing software. The evaluation session took half an hour to an hour per user.

Goals
The goal of this session was to evaluate:

- Feature completeness
- Adequacy of scenario representation
- Usability

7.1.2 Process

Test Setup
To make an adequate simulation of how Shai will be used in the field, the user was presented with the following setup:
• Functioning Shai scenario editor, complete with a predefined example scenario and correctly set up to play, edit and create new scenarios
• Supervisor testing game; a small game environment set up to communicate with Shai, in which the various scenarios can take place
• A list defining 3 test scenarios that the user has to implement
• A tutorial describing how to use Shai
• A questionnaire to fill in after the evaluation (see Appendix)

The test was performed at TU Delft and on location at e-Semble, as a decent setup could be made at these locations.

Approach
Each user was given a list of goals; a few scenarios they had to implement in Shai. First, they were given the time to read a manual and see a demonstration of how to use Shai. Then, they were given 40 minutes to implement the scenarios they had been given. Each user was given the same set of scenarios. This list was defined beforehand, and is regarded as easily doable for beginning users in 30 minutes.

Test Scenarios
Three test scenarios were developed to test the functionality and usability of Shai. These tests were built to require the user to use all the features available in Shai. This makes it a test for feature completeness. Also, this could test if users are having difficulty with the usability of the application. This Section discusses the three test scenarios.

1. Editing an existing example scenario and running it (10 min)
   Goals:
   • Loading an existing scenario
   • Adjusting some properties
   • Re-routing some nodes
   • Running a scenario

   In this session, an example scenario was provided beforehand. This example scenario was very short; it consisted of just five nodes, of which four were interlinked. The first user task was to open the scenario. Then, the user could first run the existing scenario to see how it works. The second task was then to slightly alter the scenario by re-linking one node to another. The user could then view this change by running the new scenario. Finally, the last task was for the user to make another slight adjustment to the scenario, this time by editing a property on one of the nodes.

   This session was designed to train the user in basic scenario authoring actions, and to make him familiar with the main component of the authoring application: the scenario editor. After completing this session, the user had a basic notion of how the scenario editor works.

2. Creating a new scenario, running and saving it (20 min)
   • Adding & linking nodes
   • Adding & linking states
   • Running a scenario
• Saving a scenario
This session required the user to create a new scenario from scratch. In that scenario, a painter was ordered to walk to a scaffold and start painting. Then, the scaffold might crash, depending on whether or not the player had fixed it in the meantime.

This scenario required the user to add nodes and link them together. In total, six nodes needed to be placed, so this was also a short scenario. The user would also have had to add some states and state transitions. Alternatively, the user was shown that he could also use the ‘if’ logic node. Finally, the user was asked to run the scenario to test it, and afterwards save it to an XML file.

This session was designed for the user to become acquainted with adding nodes to the scenario himself. Also, he was introduced to states and how to use them. When the user was finished with this session, he had a good idea of how to set up a basic scenario by himself.

3. Adjusting a running scenario (10 min)
• Loading a saved scenario
• Using Variables
• Adjusting game settings
• Running a scenario
• Adjusting the scenario during play

In this final session, the user was asked to edit a scenario while it was being played. Also, the scenario complexity was increased with the introduction of variables and settings.

First, the user had to adjust a weather setting; he had to change the weather from ‘sunny’ to ‘cloudy’. This was a simple example that shows the user the power of using settings. Then, a variable was added to a predefined example scenario, and was used for scenario logic. Finally, the user was tasked with linking nodes together while the scenario was being played. This gave him a brief example of the power of real-time adjustments.

When the user was finished with this session, the user also understood how he could adjust the scenario while it was being played by the player. Also, he knew how to use variables to add complexity to a scenario, and how to apply different game settings. The user was now fully acquainted with the scenario authoring environment and was ready to start developing scenarios by himself. The user now had an understanding of how the various features in Shai enabled him to control a scenario.

Total duration of evaluation: 40 to 60 minutes.

. . . 7.2 Evaluation Results

This Section discusses the results of the evaluation sessions. Feedback from domain experts is discussed, based on their filled in questionnaires. Then, results from the discussions held after the evaluation sessions are discussed in Section 7.2.2. Please note that the complete filled in questionnaires can be found in Appendix D.
7.2.1 Results

General remarks
Amongst the domain experts, the general consensus was that this scenario editor could offer significant advantages in helping instructors develop scenarios. However, the domain experts who already had some experience in scenario development noted that this application could best be used in the preparation of scenarios, because the real-time adjustment options seemed to be too complex in the current form; re-routing the scenario sequences by breaking and re-connecting links, as is done in the example sessions, can become very difficult to do, especially under the time pressures of a game that is being played. When large, complex scenarios are involved, it can be very difficult for the user to fully comprehend the long-term effects of changes he is making in the scenario, which makes it even more challenging to do real-time. Although the availability of real-time adjustments was applauded, it should be offered in a more user friendly manner. One solution could be to allow the user to make some pre-defined ‘interception points’ in his scenario and provide an easier to use interface for controlling these interception points.

Scenario representation
The use of a visual node-based causality chain for the editing of scenarios was well received by the domain experts. However, some question marks were placed concerning the use of states and settings. One user questioned whether or not users would easily understand this concept, while another user would have preferred states to be used differently. For instance, the state panel should, according to him, be used as a feedback panel, showing the player’s global progress through a scenario. Especially in more complex scenario, the current feedback that shows the player’s progress can be a little awkward (nodes that have been completed are given a different colour; this works well for small scenarios, with a lot of branching scenario structures, this might prove to be insufficient). By using a more global progress feedback system, the user might have an easier, quicker overview of the player’s progress. Also, the current implementation of settings could be improved according to several users, by making them more easily editable and more powerful, such as by providing logic based on the setting’s values.

Features
While the current features offered in the prototype were well received by these domain experts, they still had some suggestions on possible improvements (naturally, since these users had their own views on scenario editing). Some interesting improvements that should be considered as recommendations were

- Options for conversation nodes
- Pause button
- Easier turnaround
- Scoring options
- Nodes that can retrieve information from the game
- Sub-graphing options
- 3D editing with object placing

These improvements will be discussed at more length in the following Section.
Usability
Generally, the usability was rated slightly positively, with an average score of a little over 3 out of 5. This does imply, however, that significant improvements should be made in this area, since this score is not very high if it would be considered for practical use.

7.2.2 Discussion
Finalizing the evaluation sessions after the tutorial and questionnaire, the prototype was discussed with the test users in an informal setting. This Section discusses the issues brought forth during these discussions.

Additional nodes
During the discussions, several proposals were made with regard to some kind of additional nodes; conversation nodes, ‘while’ and ‘case’ logic nodes and information retrieval nodes.

A conversation node could be used for conversation in a game, between the player and NPC’s, offering multiple dialogue choices. For instance, in the Supervisor game, conversations occur between the player and several NPC’s. The player is presented with a handful of choices of how to respond to the NPC’s questions. The game logic is then branched, depending on which option the player has chosen. A conversation node could be used to implement scenario logic based on these options, while at the same time allowing the instructor to adjust the available options, via the node’s properties. Such a node could be very useful in the case of Supervisor, and in other games as well, since conversations are an important part of many simulation games.

Additional logic nodes, such as a ‘while’ and a ‘case’ node, could be used to offer the instructor even more control over a scenario. Currently, ‘while’ and ‘case’ functionality is offered in the prototype, but it would have to be implemented using a cyclic graph in the former, and a lot of ‘if’ nodes in the latter case. While the relatively small scenarios used in the evaluation did not have a need for a while or case node, it is not unimaginable that larger and more complex scenarios would find a need for these nodes.

Finally, an information retrieval node could be used for more complex communication between the scenario editor and the game. In some situations, it would be interesting for the instructor to know certain information on the state of the virtual world, such as the location of certain objects, to base scenario logic on. The instructor could theoretically do this by using a complex combination of nodes and states, but this could easily get very complicated. An information retrieval node could for instance be used as ‘retrieve the location of the barrel with chemical waste’ and could then be combined with a ‘case’ node such as ‘if the barrel is...’ ‘near the other barrels’, ‘within the chemical waste range’ or ‘outside the chemical waste range’. This could offer the instructor more freedom in scripting a scenario.

Graph complexity
Those users already experienced with visual node-based graph editing quickly identified the issue of graph complexity. When a scenario becomes more complex than the short example scenarios used in the tutorial, the graph representing the scenario can
explosively increase in size. It is predicted that any reasonably large scenario quickly becomes unreadable and difficult to navigate. A solution to this would be to implement some kind of sub-graphing options, or the ability to implode and explode sections of the scenario graph. Such a feature has been discussed in Chapter 5, as this issue was already identified in other scenario editing application in the literature study in Chapter 3. It was not implemented in the prototype however, due to the fact that this prototype was designed for an evaluation of the approach, which could also be done with short scenarios. Consequently, the implementation of a sub-graphing feature will be discussed as a recommendation for further developments.

**Environment editing**

Another important issue users found during their evaluation sessions was the lack of a (3D) environment editor. In all sessions, an environment editor was brought forth as a desired feature, which would allow instructors to freely place gameplay objects in the environment. The need for such an editor is identified best in a simulation game that recreates for instance an accident or disaster site, or perhaps a crime scene. In these games, the specific location of objects in the environment can be considered as part of the definition of a scenario. Therefore, to fully allow the instructor to have control over the scenario, he would need to have control over the location of objects as well as over the causality sequence, hence the need for an environment editor. However, allowing the editing of the environment comes with some issues as well.

First, there is the issue of reliability. While the reliability of a scenario in the current scenario editor is guaranteed by the game developer, this is not necessarily the case when the instructor is free to determine the location of objects. An example that illustrates this point is path finding. Currently, when the game developer offers the instructor several nodes requiring path finding from an NPC, he can and should make sure that every possible combination of these nodes result in a functioning game, and that no combination result in errors in path finding (this could be the case for instance, when two nodes are placed after each other ‘painter walk to gas tank’ and ‘painter walk to garage’ and there is no path from the gas tank to the garage). However, when the instructor is free to place objects wherever he wants, the game developer cannot test beforehand that there will be no errors in path finding and thus cannot guarantee that the game will function correctly. Thus, the scenario becomes unreliable.

Second, offering environment editing within the scenario editor also comes with issues regarding the scenario editor’s game independence. One of the main advantages the scenario editor prototype has is that its functionality is independent from the type of game it is used with. Implementing an environment editor within the scenario editor removes this independence, since it then operates on specific level data. Considering these objections, the question of whether or not to offer environment editing as part of the scenario editor, and how this could be done, is something that can be researched in more detail.

**User-friendliness**

Finally, a major issue identified during the discussions was the user-friendliness of the application. Significant improvements need to be made in this area if the prototype is to be developed into a commercial application. For instance, the graphics of the various nodes are not descriptive concerning their functionality. Icons could, and should, be
used for this purpose. Also, as discussed before, usability issues exist concerning scalability. When scenarios become larger and more complex, the sequence graph can increase exponentially in size, making it more difficult to comprehend. At the same time, it becomes more difficult to track the player’s progress through this scenario and it becomes more difficult to oversee the effects of adjusting the scenario in real-time. Other usability improvements that are needed to turn this prototype into a commercial application are ‘basic’ features such as keyboard shortcuts, better descriptions of the various features and an overall better presentation.

However, this poor result in the area of user-friendliness was to be expected, as the current implementation of the prototype is focused on offering functionality and is used as a testing environment for the approach discussed in Chapter 4. As such, the focus was not on usability and it was never intended to be suitable for commercial use.
8. Conclusions and Recommendations

This Chapter discusses the project’s results as to whether or not it has answered the research question and gives some recommendations for further research.

:: 8.1 Conclusions

Although simulation games are a powerful and cost-effective alternative to real-life training sessions, the instructor’s ability to adjust such training sessions to fit individual trainees’ needs is relatively limited in simulation games. The reason for this is that simulation games are often delivered as a static product with predefined scenarios over which the instructor has little control.

In response to this problem, the question that was asked at the beginning of this thesis was how can an instructor be given more control over the scenario of a simulation game?

This question was then subdivided into three sub questions. The answers to these sub questions would then in turn lead to an answer to the main research question. First, how can the scenario of a simulation game be represented at a more abstract level of scripting? The answer to this question was that we could represent the scenario of a simulation game at the more abstract level of scenario scripting, by combining scripts at the gameplay scripting level into scenario building blocks. Then, we had to find out what exactly the composition and content of these building blocks should be, and how the game developer could decide how to abstract scripts from the gameplay scripting level to the scenario scripting level. The second question was thus formulated as how can an abstracted scenario be made editable to an instructor? The answer to this question was to let the instructor (re-)arrange events and actions, with the help of logic, states and variables, and to allow him to edit certain game settings. These verbs together were called building blocks. Finally, we discussed how the arrangement and properties of these building blocks can be communicated to a game engine, so the scenario scripted by the instructor could be executed within a game. This resulted in the question how can the adjustments that have been made to a scenario be communicated to a simulation game? This was done by using an event-based communication system. The result was that the communication could happen real-time, while the game was being played, allowing the instructor to make adjustments during a training session. Also, by making the communication independent from the game, authoring a scenario was made independent of the type of game in which the scenario would be played.

So, the answer to the main research question was to let the instructor exercise control over the scenario by writing a script at the abstracted level of scenario scripting, using the language of scenario building blocks. More control was given to the instructor by allowing him to make real-time adjustments to the scenario as well.

This approach has several advantages. First and foremost, by allowing instructors to write scripts for games at the abstracted level of scenario scripting, their freedom in creating varying and rich training sessions is returned. Another advantage with the current approach is that additional control over a scenario is given to instructors in allowing them to edit a scenario while it is being played. Also, the system is designed to
operate independently from any game engine, which allows users to use the same system for multiple types of games.

To evaluate this approach, a prototype scenario editing application, called Shai, was developed. The design and implementation of this prototype was discussed in Chapters 5 and 6. When the development of the prototype was finished, several evaluation sessions were hosted, to evaluate the prototype in three areas: feature completeness, adequacy of scenario representation and usability. The results of these sessions proved to be a good starting point for the recommendations found in the following Section.

8.2 Recommendations

This project has been set up in such a way that it opens up avenues for further research, which are discussed in this Section, along with an identification of several improvements that can be made to the prototype, which would improve the use of the application and make it ready for use in the real world.

Some improvements discussed in this Section can be made simply due to the fact that several features were not implemented in the prototype, even though they had originally been incorporated into the design. The reason these features were not implemented is because –at that time- they would not improve the prototype’s use in evaluating the approach. However, if these features were indeed implemented, they would certainly improve the application when it is considered for eventual use.

Two areas in which improvements can be made are functionality and usability. These are discussed in the following two Subsections. Additionally, further research can be executed in the area of didactic support, which is discussed in the final Subsection.

8.2.1 Extending functionality

First, the editor’s functionality can be extended based on suggestions from the evaluation. Such improvements are more causality nodes, such as information retrieval nodes and more logic nodes, and possibly an environment editor. While the implementation of more causality nodes is fairly straightforward, offering an environment editor within a scenario editor that is designed to be game-independent has its issues.

Also, some features from the original design have not been implemented in the prototype. Such a feature is the library exporter, which automates the process of translating scenario elements as defined by the game developer in the game engine into a readable format to be used in the Shai designer. Currently, this process is done manually, which is usable for the proof of concept, but should be automated in a final build.

Another feature that could massively improve the usability of Shai is better handling of large graphs, for instance by using graph collapsing. Such a feature has been discussed in the literature study, and was deemed to be an interesting feature, but for the reasons mentioned above was not implemented in the prototype. Another feature is extensive player monitoring. Currently, feedback on the progress of the player is received as
messages in Shai and given to the instructor in the form of a progress indicator in the causality chain. The efficacy of the scenario editor could be improved by more detailed feedback, such as a visualization of the player’s viewport.

Finally, a useful feature would be precondition checking, discussed in Subsection 5.3.1. Precondition checking assists the user in validating scenarios, by checking each node’s preconditions to see if the user hasn’t accidentally written a scenario with inconsistencies in it. For the proof of concept, this feature was not deemed to be strictly necessary, but for a scenario editor to properly function without causing too many problems, precondition checking should definitely be included.

8.2.2 Improving usability

Secondly, the editor could be revised to be more user-friendly. The evaluation showed that this area could be improved significantly. A first step in this direction would be to use better visuals, such as descriptive icons, for the causality nodes; thus making their functionality easier to understand. Currently, the prototype’s interface is very oriented towards function over form, with little visual aids that help the user understand the program’s various functions. Also, improved handling of large graphs, as discussed in the previous Subsection, should improve the usability significantly.

Another issue with the usability of the current prototype is that it is very tiresome to test a scenario. Currently, when the instructor wants to try out a change he has made in a scenario, he can only do so by re-starting the game, which usually takes a significant amount of time (i.e. in the order of a few minutes in the case of Supervisor). This issue could be improved by implementing for instance a scenario resetting system in the game, which allows the game to restart to a specific scenario, without having to reboot the entire game every time.

8.2.3 Didactic support

Finally, an area in which further research could be performed is providing didactic support. Aside from offering instructors more control over a simulation game’s scenario, an application such as Shai could be used to offer didactic functionality as well. Such functionality could for instance be monitoring and analysis of a trainee’s progress. When such features are incorporated in Shai, it could potentially grow out to become a full suite that helps instructors in every aspect of their work; from preparing and running training sessions, to monitoring and evaluation their trainees’ progress.

Concluding, we hope to see this report functioning as a starting point for future research on developing approaches that help instructors and other didactic or creative experts gain expressive power in the exciting and developing field of games. The current implementation of Shai offers the basics of such an approach, and with the right improvements, it could very well be used to give instructors more control over scenarios in simulation games.
A. Original design & example scenario

This appendix presents an overview of the initial design for the nodes of a scenario editor, and an example of a scenario created using this system.

Figure 37 - A list of nodes available to the user
Figure 38 - An example scenario, part 1

Figure 39 - An example scenario, part 2
B. Evaluation Tutorial and Questionnaire

Session 1: Editing and running an existing scenario
Estimated duration: 10 minutes.
Goal: Getting acquainted with the causality editor.

In this session, an example scenario is provided beforehand. This example scenario is very short; it consists of just five causality nodes, of which four are linked together. Your first task is to open the example scenario. You do this by pressing Ctrl-O and selecting ‘ExampleScenario_S1.XML’.

Figure 40: Shai after booting up
The scenario is very basic; the player starts indoors. When he walks outside, a painter is waiting for him. The player can talk to the painter, and send him to either the gas tank or the garage. You can run the existing scenario to see how it plays out in the game. Do this by clicking on the ‘Play’ button. As you will see, this starts the game. The game ends when the painter has reached the gas tank.

The second task is then to slightly alter the scenario by re-linking one node to another. As you can see, one of the nodes is not linked to the rest of the scenario. Disconnect the node titled ‘Painter: Walk to gas tank’ by right-clicking on its plugs. Now, connect the right plug from the node ‘Player: talked to the painter’ to the left plug on the node ‘Painter: Walk to garage’ and from that node on to ‘End Game’. You can do this by dragging the plugs with your left mouse button (LMB). See the following three screenshots for an example.

You can view the change you’ve made by running this new scenario, using the play button again.
Figure 42: Scenario before

Figure 43: Re-linking a connection
Finally, your last task is to make another slight adjustment to the scenario, this time by editing a property on one of the nodes. Select the ‘Painter: Walk to garage’ node by clicking on it with the LMB. As you can see, some properties show up in the properties panel. Here, you can adjust the walking speed of the painter. Change it from 10 to 20 km/h, to make him run.

If you’d like, you can press play again to see your changes in action. This concludes the first session. Well done!
Session 2: Creating and running a new scenario

Estimated duration: 20 minutes.

Goal: Using the causality editor and getting acquainted with the state editor.

In this session you will be asked to create a new scenario from scratch. In that scenario, a painter will be ordered to walk to a scaffold and start painting. Then, the scaffold might crash, depending on whether or not the player has fixed it in the meantime.

Your first task is to clear the existing scenario, if there is any. Do this by pressing Ctrl-N. Now it’s time to add some causality nodes. Start by dragging the ‘Start Game’ node from the library onto the causality editor. Then, also add the node titled ‘Player: talk to the painter’, ‘Painter: walk to scaffold and paint’, ‘Scaffold: Crash’ and ‘End Game’.

As you might have noticed, these nodes were grouped into two types; Event Nodes and Action Nodes. No might be a good time to explain the difference between these two types. An event node is used to send a message from Shai to the game, indicating that an event needs to be executed. It’s telling the game that a specific object or person needs to do something. An action node on the other hand is used by the game to inform Shai that the player has done something. As you see, all these nodes start with the word player.

Now, go ahead and connect the nodes you’ve just placed to each other like in the picture below. You might need to move some nodes around a bit; you can do this by left-clicking on a node, hold the mouse button, and then move it around.

Great! You can test your newly created scenario by pressing the play button. If everything went well, the painter went to go paint on top of the scaffold when the player has talked to him. The game exits shortly after the painter has reached the scaffold.
Now we’re going to add some variation to this scenario. What happens when the player doesn’t see that the scaffold is broken? The painter will try to stand on it, but it will crash! To represent whether or not the player has fixed the scaffold, we will use states.

You can find the state editor in the bottom centre panel of Shai. Adding states is done by pressing RMB on a blank Section of the state editor and selecting ‘Add new state’ from the popup-menu. You can name the states any way you like in the properties panel. Create one new state, next to the default ‘Start’ state and name it ‘Scaffold fixed’. You can also rename the start state to ‘Scaffold broken’.

Now, to define a state transition, we’ll go back to the causality editor. From the list of action nodes in the library, select ‘Player: fixed the scaffold’ and drag in onto the causality editor. Then, from the list of other nodes, drag a ‘Switch state’ node on to the editor. Connect the action node to the switch state node and in the properties panel of that node, enter the name of your new state ‘Scaffold fixed’.

![Figure 46: Scaffold states](image)

What is left now is to define some logic based on these states. As mentioned before, the scaffold will crash when the painter stands on it, if the player hasn’t fixed it. So, we will add a ‘Check state’ node to the scenario. Drag it onto the editor and connect it to the end of the ‘Painter: walk to scaffold and paint’ node, so it will be triggered when the painter has reached the scaffold. In the properties panel of the ‘Check state’ node, enter the name of the state you want to check. In this case, that would be ‘Scaffold broken’. Now, link the top connection ‘is in state’ to the node titled ‘Scaffold crash’. Also link the other connection and the connection from scaffold crash to ‘End Game’.

Two more things need to be done to clean this scenario up. First, also connect the start game node to ‘Player: fixed the scaffold’, otherwise it will never get triggered. Also, make sure ‘Painter: walk to scaffold and paint’ is not directly connected to the end game node anymore. To following figure will illustrate how the scenario should look like right now.
Figure 47: Complete scenario

Play the scenario to test it out. As you will see, the scaffold will crash depending on whether or not the player has fixed it. To create this behaviour, you used states. Can you imagine another way you could have created this behaviour, without using states (only nodes)?
Session 3: Editing a running scenario

Estimated duration: 10 minutes.

Goal: Learn how to adjust a running scenario and using variables and settings.

In this session you will be asked to edit a scenario while it is being played. Also, you will use variables to base the scenario’s logic on. In this scenario, the player can move a chemical barrel, and an assistant will walk around.

Start by loading ‘ExampleScenario_S3.XML’.

The first thing we’re going to do in this session, is adjusting a game setting. This is the final panel we haven’t used yet and it can be found in the bottom left corner. Click on ‘Weather’ and change the Weather settings from ‘sunny’ to ‘cloudy’. Now, when you boot up the game, you’ll see the weather has changed to cloudy, as opposed to the sunny weather we have been seeing in the previous sessions.

Another new thing we’re going to use is a variable. One variable is already placed, and is linked to a logical compare node. Place a new variable node by dragging it on the scenario from the library. A variable is used to story a value, so you can use it later in the scenario. The variable’s value can be set by you, in the properties panel, or it can be set by the game. In this case, we’re going to connect the variable to the bottom exit of the ‘Player: Moved the barrel’ node. Now, the game will define the value of the variable, and it will define it as the time it took the player to move the barrel (counted from the moment he stepped outside the office).

We can then link the variable to a logical compare node, to compare the value of the variable to the value of another variable. In this example, we will use the compare node to check if the player took less, more or exactly 3 minutes to move the barrel.
Now, there is only one thing left to do before we’re going to try out this scenario. Place one more node at the end of the scenario; ‘Painter: Walk to gas tank’, but don’t connect it to anything yet, as we’re going to do that while the game is running!

Now, press ‘play’ to start the game.

When the player has moved the barrel, the assistant will walk to one of three locations, depending on how long it took the player to move the barrel. Now, while the assistant is moving to one of the locations, we will connect the last node to the selected assistant node. Be sure to do this quickly, as it will not work when you do it too late. As you will see, the result indicates that you can adjust the scenario of the game while it is running. Similarly, you can re-link nodes, place new nodes or edit their properties; with the only requirement of course that the node you’re adjusting hasn’t been reached yet.

Congratulations, that’s the end of the final session. You’ve now finished exploring the various features of Shai. Of course, this is only the first step in creating and using your own scenarios. To finish things up, stop the scenario if you haven’t done so yet, and please take some time to fill in the questionnaire on the following page.
# C. Filled in questionnaires

Name: Ronald In ’t Velt  
Profession: IT Architect for Shell Learning

<table>
<thead>
<tr>
<th>General Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>How do you feel about the approach of using a standalone application that allows instructors to adjust the scenario of simulation games?</strong></td>
</tr>
<tr>
<td>I think this type of tool is essential, as it allows course designers to create scenarios without requiring help from game designers. This has the potential of greatly streamlining the scenario design process, and puts that process where it belongs: with learning professionals instead of IT/Game experts.</td>
</tr>
<tr>
<td>From an architectural point of view there is a benefit in making this application stand-alone: it makes it lightweight and portable across different game engines.</td>
</tr>
<tr>
<td><strong>Do you see advantages of using this application in training sessions?</strong></td>
</tr>
<tr>
<td>A possible advantage is that it allows an instructor to adjust the scenario immediately, if he notices that the game is too easy or too hard for a student. The scenario can be adjusted based on questions or remarks from students.</td>
</tr>
<tr>
<td>Another use case made possible by this application is doing after-action reviews in real time: a group discussing a safety incident can set up the events leading up to the incident and replay it.</td>
</tr>
<tr>
<td><strong>Do you see any disadvantages in using this application in training sessions?</strong></td>
</tr>
<tr>
<td>A potential pitfall of using this application during training is that tweaking the scenario will detract attention away from the scenario itself. Also, using this tool effectively during a running scenario requires a good deal of proficiency in its use by the instructor.</td>
</tr>
<tr>
<td><strong>Do you feel this application can improve the flexibility of training sessions?</strong></td>
</tr>
<tr>
<td>Yes, see above!</td>
</tr>
<tr>
<td><strong>Do you feel this application gives you more control over the scenario of a simulation game?</strong></td>
</tr>
<tr>
<td>Yes, but it may provide too much control to an inexperienced instructor, or work against the learning objectives that the designer of the scenario envisioned. One way to mitigate this would be to not let the instructor tweak the whole scenario on the fly, but instead predefined a number of alternative chains of events in the scenario which the instructor can trigger at will.</td>
</tr>
</tbody>
</table>
## Scenario Representation

**Do you feel you understand how a scenario is represented in this application?**

Yes, the flow of events is clear from the representation. The use of “human” language in the description of the events helps a lot.

**Do you feel the representation of a scenario in this application is intuitive?**

Mostly yes, though by improving the graphical representation of more abstract elements such as conditions (if/then/else), these concepts will be much clearer to non-technical users.

**Do you feel the representation of a scenario in this application is useful?**

Yes, I would think that the representation is sufficiently clear for a course designer to understand and discuss.

**Do you appreciate the use of nodes for scripting a scenario?**

Nodes are a good way to represent the flow and cause-and-effect in a scenario.

**Do you appreciate the use of states for scripting a scenario?**

States are a concept that many people may struggle to understand properly, but as an advanced design option they are still necessary. It may help to represent multiple states as clear 1-of-many options.

**Do you appreciate the use of settings?**

They are useful, but should perhaps be dynamic and be available as parameters during the scenario (i.e. if it rains, then you need to have the raincoat on).

## Features

**Do you feel the current features offered in this application are enough for you to exercise control over a scenario?**

I would like to see the following:

- options to customise dialogues and trigger actions based on the outcome of those.

- real time / dynamic / conditional weather changes, and triggers based on those conditions

- an option to reset the game to the start position or roll back to intermediate "save points". Also a pause button to stop the game.
- options to add and position game objects in the scenario
- scoring. This would have to be multidimensional, for example a student can be scored on observancy, speed, correctness, etc. These scores could be updated by any of the elements in the scenario, and score levels could also fire off new events.

**Do you feel that there are features missing in this application, which would give you better control over a scenario?**

See above. Also, an “instructor mode” would be useful, in which the scenario would be made read-only, but the instructor would still be able to fire off certain nodes.

**Do you feel some features could be improved, to give you more control over a scenario?**

Not at this time.

### Usability

**On a scale of 1 to 5, how would you rate the user-friendliness of this application?**

3. Scenario design and layout is mostly intuitive, but would like to see the following:
   - Windows-style menus to open and save, etc.
   - Clearer interface to drag-drop connections between nodes, and a right-click menu to delete nodes or connections.
   - A quick way to pan across the canvas when zoomed in (didn't see how this worked in the current version).

**On a scale of 1 to 5, how would you rate the visual representation of the nodes in the scenario editor?**

4. Nodes and especially the descriptions are clear enough. The visual representation of the conditional stuff might be improved to better convey their meaning.

**On a scale of 1 to 5, how easy did you find it to create and edit scenarios using this application?**

4. I have only tried simple scenarios though.

**Finally, are there any other comments you would like to share?**

Good effort and in my opinion an important first step to address the difficulties of scenario design, especially in the area of user friendliness and abstraction level. I see this tool as an enabler for non-technical learning designers, allowing them to design a scenario from start to finish by themselves.
<table>
<thead>
<tr>
<th>General Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do you feel about the approach of using a standalone application that allows</td>
</tr>
<tr>
<td>instructors to adjust the scenario of simulation games?</td>
</tr>
<tr>
<td>Indien goed toegepast heeft het zeker toegevoegde waarde. Valkuilen zij vooral</td>
</tr>
<tr>
<td>van didactische aard.</td>
</tr>
<tr>
<td>Do you see advantages of using this application in training sessions?</td>
</tr>
<tr>
<td>Mostly in pre-training/configuration.</td>
</tr>
<tr>
<td>During training usage should be limited to “trigger” actions instead of overriding events etc.</td>
</tr>
<tr>
<td>Do you see any disadvantages in using this application in training sessions?</td>
</tr>
<tr>
<td>Didactical disadvantages. A good script should not be changed by the instructor.</td>
</tr>
<tr>
<td>This is also dependant on type of training. Exams are way more strict and should not need interventions.</td>
</tr>
<tr>
<td>Do you feel this application can improve the flexibility of training sessions?</td>
</tr>
<tr>
<td>Flexibility no. You are removing flexibility by introducing scripts and events.</td>
</tr>
<tr>
<td>You will make the training more dynamic!</td>
</tr>
<tr>
<td>Do you feel this application gives you more control over the scenario of a simulation game?</td>
</tr>
<tr>
<td>Yes, user can define his own events and is “in control” over actions.</td>
</tr>
<tr>
<td><strong>Scenario Representation</strong></td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td><strong>Do you feel you understand how a scenario is represented in this application?</strong></td>
</tr>
<tr>
<td>A complete scenario will get messy in terms of overview. The building blocks need usability/visual check to become understandable and readable</td>
</tr>
<tr>
<td><strong>Do you feel the representation of a scenario in this application is intuitive?</strong></td>
</tr>
<tr>
<td>Building blocks / Flow diagram is the best approach for this. Usability-wise it can be a lot better</td>
</tr>
<tr>
<td><strong>Do you feel the representation of a scenario in this application is useful?</strong></td>
</tr>
</tbody>
</table>
| Yeah I can see useful implementations.  
- follow state of scenario  
- make instructions  
- Build your own dynamic script |
| **Do you appreciate the use of nodes for scripting a scenario?** |
| All in favor of nodes |
| **Do you appreciate the use of states for scripting a scenario?** |
| Not in the current form.  
- I would like to see states in a more global manner  
- Keep track of the scenario state (as an overview) |
| **Do you appreciate the use of settings?** |
| - Not really |
### Features

*Do you feel the current features offered in this application are enough for you to exercise control over a scenario?*

- Currently they are too rigid I would like to see more global events based on “assets” available in a configuration

*Do you feel that there are features missing in this application, which would give you better control over a scenario?*

- Overview during training sessions
- Events will get large soon so a folder / categorization is needed
- More actions and checks (proximity checks)

*Do you feel some features could be improved, to give you more control over a scenario?*

- Mainly usability
  - > would need hands-on to give more feedback

### Usability

*On a scale of 1 to 5, how would you rate the user-friendliness of this application?*

2.5

*On a scale of 1 to 5, how would you rate the visual representation of the nodes in the scenario editor?*

2

*On a scale of 1 to 5, how easy did you find it to create and edit scenarios using this application?*

2.5

*Finally, are there any other comments you would like to share?*

<table>
<thead>
<tr>
<th>General Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do you feel about the approach of using a standalone application that allows instructors to adjust the scenario of simulation games?</td>
</tr>
<tr>
<td>Do you see advantages of using this application in training sessions?</td>
</tr>
<tr>
<td>Do you see any disadvantages in using this application in training sessions?</td>
</tr>
</tbody>
</table>
| Do you feel this application can improve the flexibility of training sessions?   | - In verhouding tot “fixed-code” zeker!  
- Vanuit XVR (waar je alle mogelijkheid to wijzigen hebt), bidet het houvast in reproduceerbaarheid |
<p>| Do you feel this application gives you more control over the scenario of a simulation game? | Yes, for the instructor in a reproduction gameplay. |</p>
<table>
<thead>
<tr>
<th><strong>Scenario Representation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Do you feel you understand how a scenario is represented in this application?</em></td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td><em>Do you feel the representation of a scenario in this application is intuitive?</em></td>
</tr>
<tr>
<td>Kan.</td>
</tr>
<tr>
<td><em>Do you feel the representation of a scenario in this application is useful?</em></td>
</tr>
<tr>
<td>Het kan een beschrijfboom geven.</td>
</tr>
<tr>
<td><em>Do you appreciate the use of nodes for scripting a scenario?</em></td>
</tr>
<tr>
<td>Is het compleet? =&gt; moet volgen uit testen.</td>
</tr>
<tr>
<td>For loop?</td>
</tr>
<tr>
<td>While?</td>
</tr>
<tr>
<td><em>Do you appreciate the use of states for scripting a scenario?</em></td>
</tr>
<tr>
<td>States: Binair (nu) -&gt; Int -&gt; Float -&gt; Graph (toekomst?)</td>
</tr>
<tr>
<td><em>Do you appreciate the use of settings?</em></td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>
## Features

**Do you feel the current features offered in this application are enough for you to exercise control over a scenario?**

- 

**Do you feel that there are features missing in this application, which would give you better control over a scenario?**

- 

**Do you feel some features could be improved, to give you more control over a scenario?**

- 

## Usability

**On a scale of 1 to 5, how would you rate the user-friendliness of this application?**

3, it has some fitness for use-interfacing

**On a scale of 1 to 5, how would you rate the visual representation of the nodes in the scenario editor?**

3. Please Test with 50 actors, 200 logic, 25 labels, 25 targets

Is it still ok?

**On a scale of 1 to 5, how easy did you find it to create and edit scenarios using this application?**

- 

**Finally, are there any other comments you would like to share?**

I would suggest a setup in which there are 4 phases:

- Use in training
- Building scenarios
- Game development
- Evaluation

First, the game developer writes the game and scenario building blocks. Then, in the scenario building phase, an elaborate scenario is defined. Then, the scenario can be used in a training session, in which the instructor has a new dashboard, showing evaluation and feedback statistics alongside a few buttons he can use to intervene with the running scenario. This relieves the difficulty of adjusting a running scenario that is very complex.
**General Questions**

<table>
<thead>
<tr>
<th>Question</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>How do you feel about the approach of using a standalone application that allows instructors to adjust the scenario of simulation games?</em></td>
<td>Positief. Maar ik zet vraagtekens bij de haalbaarheid van het aanpassen can het scenario tijdens het spel. Een groot complex schema pas je niet even on-the-fly aan en de gevolgen ervan helder te hebben.</td>
</tr>
<tr>
<td><em>Do you see advantages of using this application in training sessions?</em></td>
<td>Voor het maken van scenarios wel. Tijdens de training niet. Ik denk dat on-the-fly aanpassen van de logica in de praktijk niet gaat werken.</td>
</tr>
<tr>
<td><em>Do you see any disadvantages in using this application in training sessions?</em></td>
<td>Ja</td>
</tr>
<tr>
<td><em>Do you feel this application can improve the flexibility of training sessions?</em></td>
<td>Nee. Maar kan wel een boel werk uit handen van de instructeur nemen.</td>
</tr>
<tr>
<td><em>Do you feel this application gives you more control over the scenario of a simulation game?</em></td>
<td>Ja</td>
</tr>
<tr>
<td><strong>Scenario Representation</strong></td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td></td>
</tr>
<tr>
<td><em>Do you feel you understand how a scenario is represented in this application?</em></td>
<td></td>
</tr>
<tr>
<td>Ja</td>
<td></td>
</tr>
<tr>
<td><em>Do you feel the representation of a scenario in this application is intuitive?</em></td>
<td></td>
</tr>
</tbody>
</table>
| Voor mij als techneut wel.  
Zet vraagtekens bij of dit ook voor niet techneuten geldt. |
<p>| <em>Do you feel the representation of a scenario in this application is useful?</em> |
| Ja |
| <em>Do you appreciate the use of nodes for scripting a scenario?</em> |
| Ja |
| <em>Do you appreciate the use of states for scripting a scenario?</em> |
| Ja |
| <em>Do you appreciate the use of settings?</em> |
| Ja |</p>
<table>
<thead>
<tr>
<th><strong>Features</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you feel the current features offered in this application are enough for you to exercise control over a scenario?</td>
<td></td>
</tr>
</tbody>
</table>
| Nee, ik mis in ieder geval.  
- scripts op basis van object-type ipv object  
- node kan informatie uit spel opvragen (over andere objecten) |  |
| Do you feel that there are features missing in this application, which would give you better control over a scenario? |  |
| Ja, zie boven |  |
| Do you feel some features could be improved, to give you more control over a scenario? |  |
| Nog geen kijk op. |  |

<table>
<thead>
<tr>
<th><strong>Usability</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>On a scale of 1 to 5, how would you rate the user-friendliness of this application?</td>
<td></td>
</tr>
<tr>
<td>Voor mij als technet: 4</td>
<td></td>
</tr>
<tr>
<td>On a scale of 1 to 5, how would you rate the visual representation of the nodes in the scenario editor?</td>
<td></td>
</tr>
<tr>
<td>Voor mij als technet: 4</td>
<td></td>
</tr>
<tr>
<td>On a scale of 1 to 5, how easy did you find it to create and edit scenarios using this application?</td>
<td></td>
</tr>
<tr>
<td>Voor mij als technet: 4</td>
<td></td>
</tr>
<tr>
<td>Finally, are there any other comments you would like to share?</td>
<td></td>
</tr>
<tr>
<td>Interessante applicatie. Maar ben wel benieuwd naar of deze opzet werkelijk is bij grote complexe scenarios (veel nodes). Kan de programmeur bijvoorbeeld elke mogelijke nodekoppeling overzien en implementeren?</td>
<td></td>
</tr>
</tbody>
</table>
D. References

"Serious Game Initiative." from http://www.seriousgames.org/about2.html.