NIFV - T1

A Serious Game concerning Triage with use of an Intelligent Tutoring System

MSc Thesis

M.M. Hendriks
June 2008

TU Delft

NEDERLANDS INSTITUUT FYSIEKE VEILIGHEID Nibra

NIFV – T1

A Serious Game concerning Triage with use of an Intelligent Tutoring System

MSc Thesis

M.M.Hendriks
1150014
Man-Machine Interaction Group
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Delft University of Technology (TU Delft)

Graduation committee:
Prof.dr. L.J.M. Rothkrantz
Ir. P. Wiggers
Dr.ir. A.R. Bidarra
Th.J.A. Uffink, RN, CCRN, MSc

Abstract:
In a Mass Casualty Incident, medical personnel aren't able to help all the victims in a disaster area at once. Therefore, triage is performed to classify the casualties into groups of urgency. In the Netherlands, a new triage procedure is going to be introduced in the beginning of 2009. The triage system requires medical workers to re-learn and train the reasoning patterns of the new methodology.

This MSc thesis will describe the creation of a serious game that will help medical workers train the triage procedure. In order to track and respond to the user's actions in the game an Intelligent Tutoring System (ITS) will be implemented that is specifically designed for communicating with the game and the underlying game engine.

The global ITS framework is used for three different observation and manipulation functions: the monitoring of the player's movement, the providing of feedback on the user's actions concerning triage and the automatic creation of a disaster scenario. This last function will generate a disaster area at random for the player to search victims in. The main focus of this project is the creation of an instructional tool for learning triage and the implementation of a placement algorithm for the automated creation of a disaster area.

The complete application will be created with help from the Valve Source engine. This engine provides users the necessary functionality to create a digital game for PC. With this basic functionality already implemented, focus can be more towards the actual creation of the framework and game content.

Expert knowledge regarding triage procedures, forms of evaluation and victim data are provided by the Nederlands Instituut Fysieke Veiligheid (NIFV). Experts at this institute were also willing to participate in a small preliminary test to get a first opinion of the implemented game. These participants were excited about the created pilot and the feedback it provided. Although this was only a preliminary test, further development in this type of training and learning will be very useful for all kinds of relevant topics in disaster management.

Abstract

In a Mass Casualty Incident, medical personnel aren't able to help all the victims in a disaster area at once. Therefore, triage is performed to classify the casualties into groups of urgency. In the Netherlands, a new triage procedure is going to be introduced in the beginning of 2009. The triage system requires medical workers to re-learn and train the reasoning patterns of the new methodology.

This MSc thesis will describe the creation of a serious game that will help medical workers train the triage procedure. In order to track and respond to the user's actions in the game an Intelligent Tutoring System (ITS) will be implemented that is specifically designed for communicating with the game and the underlying game engine.

The global ITS framework is used for three different observation and manipulation functions: the monitoring of the player's movement, the providing of feedback on the user's actions concerning triage and the automatic creation of a disaster scenario. This last function will generate a disaster area at random for the player to search victims in. The main focus of this project is the creation of an instructional tool for learning triage and the implementation of a placement algorithm for the automated creation of a disaster area.

The complete application will be created with help from the Valve Source engine. This engine provides users the necessary functionality to create a digital game for PC. With this basic functionality already implemented, focus can be more towards the actual creation of the framework and game content.

Expert knowledge regarding triage procedures, forms of evaluation and victim data are provided by the Nederlands Instituut Fysieke Veiligheid (NIFV). Experts at this institute were also willing to participate in a small preliminary test to get a first opinion of the implemented game. These participants were excited about the created pilot and the feedback it provided. Although this was only a preliminary test, further development in this type of training and learning will be very useful for all kinds of relevant topics in disaster management.

Acknowledgements

Within a year of working on this thesis, I have come to meet a lot of people who were more than willing to help me with this project. I would like to take this opportunity to thank all these people for their support.

First of all, I would like to thank my supervisor Prof.dr. Leon Rothkrantz, for his valuable advice throughout this complete project. Regarding the content of this thesis and the structure of the implemented game, he has helped me make clear decisions and a global perspective on the project.

Secondly, my thanks go out to the NIFV, for providing all expert knowledge on the triage process and especially for letting me use the victim database in the serious game. Also giving the opportunity to create a serious game that might later be used to train medical workers was a great experience for me. Special thanks go out to Theo Uffink, my supervisor at the NIFV. Many ideas and thoughts have come from our meetings and brainstorm sessions in Arnhem. Another person to mention is Marco van Wijngaarden, who gave the initial go, but left the NIFV at the beginning of this project. I would also like to thank all the co-workers at the NIFV for participating in the user study. Without this, the project wouldn't be completed.

I would also like to thank the Valve Developer's Community for providing me with workable solutions and good alternatives to the problems I've encountered with the creation of the game.

Last but certainly not least, my thanks go out to all of my friends and family for supporting me. Thank you all for reading and commenting my thesis and for providing a good balance between working and relaxing. Special thanks to Alexander Fijlstra for the creation of the victim model and the audio files in the game.

Only one person is left to thank, and that is my girlfriend Linda. Lin, you have been a tremendous help throughout this project and I don't know what I would have done without you. Thank you for your love and support.

Contents

Chapter 1 - Introduction

Ambulance driver John A. and his companion care assistant Monica B. are responding to a distress call heard on the radio. It seems that a touring bus has collapsed against an approaching truck on one of the federal highways. As soon as they approach the reported area, they realize that this is a Mass Casualty Incident. The collision was on a very busy road, creating a chain reaction of cars colliding onto the bus or truck.

Fortunately, other ambulance teams have already set up a meeting point and the fire and police departments are also on scene to put out small fires and help bystanders. Once John parked the ambulance, Monica rushes out to the present medical forward incident officer. They are set on the triage of victims on the crash site. Their job is to categorize every victim into 5 easy to recognize groups indicating the urgency of treatment.

Monica walks back to the ambulance and informs John while grabbing the triage tags from the back of the car. They both walk to the disaster area and only then they realize how big this incident is. A lot of cars are crashed together on both lanes of the highway. People are screaming everywhere, some managed to get out of their car to help others, some badly injured still in their car. Parts of metal and glass fill the ground and total chaos has struck. Luckily, John and Monica both had some training on operating in such conditions!

1.1 - Relevance

Fortunately, large-scale Mass Casualty Incidents don't happen very often in the Netherlands. If they do happen, like for instance with the fireworks explosion in Enschede (2000) [86], the fire in cafe 't Hemeltje in Volendam (2001) [35] and the recent Schipholbrand (2005) [65], disasters will always be evaluated by government agencies to understand how disaster management could be further improved. This improvement requires lots of training for both personnel in the field as managers outside the disaster area.

Instead of letting medical workers only learn the theory of disaster management, practical training is also required to be prepared for the actual incident. Although a staged disaster with actors playing all sorts of injured victims gives a real sense of urgency and pressure, planning for such training requires a lot of time, space and money. A computer application that simulates a disaster area with victims and the like can be started whenever the player wants, takes up one desk and a chair of space and costs significantly less than the previous approach. For these reasons, first responders start to work more and more on computers to train the necessary skills.

In the Netherlands a new triage system (Het Nederlands Triage Systeem [58]) is going to be introduced. This system is now used as pilot in 4 test regions and after this test run the system should become operational in the beginning of 2009. With this new system, the medical sector that has to deal with Mass Casualty Incidents (MCIs) (e.g. medical workers on scene, hospitals, even general practitioners) all use the same system for classifying victims into groups of urgency.

To introduce this new system means there needs to be a lot of re-education and training before everybody is familiar with the concept. A new training facility is needed to get people familiar with the system. The Nederlands Instituut Fysieke Veiligheid (translation: Netherlands Institute for Safety; further referred to as NIFV [57]) is interested in such a facility that teaches the player this new method of triage. The NIFV is concerned with the research on and training in crisis management and fire safety. Both medical personnel and fire fighters can follow specific training courses on these subjects. The application could then be used as an addition to a course on the new triage method.

The NIFV is already using a training simulation called NIFV-ADMS (ADMS is an acronym for Advanced Disaster Management Simulator; see [60] for more information). This simulator gives an instructor total control over what type of disaster the students have to deal with. The target audience of the simulation is all disciplines in disaster management: from ambulance, fire and police departments up to crisis management teams residing at governmental organizations.

Main focus points of the NIFV-ADMS simulator are the management roles that are needed in controlling a disaster. This means that the main task of the players is to direct virtual characters to perform the correct actions. However, players of the simulation can also walk through the virtual

world seeing through the eyes of such a virtual character. Most of the actions (when ordered to perform) are performed automatically by the system, giving more room for planning how to progress with controlling the calamity.

1.2 - Serious games

The most clear-cut approach to creating a computer application for training purposes is by means of a simulation. Simulations provide a digital (mostly 3D) environment that closely resembles the real world. In most cases, additional peripherals are provided to give input like in real-life. An example of this is a flight simulator with a complete cockpit to make the impression of being in a real plane even stronger.

Although simulations work very well as training environment, a new area of research and commerce has been established in 2002. These 3D applications use PC game elements to learning or create awareness of certain topics of interest. They are called Serious Games. By using game elements used in Commercial Of-The-Shelf (COTS) games and didactical methodologies, serious games tend to mix a learning tool with a game to be played by the student.

Of course, these serious games differ from the training simulations. Entertainment games always have some sort of unrealistic components. Zombies, slow motion effects, cartoonesque figures or weapons, everything the mind can come up with has been put in entertainment games. Serious games are also more flexible in putting entertainment values before realistic behaviour, unless this contradicts with the topic to be taught.

In the case of a serious game for the new triage procedure, players must be able to learn the triage topic while walking through a disaster area. So far, this can also be presented with a simulation. However, to induce the player to keep playing, a score can be kept and displayed on the screen to indicate how well the player is doing. Information concerning the new triage process can be displayed whenever the user has done some action or when the user wants more information.

Also, the main goal of the application is to teach the student a new reasoning pattern of how to perform certain tasks and what to conclude from the results. The focus lies more on the reasoning then on the actual performing of the tasks. A simulation for instance might give a complete and detailed description of what the victim's condition is. Within this project, the user doesn't need to get a detailed description, as long as the information provided is enough to classify the victim.

1.3 - Intelligent Tutoring Systems

Although serious games are concerned with a given topic they bring to the user, a lot of those games provide the information instead of checking whether or not the message got through. A lot of research has been and still must be done in the knowledge communication aspects of serious games.

A rising branch of Artificial Intelligence (AI) is focused on conveying knowledge inside a computer to a human being operating it. A concept is designed for this purpose, called an Intelligent Tutoring System (ITS).

This system tries to obtain a model of what the student (e.g. user) has understood of the information provided so far by the tutor (e.g. the computer). Certain errors should be detected and corrected by the system; knowledge should be graduately given to the user, instead of all the information in one time; appropriate learning strategies should be applied to different students with different backgrounds. All these tasks can be accomplished more or less with an ITS.

To use this system as an underlying structure of the serious games gives the game more control of what the user is doing and how to respond to these actions in a fashionable manner. For the triage training, evaluation is crucial in order to indicate if a player has mastered the new procedure or not. Therefore, this serious game will make use of an ITS to report observations of the user both to the user himself and the trainer.

1.4 - Automated Scenario Creator

A person who plays a fair amount of entertainment games (whether it be on PC or console like the Xbox or Playstation) knows that a good game will suck you in, passing hours of game play in what seems to be minutes. This behaviour is very important to keep playing the game without getting bored.

In entertaining games, the player is constantly confronted with new components (e.g. weapons, power-ups, etc.) and/or a compelling story line that makes the game more of an interactive movie.

For serious games, these immersive components can be used to let the user play the game more often. This indirectly provides more time to learn the topic presented by the serious game. It can even induce a constant re-learning process, which keeps the user up-to-date on specific routines, skills and/or knowledge.

To use this immersion in the triage application, a level structure can be applied. This will consecutively increase the difficulty of the disaster area. The disaster area itself can also be a component that helps induce immersion. By setting a certain atmosphere, the user can feel the sense of urgency and the pressure that is also present when working at a real disaster.

An extra feature that this project will try to accomplish is the random generation of the disaster area. With this feature every level of the game will become slightly different, providing a surprise element to the user. Victims that aren't always in the same spot, an occasional fire that breaks loose, these components make the player be a little bit more on his/her toes. He/she must be always alert to deal with the unexpected, which is also true in the real world.

1.5 - Problem definition

The purpose of this Master thesis is to describe the creation of the serious game, called NIFV – T1, that uses the architecture of an Intelligent Tutoring System (ITS) to track the player's actions in the game. Based on the progress of the player, the game creates a random scenario that becomes more challenging. The underlying framework that creates the scenario and monitors the player is the main focus of this project.

In order to tackle this problem of designing and creating such an application, several tasks need to be done. In the first place more needs to be known of the new triage process, serious games and Intelligent Tutoring Systems. Once the theory is discussed, a choice needs to be made by what means this application is to be created. Can an already created game be altered to meet the specifications required? Does the game need to be created from scratch? How well can an additional feature like the ITS be added to game environments? These questions must be answered in order to see what further specifications are to be made.

A global overview of what should be created and how this should be created must be presented and specified to get a blueprint of the game's functionality. Since the Master project is no longer than a year, a choice must be made what to actually design and implement for this application. For this project, the focus lies on the ITS framework together with the automated scenario creator. These concepts will be created in the game, which also needs to be created. This means that the graphical representation of NIFV – T1 gets a lower priority.

Finally the implemented game will be tested for its playability. A small user study will take place to give an indication if the implemented parts work and how they work for the users attending the test.

1.6 - Overview

This report is subdivided into five parts. These parts all cover a different section of the problem and its solution.

Part 1 will describe the theory concerning triage, serious games, Intelligent Tutoring Systems and the Valve Source engine (chapter 2 – 5 respectively). This part contains a description of what the new triage methodology is and how it should be applied in a disaster. The new growing market of serious games is described, along with some examples and the concept of Intelligent Tutoring Systems is introduced. The last chapter of this first part describes the choices made to come to the game engine used for this serious game project.

In part 2, specifications are made to the serious game (chapter 6 - 8). Some global handles of how the game should interact with the player and how entities in the world should be represented are discussed. This part will also delve deeper in the most important entity in the virtual world: the victim.

The instructional tool is described in part 3 of this report (chapter 9 – 11). In this section the game's design and implementation is described. The focus lies mostly on the instructional tool observing and responding to the user's actions and the actual implementation of the victim agent.

Part 4 (chapter 12 – 14) will discuss the design and implementation of the placement algorithm used to create random scenarios. With this description, the reader is able to understand how a scenario is created within this serious game and how easy it is to create a new scenario for the expert.

Evaluation of the game itself and the project are described in part 5 (chapter 15 & 16). A user study will briefly discuss results of a first test run with several users and experts. Finally, chapter 16 will conclude this report.

Part I - Theory

Chapter 2 - Triage

The word ‘triage’ originally comes from the French verb ‘trier’, which means ‘to sort’ or ‘to sort out’. In the scope of this thesis, triage is the process of classifying victims by priority of further evaluation and treatment. The term was first coined in Napoleons army, with its main purpose to get soldiers with minor injuries back to the battlefield as fast as possible. Nowadays triage is used in military and disaster medicine, trying to give the victims that need immediate help the first treatment.

Triage is of great importance when there aren’t enough resources to treat all the patients simultaneously. Even in an incident with 3 victims it could be wise to do a small priority check to see which patient needs to be treated first. When the incident becomes a disaster, the need for triage is tremendous. By applying triage to these types of incidents, medical personnel can do the best possible for the most casualties and get the right patient at the right time to the place they need to be.

Because the condition of the patient can change over time, it is very important to triage multiple times throughout the disaster. Since the procedure shouldn’t take longer than 30 to 60 seconds per victim, repeatedly triaging casualties should not be a problem to the medical staff.

Triage, being such a general and abstract term, has a lot of variations throughout the world. The numbers of classes and the procedure of appointing casualties to a certain class vary from country to country. Even within countries, different triage systems can be used by different sectors that might need it (e.g. military, hospitals, disaster management).

2.1 - Methods

One of the most common used triage method is called the Simple Triage And Rapid Treatment (START) [73]. This system has four distinct triage categories:

- **Minor**, indicating the patient doesn’t need to be treated anytime soon. Victims can be treated the next day, if at all.
- **Delayed**, indicating the patient needs to be taken to a hospital within 6 hours.
- **Immediate**, indicating the patient needs to be taken to a hospital within 2 hours.
- **Deceased**, indicating the patient has lost all life signs or is so badly injured that death is imminent.

In order to classify a victim to a certain group, the medical personnel have to evaluate the state the victim is in. If vital functions are threatened, the victim would normally get assigned to the ‘immediate’ group. People who could walk by themselves should receive the minor tag, even if it could mean the collapse of the victim after several steps. Because triage is a dynamic process, the victim would be re-evaluated and assigned to another group.

For the system to work, medical workers should always know which victim is triaged and which not. START uses a colour system to indicate which patient belongs to which group (see table 2.1).

Table 2.1: Triage classes within the START triage procedure.

Minor
Delayed
Immediate
Deceased

Every victim that is triaged receives a marker (or tag) to indicate which triage group this victim should be in. Figure 2.1 shows some tags that are used within the START procedure.

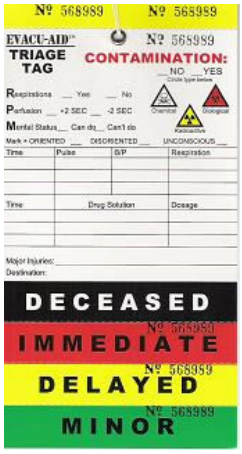


Fig. 2.1: Different ways to mark the victim belonging to a certain triage class.

From left to right: triage tag where the medical worker tears off the bottom part of the tag; triage tape that should be taped around the arm or leg of the victim; triage tag that the medical worker should fold so that the correct triage class comes up.

The new form of triage in the Netherlands is part of the Major Incident Medical Management and Support (MIMMS) [6]. This organisation is mostly active in Europe and focuses on every part of a Mass Casualty Incident (MCI). From communication inside a disaster region to the responsibilities of the different agencies working together. MIMMS uses a slightly different triage procedure then START, called the Mass Casualty Incident Management System (MCIMS) [54]. It has 6 classes,

Table 2.2: Triage classes within the MCIMS triage procedure.

Deceased	The victim has lost all life signs.
T1	The victim needs immediate treatment.
T2	The victim needs medical treatment within 6 hours.
T3 wounded	The victim is wounded, but the treatment can be postponed.
T3	The victim isn't wounded and needs to be removed from the area.
T4	The victim is in such dire conditions that further treatment will not help the recovery.

instead of 4, indicated in table 2.2.

The classes look a lot like the START procedure, except for making the distinction between victims that have a minor injury (T3 wounded) and victims that do not have any injury at all (T3). The newly introduced T4 class is only used when a disaster is too large for the fully scaled medical sector.

The triage card that is going to be used is displayed in figure 2.2. The medical worker should be able to fold the card so that the concluded triage class comes up and is visible. The tag should then be attached to the victim's arm or leg. To create a tag that indicates the victim belongs to the T4 class, The T1 card should come up with a little blue corner at the top-right. Figure 2.3 shows the end result of this class. Also the T3 wounded class is represented with a little orange corner in front of the green T3 card.

The card also contains all the information of the victim relevant for further treatment. These fields will be described in section 2.2.2.

2.2 - Triage procedure

Triage in a Mass Casualty Incident is done in two places.

The first place to triage is on the site itself. This is called the SIEVE. Although the faster this process starts the more people could be helped, the safety of medical personnel is assured first. Whenever the disaster area (or part of it) is accessible for the medical staff, the search for casualties begins.

Fig. 2.2: Front and back of the Dutch triage card.
The deceased class is a separate card.

Fig. 2.3:
Indication of the
T4 triage class.

The second place to triage is in the treatment area. This is called the SORT. Here are more supplies and resources to treat major injuries of victims that are brought in. Usually, the treatment area is for victims that were triaged T1 and T2 on the site. The victims are re-triaged and treated so that vital signs are as stable as possible for transportation. Afterwards the victims are transported to nearby hospitals according to the given triage class.

In the triage on site, a different strategy is needed than in the treatment area. Therefore different triage procedures are used for both locations. Since there are more technical resources in the treatment area, a more accurate profile of the victim can be created, while inside the disaster area it is crucial to get a clear view of who to get to the treatment area first.

2.2.1 - SIEVE

Also known as the primary triage, the SIEVE is done inside the disaster area. This has to happen as fast as possible. Accuracy is therefore of less importance, since the result can easily be corrected in the secondary triage.

Whenever medical workers start the SIEVE they first direct everybody they see who can walk to the nearest exit. This is very important, especially where the medical staff have to work in tight spaces (e.g. a plane crash, bus accident, etc.). The rule of thumb here is that for every medical worker that enters the site, one victim has to leave. People that can walk automatically get triaged in the T3 class. If there are any visible injuries or the triagist thinks the patient needs further analysis, the victim is given the card marking the T3 wounded class. Every casualty should leave the disaster region with a triage tag, so this can later be used in the secondary triage.

The most important parameters to monitor in case of making a fast judgement are the vital signs of the human body:

- Respiration,
- Pulse,
- Mental status.

In the SIEVE, only the first two parameters are used to come to a conclusion. The order of checking is given by the first 3 letters of the alphabet:

- **A**irway – Make sure the victim is breathing. Is the airway obstructed or free? Try to clear the airway when obstructed.
- **B**reathing – Record the breathing rate of the victim. Count the number of inhales for 15 seconds and multiply that number by 4 to get a respiratory rate per minute.
- **C**irculation – Record the pulse of the victim. Count the number of heartbeats for 15 seconds and multiply that number by 4 to get a heart rate per minute. Another way to measure circulation is the Capillary Refill Time (CRT). This is done by pressing the nail bed of a finger until it becomes white and counting the number of seconds that pass between releasing the finger and the finger returning to its normal colour.

The CRT is the fastest way to measure the circulation of a victim, (approximately 7 seconds, versus the 15 seconds for the heart rate) but is dependent on the temperature of the patient. In cold weather it might take longer for the circulation to fill the top of the finger. Also when the visibility is limited the CRT is hard to check, while the heart rate can be felt at the wrist. Both cases are used, but each under its own conditions.

The procedure for assigning the correct triage class giving the measurements is given in the decision tree in figure 2.4.

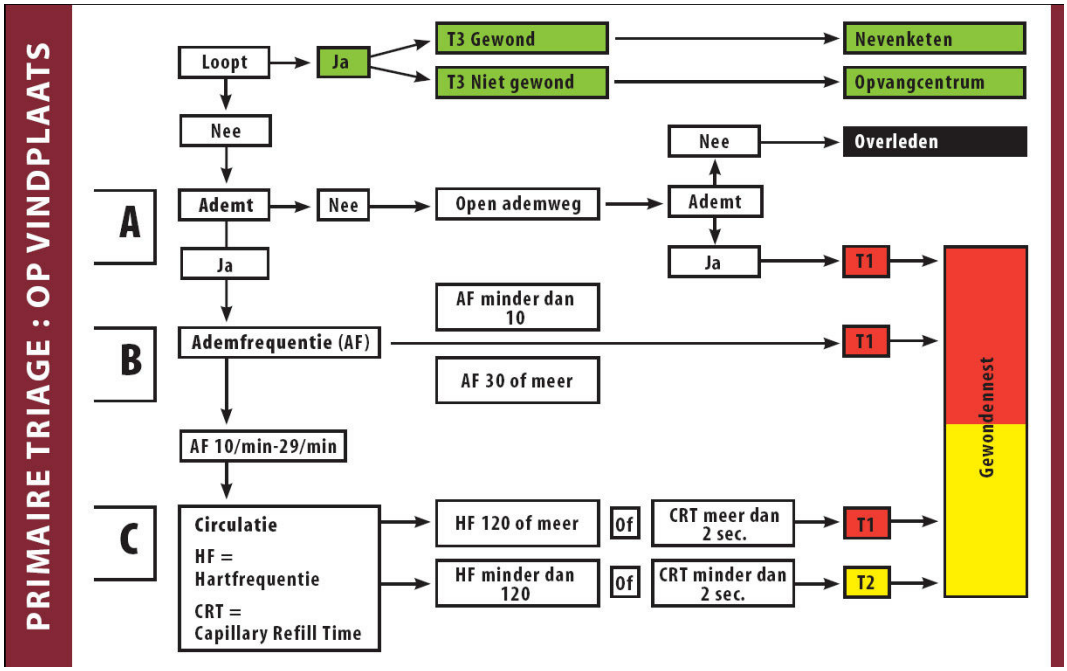


Fig. 2.4: Decision tree for the SIEVE. The ABC-order is applied as shown on the left.

Since the decision tree is in Dutch, an English translation in words is needed. The first thing to notice is the mobility of the victim. If the victim can walk on his own, he falls into the T3 or T3 wounded category. If this is not the case, check for the **A**irway. If the victim isn't breathing by himself, open the airway by means of a chin lift or jaw thrust. If the victim still isn't breathing, declare the victim as deceased. If the victim breathes after the chin lift, mark him/her as T1. If the victim was breathing on its own check the **B**reathing. If this is less than 10 or more than 30 per minute, classify the victim as T1. If it is between 10 and 29, check for the **C**irculation. If the heart rate is larger than 120 beats per minute or the CRT is more than 2 seconds, the victim should be classified as T1. If this is not the case, T2 is the correct triage class.

After this decision tree is traversed to one of the triage classes, the triagist folds the card to show the correct triage class and ties the card to the victim's arm or leg. Other medical personnel will notice the card and try to bring the victim to the corresponding treatment area as fast as needed.

2.2.2 - SORT

In the secondary triage, or SORT, patients are treated in a specially created treatment area. This could be a large building or one or more Snel Inzetbare Groep ter Medische Assistentie (translation: Fast Deployable Group for Medical Assistance; SIGMA) tents.

Fig. 2.5: Deploy of a SIGMA team.

Casualties from the disaster region are transported to this treatment area and the victims are re-triaged there. Instead of doing the same routine as in the SIEVE (ABC), the medical worker now adds an additional **D**isability to get a better and more accurate profile of the victim's injuries. The rest of the triage card is filled in, giving information concerning personal identification, treatment procedures and transportation directions.

The secondary triage procedure inside the treatment area is as follows:

- **A**irway – Make sure the victim is breathing.
- **B**reathing – Record the breathing rate of the victim, in minutes.
- **C**irculation – Record the systolic blood pressure (RR_{syst} in mmHg Riva-Rocci) of the victim. This is done by means of a regular stethoscope and sphygmo-manometer or Non-Invasive Blood Pressure (NIBP).
- **D**isability – Measure the conscience level of the victim. This is done with help from the Glasgow Coma Scale (GCS), which uses the best eye, motor and verbal response.

The end result of the secondary triage is the Triage Revised Trauma Score (TRTS). This score (ranging from 0 to 12) indicates how well the patient is considering vital signs and neurological damage. The better the score, the better the person is and the lesser the priority is to treat the victim. A score of 12 means the victim can be classified as T3 (wounded). If the TRTS is 11 the victim is triaged as T2. A score ranging from 10 to 1 is T1 and the patient is deceased when the TRTS is 0. In the case where the T4 class is used, this class is assigned to victims with a TRTS score between 3 and 0.

To come to the TRTS, a summation has to be made over three scores:

- The score of the Glasgow Coma Scale.
- The score of the breathing rate.
- The score of the systolic blood pressure.

In order to get a score of the GCS, the medical worker needs to obtain the GCS of the victim. This is done by adding three scores of reaction:

- The best possible eye opening of the patient.
- The best possible motor response of the patient.
- The best possible verbal response of the patient.

To get these reactions the medical worker needs to observe the victim closely when certain tasks are performed (e.g. asking questions, administering a painful stimulus to the patient). The medical worker classifies the best-observed reaction from the victim into the following groups with attached scores (see table 2.3 – 2.5).

Table 2.3: Score table for the best eye response.

Eyes	
4	Eyes are opened spontaneously.
3	Eyes are opened when spoken to the victim.
2	Eyes are opened in response to pain. This is done by applying pressure to the victim's fingernail bed or supraorbital (right above the victim's eye).
1	No eyes are opened, spontaneously nor in response to pain.

Table 2.4: Score table for the best motor response.

Motor	
6	Obeys commands (simple things that are asked by the triagist).
5	Localizes to pain. The free hand (where no pressure is applied to) should cross the mid-line of the body.
4	Withdrawal to pain. The victim pulls the body part away from the painful stimuli.
3	Flexion to pain. After pain stimuli, the victim's hand will move slowly towards the chest, with adduction in the shoulder and hyper flexion in the wrist (see figure 2.6).
2	Extension to pain. After pain stimuli, the victim's arm will be stretched, often with endorotation in the shoulder, pronation in the lower arm & hyper flexion in the wrist (see figure 2.6).
1	No motor response.

Table 2.5: Score table for the best verbal response.

Verbal	
5	Oriented. The patient knows his name, age, the year, month, etc.
4	Confused. The patient responds to questions but there is some confusion.
3	Incorrect word use. Random speech, but no conversation.
2	Incomprehensible sounds. No words.
1	No verbal response.

Once these scores are noted, the GCS is obtained by adding these scores together. To obtain the earlier mentioned scores the three conversion tables (see table 2.6 – 2.8) are used.

Adding these three scores together will give us the TRTS, and an indication of which triage class the victim is during the SORT. Instead of mesmerizing all the scores and conversion tables, the triage card has its own section where the medical worker can write down the observations. This section is shown in figure 2.7 and has multiple columns intended for the multiple times this triage can be done over time.

After filling in the rest of the card (noting personal identifiable information and extra information about allergies etc., location of injuries, treatment the patient had & transportation directions) the medical worker folds the card to show the correct triage class and attaches the tag to the victim's arm or leg.

Table 2.6: Conversion table to a GCS score.

GCS score	GCS
4	13 – 15
3	9 – 12
2	6 – 8
1	4 – 5
0	3

Table 2.7: Conversion table to a breathing rate score.

Breathing score	Breathing rate
4	10 – 29
3	> 29
2	6 – 9
1	1 – 5
0	0

Table 2.8: Conversion table to a blood pressure score.

RR score	RR_{syst}
4	>= 90
3	76 – 89
2	50 – 75
1	1 – 49
0	0

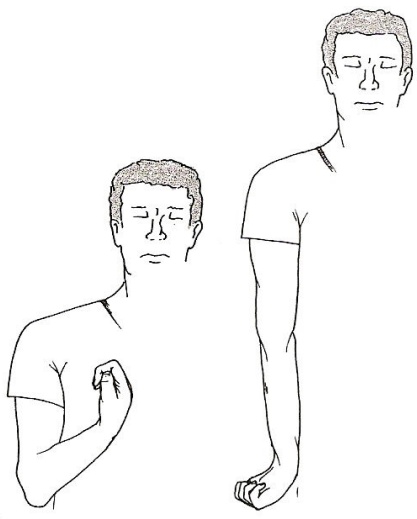


Fig. 2.6: Difference between flexion (left) and extension (right) to pain.

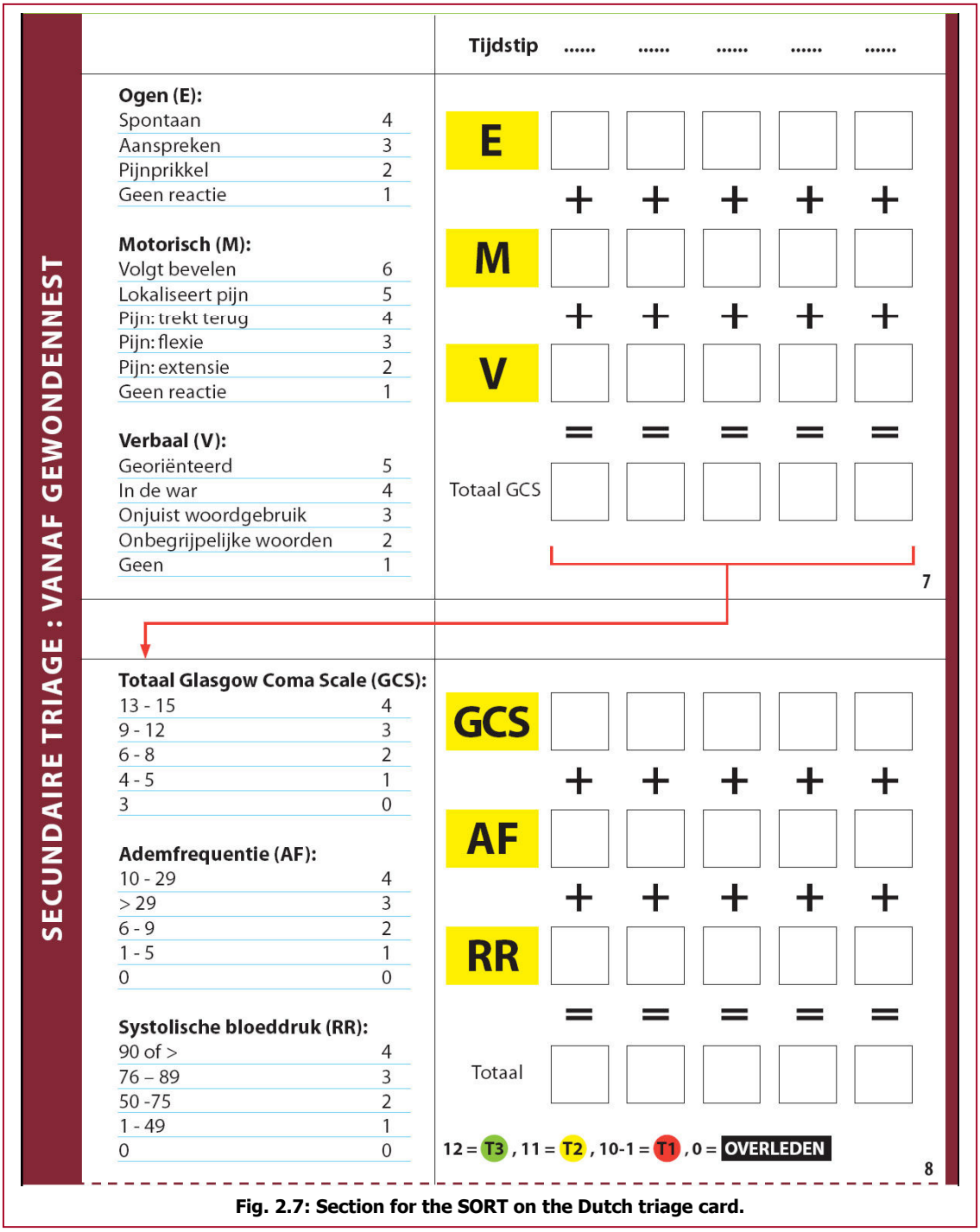


Fig. 2.7: Section for the SORT on the Dutch triage card.

2.3 - Summary

Triage is the process of classifying victims in a Mass Casualty Incident in groups of urgency. With this triage procedure, medical personnel can do the best possible for the most casualties. A wide range of procedures is used throughout the world.

In the Netherlands, a new triage system is going to be introduced in the beginning of 2009. This new triage system is using the MCIMS triage procedure, classifying the victims in 6 distinct classes: T1, T2, T3 wounded, T3, T4 and Deceased. These classes are ordered starting with the group that needs immediate treatment and ending with the collection of the deceased victims.

Since triage is a dynamic process, victims are triaged multiple times throughout their transportation from the disaster area to an appropriate hospital or other treatment centre. The procedure itself should take no longer than 30 to 60 seconds.

Two different procedures are performed for the distinct locations in a disaster: The SIEVE is used for triaging victims at the disaster site; the SORT is used in the nearby located treatment area.

The main purpose of the SIEVE is to get a rough idea of how many victims there are and in which group they belong. Medical personnel use a decision tree that is only concerned with the vital signs of the victim (ABC). The classification is done as quickly as possible to get the victims to a nearby treatment area.

In the treatment area, more medical equipment is present to actually treat a victim. But this treatment is only to stabilize the victim for transportation to a hospital. In this area, the SORT triage procedure is performed to get a more accurate description of the victim's injuries (ABCD).

Chapter 3 - Serious Games

There is no clear definition for serious games. Susi, Johannesson & Backlund [31] proclaim "there seems to be as many definitions available as there are actors involved". In their article multiple definitions from multiple sources are stated and they found that "most agree on a core meaning that serious games are (digital) games used for purposes other than mere entertainment".

The video and computer game industry has seen its sales steadily growing every year (see figure 3.1 and 3.2 for some statistics of the United States of America only). According to [22] 67% of the American heads of households played computer or video games in 2007 and 33% of the homes in America had a video game console.

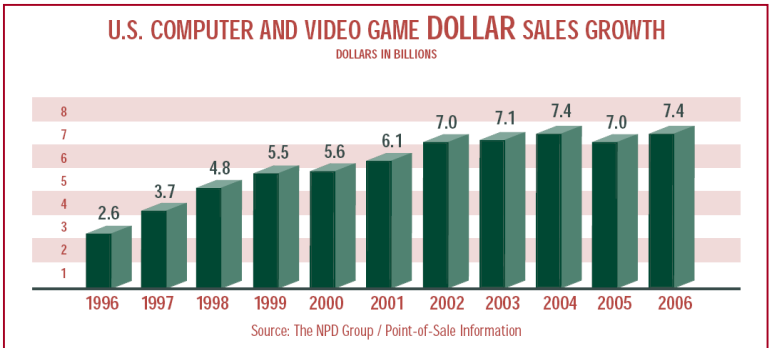


Fig. 3.1: U.S. computer and video game dollar sales growth.

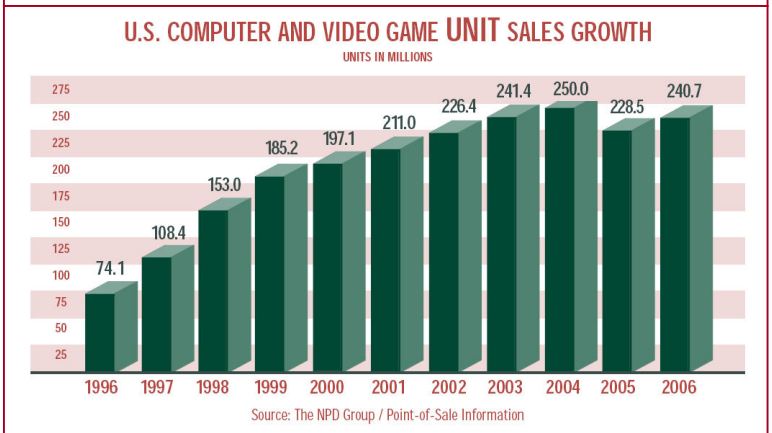


Fig. 3.2: U.S. computer and video game unit sales growth.

In the Netherlands, 41% of 536 questioned children between 8 and 18 years old say they like to play digital games, while 56% really likes this form of entertainment [29]. 59% of those children play digital games every day and 31% play several times a week. The most used medium for playing these digital games is the PC (91%). 43% of the children say they often play on a console like the PlayStation (1 or 2), Nintendo (64, GameCube) or Xbox.

Since practically every child has played one or more computer or video games, it is very useful to use this medium as an educational tool in class rooms, or simply at home on your own PC or game console.

Although there are numerous serious games on the market (some with a very high quality), the serious game business is small compared to the game industry that creates games for entertainment purposes. A lot of research and development needs to be done in order to get the correct formula for creating games with a purpose to educate or create awareness around a certain topic. Only then can games have a more educational value for the player.

This concept of using games for educational use has already been done a decade ago (in the late 1990s). Under the name of Edutainment, these games were mainly focused on learning a preschool topic like arithmetic, science, reading, etc. Unfortunately, because the focus of such games was less (if not) on how to make the program interesting to play, these games were referred to as "skill and drill" games. Attention was quickly lost and it seems that educational games weren't a feasible step to proceed with.

Then, in 2002, the Serious Game Initiative [67] started the introduction of serious games. This initiative has the goal "to help usher in a new series of policy education, exploration, and management tools utilizing state of the art computer game designs, technologies, and development skills. As part of that goal the Serious Games Initiative also plays a greater role in helping to organize and accelerate the adoption of computer games for a variety of challenges facing the world today." This new rise of games for educational means let to an explosion of new ideas and research. Nowadays, serious games are even part of the Game Developers Conference [44], called the Serious Games Summit [68].

Combining all that can be used from the entertainment business of digital games with pedagogy and learning strategies turns out to be a major improvement on the results of such serious games. It can be even so that games with an entertaining purpose can also be used for a more 'serious' purpose, without changing the game itself [14][15]. With serious games it is also possible to place a user inside a potentially dangerous environment without physically experiencing the consequences of ones actions. As long as the player of the game has the idea that actions in the game are just as important as in real-life, the player can learn from its mistakes. In [23] Ben Sawyer (co-director of the Serious Games Initiative) states "We are trying to be better than books or tapes. An [interactive] trainer won't let you say, 'Your guy died, start over'. He'll say, 'Your guy died. That's a big deal. What are you going to do about it in real time?'".

The serious aspect of such phrases by Sawyer might let the reader think serious gaming isn't about the fun anymore. This is not the case. Although the focus point of the game has shifted from entertaining to education, marketing, awareness, etc. this doesn't mean entertainment is out of the picture. The player has to be drawn to the game in order to play the game more often (where more learning can be done). To get the player's attention in even finishing the game, entertainment and 'fun' can be a very good strategy. But fun isn't the only way of immersing people in the game. Competition, personal feedback, character evolving, sense of a real-life situation, and many more tactics can be applied in order to draw the attention of the player into the game (see [17][20] for more information on immersion in games).

Because the above-mentioned general definition only refers to serious games as games without entertainment as main purpose, there are a lot of categories within the serious game genre. Games that advertise for a certain product are in this sense also serious games. Bringing awareness concerning a certain topic in the world via a game (Darfur is Dying [39] is a good example of this) is also a form of serious gaming. So education is not the only goal serious games can have.

With so many forms of serious games, there are a lot of areas in which serious games can operate. Michael and Chen in [9] used 6 groups to categorize serious game areas. Other (similar) groupings are available (see [31]). This thesis uses the 6 groups by Michael and Chan, because the author thinks these are the core segments of extensive use of serious games (also in the Netherlands).

The application areas of serious games are:

- **Military games** – even before the concept of serious games, the military used simulators and games to train soldiers, pilots, officers, etc. Nowadays the military still plays a large role in the creation of serious games as training tools.
- **Government games** – this category concerns all other governmental departments. Ranging from national to regional importance and scale, serious games in this group focus on for instance fire fighter training, city planning, public health etc.
- **Educational games** – these serious games are focused on children and students for teaching all kinds of topics that are taught at schools.
- **Corporate games** – for big corporations, serious games can be an attractive form of teaching people to do certain jobs or tasks within a job.
- **Healthcare games** – games that are concerned with the health of the individual. This can range from awareness of certain diseases to staying fit by doing exercises with a game as trainer.
- **Political, Religious & Art games** – this group contains mostly games that should create a certain awareness of some major topic.

Of course, not all serious games can be distinctly categorized into one of these areas. A mix can very well occur, for instance with a game that let the user do some exercises to stay fit. This game could also be applied in the business section to prevent certain types of Repetitive Strain Injury.

3.1 - Examples

3.1.1 - America's Army

One of the best serious games that are nowadays available is called America's Army [34]. In an article concerning this game (see [30]) the game is simply described as follows: "America's Army lets users play soldier online, band together with other Internet warriors and battle enemies in detailed 10-minute scenarios that the Army says are more realistic than any other game."

Fig. 3.3: Weapon familiarization in one of the training courses of America's Army.

Fig. 3.4: Medical training where the player has to answer some questions concerning medical treatment.

Fig. 3.5: Actual combat mission played online with several missions.

This game was created by the American army to promote a job at the military to adolescents. Since its release in 2002 the serious game has 9,253,300 registered users, with 3,880 users newly registered this month [34]. The secret behind this many users, is that the game is free to download from the website and offers a good quality multiplayer game where players team up against each other to win the match.

Before the player is allowed to play the online game, he/she must first go through several training courses. These courses bring more information on how to handle certain weapons, giving first aid to wounded soldiers, staying unnoticed in a hostile environment, etc. With these courses the player gets a deeper look in how it's like to be in the army and how to become a soldier. After the courses are completed with success, the player can enter online missions with and against other registered players.

The development of the game hasn't stood still after the release. Recently, the game America's Army: True Soldiers is available for the Xbox 360, giving more gamers a change of playing the American soldier. Actual soldiers of the United States army are also using the game to practise certain missions with each other before putting their knowledge to use in the actual battlefield.

3.1.2 - Hazmat: Hotzone

As the site of the game [47] proclaims itself, Hazmat: Hotzone is "an instructor-based simulation that uses video game technology to train first responders about how to respond to hazardous materials emergencies". The game (or simulation) is created by the Entertainment Technology Center at Carnegie Mellon University and is now under further development by Sim Ops Studios [72].

The game requires for an instructor to create a certain disaster. These disasters are mostly focused on toxic hazardous materials. Once the instructor created the situation, a group of first responders start playing the game. The goal of the game is to locate and contain the source of the hazardous material. Secondary goals can be to bring victims to safe areas where they can be treated.

Fig. 3.6: Fire fighter in Hazmat: Hotzone finds victims convulsing on the ground.

Eddie Zagajesky, a 19-year fire department veteran says in [23] that “once you’re in [Hazmat: Hotzone], you feel the real situation ... You can only show and tell so much, but if you can put [first responders] in the real situation, choose your equipment, help your victims, then maybe when it comes to the real thing, you’ll make decisions that much faster”.

3.1.3 - Global Conflict: Palestine

Global Conflict: Palestine [45], created by the Danish game company Serious Game Interactive, is concerned with the Palestinian-Israeli conflict in the Middle East. The game has its primary focus on school children following a class of history or politics. It gives a better understanding of how difficult the conflict is to solve and how two competing groups think about this problem.

Fig. 3.7: Screenshot from Global Conflict: Palestine.

In the game, the player plays a news reporter. In order to write good articles, the player must question certain people in the region and experience the conflict from up close (e.g. the problems occurring at a checkpoint to enter a city). The player is able to choose if he/she wants to write for a Palestinian, Israeli or International newspaper. All papers want something else reported about a certain event and writing for one newspaper could mean that certain sources are more willing to talk to you then others.

This perspective of the game gives people a good view of the difficulties of the conflict in the Middle East. For High school students, this game might introduce new views on the topic and getting them aware of what they might do in certain situations.

3.1.4 - Serious Gordon

Food safety in the kitchen of a restaurant is a very important aspect of working in this sector. Serious Gordon is a serious game that teaches basic principles of this food safety. The player starts out on a new day of work and gets an explanation of the chef what he needs to do. These tasks are all related to personal hygiene in the kitchen and the conservation of certain food types in certain areas of the kitchen.

Unfortunately, there is no site where one can play Serious Gordon. There is however a movie on YouTube where the reader can see how the game looks like and what tasks are performed in order to secure food safety [70]. Both [26] and [69] shows the development of this game by students of the Dublin Institute of Technology.

3.1.5 - Dance Dance Revolution

Although this game was meant to be entertaining, Dance Dance Revolution also has a serious game aspect. The meaning of the game is to step on the right buttons on a dance mat positioned on the floor. What button to push is displayed on a (TV) screen and this is all done on the rhythm of a song chosen by the player.

Fig. 3.8: People playing the arcade version of Dance Dance Revolution.

Although this game was first introduced in arcade centres, Konami (the creators of the game) soon realized this game was very suitable for game consoles like the PlayStation or the Xbox. Nowadays, several dozen game versions of Dance Dance Revolution are produced with new songs and faster rhythms and the game itself is available for every gaming console and PC. The game has a thriving online community [37] and movies where players show off their skills on extremely difficult songs [38].

Because this game is so popular amongst the youth, children keep playing in order to get better. While they are playing they are also exercising, which is of course very good for their condition. So by playing an amusing game, children can get the necessary exercise.

3.1.6 - September 12th

"September 12, does not behave like a regular video game. It does not try to grab you; it's not even particularly enjoyable. It exists purely to intrigue you long enough so you poke around and figure out the underlying argument: an op-ed composed not of words but of action." [32].

In September 12th [66] the player has the option to bomb an Arabic town with several terrorists running around. Once a rocket is launched, the player realizes that the impact of the blast also killed several civilians. After some while the relatives find the casualties and in rage also become terrorists. The player is confronted with the question if bombing is indeed a good thing to do.

Fig. 3.9: Screenshot of September 12th.

This simple game (created by newsgaming.com [59]) serves only as a statement to think about what we are doing. It's trying to bring awareness of how we are dealing with terrorists and has started many discussions among players and politicians [32].

3.2 - Concerning triage

Nowadays in the Netherlands, triage is learned from a theory book. At the NIFV, training is done by means of questionnaires describing the victim's condition. Based on this textual explanation, the student is asked to classify the patient into the triage classes described in section 2.1. Another way for training the triage methodology is by staging an actual disaster. Landelijke Opleiding Tot Uitbeelding van Slachtoffers (translation: National Training for Acting as Victims; LOTUS) victims are used to create lifelike situations where triage can be applied. Although this doesn't actually test the knowledge of triage the student has (since individual actions aren't evaluated), it is a way to apply the learned lessons to a close-to-real-life situation.

Serious gaming can provide a good way that is in between the questionnaires and the staging of a disaster. Within the game, the student is closely monitored as to how he/she is doing personally. While also providing a virtual disaster area, players are confronted with the pressure that they have to deal with in a real disaster. Instead of reading the symptoms of the victim from a sheet, the serious game can provide a multi-modal experience where the player has to discover the condition for him/herself. And since a serious game can easily be played nearly anywhere anytime with little cost, this form of learning could be very well used for classes in triage procedures.

Fig. 3.10: Fire fighter performing triage in UnrealTriage.

Of course, a serious game concerning triage cannot be the only medium of training and wont certainly remove the need of instructors. Actually being in a disaster region triggers much more senses and feelings than sitting behind a computer looking at a screen and pressing buttons. It is still very hard for a computer to teach certain topics to a human student, therefore instructors are needed to discuss the actions made in the game even further. An evaluation after the game has been played is needed to discuss progress and any questions the student has. This serious game has not the intention of replacing the entire learning system, but must be viewed as an additional educational tool instructors can use in their curriculum.

In America a serious game about triage has already been created. The game is called UnrealTriage and in [28] an explanation is giving to how this game was created. The main purpose was to train fire fighters in dealing with a small plane crash with 30 victims. The fire fighter both has to extinguish the fire and triage the victims in the area. In figure 3.10, the fire fighter reads the vital signs from a Heads Up Display (HUD) and presses the key 1, 2, 3 or 4 to assign the START triage class (see section 2.1 for the START triage procedure).

Note that this game only trains the SIEVE part of the triage procedure. Fire fighters are only part of the SIEVE procedure since they are also in the disaster area. The secondary part of the triage (the SORT) is only done by medical personnel.

3.3 - Summary

With the rise of the PC and console game industry, more and more people are confronted with digital games. While this medium is heavily reliant on entertaining the user, it could also be used for other purposes. Serious games are digital games that try to use game elements from entertaining (commercial off-the-shelf) games to educate, train and/or inform the player of a certain topic.

Although numerous serious games are created (described in section 3.1 and a list of games can be found in appendix A), a lot of research and development still needs to be done in this area. Finding the correct mix between an entertaining game and an instructive learning program still requires a lot of attention in this field.

Concerning triage, training of this topic is nowadays done by means of questionnaires and staging a disaster (this is usually part of a large-scale training operation for all disciplines of disaster management). Specially trained LOTUS victims mimic injured people, creating a close-to-real-life situation for a medical worker to perform triage. Serious games can however provide the important advantages from these two training forms, while reducing the disadvantages of these systems.

Serious games can provide a user-specific training module, evaluating the user instead of the overall process. It also creates a virtual world the player can walk around in, search for victims, deal with unexpected events, etc. providing a sense of realism, without the costs of arranging a full-scale disaster scenario. Of course, serious games should not be the only form of evaluation or training. Sitting behind a desk with a computer simulating a disaster scenario doesn't learn the actual performance of certain actions in real-life. But serious games can be a very helpful tool to train routines, reasoning patterns and factual knowledge also crucial for the medical worker in such situations.

Chapter 4 - Intelligent Tutoring Systems

In all the serious games described in the previous chapter, focus was more on the transmission of information. If America's Army and Hazmat: Hotzone are seen as training tools an instructor or trainer is required to provide the learning material. The instructor creates the problem for the players of the serious game to solve. Also, evaluation of those players tends to be done by the instructor himself.

In the Serious Gordon game, some evaluation is presented by the game itself. In the video (see [70]) the player makes a mistake by placing a product in a wrong conservation cell. The chef inspects these cells and alerts that the user made an error, asking the player to do it again. Unfortunately, this is the only form of evaluation that is seen on the video (the rest of the actions are performed correctly).

The form of evaluation in Serious Gordon is checking whether the player did something wrong or not. The reasoning about why a certain error was made and what underlying thoughts the user had, isn't relevant for this case. However, in a lot of topics this can provide a better understanding of why the mistake is made. Pinning down the arguments of this reasoning might prove useful in explaining what went wrong.

An Intelligent Tutoring Systems (ITS) will take the forms of evaluation seen in the previous sections a step higher. Its main goal is to mimic or simulate a human teacher or instructor into a computerized form. Having expert knowledge of a certain topic, techniques of how to present such knowledge to the user and probing methods to infer the student's understanding are crucial for an ITS to function properly.

Of course, in most cases the human instructor cannot be replaced by an ITS and this is not the underlying thought of such a system. It can work together with a trainer or instructor to make the task of teaching students easier and more thorough. An ITS will not be able to respond to questions or other actions from the student in a way its human 'colleague' does. This seems even impossible at the time being. But these systems can be a very useful tool in classrooms or courses, providing personal treatment, extensive logging capabilities and the global information concerning a topic.

4.1 - Framework

An ITS has a particular architecture in order to provide the necessary functionality. Although previously designed systems use different names, most ITSs use three common models. While

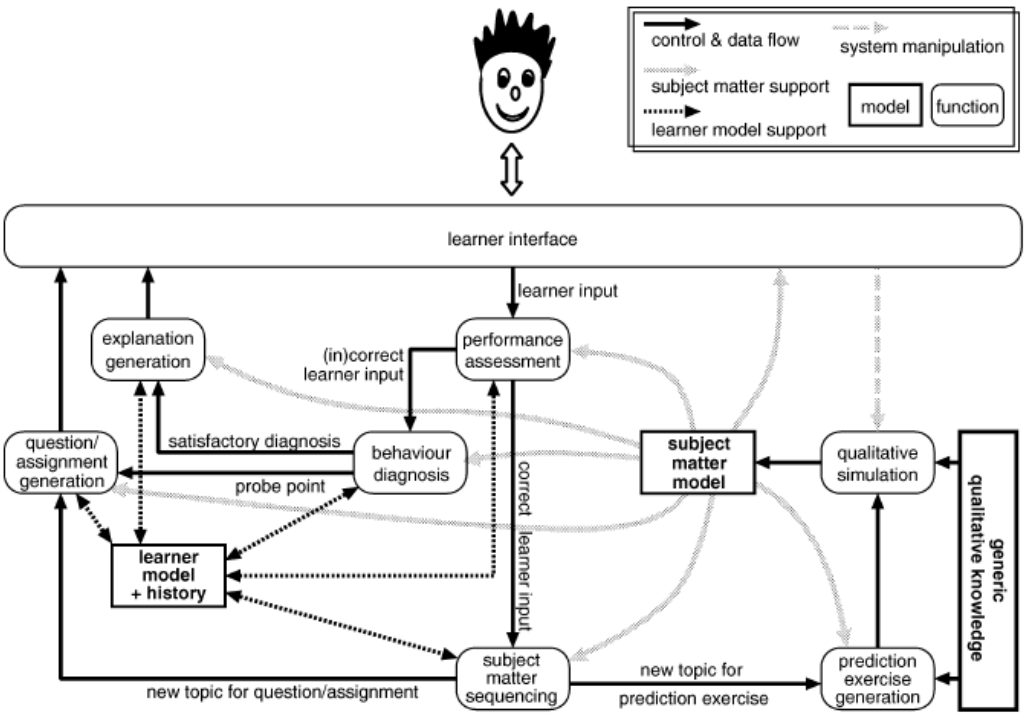


Fig. 4.1: Global architecture of a specific example of an Intelligent Tutoring System.

functionality changes regarding specific purposes or topics, these three models are crucial for the ITS. In figure 14.1, an example framework is provided for an ITS where these three models are also present. Please note that this is only an example of an ITS.

4.1.1 - Models

The models in the figure are marked with bold-lined rectangles. These models contain information about a certain subject relevant for knowledge communication. Other modules (here called functions) will use this information to find errors, explain difficulties, store user actions, create a learning strategy, etc.

Student model

The student model (displayed in the figure as learner model) is the model that keeps track of the acquired knowledge of the user. To give information to the user doesn't necessarily mean that the information is interpretated correctly. Lots of errors can occur, leading to erroneously created knowledge. This model can be used to compare the user's knowledge to that of an expert to see if certain information is missing and if everything is understood correctly.

The mentioned purposes are very locally (e.g. related to one session in the game). It is concerned with storing what information the student has mastered. The student model can also keep track of more global features of the user, like the name and the number of errors made during a session. These global features can be used when the same user restarts the application, or when a supervisor wants to know if there is progress in the student's work.

Didactical model

The didactical model (or subject matter model in figure 4.1) is the model that keeps track of the conversation. It is the main drive of the communication part. The model contains rules of how to engage in a social conversation and how to present information didactically correct.

This model provides the teacher's reasoning as someone who tries to explain something to a student. Learning strategies and alternative sideways to explain a topic are defined in this model.

Expert model

The expert model (described as generic qualitative model in the figure) contains all the expert knowledge on the subject to be taught. This can be seen as the teacher's facts and rules concerning the topic.

Usually the expert model is created by an expert on the subject in a description that suits the topic. Knowledge about arithmetic could very well be presented in facts (e.g. numbers, variables, etc.) and rules that apply to the facts (e.g. addition, subtraction, etc.). Knowledge concerning the evolution of animals might be better represented in a hierarchical network with some properties per node.

4.1.2 - Flow

The functions shown in figure 4.1 are marked by the round-edged rectangles. These functions process the information retrieved from the above-mentioned models. The results of such functions are reported back to the model and/or send over to other functions for further reasoning.

The architecture depicted in figure 4.1 is from an article creating an ITS with help from a simulation based on qualitative models [24]. Although the functions are described for this specific ITS only, the framework gives a good sense of how an ITS should work.

As you can see, there are two main flows of interaction between the user and the ITS. These two flows both have different purposes and it is up to the system to choose which flow it will apply given certain input from the user.

The first flow will be from the interface, to performance assessment, to subject matter sequencing, to question/assignment generation, back to the interface. In this flow, the user will learn something new about the specific domain. This can range from a textual explanation on screen describing a certain subtopic, or a complete video presentation with animations and sound. However, between those chunks of information the user and the ITS are allowed to ask questions to one another. The user

can for instance ask the system to elaborate further on a certain subtopic, or can give an indication the user is finding it hard to understand all the information. The ITS can probe the user to see if all the information provided is actually understood correctly.

Once a student gives a wrong answer or doesn't understand certain aspects of the domain, the ITS will end up in the second flow: from learner interface, to performance assessment, to behaviour diagnosis, to question or explanation generation. The flow can choose to ask a follow up question to the user in order to get a better view on what is wrong. When the system knows the misconception, it can explain the subject matter in more detail by the explanation generation function.

The two flows of the system will be used interchangeable. For instance, a student can follow an instruction by the system, gaining new knowledge. When the system tries to understand whether the student has received all the given knowledge correctly there is a misconception about a certain topic. The system then tries to pinpoint the bug by asking the student questions. When the system finds the bug, it will explain the misconception and the solution, and the system resumes the session.

4.2 - Examples

4.2.1 - Wusor

In 1972 a text-game called *Hunt the Wumpus* was created. This game features the player in a cave system. Every cave in the map has two to four other caves connected to it. In one of those caves the wumpus lays sleeping. The wumpus is a creature that will eat you whenever you enter its cave. The goal of the game is to use your one arrow to shoot the wumpus. In order to actually hit the wumpus, the player needs to be in one of the adjacent caves shooting in the direction of the wumpus' cave.

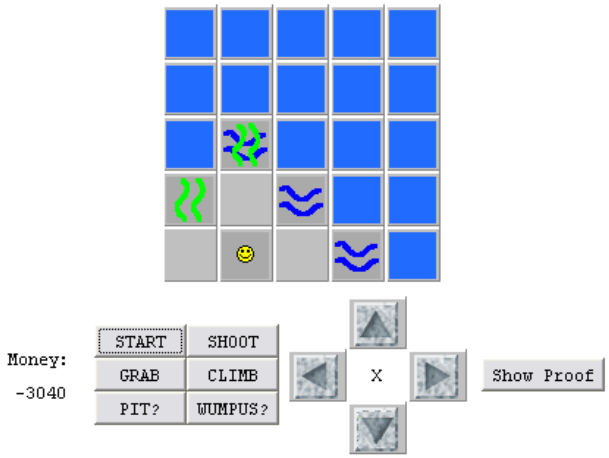


Fig. 4.2: Screenshot of the Wumpus World game. The yellow smiley is the player, the blue waves are indications of nearby pits and the green fumes are indications of the sleeping wumpus. Blue squares represented uncharted territories.

The user is able to smell the wumpus whenever he/she is in one of the adjacent caves. The player has to think logically in order to located the wumpus cave and win the game. Additional hazards can also be introduced, like pits the player can fall in or bats that drop the player in a cave where there is another bat, a pit or a wumpus. Pits cover a complete cave and can be spotted by feeling a draft in the surrounding caves; bats are recognized by a squeaking sound in the neighbouring caves. A (simplified) web-based version of the game can be played with a Graphical User Interface and an additional option to win: to grab the gold in one of the caves [87].

Although this was at first a fun text-based game for the early computers, Brain Carr created it into a ITS that teaches children some elementary form of logic, called Wusor (the improved Wusor II is described in [19]). This Wumpus Advisor asks questions and advises the player to take the cave with the fewest risk of losing.

4.2.2 - LISP Tutor

LISP tutor [12] is an Intelligent Tutoring System that allows the user to perform exercises in order to learn the LISP programming language. This ITS is widely used by the Carnegie-Mellon University to research effects of an ITS on the student.

With help from undergraduate students, research was done on how effective an ITS is in learning a programming language. Results concluded that learning with a human tutor was the fastest way, but the LISP Tutor provided a better learning rate when opposed to learning from text.

A simplified version of the LISP Tutor, called LISP Tutor Jr. can be found at [52]. This version contains all the text and exercises the original LISP Tutor had, but lacks the sophisticated feedback and reasoning.

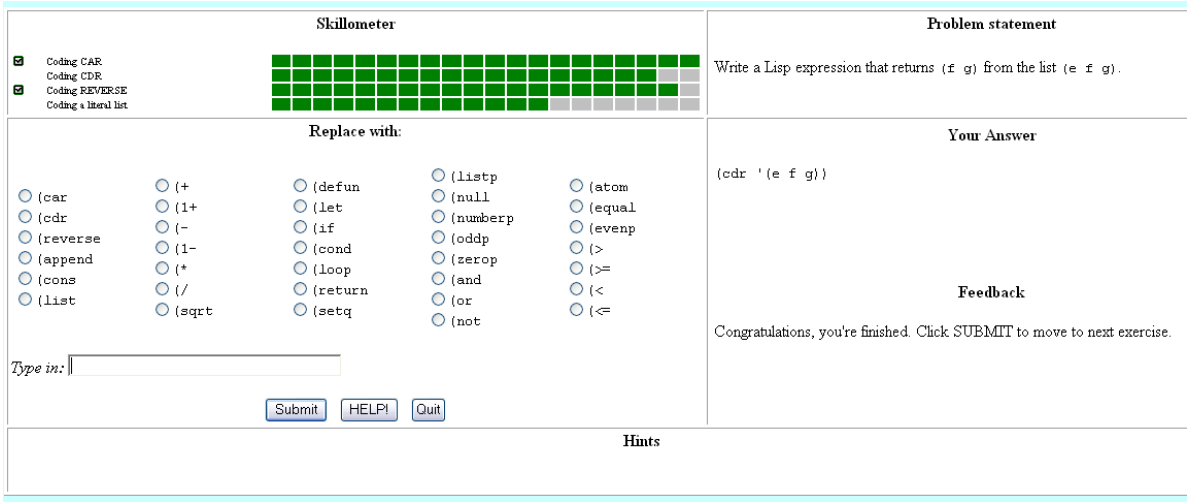


Fig. 4.3: Screenshot of the LISP Tutor Jr. On the right an exercise is presented the user must solve with LISP code (displayed on the left). Feedback is provided in the lower-right.

4.2.3 - Advanced Geometry Tutor

The Advanced Geometry Tutor [27] is another ITS with again a different topic to inform the user about. With this ITS students are stimulated to prove certain geometric theories in order to improve their reasoning. Unfortunately, an actual working version of this ITS cannot be found on the Internet.

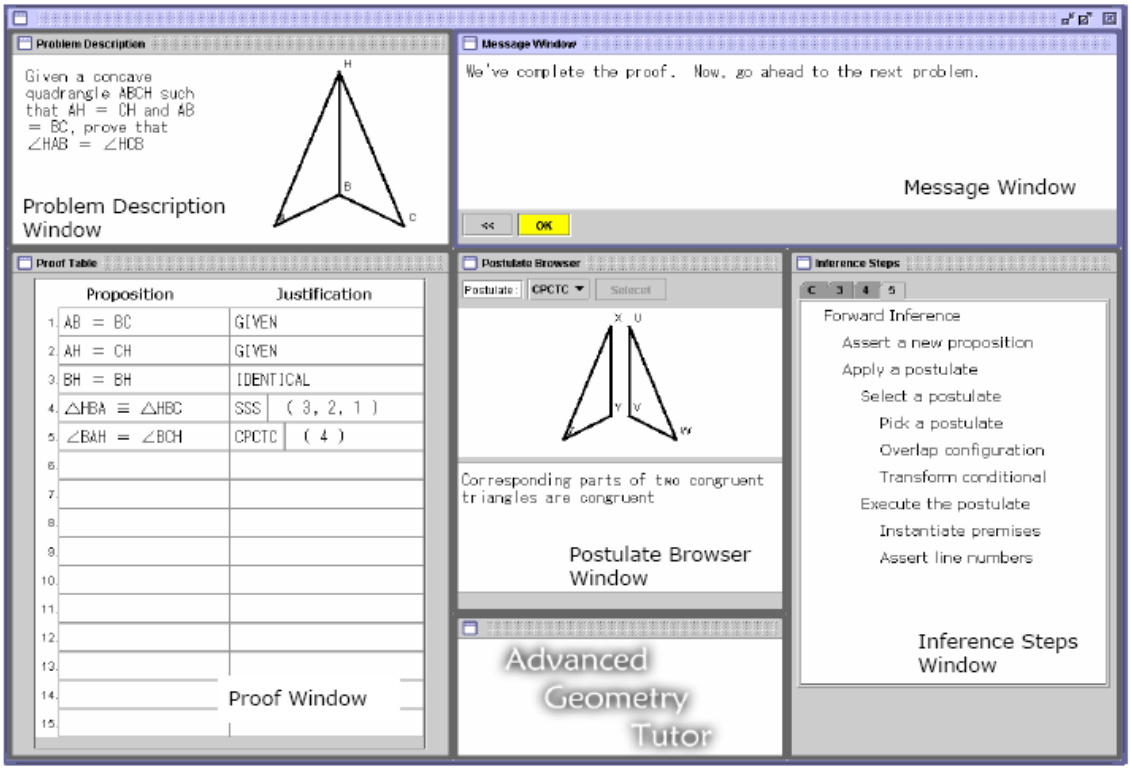


Fig. 4.4: Screenshot of the Advanced Geometry Tutor. A problem description is stated in the upper-left part of the screen. The user is trying to prove a certain geometric statement in the proof window (shown on the left) with help from given postulates (the centre screen).

4.3 - Summary

Intelligent Tutoring Systems are computer applications that try to mimic a human instructor or teacher. Instead of only indicating something went wrong, ITSs are more focused on the reasoning that lies behind the misconceptions or misinterpretations. This way, an ITS must not only have expert knowledge about a certain topic. It must also keep track of the student's progression through the topics and knowledge of the subject matter. Learning strategies are also needed to provide a good knowledge communication between computer and user.

While a lot of ITSs use different functionality and architectures, most of them use three main models to store information in: the Expert model, Didactical model and Student model. The expert model contains all the information concerning the topic to be taught. The didactical model describes rules on how to present this information to the user. The student model is the model describing all relevant information concerning the user.

As can be seen in section 4.2, a lot of topics can be discussed with Intelligent Tutoring Systems. The examples are mainly focused on reasoning patterns that are taught to the student. The triage procedure described in chapter 2 is also a new reasoning pattern that needs to be learned by medical worker. An ITS can very well be used for the training of the triage procedure as well as all other routines that need to be trained in disaster management. For this project, the ITS to be created needs to work together with the serious game content in order to perform its tasks. Part 3 of this project is dedicated to the design and implementation of this ITS.

Chapter 5 - Valve Source Engine

The creation of a serious game can actually be seen as a game project with additional features. The development of a Commercial Off-The-Shelf (COTS) game requires a lot of hard work [16] and it may take several years to create such a game with a complete team of specialized programmers and artists. Although it takes a long time to create such games, the product is a state-of-the-art game that uses the latest technologies on graphics, physics, sound, gameplay elements, etc.

For serious games this process can be simplified since the game doesn't need the most beautiful graphics and other state-of-the-art components. Moreover, it most definitely narrows down the target audience of the game since using the latest technologies also means that the player must have a computer or console that is capable of working with these technologies. If a serious game was to be created that should be played by 30 students in a classroom, creating an application that uses all the latest technology implies that 30 PC's with the latest hardware must be purchased. Usually there isn't enough budget at the client's side (e.g. a high school, a hospital, non-profit organisations) to make such an investment.

Serious game developers must spend their limited money and time in the best possible way to still create a stable game that also provides information to the player. One of the options the developer can choose is to take relevant components from earlier produced games. With the rise of tools to create user-generated content for games (maps, models, textures, etc) this option is a very attractive solution for the posed problem.

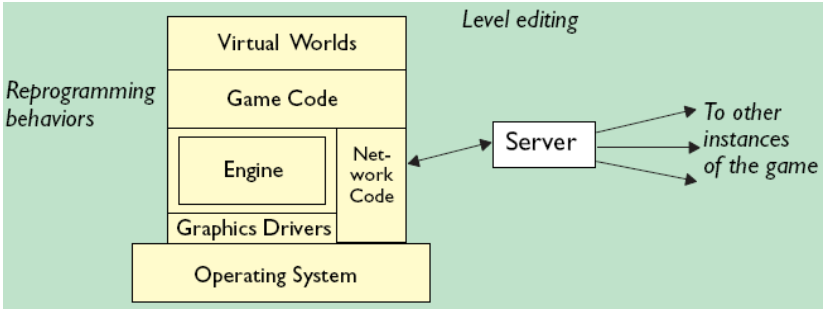


Fig. 5.1: Schematic overview of the game engine interacting with other parts of the system and the game. The game code and the virtual worlds are part of the game content.

More and more entertainment game studios tend to create their games with the concept of re-using essential elements like for example the render-pipeline (i.e. the functionality of displaying the full 3D world on screen) for later games. This concept splits up the low-level functionality of displaying entities on screen, emitting sound from the speakers, etc. from the actual game content like levels, weapons, story line, etc. The first component is referred to as the game engine, the latter as the game content. Figure 5.1 shows a schematic overview of a modular game structure with a game engine.

With this scheme, it is very easy to create a new game without implementing it from the base up. A development team simply has to update or replace the game content and build the new game with help from the existing game engine. This turns out to be very effective, as many game studios tend to follow this scheme nowadays [53].

The rise of this approach to game design also let to the open-mindedness of game studios allowing the player to use parts of the published game to create its own content [18]. With this idea the published game is more attractive to the consumer, since it offers a lot more than only the game. Additionally, the game studio can use good user-generated content (new types of weapons, levels, game concepts, etc.) in new games by contracting the developer of that new feature.

Also for the triage serious game, a game engine can alleviate the development. By using the pre-defined code libraries in a game engine, the project can focus more on the design and implementation of the serious game with the Intelligent Tutoring System. In order to create an ITS to the existing platform, the engine itself needs to be very modular and implemented so that the programmer can easily add self-created code.

The game studio Valve [82] (resided in Bellevue, Washington) also applied this scheme of a separate game engine to all its published COTS games. From the beginning, this company has created its own game engine by expanding from the QuakeWorld game engine [63]. Initially named GoldSource (the engine with which the popular game Half-Life was created, published in 1998), the

game engine got more and more evolved and is now called the Source engine. The Source engine is the driving force behind big game titles like Half-Life 2 (with Half-Life 2: Episode One and Half-Life 2: Episode Two), Portal & Team Fortress 2 (now all sold in the Orange Box [61] and winner of over 100 game awards). The Source engine is also licensed to other game developers, which have created games including SiN Episodes [75], The Ship [71] and Vampire: The Masquerade – Bloodlines [85].

Valve has chosen to make certain elements from the Source engine freely available for gamers who have bought one of the games in the Orange Box. The so-called Source Software Development Kit (Source SDK) provides the main tools for creating new game content like levels, weapons, models and game logic. The actual low-level functionality (e.g. the render pipeline, lighting, shaders, etc.) can be used, but not altered by the game developer. For game developers, this is enough to create a game with the engine. For others (e.g. programmers that want to change the render pipeline or delve deeper into the physics system) the Source SDK doesn't provide enough functionality.

The downloadable Source SDK has created a massive explosion of gamers teaming up to create a modification (i.e. mod) of one of the original games. Some even try to make a total conversion (a game that is only created with some basic functionality of the game engine) in order to create new gaming concepts and art.

5.1 - Features

Although the Source engine has evolved to a game engine that can produce state-of-the-art games, the flexibility of the system preserves the option to create games for older PC's. According to Valve, Half-Life 2 can run on a computer with a 1.2 GHz processor, 256 MB RAM and a DirectX 7 graphics card. The recommended system requirements are a 2.4 GHz processor, 512 MB RAM and a DirectX 9 graphics card. For the newer released games (like Portal, Team Fortress 2 and Half-Life 2: Episode Two) the minimum requirements have gone up to 1.7 GHz, 512 MB RAM and a DirectX 8 card [62].

But what does this engine provide? In [33][76] all the important features of the Source engine are listed.

Fig. 5.2: Screenshot of the game Half-Life 2.

Not only has the Source engine all the latest technology on shaders and lighting like support for the High Level Shader Language (HLSL) and High Dynamic Range (HDR) lighting, it also contains already defined shaders and lighting properties for the developer to use. With advanced tools like Faceposer and Valve's Modelviewer, the developer can create complex animations to characters and other objects that can be imported from the standard 3D model programs (e.g. 3D Studio Max, Maya & Cinema 4D, Milkshape 3D, etc.). With faceposer, the developer can create choreographed scenes for all the faces of Non-Playable Characters (NPCs) in the mod, making realistic facial animations that can be used in the game.

Another major feature within the Source engine is the implemented physics system. Because Half-Life 2 heavily relates to game elements with extensive use of physics (i.e. the game introduces a completely new weapon, called the gravity gun. This gun is able to pull certain objects towards the gun in order to shoot it in the direction the player wants) the physics system in the engine is very well implemented and calibrated.

The AI in the Source SDK is remarkably simple in structure and easy to use. Source has a lot of basic functionality implemented, like a path finding and navigation system, an AI sensory system and an event-based Input/Output system. The code of the Source SDK as well as the engine is written in

C++, and is suitable for both PC games and Xbox 360 games. With a modular structure and a well-defined hierarchy the Source engine is very easy to extend for programmers.

Audio and networking facilities are all implemented and usable for developers with the Source SDK. With surround sound and 3D spatialization, the player can really experience the game to the fullest extent.

Creating maps can easily be done with the Hammer editor, also provided with the Source SDK. Instead of only creating the geometry of the map (e.g. walls, staircases, buildings, etc), Hammer lets the designer place all entities in the same map. This makes it possible to create a level, or even a complete game with only one application.

5.2 - Mods

The Source SDK has provided many gamers with an opportunity to create a mod. Although there are many mods, only few will get a chance to be famous enough to build up a reputation in the mod community. A developer can choose to create a single-player or a multiplayer mod. A template of the original game is created, in which the developer can alter and create new code, models, sounds and levels.

5.2.1 - Single-player mods

Single-player mods can only be played by one player. The complete game runs on the PC of the player and no Internet connection is required. Half-Life 2 and its additional episodes is a single-player First-Person Shooter (FPS). This genre has the property that the player always perceives the game from the eyes of the leading role of the game (in this case Gordon Freeman). The overall goal of the game is not to get killed by all kinds of enemies by means of shooting those enemies with different weapons. In the case of Half-Life 2, the world has been taken over by aliens (called the Combine) and it's up to the player to help the few human rebels who try to liberate an important city.

Fig. 5.3: MINERVA: Metastasis.

The single-player mods that are created from this game usually expand this story with new insights and characters. Two of the most renowned single-player mods on the community are MINERVA [55] and Half-Life 2: Riot Act [46]. Both of these games practically use all the game elements in Half-Life 2, except for the levels, which where created by the developers. This makes the mod feel like an extension of the original game.

5.2.2 - Multiplayer mods

Fig. 5.4: Team Fortress 2.

Within the Source SDK, it's also possible to create a multiplayer game. This game can only be played over an Internet connection and with other people. A server runs a game where players can connect to. Depending on the game, the player can play in teams or all by himself and must score the most points, usually by killing other players although other goals also give points.

An example of a multiplayer game created with the Source engine is Team Fortress 2. In this game the player can play one out of 9 different classes with each their special abilities. Playing for a team (RED or BLU) the player must score points by occupying certain points in the level or retrieving a briefcase

located in the enemy's base. Since the release of this game by Valve, there are a lot of gamers that create new maps for Team Fortress 2 with the Source SDK, to be played at local game servers.

Total conversion multiplayer mods have also been created with the Source SDK. Two of the most influential mods created are Counter-Strike [36] and Day of Defeat [40]. Both of these mods were created with the old GoldSource engine by gamers.

In Counter-Strike, the players are put in two teams, the terrorists and the counter-terrorists. A player has one life per round and has to wait for the next round when killed. The goal is to annihilate the opposing team, or do a specific action before the time runs out. These actions would for instance be for the terrorists to plant a bomb at a certain area, or for the counter-terrorists to rescue some hostages. The team who has won the most rounds has won the game. The setting of the game was, instead of the science fiction of Half-Life and Half-Life 2, more realistic with all kinds of weapons from the real world.

Fig. 5.5: Screenshot of Counter-Strike: Source.

In Day of Defeat, the players of the game are also put in teams: Axis and Allies. The setting of the game is placed in a WWII scenario. Here, players have to fight each other to capture and occupy certain control points in the map. Again, the whole concept of Half-Life has been removed and a completely new game has been created on the engine.

When Valve decided to create the new Source engine, they've asked the makers of the two very popular multiplayer mods to create that same game with the new engine. The new Counter-Strike: Source and Day of Defeat: Source are now published by Valve with help from the original developers of the games.

5.2.3 - Serious mods

Using the Source engine for serious game purposes is a very attractive approach to take, and has already been done several times [21]. The Serious Gordon game, discussed in section 3.1.4 is a serious game mod of Half-Life 2. Most of the models (e.g. characters and the hands of the player) seen in the movie ([70]) are alterations of models used for Half-Life 2. [26] describes the changes that have been made to the original game to create Serious Gordon.

Another serious game that has been made with the Source SDK is DoomEd [43]. As a research project from DESQ and the University of Wolverhampton School of Education, this game tries to blend education with games. The player of the game is confronted with bio-terrorism and WWII chemical experimentation in a FPS. These mods show how well this game engine can be used to create serious games.

5.3 - Choosing the Source engine

The Source engine isn't the only engine that can help with the creation of a game or mod. As [53] has shown, a lot of game studios have provided an engine for the game community to create games with. There are even companies that are specialized in the creation of game engines, selling them for a certain price to let the developer distribute a created game commercially [41][64][79][80]. Usually, mods must be distributed freely among the mod community and the player must always have a version of the original game to play a certain mod. Why use the Source engine and the Source SDK for creating a serious game concerning triage?

Appendix A compares several game engines (Delta3D[41], Quest3D[64], Source, Torque[79], Unity[80] & Unreal[81]) for some given features. A survey has been performed with several serious game developers who were willing to cooperate in answering some questions concerning their game engine. This and a qualitative comparison of the engines' capabilities show that the Source engine is very suitable for serious game in general.

In order to implement an Intelligent Tutoring System and an automated scenario creator a lot of code has to be programmed. An engine built on the C++ programming language can contribute to a proper merge between the engine and the to be programmed modules. With the Source engine being built very modular and with lots of design patterns implemented, this engine has an extra advantage over other game engines.

Another big plus of the Source engine is its thriving community. With many sites like Valve-ERC [84], The Valve Developers Community [83], Steam's Tool Discussion Forum [78], Mod DB Tutorials [56] & Interlopers [51] a lot of developers are helping out each other with tutorials and forums. This lowers the learning curve for the Source SDK and provides a rapid recovery from problems encountered during the implementation.

Fig. 5.6: Screenshot of Half-Life 2.

For this project, Artificial Intelligence techniques will be implemented in order to provide basic reasoning. The Source engine already has a basic AI framework. Although this is mainly focused on NPC behaviour, all code is provided for the programmer to use. As will be seen later, certain modules of the Intelligent Tutoring System will use an extended form of an AI system provided by Source.

Since this project isn't focused on the actual appearance of the 3D world and the victims in the game, the models in the Source engine can very well be used for the serious game. Mainly because Half-Life 2 is set in an urban environment (instead of a futuristic space setting), all models that are provided by the engine can fit easily in a disaster area. The citizen NPCs of the game are very suitable for a simple representation of a victim. Especially the animations the citizens can perform can very well be used to act out a wounded person lying down.

All these arguments together make the Source engine a very good engine to create this triage serious game with.

5.4 - Outline

Before design and implementation is explained in the following parts of this thesis, it might be useful to address certain topics of the Source engine.

The main application for creating a game with the Source SDK is the Hammer editor. Within this editor levels (or maps) are designed. A level designer can create the 3D geometry for the level consisting of a collection of primitive shapes like blocks, spheres, cylinders, etc. This geometry can be textured with all kinds of textures, both from the games published by Valve as well as the developer's own created images.

Instead of creating the map with only the solid geometry, Hammer lets developers place entities in the same map as well. These entities are created by programmers that code additional functionality for the game. Entities can range from static props, being a complex version of a geometry with an attached model (e.g. a sink, lamppost or tree), to a complete NPC (e.g. an enemy like the Combine or a citizen). When the developer compiles the map, Hammer calculates lighting and optimizes certain areas for the Source engine. When the map is compiled, it can be run in the game where the Source engine displays all the geometry and handles all the game logic provided by the entities in the world. In figure 5.7 a screenshot of the Hammer editor is displayed.

Entities can be seen as building blocks. All entities are descendant from the most basic class in the Source engine: *CBaseEntity*. All basic functionality is coded in this class. This entity has three main child entities: *CLogicalEntity*, *CBrushEntity* and *CModelEntity*. Practically all entities that can be placed in Hammer are descendants of these three entities. The logical entity is an entity that will not be seen by the player. It provides game logic like for instance the number of kills the player has made. The *CBrushEntity* class needs to be attached to a geometry (or brush) in order to function. It can use the brush to alert other entities of certain changes in the world. An example of the use of a brush entity is triggers. Whenever the player enters a specific part of the world, a trigger alerts other entities (like

sound entities or light entities) to indicate the player has entered that area. The last entity is the model entity. As the name implies this entity is represented by a 3D model with or without animation. All NPCs for instance are descendants from *CModelEntity*.

All common entities that are present in the Source SDK are hierarchically related to the *CBaseEntity* class. A programmer is allowed to add extra entities to this hierarchy, thus providing extra functionality in the game's mod.

A specific programming notation is used in the Source engine to categorize certain variables in a class. This so-called Hungarian notation [49] adds pre-defined prefixes to variable names. A boolean value named 'done' for instance is represented by `bDone` in Hungarian notation. For each variable type a special prefix is defined so the programmer can see which type a certain variable is only by looking at its name. Additional prefixes are made for variables that are defined in a class (i.e. `m_bDone` for the boolean to be a member of a class) or represented globally (i.e. `g_bDone` for the boolean to be globally accessible). The C prefixes in front of the class names are also from the Hungarian notation, describing the class is implemented on the server-side of the project (e.g. *CBaseEntity*). *C_BrushEntity* would mean the class is client-side.

The programming solution provided by the Source SDK has 2 projects: one server project and one client project. For multiplayer mods, this is very important since the two projects will be run on different computers. The server code is the main game, keeping track of the complete world all the players are playing in. The client code resides on the player's own computer and communicates with the connected server. This code displays only relevant information for the player, given its location in the world. The server-sided code is also some sort of judge over the players, keeping control over the players, preventing them from cheating. For single-player games, like the triage game this project is focussed on, both client and server are run on the computer of the player. Communication is therefore less difficult to program, since sending messages from one module to another on one computer is more reliable than sending it over a network with noise.

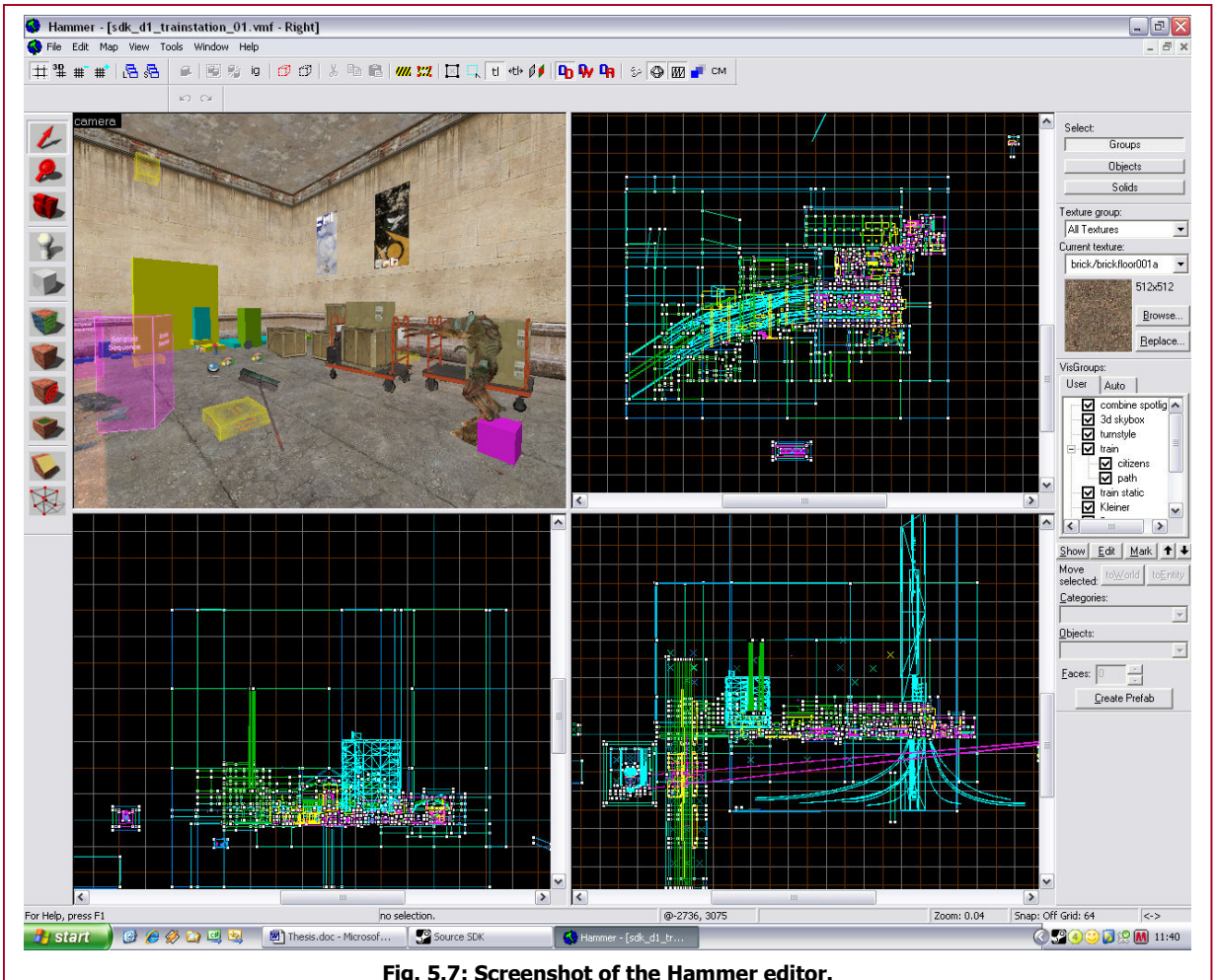


Fig. 5.7: Screenshot of the Hammer editor.

5.5 - Summary

Instead of creating every COTS game from the bottom up, more and more game development studios tend to create or use a game engine. A game engine provides all the basic functionality for creating a game on one or more gaming platforms (PC or console).

For this project, a game engine can be very useful. This way, development can be focused on the game and the ITS modules instead of first creating all the basic handles to for instance present certain figures on screen. However, implementing a self-created ITS will give problems when a game engine isn't compatible enough to handle an external library.

The game studio Valve has also created its own game engine, called the Source engine. All players who bought one of the published games created with this engine can use this engine via the Source SDK. A lot of gamers have used the Source SDK in order to create a (total conversion) mod of one of the games made by Valve. Several are discussed in section 5.2.

While the Source engine has many state-of-the-art features that developers can use, it doesn't necessarily require a high-end PC to play a mod. This feature is very useful for the creation of serious games, where the users usually don't own a PC with the latest hardware.

With a very lively developer community, a modular code project in C++ with clear comments and descriptions and a basic AI framework the programmer can alter, the Source engine and SDK seem a very good choice for creating a serious game about triage. Therefore, this project is going to use the Source SDK as main tool to create the serious game and the ITS with.

Part II – Game model

Chapter 6 - Ontology

To create a (serious) game it is important to specify entities in the virtual world. Especially when those entities should represent an object in the real world. It is impossible and most of the times irrelevant to represent all the information that is known of a real-live object into a virtual entity (e.g. the inner workings of the brain in a serious game about assessing the state of a victim). So it's up to the designer of the game to take into account certain important features of such objects.

For simulations, this is the most important part of the creation process. Although (serious) games are less focussed on life-like behaviour, it is still very important to create realistic entities the user can relate to. The user should be able to understand the entity and its behaviour. Only then the player can learn something from the application. For instance, a fireman is playing a serious game where he should be able to learn when to use different types of fire management plans. If the designer didn't create a good representation of the different types of fire, the fire fighter isn't able to distinguish between them and cannot apply the appropriate actions.

The actual implementation of this project is concerned with the creation of a serious game for the new triage method. However, this specification part describes a more global design of serious gaming and will take examples in both specific, defined topics and global, abstract topics to teach concerning the management of a disaster. The next part of this thesis will bound the ideas described to the triage game.

6.1 - Regions

Although the user will only interact with objects (be it other humans in the same virtual world, artificial intelligent characters or simple objects like a fire hose), in disaster management a lot of information can be represented by defining areas or regions. These regions are used for agents to know what kind of behaviour is necessary. For instance, a region that should indicate an area where an explosion has occurred should contain objects that are damaged or set on fire by the blast.

In disaster management, regions are also used to create more order in the chaos. The police usually seals of the area where the disaster has struck. Some regions are classified as unsafe, only to be entered by the proper specialists. Outside the disaster area(s) the treatment areas, waiting stations for ambulances, etc. are set up.

By the notion of a virtual region the application can direct the entities, forming a more realistic environment for a player to work in. This can both serve the purpose of emerging the player in the game and learning the player to distinguish between certain areas and their location.

Some of the regions that could be present in a serious game concerning disaster management are presented in table 6.1. Note that there are more possible regions to define, and that this table is just an indication of what is meant by a region in this game world.

Table 6.1: Some virtual regions and their description.

Region name	Description
R_WORLD	The game world the player is playing in.
R_DISASTER	The area where the disaster has struck.
R_CRASH	A certain region where a vehicle has collided with another agent.
R_FIRE	The area where fire is likely to be found or where a fire has been.
R_EXPLOSION	The area where the impact of an explosion is felt, seen, heard, etc.
R_TREATMENT	A treatment area for victims.
R_UNSAFE	A region that is unsafe for the player.

A region only has its core as main feature. Since a region must be very flexible in its shape a radius or other more complex parameter isn't given. The designer must be able to create this parameter for every type of region. Also note that the core is not the centre of a region. Section 6.3 will try to explain this by means of spatial relations between objects.

6.2 - Agents

Agents are the objects that interact with the virtual world. This could be both non-responsive, to be placed in the world as decoration, as well as intelligent agents like victims or fire that do respond to certain events in the environment.

Since a virtual disaster can change over time (consider for instance a fire that keeps on spreading) most of the agents have some sort of intelligence. This can be very reflexive, like changing the model of a tree to a burned down tree when the fire has reached it. Some agents however require more intelligence by making goals, planning actions and anticipate on what could be coming. For more information on different types of intelligent agents, the reader is referred to [11].

For every agent type there also must be a set of features or parameters that characterizes that agent. One of the most simple features practically all virtual agents should have is a position and an orientation in the 3D-world. A model (with matching animations) that represent the looks of the agent is usually also needed. Most of the other features that represent an agent are agent specific. For instance, with a toxic gas agent a feature could be the spread rate, the type of gas, if the gas is flammable or not, the concentration, etc. These features must be carefully selected by the designer of the application and must be relevant for the goal of the game.

A list of several agent types is represented in table 6.2.

Table 6.2: Some virtual agents and their description.

Agent name	Description
A_TREE	A tree.
A_VICTIM_T1	A victim that should be triaged as T1.
A_MEDICAL_STAFF	A medical worker that is not played by a human player.
A_FIRE	A virtual fire.
A_SMOKE	An agent representing the smoke coming from a fire.
A_WIND	An agent representing the wind.
A_TRAIN	A train in the virtual world.

6.3 - Relations

Once defined regions and agents in the virtual world its time to link agents and regions together to form a network. To do this relations are needed between the multiple types of regions and agents. These relations can be very local and object specific (e.g. a fire fighter carries a fire hose) or global and affecting large groups of entities (e.g. the wind agent that has an effect on all of the gasses).

With these functional relations, agents and regions can change certain properties of other agents and regions. A medical worker can change the vital signs of a victim if proper treatment is administered. The medical worker can therefore affect the state of the victim. Gasses are also affected by the wind agent, changing their position and concentration according to the speed and direction of the wind.

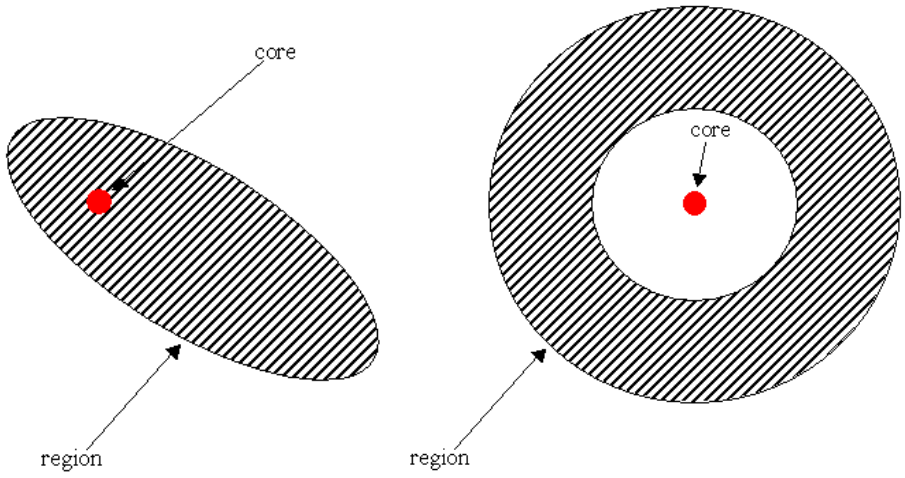


Fig. 6.1: Spatial relations between regions and agents. The core of a region doesn't need to be the centre of the region or inside the region at all.

Instead of these functional relations, spatial relations are also very important to consider. Spatial relations are used to place certain objects in, close to, or away from certain regions or agents. The crash region for instance attracts vehicles in the world to be placed at the core of the region. In figure 6.1, two situations are given of spatial relations between a region and an agent.

In the left image an elliptic region is drawn with the core in the top-left part of the region. The elliptic region could for example describe a relation between a crash region and debris of car parts. The core of the crash region describes where the crash has happened. The marked area describes where the car parts from that crash should be placed.

In the right image of figure 6.1 a spatial relation is defined for T2 victims regarding a disaster region. The disaster has happened at the core, but T2 victims are hardly found close to the origin of the disaster. Therefore the relation is a ring around the core. Agents can also use spatial relations between other agents. This is mostly used to avoid agents being in collision with each other.

These spatial relationships can very well be used to create dynamic scenarios of disaster areas. If the relations are formulated well enough, the system can create a random disaster area that should look realistic to the user. Part 4 of this thesis will discuss the use of spatial relations in this project.

In table 6.3, some examples are given of the relation types discussed on the entities defined.

Table 6.3: Examples of both functional and spatial relations between entities.

		To **Agent**	To **Region**
From **Agent**	Spatial	Trees should not be placed in cars	A mobile gas detector has a detection range
	Functional	Smoke decreases the breathing rate of victims	Fire fighters can declare a previously unsafe region to become passable
Region	Spatial	Debris that should be placed around a crash site	A crash region has a position relative to the origin of the disaster
	Functional	Objects are damaged because of a recent explosion	

6.4 - Events

With entities and relations between those entities described, the system and the user must also be able to alter the environment. In a dynamic world, victims for example are affected by time. If a person with an injury isn't treated in time, his/her state will deteriorate until the victim eventually dies. Fire and toxic gasses or liquids (if not controlled) are likely to spread across the area.

These situations are described by events. Events change features of affected agents and regions or relations between certain entities. This way, the system can manipulate the entities in the environment to pose different problems or challenges to the user.

For instance, if the player hasn't noticed a fire that is heading for a gas station, the system can send out the event that creates an explosion, maybe even sooner than that should have happened in the real world.

The user itself also creates events, although indirectly. If a user triages a victim, the victims' state has changed to being triaged with the triage class recorded. This event will be send to the system to inform what the user just did. In this way the application can reason about the user's actions.

Chapter 7 - Victims

In the previous chapter agents were discussed, with their variety of artificial intelligence. In the case of a serious game about the triage process, the most important agents in the world are the victims the player has to treat. Therefore, this chapter discusses the victim agent in more depth.

7.1 - Parameters

As chapter 2 has shown, the medical worker checks a lot of vital signs in order to form a conclusion concerning the proper triage class. If this medical worker would like to train the triage procedure, the training application must also present these parameters. Therefore, for every victim that is present in the virtual world individual parameters like heart rate, respiratory rate, etc. must be stored and accessed whenever the user wants to see them.

There are also parameters that don't relate to the actual triage procedure, but are used by the engine to keep track of all the entities in the world. Every entity for instance must have a unique identification number or name. The position and orientation in the virtual world are also parameters concerning the entity. In table 7.1 are the global parameters shown for a victim agent. The bottom 5 parameters are parameters used by the engine, while the top 4 are global parameters concerning the triage process.

Table 7.1: Global parameters for a victim.

Parameter name	Description
`Index`	The corresponding database index of this victim
`Gender`	The gender of the victim (male/female)
`Age`	The age of the victim
`Injuries`	The injuries this victim has
`Name`	The name of this victim
`Model`	The 3D model that represents this victim in the virtual world
`Animation`	The model animation this victim is performing
`Origin`	The position of the victim
`Angles`	The orientation of the victim

The SIEVE and the SORT triage procedure (described in section 2.2.1 and 2.2.2 respectively) both use different parameters to come to the correct triage class (although some overlap exists). Therefore, the victim must also contain different parameters for each part. The SIEVE parameters are described in table 7.2, while the SORT parameters are described in table 7.3.

Table 7.2: SIEVE parameters for a victim.

Parameter name	Description
`Speaking`	Indication of how well the victim can speak
`Walking`	Indication if the victim can walk on its own
`Airway`	Is the victim able to breathe on its own? (free/obstructed)
`CanOpenAirway`	Indication if the user can open the airway of the victim
`Breathing`	The breathing rate of the victim (in breaths / minute)
`CirculationHR`	The Heart Rate of the victim (in beats / minute)
`CirculationCRT`	The Capillary Refill Time (in seconds)
`TriageClass`	The triage class this victim is in

The index parameter (in table 7.1) is a number referring to one of the victims described in a victim database. The NIFV already had a database of 62 victims that are used for the current triage training. These victims contain all the relevant information for both global, SIEVE and SORT parameters and could very well be found in a real-live incident. All victim parameters were based on actual incidents and evaluated by several experts. LOTUS victims were asked to simulate the 62 victims in order to

obtain photo and video material. The NIFV gave permission to use this extensive victim database for this project. The database (stored in Excel-file) had to be converted to an XML-file in order to read the values. An example of all the parameters stored in XML for one victim is given in figure 7.1.

Table 7.3: SORT parameters for a victim.

Parameter name	Description
Speaking	Indication of how well the victim can speak
Airway	Is the victim able to breathe on its own? (free/obstructed)
Breathing	The breathing rate of the victim (in breaths / minute)
BreathingScore	The breathing score that is filled in on the triage card
CiruclationRR	The systolic blood pressure of the victim (in mmHg)
CirculationRRScore	The blood pressure score that is filled in on the triage card
DisabilityEyes	The best eye response of the victim
DisabilityMotor	The best motor response of the victim
DisabilityVerbal	The best verbal response of the victim
DisabilityGCS	The value of the Glasgow Coma Scale of the victim
DisabilityGCSScore	The GCS score that is filled in on the triage card
DisabilityTRTS	The Triage Revised Trauma Score of the victim
TriageClass	The triage class this victim is in

This figure shows the representation of a 30 year-old male victim, who has burns as injury type. For the SIEVE part the victim has a respiratory rate of 32 breaths per minute and a heart rate of 112 beats per minute. The victim should be classified as T1, because he cannot walk and his breathing rate is above 30. For the SORT part, the victim still has a breathing rate of 32 breaths per minute, and now has a systolic blood pressure of 130 mmHg. All the scores are also given, except for the Glasgow Coma Scale and the Triage Revised Trauma Score. These should be calculated by the system.

```
<victim index="16">
	<gender>male</gender>
	<age>30</age>
	<injuries>burns</injuries>
	<sieve>
		<class>T1</class>
		<walking>no</walking>
		<airway>free</airway>
		<breathing>32</breathing>
		<circulation type="HR">112</circulation>
	</sieve>
	<sort>
		<class>T2</class>
		<walking>no</walking>
		<airway>free</airway>
		<breathing>32</breathing>
		<breathing_score>3</breathing_score>
		<circulation type="RR">130</circulation>
		<circulation_score>4</circulation_score>
		<disability>
			<GCS>
				<E>4</E>
				<M>6</M>
				<V>5</V>
				<GCS_score>4</GCS_score>
			</GCS>
		</disability>
	</sort>
</victim>
```

Fig. 7.1: XML representation of a victim.

Note that these parameters are the actual parameters the victim has. If an instructional tool must be able to check if the user of the game had made a mistake in the triage procedure, the system must also store the parameters entered by the user. Practically all parameters described in table 7.2 and 7.3 must

have an ‘actual’ and a ‘perceived’ value. The actual value refers to the value described in the XML database. The perceived value refers to the user’s perception of this value in the game.

7.2 - Interactivity

Now that the parameters of a victim are described, there must be a proper way to present these parameters to the user. If the player is able to walk freely in a 3D disaster area, there need to be a connection between the parameters of a victim and the representation of that victim in the world. This representation must give the same (or must come close to the) sense of urgency to the player compared to the real-life situation.

As in the real-life situation, medical workers need to perform some action in order to retrieve a certain parameter. In order to measure the breathing rate, medical personnel tend to look at the chest area for movement and listen to the victim. After 15 seconds of counting, the number of breaths are multiplied by 4 and recorded as the respiratory rate. In the virtual training situation this procedure should also be used. Instead of just giving a textual description of a victim, the player must search for the parameters.

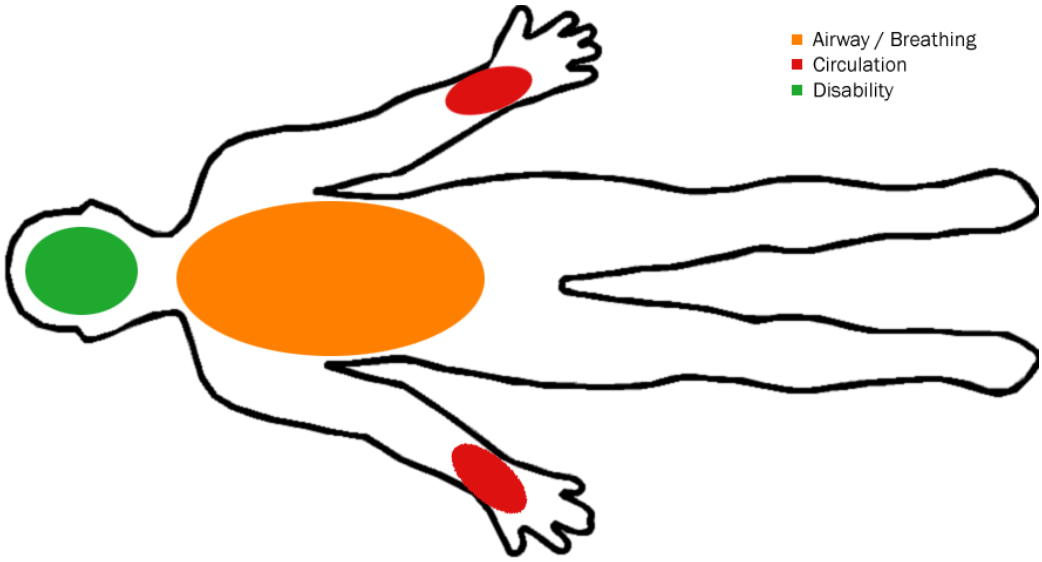


Fig. 7.2: Regions of interest to present parameter values of a victim.

This search for parameters needs to be done on the model of the victim. In figure 7.2 a victim is schematically drawn with possible regions where a user must be able to find the corresponding parameters. Here, checking if the airway is free and checking the breathing rate is in the same region. This is because a countable breathing rate automatically implies that the airway is free. If the user can’t hear the victim breathing, the airway must be obstructed or the victim has died.

By looking at (i.e. placing the mouse over) a certain region, the system should display the corresponding parameters. But displaying isn’t the only option. To get a more multimodal experience, the system can also emit sounds to represent certain parameters. The breathing rate for instance could very well be a sound of a person breathing at a given interval. Another option for multimodality could be the heart rate. The heartbeats could be heard from computer speakers, but it could also be applied as haptic input for the user. If the user wears a haptic feedback glove, the system could apply pressure to simulate the actual wrist and the heartbeat.

Instead of textual explanation an image or animation could also very well be used to represent certain parameters. The blood pressure in the SORT for instance can be represented by a pressure gauge with a pointer indication the actual pressure. The eye response can also be shown as an animation of the eye.

For the complete triage procedure, all injuries need to be written down so nothing is forgotten while giving treatment. The victim in figure 7.1 has as main injury burn marks. The model of the victim should show this to the user. Some injuries however aren’t detectable by the eye only. A broken

leg must be felt by the medical worker in order to confirm its location. Unfortunately, the user isn't able to feel the complete virtual victim. This problem can however be solved by letting the player check the complete body with the mouse. Whenever something should be felt or seen, the system could alert the user by text or symbols what the victim has. So for instance, by going over the right leg the user gets a message that a broken bone has been found at the current location.

For the disability parameters, the user has to watch a response of a stimulus given by the user itself. This stimulus can be a sentence spoken by the user (e.g. "Hello sir! Are you alright?"), or a pain stimulus whenever the victim isn't responding to this sentence. Maybe in the near future it is possible to actually let the person speak to the virtual victim, which would create a massive impact on the experience of the game. For now however, a simple press of a button might produce on of the two stimuli.

For a victim to respond on these stimuli, intelligent behaviour is necessary. The victim must reason that a motor response score of 3 should induce an animation to the model that represents a flexion of the body when a pain stimulus is administered. All the parameters of the disability part of the triage procedure can be expressed by animations of the victim model. Audio is also needed to represent the verbal response of the victim.

Another behaviour that requires intelligence of the victim agent is the response of the player in the vicinity. Victims with a broken or even amputated leg are in constant pain. This victim should be screaming out loud, something that could be heard over a certain distance. Walking victims should respond to the presence of a medical worker by walking or running towards him/her, getting his/her attention. Victims that don't have any injuries could help other victims or could keep searching for a lost relative, refusing to leave the area. All these types of behaviour are part of the victim agent. The victim agent needs to reason about the environment and should form goals concerning its own history.

Chapter 8 - Game structure

Instead of looking for a specific solution for the serious game concerning triage, this chapter takes on a global perspective. With this specification an implemented version could be more modular so additional changes are easier to design. With a global design and the triage game as an instance of that design, someone else can attach another instance of a game (e.g. a fire fighter training or a disaster management training) with use of the global design. This can be seen as a template that is used for every instance of a game in this framework.

8.1 - Use cases

In order to specify what the game should be able to do a use case diagram is created. Figure 8.1 shows this diagram containing 3 distinct users with different global actions they can perform. This diagram was constructed with help from the NIFV.

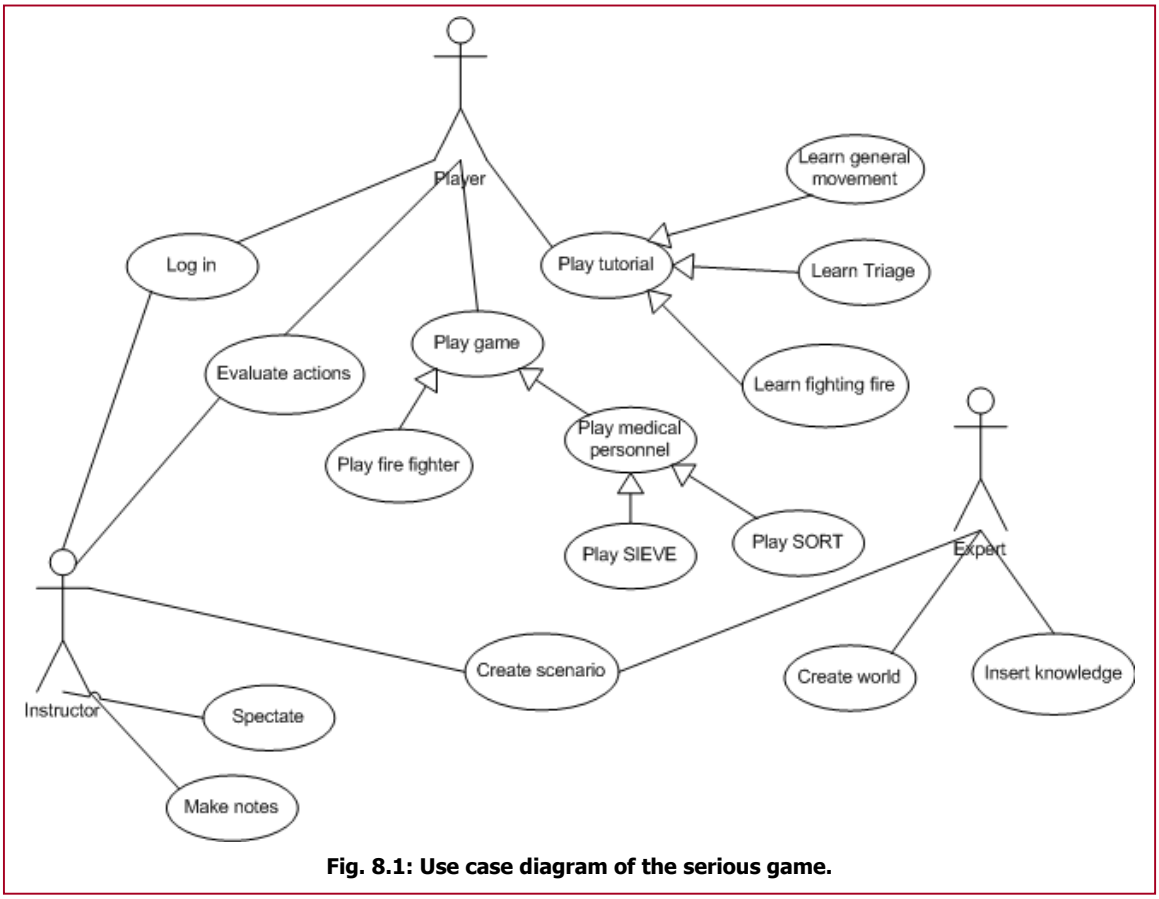


Fig. 8.1: Use case diagram of the serious game.

For instance, the player of the game (depicted at the top part of the figure) must be able to log into the system. This is used by the system if the player has played a previous session before. It also functions to store the evaluation in a fashionable manner, so that the instructor can evaluate all sessions of one person.

The player must also be able to start a game or a tutorial on one type of game. After a game has been played the player can check how he/she did by asking the system an evaluation report per victim. The instructor (shown at the bottom-left corner of the figure) should also be able to check these evaluation reports, plus some more global information on how the player did in a certain session.

In this case the instructor is a person that monitors the progress of the player from outside the system. This must not be confused with the instructor introduced in the Intelligent Tutoring System. The instructor described here can help the player (or multiple players in a classroom) go through some of the difficulties given by the game. Different opinions on how to triage can also be discussed with this instructor, instead of the game itself.

The expert in the use case diagram (depicted at the bottom-right corner of figure 8.1) is the person that puts in new knowledge about a certain topic. This doesn't have to mean the expert is confronted with the game itself. Some text files with the rules of triage described in a specific format the system is able to interpret could be sufficient. It can even be possible that the expert uses the Hammer tool in the Source SDK to create a completely new world or new model of an agent.

For the textual descriptions of the use cases shown in figure 8.1, the reader is referred to appendix B of this thesis.

8.2 - Blueprint

In figure 8.2, a global picture of the system is depicted. In this picture there are four main components of the system: the Source engine, the game the player is playing, one or more Intelligent Tutoring Systems that monitor and evaluate the user's behaviour and the Scenario Creator.

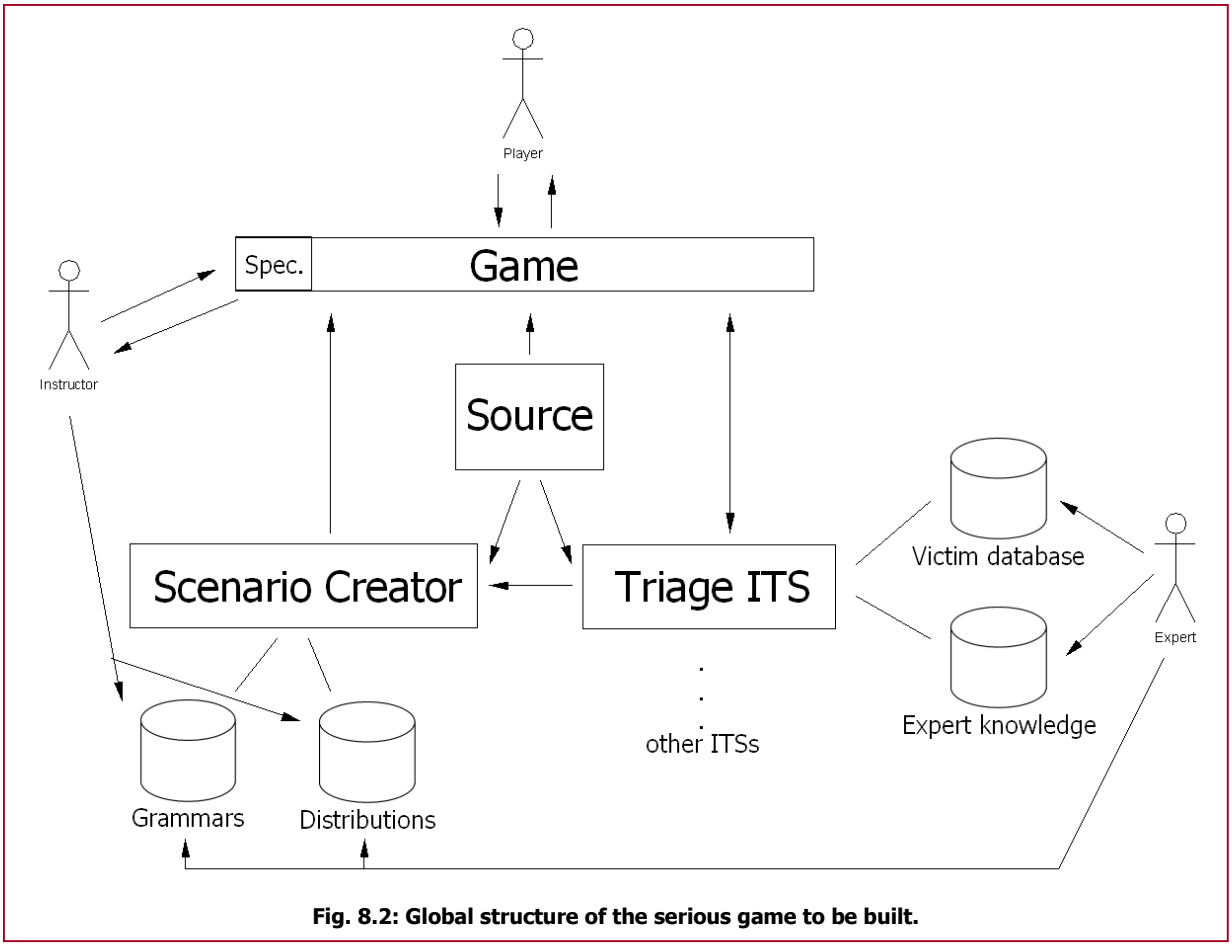


Fig. 8.2: Global structure of the serious game to be built.

The Source engine is the thriving force behind the system. It starts the game and provides the other components with basic functionality. It also keeps track of the time spend on certain operations and informs the components when to act.

The ITS components are constantly checking the game for alterations made by the player. In order to give feedback or to change the virtual world (e.g. alter the victim's state, continue the spreading of a fire, etc.) the ITS has some control over the game. With data created by the human expert, the ITS can reason about the user's actions and provide adequate feedback to errors made. Part 3 will discuss the inner workings of the ITS.

The scenario creator is the module that is used to create a disaster area with. At the start of the map, this component designs and places entities in the game by means of grammars and distributions provided by both the expert and the instructor. After this process has been completed, the game starts and the ITS begins to do its task. The automated scenario creator will be explained in part 4 of this thesis.

8.3 - Level structure

When the player starts the game created with this project, it can either choose between playing the SIEVE or the SORT. After entering the name of the player and some additional options for the game, the player can start a new game. For the SIEVE part, the game starts out at the edge of a disaster area. In the SORT, the player is located inside a medical tent with victims lying down on several stretchers.

The player takes over control and can walk freely in the virtual world. There should however be some restrictions, because else the player is able to wander of and forgets about the purpose of the game. The main task of the player is to triage all the victims that are placed in the virtual world.

In order to let the user get familiar with the game, a level structure should be created. This starts out easy and increases in difficulty whenever the player is mastering the game and the triage process. This increase in difficulty can only be done by the number of victims presented in the virtual world. This has to do with the fact that the triage procedure the medical worker has to learn is the same for all the victims. No one victim is more difficult than the other in the view of the procedure.

The difficulty increases when the system allows only a fixed number of mistakes, while the number of victims placed in the world is increasing. For this reason, a skill level is introduced to the player. The main goal of the player is to keep this level above a certain threshold to continue to the next level. Every new level, the skill level is set to the initial value and for every mistake in the triage process the player makes, this value will go down accordingly.

For this project, 5 levels are created for both the SIEVE and SORT part of the game. The number of victims the player starts out with in level 1 is 5. This will increase to 10 in level 2, 15 in level 3, 20 in level 4 and eventually 30 victims in level 5. If the player completes all the levels, he/she has triaged a minimum 80 victims without making to much mistakes. This should be enough to master the routine of the triage process.

Part III – Instructional tool specifications

Chapter 9 - Actual Victims

In chapter 7 specifications were created for the victims of the triage game. Because the focus of this project is on the underlying frameworks and reasoning, the representation of the victims aren't implemented as specified. Practically all the parameters are used, but the interactivity and the intelligence of the victims described in section 7.2 aren't implemented in such fashion.

The interactivity between the user and the victim is still needed. A user must still be able to retrieve information from a victim and report the perceived parameters to the system. This chapter will describe how this can be done.

9.1 - Design

A basic outline of what classes are needed to describe a victim is always useful when starting to implement. In figure 9.1 this basic outline is sketched in an UML class diagram. The main focus of this figure is the *CVictim* class in the lower-middle section. Please note that the 'C' prefix in the class name means that this class is on the server side of the game. Multiple players therefore always perceive the victim as the same. Also, when the reader wishes to get a deeper insight in variables and functions used, appendix C displays the UML diagrams with this extra information.

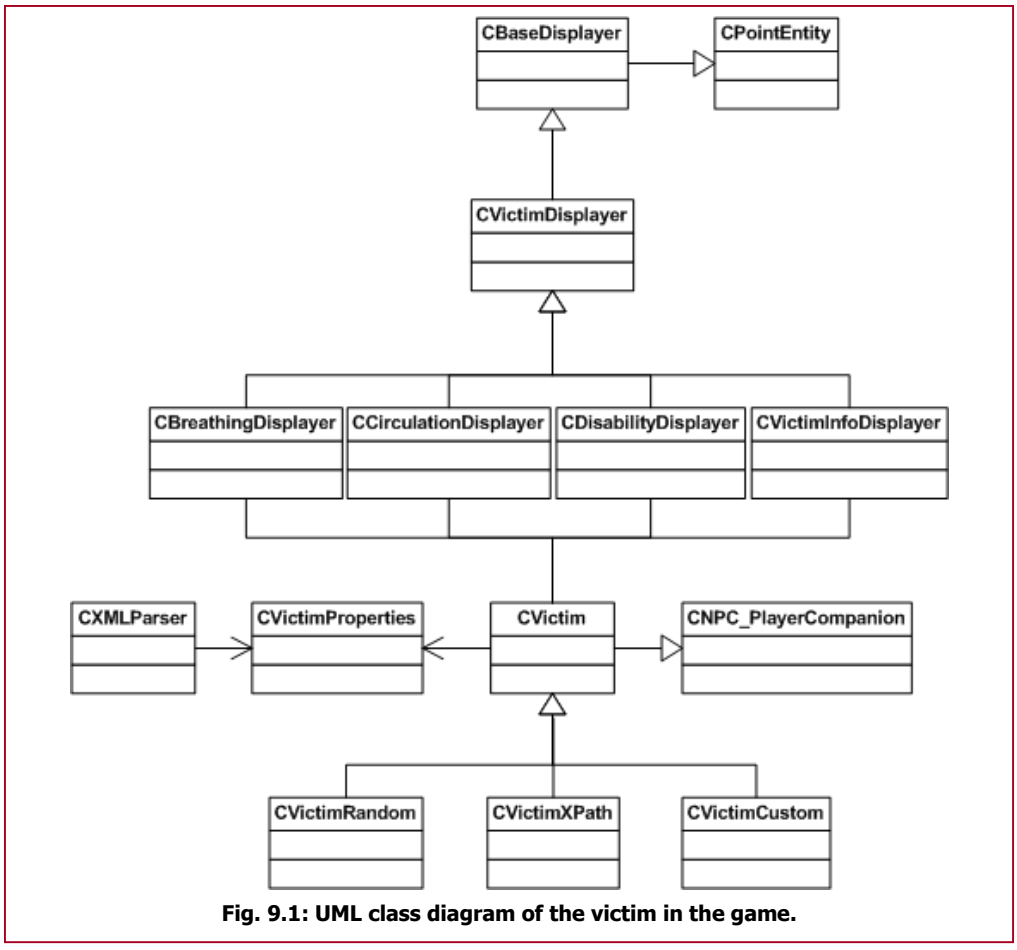


Fig. 9.1: UML class diagram of the victim in the game.

The main class is a descendant from the *CNPC_PlayerCompanion* class provided by the Source engine. This Source class already has the functionality of a model entity and embedded AI. The class is used for Non-Playable Characters (NPCs) that fight along the player in the game Half-Life 2.

The parameters described in chapter 7 are stored in the *CVictimProperties* class. The XML parser (*CXMLParser*) retrieves the parameters of an appropriate victim from the database (see also figure 7.1). It puts these parameters in a *CVictimProperties* class attached to an instance of the *CVictim* class.

The victim class itself is never used in the system. 3 classes below the *CVictim* class in the figure are used to represent different types of accessible classes. The *CVictimRandom* class asks the database for a completely random victim. The designer of the game doesn't have any influence on the outcome of these parameters. The middle class is *CVictimXPath*. This class has an additional search string that can be provided by the designer as well as the system itself. The search string must be conform XPath standards [88] and queries the database to search for a victim with the specified conditions. *CVictimCustom* is the class that gets all the parameters from the game designer. It bypasses the victim database and uses everything that was entered in the Hammer editor.

Every victim makes use of 4 displayer classes. These classes are *CBreathingDisplayer*, *CCirculationDisplayer*, *CDisabilityDisplayer* and *CVictimInfoDisplayer*. All these classes inherit functionality from the *CVictimDisplayer*, which is a descendant of the *CBaseDisplayer* class, which is in its turn a descendant of Source's *CPointEntity*. The four distinct victim displayers all have a separate set of parameters they display or emit. These classes take care of when to display these parameters and provide the main output of the victim to the user. The following section will explain how these classes affect the game while the user is playing.

9.2 - Output to the user

When a medical worker starts triaging, the looks of the victim are very important to get a first impression of the injuries. Therefore it is very important to have realistic models of the victims in the virtual world. A victim that should have an amputated leg (and acts like it has one) should of course have a model that misses that leg.

Unfortunately, it was impossible to use earlier created models of the 62 pre-defined victims in the database. However, the NIFV also has photo and video material for every victim in the database, making it easy to texture a model with the photos. Only one model was created to get a better impression of how the game would look like with these models. This victim is victim 16 (data represented in figure 7.1) and is shown in figure 9.2. The rest of the victims will use Source's citizen models. A difference in model has been made between male and female and burn victim or not (the model has a blue overall or a white shirt, respectively).

Fig. 9.2: Model representation of victim 16 with burns.

Along with the lack of realistic victim models, the accurate animations concerning flexion or extension of certain body parts also aren't implemented. For this problem, the solution also had to be found in the standard animations the Source engine is providing. For instance, in figure 9.3, a lying animation is executed on a regular female victim.

Instead of using haptic feedback gloves or other peripherals, this version of the game can only give output via the monitor (visual) and the speakers (auditory).

Fig. 9.3: Female victim lying down (without burns).

Because the appearance of the victims is an important factor in the realism of the game, a special displayer class was created. *CVictimInfoDisplayer* checks (as any other displayer) if the user is within a certain range of the victim this displayer is attached to. Once this is true, the eye direction (the direction the player is looking at) is retrieved and a calculation will be made to check if the player is

looking (within a certain range) at the victim. If all these premises are met, *CVictimInfoDisplayer* brings a pop-up window with pictures of how the actual victim should be like. For victim 16, this window is shown in figure 9.4. These photos are also from the NIFV victim database.

Fig. 9.4: The pop-up window with photos of the victim in the bottom-left corner of the screen.

To help the user with the search for vital signs of a victim, the regions of interest in figure 7.2 must be made visible to the user. Therefore the crosshair displayed in the game is altered whenever a player goes over such a region. In figure 9.6, a red magnifying glass is shown as the 'exploration crosshair'. The crosshair is seen whenever the centre part of the screen hits a region of interest. This makes it much easier to detect emitting sounds, especially when a victim is breathing very slowly or has died. When the magnifying glass is shown, the user knows he/she is in the right spot to detect any output.

An extra feature that has been added is that the victim changes its model to represent the triaged class. So, whenever a user defines a triage class to a victim, the victim will get the colour of the entered triage class. Figure 9.5 shows the different models corresponding to the different classes. Please note that this male model is used for all the victims. Females also become this male model with the correct class colour. This has to do with the fact that only the male model was given by the Source engine to alter, instead of all the models of all the citizens.

Fig. 9.5: Models that indicate the victim has been triaged in the triage class of the colour shown.

9.2.1 - SIEVE

Both the SIEVE and the SORT part of the game require different output parameters from the victim. The displayers discussed will deal with both parts of the game. For the SIEVE part, two displayers (aside from *CVictimInfoDisplayer*) are needed: *CBreathingDisplayer* and *CCirculationDisplayer*.

For the breathing rate, 8 audio files with distinct breathing rates (ranging from 7 to 36 breaths per minute, covering most of the values in the database) were created. Each audio file contains 1 breathing cycle (i.e. one inhalation and one exhalation). This audio file is loaded in and played repeatedly whenever the player is close enough and aims at the torso of the victim. No further output is given. The player must deduce the breathing rate by counting the number of breaths from only the audio file coming from the speakers.

For the circulation rate, both Heart Rate and Capillary Refill Time can be presented to the user. The HR uses (just like the breathing rate) audio files to mimic a pulse. 13 files were created with HRs varying from 55 to 140 beats per minute. The audio file itself contains 2 to 4 beats, making the files a bit longer so repetition isn't needed every 0.5 second.

The CRT is displayed in text, as can be seen in figure 9.6. So the user both hears the heartbeat of the victim and watches the CRT on screen. The CRT however isn't provided with every victim. Most of the victims only have a HR defined and therefore these victims only emit a sound to the user.

Fig. 9.6: Textual output of the CRT.

Fig. 9.7: Textual output of the systolic blood pressure.

9.2.2 - SORT

In the SORT part of the game all the above-mentioned displayers are used. Where *CBreathingDisplayer* keeps on emitting breathes per minute via audio files, the *CCirculationDisplayer* class must produce a different output. This displayer must present the systolic blood pressure of the victim. This is done in the same way as the CRT parameter. Figure 9.7 shows the result in-game.

Instead of using animations for the disability parameters (the EMV-scores), the user gets textual information of the best responses. Figure 9.8 gives an impression of how this is represented. Per response the best description described on the triage card is given. The user has to mesmerize the descriptions and look them up on the triage card to get the corresponding scores.

Fig. 9.8: Textual output of the EMV-scores.

In this way, the user still has to search for the parameters. This is only done in a different way that what actually should be done. In order to improve this representation, animations should be created for the models. In that case, the user is able to infer the parameters from the behaviour of the victim.

9.3 - Input from the user

In order to triage a victim, a medical worker must be able to fill in a card and/or give the card to the victim. The parameters that were given to the user described in the previous section, need to be entered by the user to know if the user has correctly read them. Several menus are created for this purpose and the following subsections will discuss them for both the SIEVE and SORT parts of the game.

9.3.1 - SIEVE

In the SIEVE part, the user has multiple actions he/she can do in the virtual world regarding the victim. A choice menu is given whenever the player presses and holds the use key (normally <e>) in the vicinity of a victim. This choice menu is shown in figure 9.9.

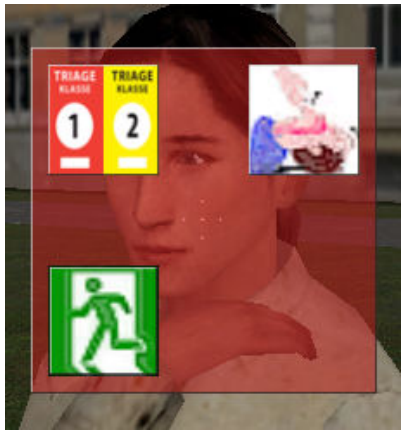


Fig. 9.9: Choices menu in the SIEVE part of the game.

As depicted by the symbols, the player has 3 options he/she could perform. The top-left symbol represents the classification of the victim into one of the triage classes. This will give a new menu that will be discussed later.

The top-right symbol represents the opening of the airway of the victim. Whenever the user cannot hear a breathing rate, he/she should open the airway of the user. If a breathing rate isn't heard after this, the victim should be marked as deceased. When the user presses this button, the system will open the airway of the user.

The third option (depicted in the lower-left symbol in figure 9.9) is to order the victim to leave the area. Whenever a victim is triaged as T3 or T3 wounded, the victim should walk to the nearest exit point. Pressing this button will give a signal to the victim to walk (whenever possible) to an exit point.

Fig. 9.10: The classification menu to indicate what triage class the current victim is in.

When the user presses the triage classify button, a new menu will pop up in the top-left corner of the screen, depicted in figure 9.10.

Here all the possible triage classes are represented and the user is allowed to click on one of the classes. Once clicked the system is informed that the user thinks the victim is in the given triage class.

It is important for the system to know why the user has chosen for this particular triage class. Therefore, after the triage classification menu closes another menu pops up that asks for an explanation. Figure 9.11 shows this menu. The user can enter all the parameters that could have played a role in the triage process. Once the player closes this window the victim will update all its data and will change its model to resemble the users' choice.

Fig. 9.11: The explanation menu for the user to elaborate on his/her decision.

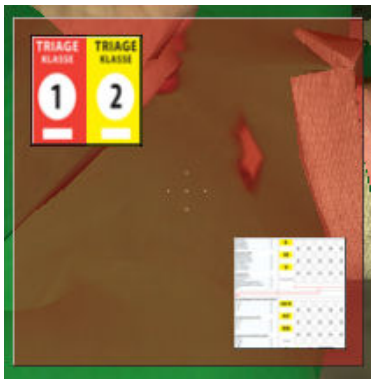


Fig. 9.12: Choices menu in the SORT part of the game.

9.3.2 - SORT

For the SORT part of the game, some of the menus have an overlap with the menus in the SIEVE part, although with some slight changes.

In figure 9.12, the choices menu is displayed for the SORT part. The user can now only perform 2 actions: Classify the victim to be categorized into one of the triage classes (top-left symbol) and edit the triage card (bottom-right symbol).

If the user presses the top-left button, the same menu will appear as depicted in figure 9.10. After entering the triage class, the menu will disappear and the victim is classified. There is no explanation menu in the SORT, because the user must first enter all the data on the triage card. This card will appear in the bottom-right corner of the screen whenever the player presses the corresponding button in the choices menu. This menu is shown in figure 9.13. The user can simply type in all the scores, which comes to the concluding TRTS. Once the player is finished, he/she can close the window and the data will be entered by the system.

After the player has first entered all the data on the triage card menu and then classified the victim in the appropriate triage class, the victim will change its model according to the given class. This is shown in figure 9.14.

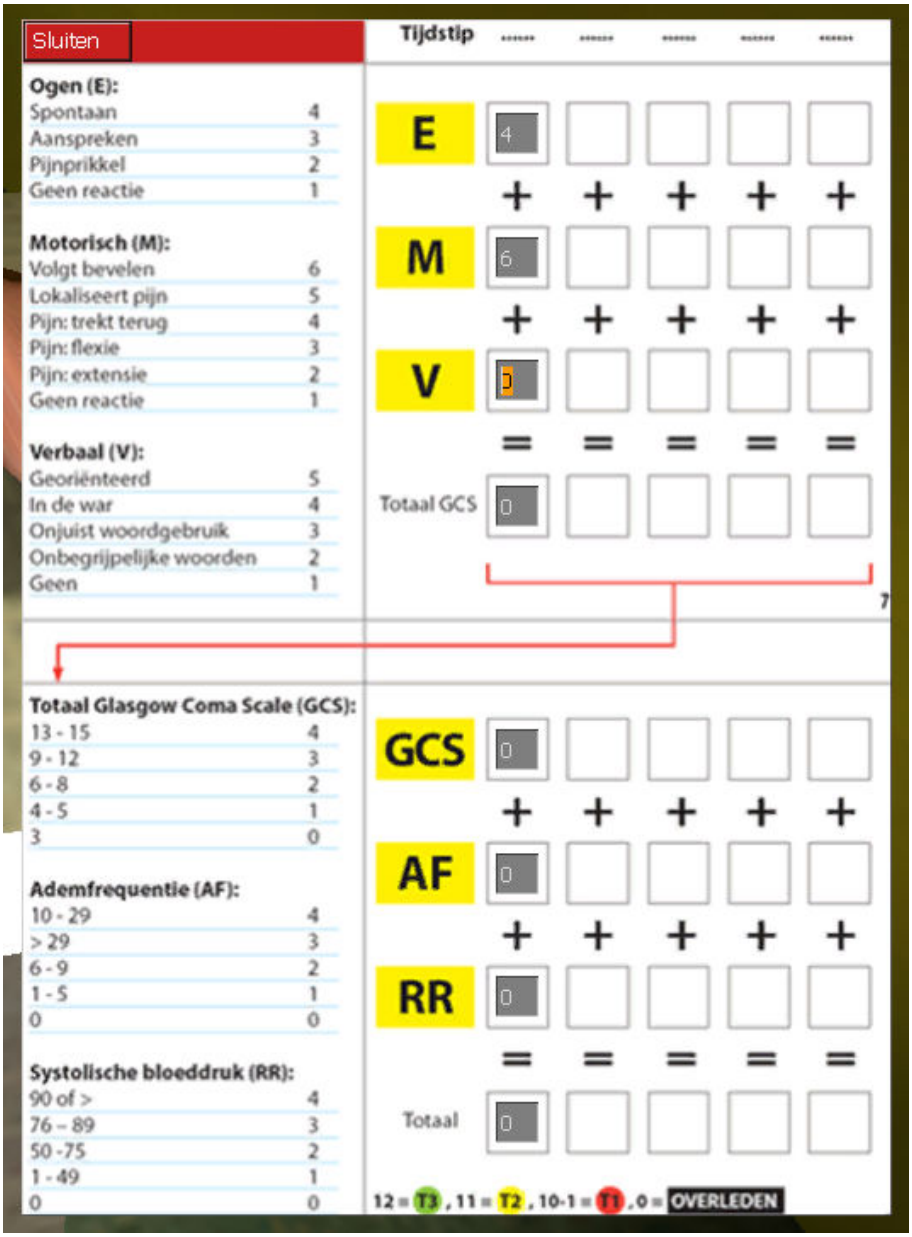


Fig. 9.13: The triage card where the user can enter all the observed parameters.

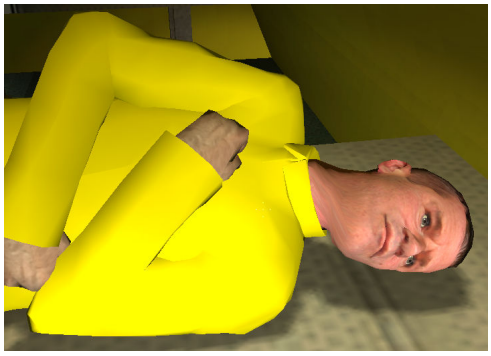


Fig. 9.14: A triaged T2 victim.

Chapter 10 - ITS framework

For a serious game that must be able to present a topic of interest to the player, reasoning about the user's behaviour, detection of errors and responding to this behaviour is very important. This chapter will explain the underlying structure of how the system records and responds to certain user behaviour.

Because a general outline is set, a lot of modifications or additions can take place to create more functionality for the system. This project is just a subset of what could have been implemented with the current model.

10.1 - Base design

The designed structure has a lot of overlap with the example Intelligent Tutoring Systems (ITS) discussed in chapter 4. The framework consists of modules and models. The models in the system contain the necessary information about specific experts on a certain topic (like in chapter 4). The modules use these models to convert input from the user into certain output to the user. However, where the example in chapter 4 uses the models only to store information, in this scheme the models also provide additional functionality.

Figure 10.1 shows the basic flow chart of the designed ITS connected to the game the user is playing. It contains 4 modules (Monitor, Evaluator, Instructor & Generator) and 3 models (Expert, Didactical & Student).

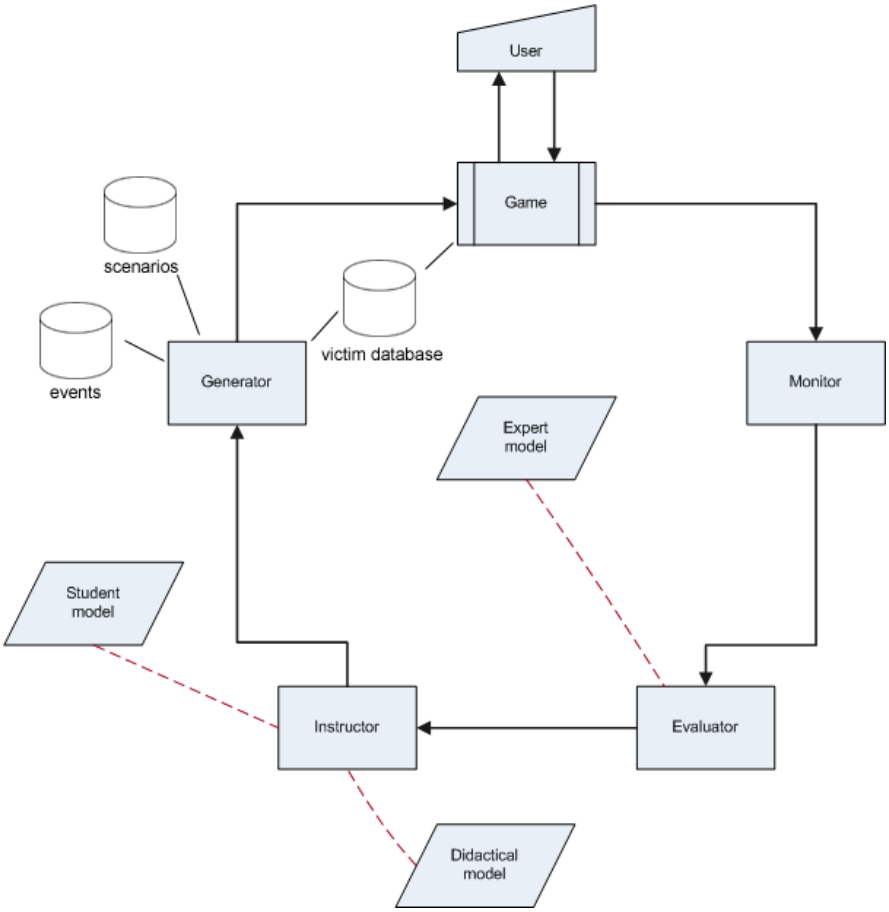


Fig. 10.1: Flow chart depicting modules in rectangles and models in parallelograms.

The main ITS loop is shown in black, whereas the dotted red lines represent the connection with the models. Additional sources of information are also depicted in the figure. The user is manipulating the game by his/her keyboard and mouse. He/she receives input from the game by means of the monitor and speakers. The game (managed by the Source engine) will give at pre-defined intervals

control to the ITS loop. It is possible to let the ITS loop run every frame displayed on the screen (i.e. approximately 30-40 times per second), except this will probably slow down the complete game. As will be seen in the next sections, it isn't even necessary to run the loop every frame.

In the actual UML design (shown in figure 10.2) a control class is needed to manage all the ITS classes (both modules and models). This class, called *CITS*, is also the class that indicates whenever the ITS loop should be fully executed. *CITS* is the only class in the loop that inherits functionality from the Source engine's entity hierarchy. It's a descendant from *CLogicalEntity* and needs to be placed in the virtual world created in Hammer to let the ITS run in the game. The designer of the game can alter some of the ITS parameters. This includes the mentioned interval for the execution of the complete ITS loop. 2 other parameters that can be set by the designer are the game type (e.g. SIEVE or SORT) and the time interval between the executions of the monitor loop.

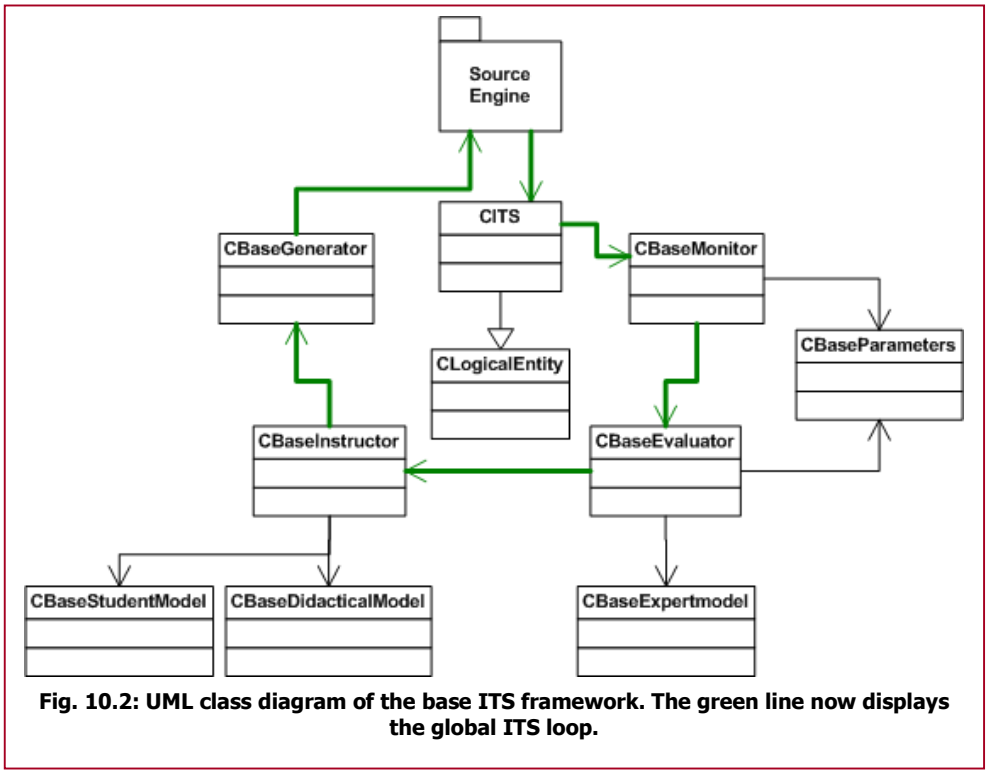


Fig. 10.2: UML class diagram of the base ITS framework. The green line now displays the global ITS loop.

This monitor loop is specially made for probing monitors. These monitors need to update certain variables at small intervals of time. An example will be given in section 10.2, where a movement monitor updates the player's velocity, position, etc. to gain more information that can be reported to the rest of the system.

The monitor's main task is to describe what the user has been doing during the time interval the ITS loop is executed. This description is called a user action. To come to a conclusion, the monitor stores parameters concerning the object that needs to be monitored. For instance, a movement monitor should monitor the player; an example of a parameter that needs to be measured is the player's position in the world. Once the monitor has inferred the appropriate user action it sends this user action along with these parameters (*CBaseParameters* in figure 10.2) to the evaluator.

The evaluator's goal is to assess the user action for possible errors. With help from the expert model, this evaluator can give a score and an explanation to the instructor. The expert model contains all relevant knowledge concerning a certain topic. This can be described in factual statements, rules, a semantic network, etc. Multiple experts (and thus multiple expert models) can help the evaluator to come to a certain score and explanation.

Once the instructor receives this information, it updates the student model and asks the didactical model for advise. The student model contains the information the player knows. The didactical model tries to form a strategy on how to proceed with the game. If the player has made an error, the didactical model tries to find a suitable way to present this error.

The generator module gets commands from the instructor in what to do in the game. If the didactical model of the instructor suggests that errors made should be displayed in text, the generator makes sure the text is displayed on the screen. Another example could be when a player has triaged a victim with success and the instructor tells the generator to construct a more difficult victim for the player to triage. Events discussed in section 6.4 are also the generator's responsibility.

Within this structure, every module has its own task. Because of this division of tasks, it can be very easy to for example create a new monitor and replace that with an existing monitor. It is also possible to attach multiple monitors to one evaluator or to use one monitor as input for two or more evaluators. In the case of this triage game, two paths are created: one for the movement and one for the triage procedure. These paths will be described in section 10.2 and 10.3 respectively.

Connections with the actual engine are tried to keep at a minimum to prevent too much coupling. The *CITS* class is the only class that can be controlled by the designer in the Hammer editor. All the other classes are hidden. Of course, in order to gain input of what the player has been doing and to change the game accordingly coupling is needed with the game (and engine). But this is only needed in the monitor (where input is gathered) and the generator (where the output is entered in the game). This makes it possible to use this structure in other engines than the Source engine. If a programmer changes the classes to function with the protocols defined in the other engine, the ITS loop prevails.

10.2 - Movement implementation

Two instances are implemented of this basic ITS framework described above. This section will describe in detail the first instance that is used in the game: the movement of the player. The UML class diagram is shown in figure 10.3. Readers that also want variable and function declaration of the classes are referred to appendix C.

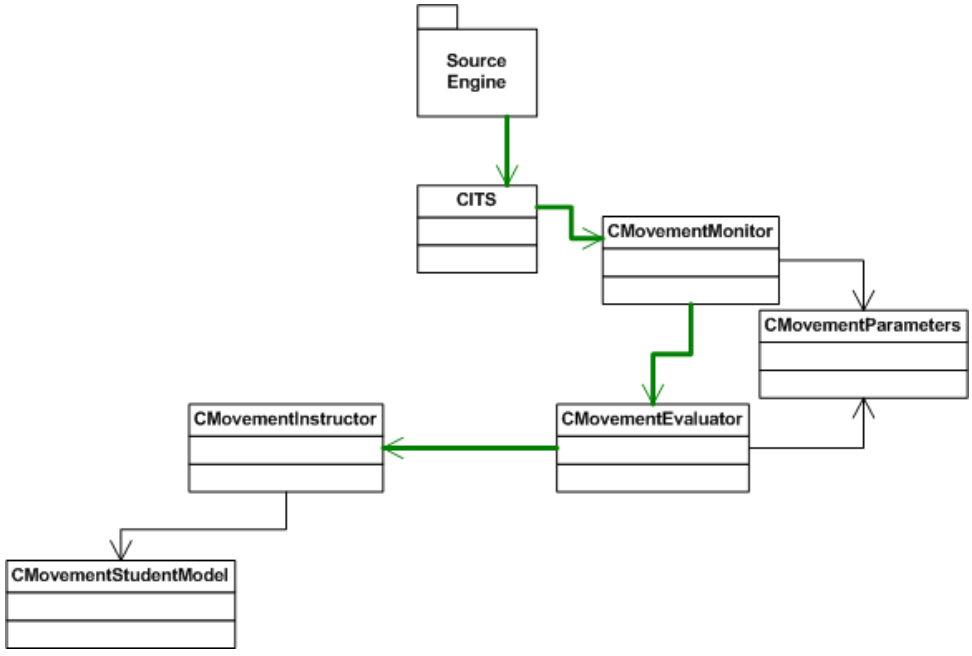


Fig. 10.3: UML class diagram of the implemented ITS loop concerning the movement topic.

As can be seen right away, the diagram is missing the generator module and 2 of the 3 models. By not implementing the generator it is shown that the ITS loop discussed doesn't necessarily has to be a loop. This means that the instructor in this case doesn't have to alter the game. The movement of the player doesn't have to be shown to the user and it doesn't influence the game. It's like the instructor only sits there, observing what the student is doing.

10.2.1 - Monitor

The *CMovementMonitor* class inherits its functionality from *CBaseMonitor* (shown in figure 10.2). It therefore has two important methods: `Observe` and `Report`. These two methods represent the monitor and the ITS loop, respectively.

In the `Observe` method information is gathered to come to a conclusion about the user action the player has been performing. For movement, these parameters are position, velocity, position of the camera, looking direction of the camera and keyboard buttons being pressed. Figure 10.4 displays the C++ code of retrieving these parameters.

```
// Retrieve movement information from the player
CBasePlayer *pPlayer = UTIL_GetLocalPlayer();

// Store the current values in the arrays
m_pvecOrigins[ m_nCycleCount ] = pPlayer->GetAbsOrigin();
m_pvecVelocities[ m_nCycleCount ] = pPlayer->GetAbsVelocity();
m_pvecEyes[ m_nCycleCount ] = pPlayer->EyePosition();
pPlayer->EyeVectors( &m_pvecForwards[ m_nCycleCount ], NULL, NULL );
m_piButtonsPressed[ m_nCycleCount ] = pPlayer->m_nButtons;

// Make sure the next values are stored in the arrays
// NOTE: If there are more than the allowed cycles, the last value
will be overwritten
if( m_nCycleCount + 1 < MONITOR_MOVEMENT_MAX_CYCLES )
{
	m_nCycleCount++;
}
```

Fig. 10.4: Code sample of the Observe method in CMovementMonitor.

As can been seen in the figure, all the parameters can be retrieved from Source's player class. For every call to the `Observe` method, all parameters will be stored separately in an array.

The reasoning concerning the observed parameters will take place in the `Report` method. First of all, the parameters that are sent to the evaluator are created. In table 10.1 all those parameters are described.

Table 10.1: Movement parameters sent to the evaluator (via CMovementParameters).

Parameter name	Description
`m_vecOrigin`	The last position of the player.
`m_vecVelocity`	Vector indicating in what direction the player is moving with a certain speed.
`m_vecEyes`	The last position of the camera.
`m_vecForward`	Vector indicating the direction of the camera.
`m_pLookTarget`	The entity the player is looking at.
`m_pWalkTarget`	The entity the player is walking towards.

All these parameters are taken or calculated from the information retrieved in the `Observe` method. The only task that needs to be done for the monitor is to find an appropriate user action for the observed behaviour. There are 7 user actions this monitor can choose from, covering all the possible user actions concerning movement the player can perform in the game. Table 10.2 displays these user actions for movement.

Table 10.2: User actions concerning the movement of the player.

User action	Description
`STANDING`	The player hasn't moved.
`WALKING`	The player walks forwards or backwards.
`SPRINTING`	The player sprints in any direction (pressing <Shift> while walking).
`CROUCHING`	The player crouches (pressing <Ctrl> with or without walking).
`JUMPING`	The player jumps (pressing <Space>).
`STRAVING`	The player moves sideways.
`MOVING`	The player isn't pressing any buttons, but is still moving (e.g. escalator).

The appropriate user action is selected by checking if the player has moved in the time interval. Subtracting the first observation from the last observation of the position of the player, gives an indication of how far the player has travelled. The recorded buttons being pressed by the player checks whether a player has been sprinting, crouching, jumping, straving or walking. After finding the user action, the user action along with the created parameters is passed on to *CMovementEvaluator*.

10.2.2 - Evaluator

The *CMovementEvaluator* class receives the user action and the related parameters by means of its `Evaluate` function. Again, this function (and the `CalculateScore`) is inherited from its base class (*CBaseEvaluator*). However, the evaluator is not evaluating the user action. It is merely recording the time. In the *CMovementParameters* one extra parameter is stored (because of the inheritance with *CBaseParameters*). This is a timestamp of when the parameters were created and passed on by the monitor. This timestamp comes from the Source engine and is the amount of time spent from the start of the level. The variable is recorded in seconds. Figure 10.5 shows the `CalculateScore` method where the evaluator calculates the time spent performing the user action.

The time elapsed is returned and sent with the user action to *CMovementInstructor*.

```
//-----------------------------------------------------------------------------
// Purpose: Calculate the score for the player's action
//-----------------------------------------------------------------------------
int CMovementEvaluator::CalculateScore( CMovementParameters *pParameters )
{
	// Calculate the time spent between two calls
	float flTimeElapsed = pParameters->m_flTimeStamp - m_flPreviousTimeStamp;
	// Update the new time stamp
	m_flPreviousTimeStamp = pParameters->m_flTimeStamp;

	// Return the time elapsed in cs as the score
	return (int) ( flTimeElapsed * 100 );
}
```

Fig. 10.5: Code sample of the CalculateScore method of the CMovementEvaluator.

10.2.3 - Instructor

Every instructor that is a descendant from the *CBaseInstructor* class should have a method called

```
//-----------------------------------------------------------------------------
// Purpose: Find a proper instruction for the generator to do
// NOTE - sExplanation is always "" and nScore gives the time elapsed in cs
//-----------------------------------------------------------------------------
void CMovementInstructor::Instruct( const char *szUserAction, int nScore,
                                    char *sExplanation )
{
	CMovementStudentModel *pStudent = dynamic_cast<CMovementStudentModel *>(
m_pStudentModel );

	// Update the student model for each user action seperately
	if( !Q_strcmp( szUserAction, "JUMPING" ) )
	{
		// We're counting the number of jumps, so add one for every jump
		pStudent->AddToValue( szUserAction, 1 );
	}
	else if( !Q_strcmp( szUserAction, "MOVING" ) )
	{
		// DO NOTHING
	}
	else
	{
		// Add the time elapsed to the given user action
		pStudent->AddToValue( szUserAction, nScore );
	}
}
```

Fig. 10.6: Code sample of the Instruct method of the CMovementInstructor.

`Instruct`. This method is used by the evaluators to provide the instructor with information like the user action, score and an explanation. For *CMovementInstructor*, the explanation is left empty and the score is the time elapsed in centiseconds.

Since the evaluator didn't assess the users behaviour, the instructor is only left with updating the student model. Figure 10.6 displays this code of the `Instruct` method. Once the game is finished, the instructor writes the student model to a text file. This will be discussed in chapter 11.

10.3 - Triage implementation

The movement implementation doesn't show the complete structure of the base ITS. Especially when it's meant to only record the player's actions. The implementation that is needed for the triage procedure is using the complete structure. The class diagram is presented in figure 10.7.

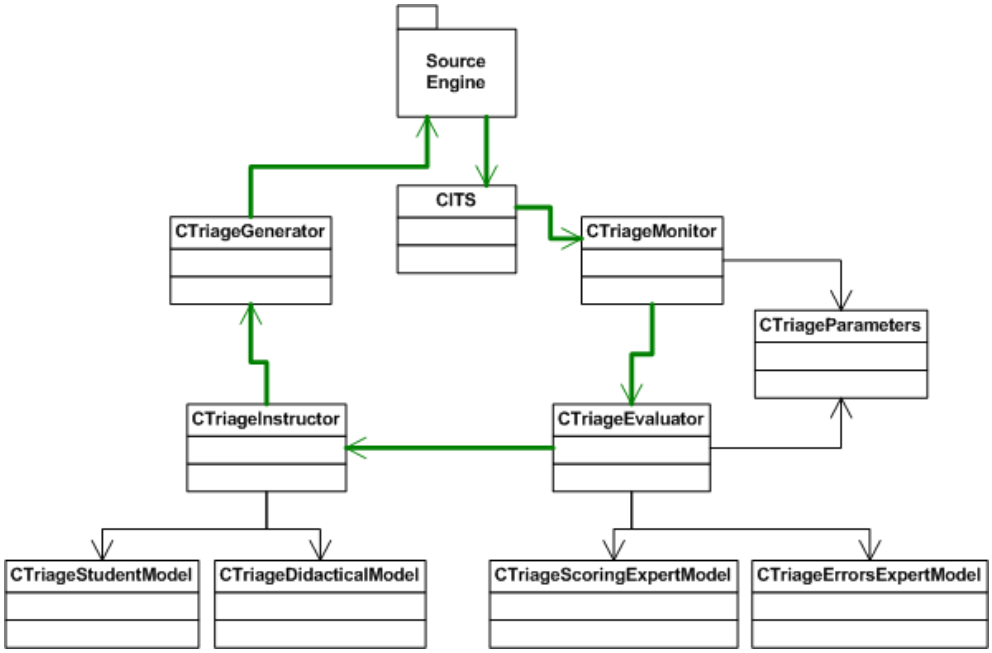


Fig. 10.7: UML class diagram of the implemented ITS loop concerning the triage procedure.

10.3.1 - Monitor

Unlike *CMovementMonitor*, this monitor doesn't need to probe for its information. The *CTriageMonitor* class is an event-based monitor that processes information given from other sources.

Table 10.3: Triage parameters sent to the evaluator (via CTriageParameters).

Parameter name	Description
`m_pVictim`	The victim in focus with all victim parameters.
`m_bChangedAirway`	Did the user change the perceived airway?
`m_bChangedBreathing`	Did the user change the perceived breathing rate?
`m_bChangedCirculationHR`	Did the user change the perceived heart rate?
`m_bChangedCirculationCRT`	Did the user change the perceived CRT?
`m_bChangedCirculationRR`	Did the user change the perceived syst. blood pressure?
`m_bChangedDisabilityEyes`	Did the user change the perceived eye response?
`m_bChangedDisabilityMotor`	Did the user change the perceived motor response?
`m_bChangedDisabilityVerbal`	Did the user change the perceived verbal response?
`m_bChangedDisabilityGCS`	Did the user change the perceived GCS?
`m_bChangedDisabilityGCSScore`	Did the user change the perceived GCS score?
`m_bChangedDisabilityTRTS`	Did the user change the perceived TRTS?
`m_bChangedClass`	Did the user change the perceived triage class?
`m_bVictimLeavingArea`	Did the user ordered the victim to leave the area?
`m_szCheckedParameter`	The victim parameter that was checked by the user.
`m_sOrder`	The order of user actions starting from the focus.

The sources in this case are the victims and menus described in chapter 9. The `Observe` method of this monitor is therefore not implemented.

In order to know which victim the player is treating, the victim classes report this whenever a player presses the use button when in range with a particular victim. This is done via the `FocusVictim` method in *CTriageMonitor*, providing the unique name of the victim in focus. Once the monitor has received this message, it sends out the user action TRIAGE_VICTIM with the corresponding *CTriageParameters* to the evaluator. The parameters that are sent along are described in table 10.3.

Not every parameter is used for every user action. The TRIAGE_VICTIM user action for instance only fills in the `m_pVictim` parameter and the timestamp provided by *CBaseParameters*. Its up to the evaluator to use the correct parameters per user action.

Once the *CTriageMonitor* class has focused a victim, the menus can give certain information on what the user thinks or does with the victim. All the choices the user can make in the choice menus (see figure 9.9 & 9.12) are user actions defined in the monitor. An example could be when a player brings up the triage card in the SORT part of the game. After editing the card and closing the window, all data entered are sent to the monitor in individual commands. The monitor alters the appropriate perceived parameter value (since this is the perception of the user) of the focused victim and marks the parameters changed in *CTriageParameters*. This action is defined as user action EDIT_CARD.

For the classification of the victim into one of the triage classes the same procedure occurs. Once the player has pressed the corresponding button the menu sends the message with the triage class to the monitor. The monitor alters the perceived value of the triage class, sets the `m_bChangedClass` parameters to 'true' and sends the evaluator the user action GIVE_CARD, along with the constructed *CTriageParameters* class.

For the SIEVE part, the classification menu waits with sending of the command until the explanation menu has sent its information. This way the evaluator can still reason about why the user has triaged a victim in a particular class. Else, the GIVE_CARD user action is sent before the EDIT_CARD user action, creating all kinds of reasoning errors discussed in the next section.

Once the player is too far away from the focused victim, or is starting to treat another victim, the previously focused victim gives the monitor the message to unfocus. The monitor will clear all the information of the focused victim, but not until it has sent a last report to *CTriageEvaluator* with the LEAVE_VICTIM user action. This gives an indication that the user has left the victim and reasoning can be done on the order of the triage procedure.

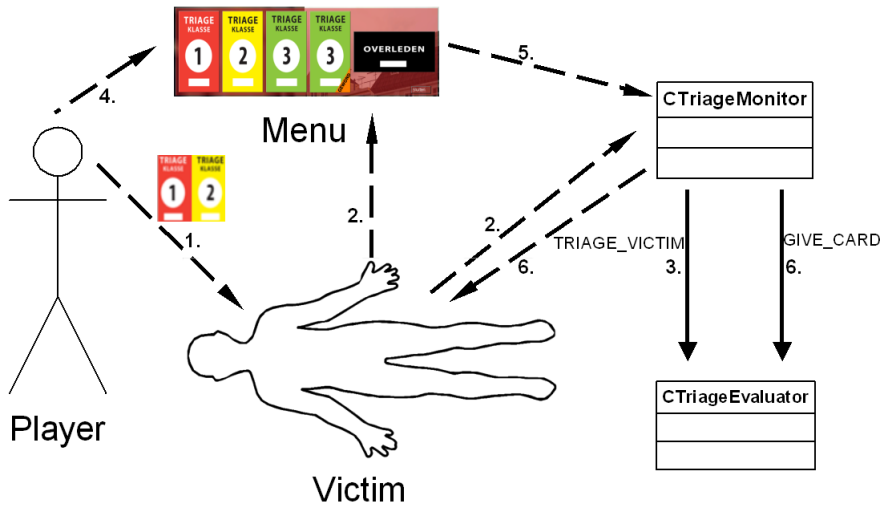


Fig. 10.8: Interaction example between the classification menu, victim and triage monitor.

The displayers discussed in section 9.2 also create messages for *CTriageMonitor* to receive. Whenever one of the displayers becomes activated by the player (because the player looks at the region of interest on a victim) a message is sent to the monitor to indicate the player is checking for a victim parameter. This is passed on to the triage evaluator as a TRIAGE_VICTIM user action. When

the player stops checking for this parameter, the displayer notifies the monitor and the monitor will pass this on to the evaluator with a CHECK_VICTIM user action.

Figures 10.8 & 10.9 display some interaction examples between a victim, menu, displayer and the *CTriageMonitor* class. In figure 10.8, the player wishes to classify a victim (1). The victim reports this to the triage monitor and displays the appropriate menu (2). The triage monitor focuses the victim and sends the TRIAGE_VICTIM user action towards the evaluator (3). Whenever the player has chosen the triage class (4), the classification menu sends this triage class to the monitor (5). In response, the monitor alters the perceived value of the triage class in the victim's properties and sends the GIVE_CARD user action to the evaluator (6).

In figure 10.9, the player starts out by pointing the mouse at a wrist of the victim (1). The *CCirculationDisplayer* class gets activated and emits the heart rate sound, while alerting the monitor via the attached victim (2). The triage monitor focuses the corresponding victim and alerts the evaluator with a TRIAGE_VICTIM user action (3). When the player stops checking (4), The displayer stops emitting sounds and notifies the triage monitor (5). This will send over the CHECK_VICTIM user action with all the relevant information (6).

Instead of asking every class involved for an update, this monitor waits until the appropriate class sends out a signal. It's therefore less demanding on the ITS loop and on the game itself, making the game run faster than a triage probing monitor. With this triage monitor, it also becomes easier to create a new menu or displayer or replace an existing menu. The only thing that needs to stay the same is the commands sent over to the monitor. The monitor doesn't need to know what menu or displayer is sending the command, as long as the command is sent properly.

Table 10.4 shows all the user actions this triage monitor can send over along with a description of the action performed by the player.

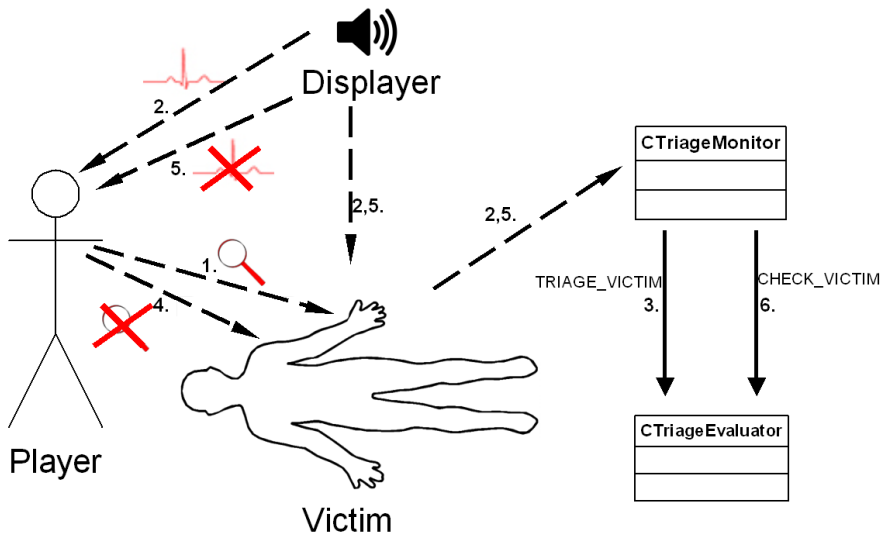


Fig. 10.9: Interaction example between a displayer, victim and triage monitor.

Table 10.4: User actions concerning the triage procedure.

User action	Description
TRIAGE_VICTIM	The player starts a certain action.
CHECK_VICTIM	The player has checked a victim parameter.
EDIT_CARD	The player has entered data on the triage card.
GIVE_CARD	The player has classified the victim into one of the triage classes.
LEAVE_VICTIM	The player has left the victim.
OPEN_AIRWAY	The player opens the airway of the victim.
COMMAND_LEAVE_AREA	The player commands the victim to leave the area.

10.3.2 - Evaluator

As with *CMovementEvaluator*, the *CTriageEvaluator* class has the `Evaluate` and `CalculateScore` methods as main focus. These methods are however a bit more complicated than with the movement evaluator.

The monitor gives the TRIAGE_VICTIM user action every time the user starts a new user action. It functions only as a time stamp so the evaluator can measure the time it took for the player to perform a certain task. The evaluator will store 2 variables derived from the TRIAGE_VICTIM user action. One is the start of a particular action that will be given by the monitor. The other is the overall start of the treatment time of one victim. It is very important for an instructor to know how long a player has been treating one victim. It gives an indirect skill factor of the player. The faster the triage process per victim, the better the player is.

Of course, speed in the triage procedure isn't the only factor that will make a good medical worker. Therefore the evaluator tries to find errors in the user's actions. *CTriageErrorsExpertModel* helps the evaluator with this task. The evaluator will pass all information given by the monitor (the current user action and the triage parameters) plus the spent time on the user action to the expert model and will retrieve a list of errors that the user has made.

The expert model uses Valve's *CResponseSystem* class. This is a simple and effective system that generates responses given a set of criteria and rules. The system is used to help the NPCs in the game Half-Life 2 respond to certain events. The responses in that case were mainly links to audio files the NPC would say. With some extra programming to the *CResponseSystem* class, the expert model could very well use this class to find errors.

The criteria, rules and responses have to be declared in a text file. This has the major advantage that expert knowledge isn't merged into the code project, so experts can change a simple text file to alter certain parts of knowledge. This can be done without editing the code project, thus without intervention from a programmer. An example of the criteria rules and responses in such a text file is shown in figure 10.10.

```
criterion "EditingCard" "user_action" "EDIT_CARD" required
criterion "BreathingChanged" "breathing_changed" "1" required
criterion "BreathingTooHigh" "breathing_actual_max" "<key:breathing_perceived"
required

rule "BreathingTooHigh"
{
	criteria	EditingCard BreathingChanged BreathingTooHigh
	response	ERROR_BREATHING_TOO_HIGH
}

response "ERROR_BREATHING_TOO_HIGH"
{
	print "ERROR_BREATHING_TOO_HIGH"
}
```

Fig. 10.10: Example of a rule and some criteria to come to the ERROR_BREATHING_TOO_HIGH response.

In this example, the expert model is checking if the user made an error of perceiving the breathing rate higher than it actually was. 3 criterions are used for this check: `EditingCard`, `BreathingChanged` and `BreathingTooHigh`. `EditingCard` is a criterion that checks if the current user action is the EDIT_CARD user action. In table 10.3, a parameter is given to indicate if the user has changed the perceived breathing rate (`m_bChangedBreathing`). This parameter is used to check if the entered breathing rate was incorrect or not. If the user didn't change the breathing rate (incorrect or not) the expert should not check if the breathing rate is correct. The last criterion is the actual criterion that checks if the perceived breathing rate (`breathing_perceived`) is higher than the actual breathing rate of the victim.

`breathing_actual_max` is a parameter indicating the maximum breathing rate that is considered to be correct. A margin of error is allowed for counting breaths and heartbeats per minute. If a player was to count and missed one beat for the 15 seconds he/she was counting, the error would be 4 beats per minute. This is the margin of error that is allowed for the user.

If all the criteria are checked with the entered parameters and all turn out to be true, the ERROR_BREATHING_TOO_HIGH rule is fired and the system responds with the string "ERROR_BREATHING_TOO_HIGH". This is done for every rule (i.e. possible error that could occur) in the text file. In total there are 32 errors, described with 47 rules, 112 criteria and 32 responses. To see all these rules, the reader is referred to appendix D.

The *CResponseSystem* class only gives one response per run. Unfortunately, there can be more than one error made by the player. So the expert system must be run multiple times. To prevent reporting the same error multiple times, an error resolver is used to disable the fired rule. This is done by means of the `breathing_changed` parameter. Whenever a certain error pops up, the expert model looks up in a text file which `..._changed` parameter needs to be turned off. In the case of the ERROR_BREATHING_TOO_HIGH or ERROR_BREATHING_TOO_LOW errors the `breathing_changed` parameter is set to 0, to prevent the rules from firing. Whenever the ERROR_NONE response comes up, the expert model knows there are no (more) errors and reports all the errors found in a list to the evaluator.

For the `CalculateScore` method another expert model is used: *CTriageScoringExpertModel*. The user action will be rated with a score between 0 and 100. The method will start out with 100 points and will deduce points for every error made. *CTriageScoringExpertModel* is the class that, given the error made finds the appropriate penalty. The expert model uses Valve's altered *CResponseSystem* as well and this time the response is a number that is subtracted from the total score. This gives the expert, who can change this text file, more control over what error is more important than other. For instance, a player checking a vital sign for 30 instead of 15 seconds can get a penalty less than someone who is forgetting to check a vital sign at all.

Once the errors are found and the score is calculated, the *CTriageEvaluator* class sends over the current victim's unique name, the user action, the calculated score and the list of errors to the instructor.

10.3.3 - Instructor

The instructor receives information from *CTriageEvaluator* via the `Instruct` method. This method updates the student model, updates the skill level of the user and asks the didactical model for feedback.

The triage student model mainly records errors made by the student. Other parameters like the time spend on the SIEVE or SORT part of the game, the number of victims triaged and the average score per user action is also stored. More information on how these variables are used for evaluation is discussed in chapter 11.

The score calculated by the evaluator is updated into a global skill level for the current level. This global skill level is always visible in the top-right corner of the screen while the player is playing a game. It starts out with 100 points and drops down whenever the player makes mistakes. The instructor updates this skill level via the *CTriageGenerator* class.

It is also possible to show the actual score per user action. The user must have this option checked in the main menu. Whenever it is checked, the instructor also passes the received score from the evaluator on to the screen via the generator.

Whenever the instructor has updated all the models and variables, it asks the didactical model to form a strategy on if and how to approach the player. For now, the didactical model only responds

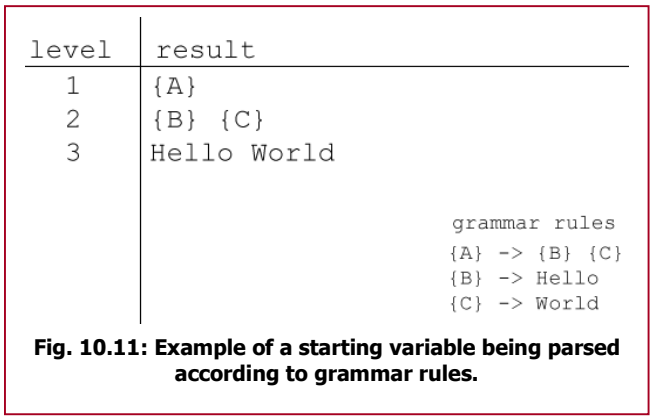


Fig. 10.11: Example of a starting variable being parsed according to grammar rules.

when a LEAVE_VICTIM user action is performed. If this is the case and the user has reported he/she wants direct feedback in the game, the model transforms the list of all errors made on the victim into a feedback sentence, which will be displayed on the screen by the generator.

This procedure is done by means of a grammar class. This class (*CGrammar*) again uses the *CResponseSystem* class like the expert models in *CTriageEvaluator*. The difference however is that the rules described in the grammar's text file can be seen as rules to parse certain variables with. An example is described in figure 10.11.

CGrammar loads in a certain set of grammar rules defined by the expert in a text file just like figure 10.10. Whenever the `Parse` method of the class is called with a certain start variable and start level, *CGrammar* will replace the starting variable according to the set of grammar rules for a given number of levels. Additional parameters can also be sent with the grammar to be used in the grammar rules. This can generate more specific results in certain situations.

For the case of a feedback sentence, special parameters like the number of errors and the errors that were made are entered in the *CGrammar* class. An example of some of the grammar rules is shown in figure 10.12. In this example, there are 3 grammar rules: `OneError`, `TwoErrors` and `BreathingTooHigh1`. The first two rules can only be fired whenever the variable is '`Start`'. The rules create a layout of how this sentence should be build. The latter rule parses one of the slots into a sentence where the ERROR_BREATHING_TOO_HIGH error is explained.

```
criterion "VariableStart" "variable" "Start" required
criterion "VariableSlot1" "variable" "Slot_1" required
criterion "NumErrors_0" "num_errors" "0" required
criterion "NumErrors_1" "num_errors" "1" required
criterion "NumErrors_2" "num_errors" "2" required
criterion "Slot1_ERROR_BREATHING_TOO_HIGH" "slot_1" "ERROR_BREATHING_TOO_HIGH"
required

response "ONE_ERROR"
{
	print "{Slot_1}"
}
response "TWO_ERRORS"
{
	print "{Slot_1},\n{Slot_2}"
}
response "ERROR_BREATHING_TOO_HIGH"
{
	print "De ademfrequentie zou \<breathing_act\> moeten zijn"
	print "#NIFV_Feedback_Breathing_Too_High"
}

rule "OneError"
{
	criteria	VariableStart NumErrors_1
	response	ONE_ERROR
}
rule "TwoErrors"
{
	criteria	VariableStart NumErrors_2
	response	TWO_ERRORS
}
rule "BreathingTooHigh1"
{
	criteria	variableSlot1 Slot1_ERROR_BREATHING_TOO_HIGH
	response	ERROR_BREATHING_TOO_HIGH
}
```

Fig. 10.12: Example of some grammar rules that are applied in the feedback grammar.

In the response of the `BreathingTooHigh1` rule there are two options to be printed. *CResponseSystem* automatically chooses one of these options at random. The first sentence (in Dutch) makes use of one of the parameters of the victim. The didactical model receives the complete sentence and enters the variables used into the actual parameter values. The second sentence is a reference to a

stored sentence in another text file. This is a special localisation file used in the source engine to provide text in a language of the user's choice.

The references are used, because else the complete sentence (that could describe 6 errors at its maximum) would be too large to send over. The Source engine keeps a strict policy on how large a string can be for it to be send over from server to client or vice versa. With the references this maximum is never reached.

Whenever the player has triaged all the victims in the virtual world, the didactical model uses the feedback grammar to inform the player of the ending of the game. If the player didn't make too many errors, the feedback will be to go on to the next level. If not, the user is kindly requested to perform the triage procedure at the current level again.

10.3.4 - Generator

The *CTriageGenerator* class is the executive force of the ITS. It receives orders from *CTriageInstructor* and this class manipulates the game to get to the desired goal.

For this project, the generator only displays text on screen. A simple function that addresses the correct methods within the Valve Source engine. This could be easily extended to play audio files with certain feedback, or to show images instead of text.

Another functionality of this generator could be to dynamically change certain objects over time, like discussed in section 6.4. A victim with deteriorating vital signs should be implemented in this generator. The instructor could then simple give the order to create such events.

The *CScenarioGenerator* class also has more functionality to change the world. This generator will be introduced in part 4 of this report.

Chapter 11 - Evaluation

When trying to learn a new topic or training an already learned topic, evaluation of the student's actions is not to be missed. For a serious game it is a crucial factor and it must also be extended outside the game. In a learning environment where an instructor must be able to present solutions and explanation to previous made errors or questions, it is also very important to note everything that has happened.

Creating a solid environment where the student can ask virtually any question concerning a topic is something that is not yet possible for computers to do. A real-life instructor is needed to provide additional information and explanation that are left unanswered by the system. Especially when multiple points of view can be applied to explain certain topics.

Therefore, playing the serious game is in this case not the only thing that provides learning. After the game has been played, an instructor could very well discuss the results with the student to go deeper into things that went wrong or right. Instead of providing an evaluation in the game, it must also be possible to print out certain information to discuss afterwards.

11.1 - In-game evaluation

As shown in section 10.3, the ITS loop provides certain information on how the student is doing while playing the game. The student however has the ability to show this feedback or not. When starting a new game (see figure 11.1) the player is asked if it wants certain feedback presented or not.

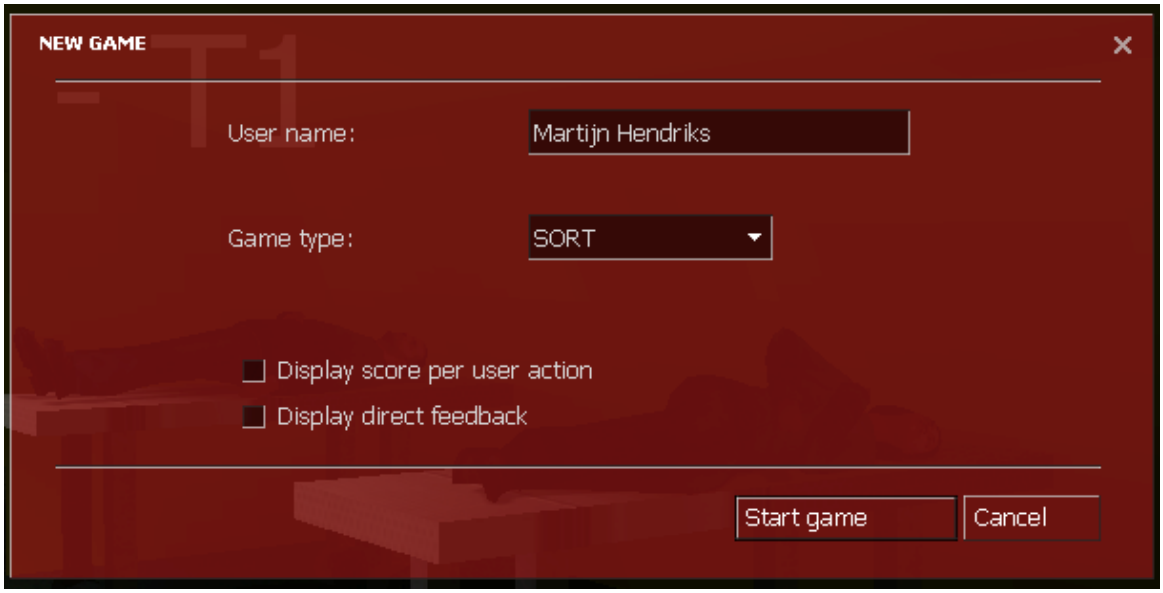


Fig. 11.1: The new game panel at start up of the game. In the middle are two check boxes (one for the score of every user action and one for the feedback sentences) that can be checked whenever the player wants more feedback.

Before introducing these options, there is one feedback parameter that is always shown to the player. This is the skill level of the player and it's shown on the top-right corner (see figure 11.2). The skill level is a score that represents the skill of the user for a given level. Every start of a level, the skill is set to 100. Whenever a player does something that is considered as wrong by the evaluator and instructor, the skill level will decrease. The main goal of the game is to keep the skill level at 100 points. The skill level will eventually decide whether or not the player is allowed to go to the next level.

Fig. 11.2: The skill level of the user always shown in the top-right corner of the screen.

The first feedback option displayed in figure 11.1 is to display all the scores evaluated by the evaluator. This gives the score of all the user actions the player is performing. Whenever the player for

Fig. 11.3: A score for a performed user action is shown below the skill level.

instance has edited the triage card in the SORT part of the game, a number below the skill level is shown like in figure 11.3. It gives an indication if the user is on the right way of triaging the victim as supposed to. It could be useful for the player to see for instance if the edited values on the triage card where correct or not. If the score isn't optimal, the player can alter the values that he/she thinks are incorrect before the actual feedback is given. The score however doesn't change whenever a player has changed the values to the correct ones. The error still needs to be reported since the player should have entered the wrong values when feedback wasn't provided.

The second feedback option is to allow the instructor to point out what went wrong for each victim while playing the game. Figure 11.4 shows an example of a player that has made three errors to the last triaged victim. With this feedback

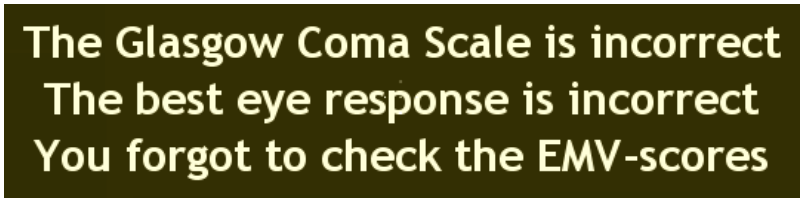


Fig. 11.4: Example of a feedback sentence displayed after a victim has been triaged.

the student knows what went wrong. He/she doesn't need to alter the values entered into the correct ones. This feedback is only provided to avoid repetition of certain mistakes.

After the player has triaged all the victims in the world, *CTriageInstructor* (together with the didactical model) gives a concluding advise. If the user kept the 100 points throughout the game, the instructor will give the advise to go on to the next level. If the points range between 65 and 100 points, the system lets the user decide whether to go one or to repeat the current level. Below the 65 points, the user is advised to repeat the current level, as shown in figure 11.5.

Fig. 11.5: The user is informed to go to the evaluation report and replay this level again.

11.2 - Evaluation report

Even with all the feedback options turned on in game, the user must be able to view his/her mistakes after the game has been played. Even during the game an evaluation report must show the errors made in a clear and understandable fashion.

For this purpose the *CRecorder* class is created. This class records all relevant information concerning victims. For every created victim in the virtual world there is a *VictimRecord* being managed in *CRecorder*. This *VictimRecord* contains all the parameters stored in *CVictimProperties* plus the treatment time and the errors made during the treatment.

CRecorder keeps these *VictimRecord*s in two separate lists: one list for the victims who already are triaged, one list for the rest. This way, the *CRecorder* class knows when all the victims are triaged. *CTriageEvaluator* logs the errors made to the *CRecorder* class, who logs them to the corresponding *VictimRecord*.

If the player wishes to check the current state of victims triaged, he/she can press <Esc> while in the game and select the 'evaluation report' button in the main menu. The evaluation report pops up as shown in figure 11.6.

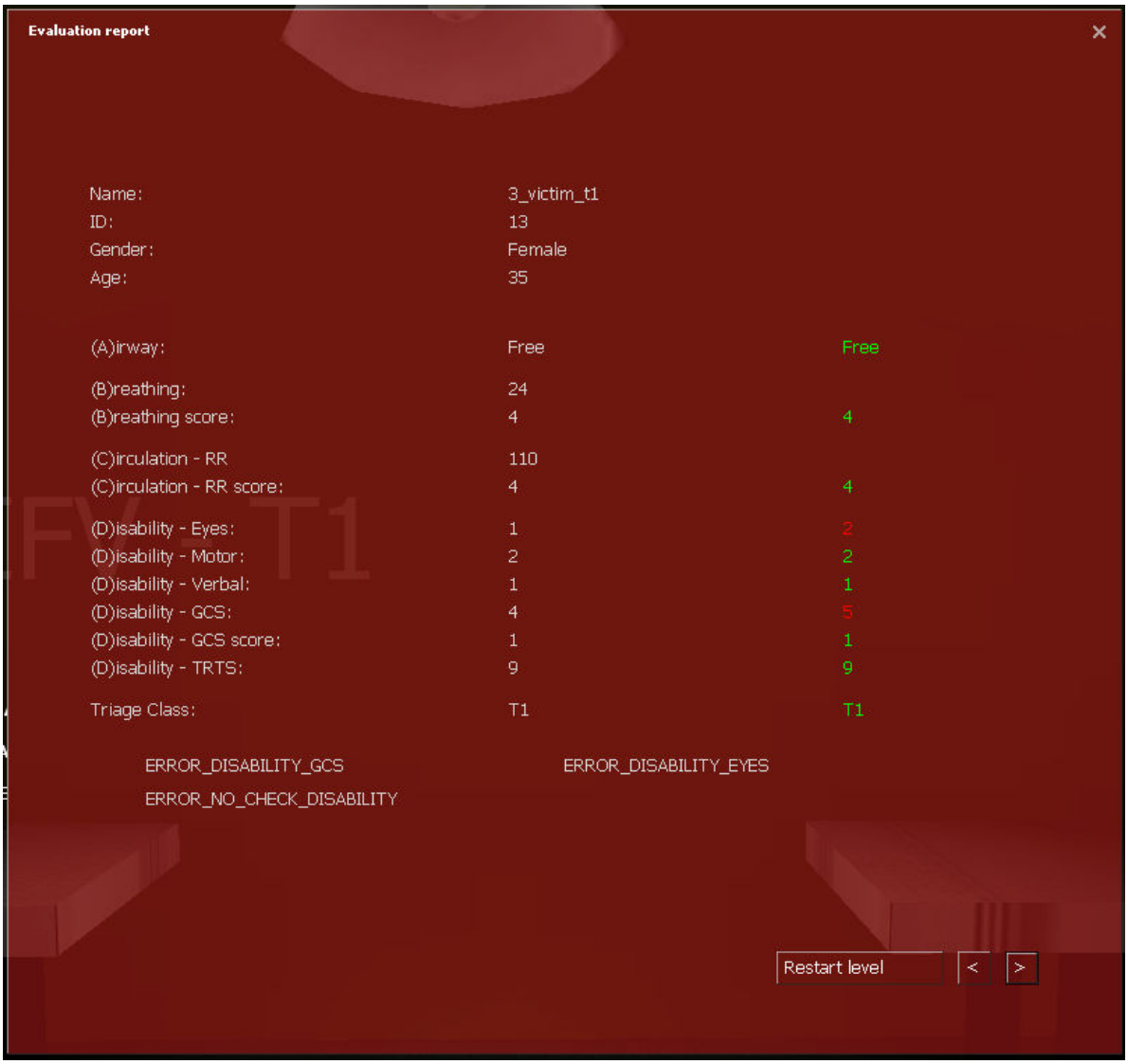


Fig. 11.6: The evaluation report displaying the user's conclusions compared to the actual victim parameters.

Here, all parameters from one victim are shown, both the actual and the perceived values by the user. Colours are used to highlight the parameters that match (in green) and don't match (in red) with the actual value. At the bottom of the panel are the errors made by the user.

The player can browse through all the victims that he/she has triaged in the current game with the two bottom-right buttons. Next to these buttons is the button that restarts the current level. If the user has kept his skill level high enough, another button will appear next to the restart button to go to the next level.

11.3 - Logs

In order to provide an evaluation to the instructor of a course featuring this game, the system stores all important values to text file. Both *CRecorder* class discussed above and the student models within the ITS report their findings to a text file. In total, there are 4 text files with different information per student. These files can be found in a special user map within the game directory. All the information that is recorded in the text files is carefully discussed with instructors from the NIFV to get a better

view of what parameters are needed for such instructors to provide relevant feedback after a game has been played.

The first file that is presented for each user is the general student model. Figure 11.7 displays the parameters stored in this text file. This general model gives the global skill level of the player (both movement and triage). This parameter is calculated by means of the two `skill` parameters discussed below. The `time_played` parameter is the total time the player has been playing the levels.

```
"parameters_general"
{
	"skill"			"67"
	"time_played"		"608"
}
```

Fig. 11.7: Example of a general student model output. The skill value is between 0 and 100 and the time_played is in seconds.

The second file is provided by the student model concerning movement. In this file (see figure 11.8) all the user actions performed by the player on the movement part are noted with the time in seconds spend on this user action. The `JUMPING` parameter is accompanied with the number of times jumped. The first parameter in this file is one of the `skill` parameters the global `skill` value is taking into account. Since it is very hard to determine when a player has mastered the movement part of the game, a simple algorithm has been created to calculate a skill level. The number of seconds the player has walked is divided by 180 (3 minutes) and this outcome is multiplied by 100 to come to a `skill` value. This means the player is 100% skilful whenever he/she has walked 3 minutes or more in the game.

```
"parameters_movement"
{
	"skill"			"14"
	"STANDING"		"569"
	"walking"		"26"
	"SPRINTING"		"0"
	"JUMPING"		"0"
	"CROUCHING"		"0"
	"STRAVING"		"2"
}
```

Fig. 11.8: Example of the movement student model output. This player has been standing for almost 10 minutes and walked for only 26 seconds.

The third text file for the further evaluation of the player is the triage part of the student model. An example of this is displayed in figure 11.9. This file contains the time played specified in the two parts of the game, followed by the skill level of the user for each user action. The number of times the user action was performed is also noted. Then all the errors that were made during all the sessions are recorded in this file. The skill value is calculated by averaging the skills of the user actions EDIT_CARD, GIVE_CARD, LEAVE_VICTIM and CHECK_VICTIM. These user actions are the most important user actions for the triage process and should be carried out best by the player.

These 3 student model files are all updated while the user is playing the game and constantly adds new information to the list. The files are created once per user and are used throughout all the sessions the player will be playing.

The fourth file that is recorded is the log file of the *CRecorder* class. This file is appended with new information once a level has been played. It provides information about the level that was played and how victims were treated. In this log file, multiple entries are possible whenever the player has played the game several times. Figure 11.10 shows an example of the log's output.

```
--------------------------------------------------------------------------
User:  Martijn Hendriks
Map:   sort_final
Date:  Fri Apr 18 17:40:02 2008
--------------------------------------------------------------------------
17:43:41	Total time played: 608 seconds
17:43:41	Playing SORT
17:43:41	Current level: 1
17:43:41	Playing time: 133 seconds
17:43:41	Average treatment time: 44 seconds
17:43:41	Longest treatment time: 44 seconds
17:43:41	Shortest treatment time: 44 seconds
17:43:41	Variance in treatment time: 0 seconds
17:43:41	0 / 5 victims triaged correctly
17:43:41	ERROR_DISABILITY_TRTS	ERROR_NO_CHECK_BREATHING
	ERROR_DISABILITY_GCS	ERROR_DISABILITY_MOTOR
17:43:41	16	x	x
17:43:41	24		x	x	x
```

Fig. 11.10: Example of a user's log. The last three lines represent a table of errors made per victim. The victim is indicated by the index number of the database.

Instead of only the total time the player has been playing, the time for this level is also recorded. The average, longest, shortest and the variance in treatment time are recorded and the number of victims triaged correctly is logged. The final part of the log is a table of all the errors made concerning the victims triaged. This table (when copy-pasted into an Excel file) gives a clear overview of what errors were made to what victims. It gives the instructor possible misconceptions of the player. It is very easy to discover errors made constantly, by just scanning the columns of the table. With these 4 text files, the instructor should have an understanding of what happened in the game and how the player of the game applied his knowledge to the virtual environment.

Another file is kept for other reasons than tracking the player's performance throughout the game. An error log is created for each game started. This log keeps track of possible errors that might occur during the game. If the system crashes, the programmer is then able to correct the system with information from the error log.

```
"parameters_triage"
{
	"skill"                         "90"
	"time_played_sieve"             "0"
	"time_played_sort"              "608"
	"EDIT_CARD"                     "80"
	"num_EDIT_CARD"                 "10"
	"GIVE_CARD"                     "89"
	"num_GIVE_CARD"                 "9"
	"LEAVE_VICTIM"                  "94"
	"num_LEAVE_VICTIM"              "11"
	"CHECK_VICTIM"                  "99"
	"num_CHECK_VICTIM"              "29"
	"OPEN_AIRWAY"                   "0"
	"COMMAND_LEAVE_AREA"            "0"
	"num_victims_triaged"           "9"
	"ERROR_AIRWAY"                  "0"
	"ERROR_CAN_OPEN_AIRWAY"         "0"
	"ERROR_BREATHING_TOO_HIGH"      "0"
	"ERROR_BREATHING_TOO_LOW"       "0"
	"ERROR_CIRCULATION_HR_TOO_HIGH" "0"
	"ERROR_CIRCULATION_HR_TOO_LOW"  "0"
	"ERROR_CIRCULATION_CRT"         "0"
	"ERROR_CIRCULATION_RR"          "3"
	"ERROR_DISABILITY_EYES"         "1"
	"ERROR_DISABILITY_MOTOR"        "3"
	"ERROR_DISABILITY_VERBAL"       "1"
	"ERROR_DISABILITY_GCS"          "3"
	"ERROR_DISABILITY_GCS_SCORE"    "0"
	"ERROR_DISABILITY_TRTS"         "6"
	"ERROR_DISABILITY_AVPU"         "0"
	"ERROR_TRIAGE_CLASS"            "5"
	"ERROR_CHECKING_TOO_LONG"       "0"
	"WARNING_CHECKING_30_SECONDS"   "0"
	"ERROR_REASONING_NOT_T3"        "0"
	"ERROR_REASONING_T3"            "0"
	"ERROR_REASONING_NOT_DECEASED"  "0"
	"ERROR_REASONING_DECEASED"      "0"
	"ERROR_REASONING_T1_STEP_B"     "0"
	"ERROR_REASONING_T1_STEP_C"     "0"
	"ERROR_REASONING_T2"            "0"
	"ERROR_MATH_GCS"                "0"
	"ERROR_MATH_GCS_SCORE"          "0"
	"ERROR_MATH_TRTS"               "1"
}
```

Fig. 11.9: Example of a triage student model output. The time played, skill per user action and the total errors made are all recorded.

Part IV – Placement Algorithm

Chapter 12 - Designing a scenario

For a medical worker that plays the serious game several times, the game might become dull. The same process is repeated over and over again until the user has finished the game with success. It could very well be that after the second level in the game, the user is bored and stops playing the game.

In order to keep the medical worker immersed in the game for each level, the disaster area is created at random. This way, the disaster looks like something else in another region. When more disaster scenarios are created, the player must be alert every level to find the victims and triage them.

The same thing could have been accomplished by creating different levels by a level designer for each level in the game. Then, all victims and scenery are to be found at the same locations whenever a level is reloaded. The advantage of this approach is that once the level is finished, no unforeseen errors can be created. With the automated scenario creator, it could happen that a victim is misplaced and never to be found by the player.

However, when looked at the development of such maps, a lot of time is spent by communication between the level designer and the expert and new levels cannot be simply created by the expert himself. With this approach of an automated scenario creator, the expert can create the world in his/her own time with simple text files. As will be seen in the following sections, the expert also doesn't have to reinvent the wheel and can use earlier created entities in new scenarios.

12.1 - Structure

In order to automatically create a disaster scenario, another class structure needs to be build with use of the ITS structure in chapter 10 (see figure 10.1 & 10.2). Figure 12.1 displays the designed structure in an UML class diagram. Note the *CBaseGenerator* class at the bottom-left that is a base module from the ITS.

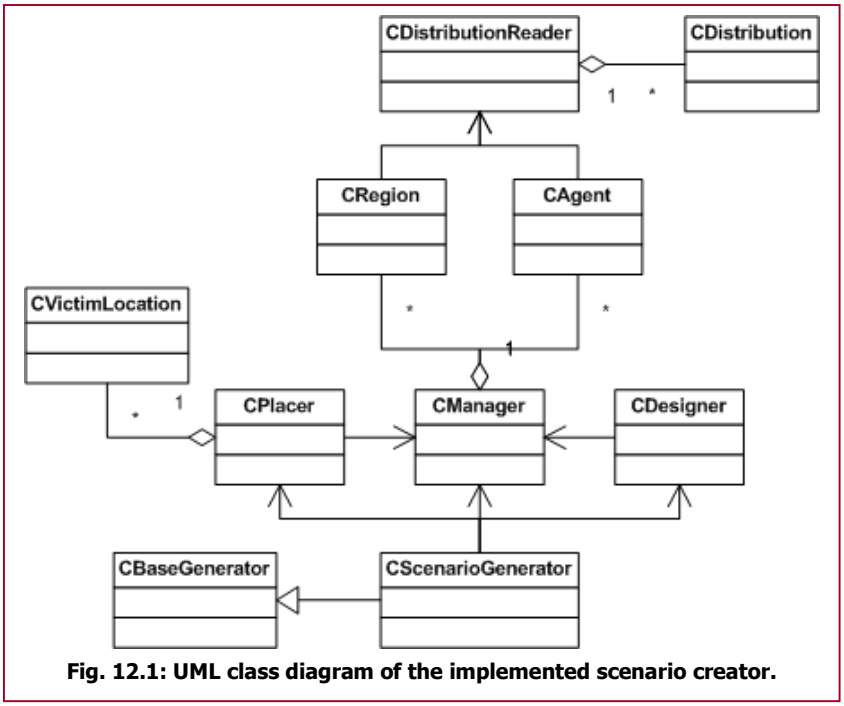


Fig. 12.1: UML class diagram of the implemented scenario creator.

At the bottom of the figure, a new generator is created: *CScenarioGenerator*. This generator inherits functionality from the *CBaseGenerator* class discussed in chapter 10. *CScenarioGenerator* is the main class in this scenario creator. However, the creation process is divided into subclasses with each their own task.

Whenever the Source engine has loaded the empty map, the *CITS* class instructs *CScenarioGenerator* to build the scenario. The generator will call *CDesigner* to create all the relevant entities according to the game type (e.g. SIEVE or SORT) being played. The designer stores these

entities in the *CManager* class and passes this back to the generator. Then *CPlacer* will be called to place all the entities in the manager towards their appropriate location.

12.2 - Regions and agents

To create the entities in the world, the *CDesigner* class uses a user-defined set of regions and agents. To categorize these regions and agents, the entities are placed in 7 groups. These groups will later be used by *CPlacer* to place the entities in order. Table 12.1 displays the groups and some examples. In total, there are 2 region groups and 5 agent groups. Groups have to be provided by the expert to each new agent or region that is created.

In figure 12.1, there are only the *CRegion* and *CAgent* classes. They are the classes representing all the entities that are needed for the game. This is because the expert creating the scenario must have all the control for the creation of new entities. If for instance the expert wishes to add a train to the scenario, he/she must be able to do so without entering new code or creating a new class. Of course, this cannot apply for all entities. Especially agents with complex behaviour (like for instance the victim) must be created in code. The focus lies here in creating simple objects, not including complex behaviour, without the intervention of a programmer.

Therefore the *CAgent* and *CRegion* classes have a primary type that specifies which model and additional information is needed. To prevent creation of a lot of types, an agent can have a secondary type that gives more information to the *CDesigner* and/or *CPlacer* class. The creation of a victim agent that has burns can be created much easier with this secondary type. Instead of creating an additional type for every victim agent (e.g. A_VIC_T1, A_VIC_T1_BURNS, A_VIC_T2 & A_VIC_T2_BURNS), the expert only needs to declare a secondary type and attach it to the victim agents that have burns (e.g. A_VIC_T1, A_VIC_T2 & A_VIC_BURNS).

Table 12.1: The groups that agents and regions must belong to.

Group name	Examples	Prefix
`Disaster`	The disaster region.	`R_DISASTER`
`Rest`	The other remaining regions like fire, crash of explosions.	`R_`
`Buildings`	All types of building entities or infrastructure.	`A_BUI_`
`Foliage`	Trees, plants, bushes, etc.	`A_FOL_`
`Vehicles`	Cars, trains, trucks, etc.	`A_VEH_`
`Victims`	Victims that need to be treated by the player.	`A_VIC_`
`Debris`	Fallen rocks, litter, etc.	`A_DEB_`

For the system to create the expert-defined entities, some protocol must be followed. In table 12.1, there are some prefixes defined. These prefixes must be applied to the types in order to create entities belonging to the given group. For instance, A_FOL_TREE is categorized in the foliage group. So is A_FOL_TREE_1, A_FOL_TREE_OAK, A_FOL_BUSH and A_FOL_GRASS. As long as this naming convention is applied, the expert can enter any name he/she wants. This name is then to be referenced later in the *CDesigner* class.

The victim agent is an agent type that is defined in the game itself. The expert is able to use types like A_VIC_T1, A_VIC_T2, A_VIC_T3, A_VIC_T3_WOUNDED and A_VIC_DECEASED to create a victim that must be classified as T1, T2, T3, T3 wounded and deceased, respectively. It is however possible to extend this type to for example A_VIC_T1_SPECIAL. For the system this type creates the ordinary T1 victim, but the expert can have special plans for this agent concerning the location in the world. This will be explained in later chapters.

Once the expert defines an agent without complex behaviour or a region, its representation and location can be altered to fit specific needs. For the placement of the region or agent, the reader is referred to chapter 13 & 14.

A region does not have a representation in the virtual world. The expert can only give agents a model in the world. This is done in the `models.txt` text file, which is located in the game's directory. An example of an entry in this file is given in figure 12.2. This example describes a car. There are 4 distinct models *CDesigner* can choose from at random. The entity that is tied to the chosen model is Valve's *CPhysicsProp* entity (called `prop_physics` in Hammer). This means that the entity is affected by physics in the game. The solid value indicates how other entities should collide with this agent (in

```
"A_VEH_CAR_TR"
{
	"count"		"4"
	"0"		"models/props_vehicles/car002a_physics.mdl"
	"1"		"models/props_vehicles/car003a_physics.mdl"
	"2"		"models/props_vehicles/car004a_physics.mdl"
	"3"		"models/props_vehicles/car005a_physics.mdl"
	"entity"	"prop_physics"
	"solid"		"SOLID_VPHYSICS"
	"offset"	"-15"
	"rotation"	"-1"
}
```

Fig. 12.2: An example entry of models.txt. This entry describes a car belonging to the vehicle group.

this case according to the model). An offset can be provided to place the agent in the vertical direction. This is sometimes needed to align the model to the surface of the level. Another additional parameter is the rotation of the entity. This ranges from –1 to 359 and indicates the angle in degrees by which the entity is rotated around the vertical axis. A value of –1 indicates the rotation is set randomly.

With these entries, an expert can create simple objects that will be placed by the *CPlacer* class according to user-defined distributions discussed in chapter 13.

12.3 - Grammars

To design a scenario, it is first important to know what entities the scenario needs. For both the SIEVE and the SORT part of the game, different scenarios must be created. As will be seen in this section, the SIEVE part needs more defined entities in the world, because the disaster area can vary from time to time. The treatment area however can be in the same setting every level. The only thing that must change is the selection of victims.

Both SIEVE and SORT use the same structure to the entities that needs to be created. This is the *CGrammar* class discussed in section 10.3.3. The following subsections will explain how this is done per game type.

12.3.1 - SIEVE

In the SIEVE, the game must create a disaster area where the player can walk around, searching for victims. For this project a scenario was created where a car has collided with several bicyclers. To produce such a scenario, an empty map needs to be created with a simple road scheme. This map can be seen in figure 12.3.

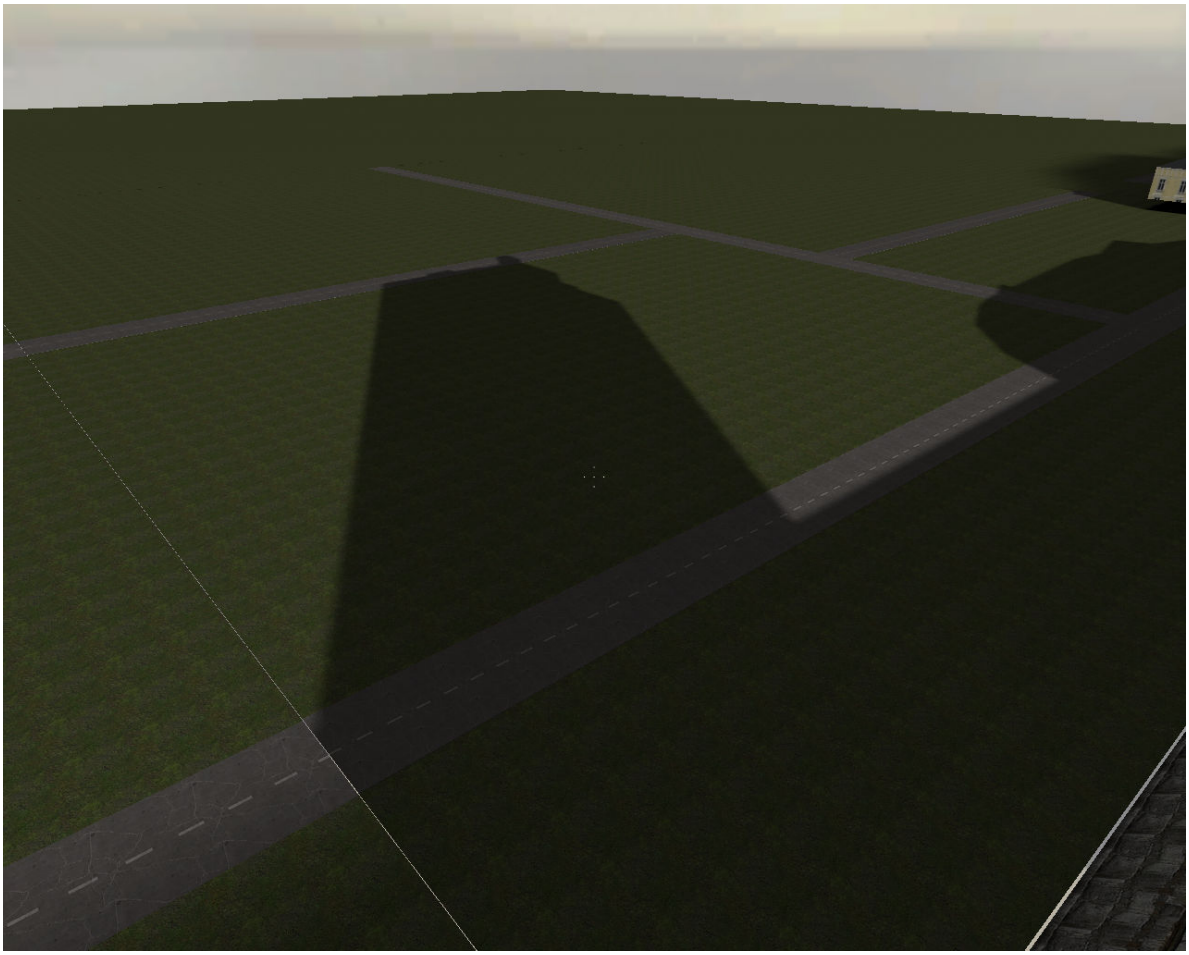


Fig. 12.3: Empty SIEVE_Traffic map that needs to be filled with regions and agents.

Another important factor are the entities that can be put into the world. For the case of the traffic accident, table 12.2 shows all the entities defined. All entities are marked by a _TR at the end of each type. This is to indicate that these entities are created for the traffic scenario. If another scenario would be created, the expert can make a distinction between the entities created for the new and the existing scenario.

There are 2 regions defined. One is the disaster region; the other is the region where the actual crash has happened. There are 3 buildings defined that can be placed along the roads of the map. A foliage agent is defined to create certain flora in the world. The car defined in figure 12.2 is also used, as well as the bicycle agent. Note that the bicycle isn't grouped as a vehicle, but as debris. This has something to do with the order of placement, which will be discussed in chapter 14. At last, the victims are also used for this SIEVE scenario. The scenario misses the T3 victims. This is because the

Table 12.2: All entities that can be used for the traffic accident scenario.

Entity type	Description
R_DISASTER_TR	The disaster area where everything is happening
R_CRASH_TR	The crash region where the car has hit the bicyclers
A_BUI_CORN_1_TR	A specific corner building
A_BUI_CORN_2_TR	Another specific corner building
A_BUI_ROW_TR	A row building
A_FOL_TR	A foliage agent represent all kinds of flora
A_VEH_CAR_TR	The car that has crashed into the victims
A_DEB_BICYCLE_TR	The bicycles of the victims
A_VIC_DECEASED_TR	A deceased victim
A_VIC_T1_TR	A T1 victim
A_VIC_T2_TR	A T2 victim
A_VIC_T3_WOUNDED_TR	A T3 victim who is wounded

victim database only has wounded T3 victims. Victims that aren't hurt must be added to the victim database in order to use them in a scenario.

Once the entities are defined, *CDesigner* must find out which entities to use for which scenario. The expert must steer this process by means of a grammar. The output of the grammar is the list of entities the designer is going to create in the world. The parse tree of the traffic accident scenario is provided in figure 12.4.

In this way, the expert can describe a hierarchy that is easy to create and understand. A lot of different outcomes can be created with relatively few rules. Providing randomization in which grammar rule to be fired can give a completely different set of entities while still providing some control to the expert.

In figure 12.4, the number of victims is fixed. This is however based on the level the user is playing. For level one, the accident has 5 victims. The last level (level 5) has 30 victims the player must triage. Also when the player has reached level 3 and higher, there will be two cars instead of one.

For this scenario, the distribution of victims categorized in triage classes is reported in [13]. According to this report 50% of the victims in a traffic accident are categorized as T3 (wounded). One sixth of the victims are T1 and a third should be T2 victims. This distribution pattern is also described in the SIEVE grammar.

These choices are described by the expert. Other decisions can be made randomly by the system. The extra variable {`Foliage`} is for instance added at random. Another example is the choice of building types. The system decides whether to use the row building or the second corner building.

Once a set of entities are given as output, *CDesigner* looks up the corresponding models in the `models.txt` file and creates the *CRegion*s and *CAgent*s accordingly. Storing the created entities in the *CManager* class the designer returns this manager to *CScenarioGenerator* and the entities are ready to be placed. For the complete SIEVE grammar and the `models.txt` file, the reader is referred to appendix E.

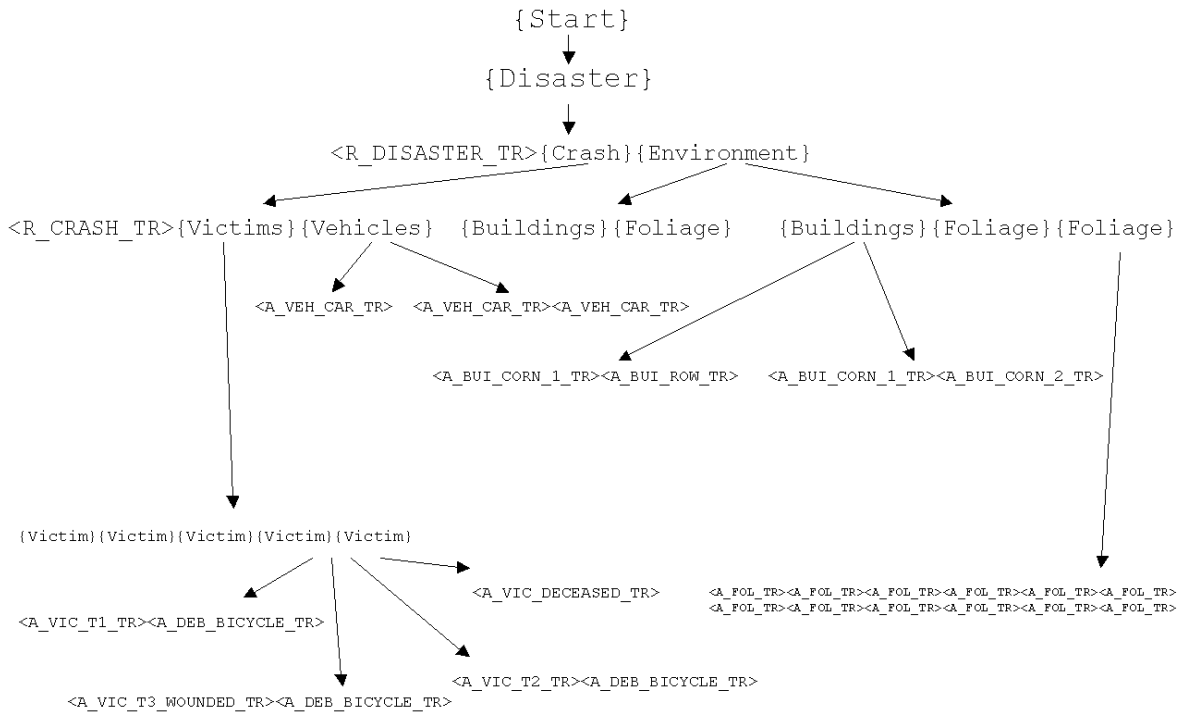


Fig. 12.4: Parse tree for the SIEVE grammar from start variable to the entities being created in the world. In the grammar entity types are between < and > while variables are between { and }.

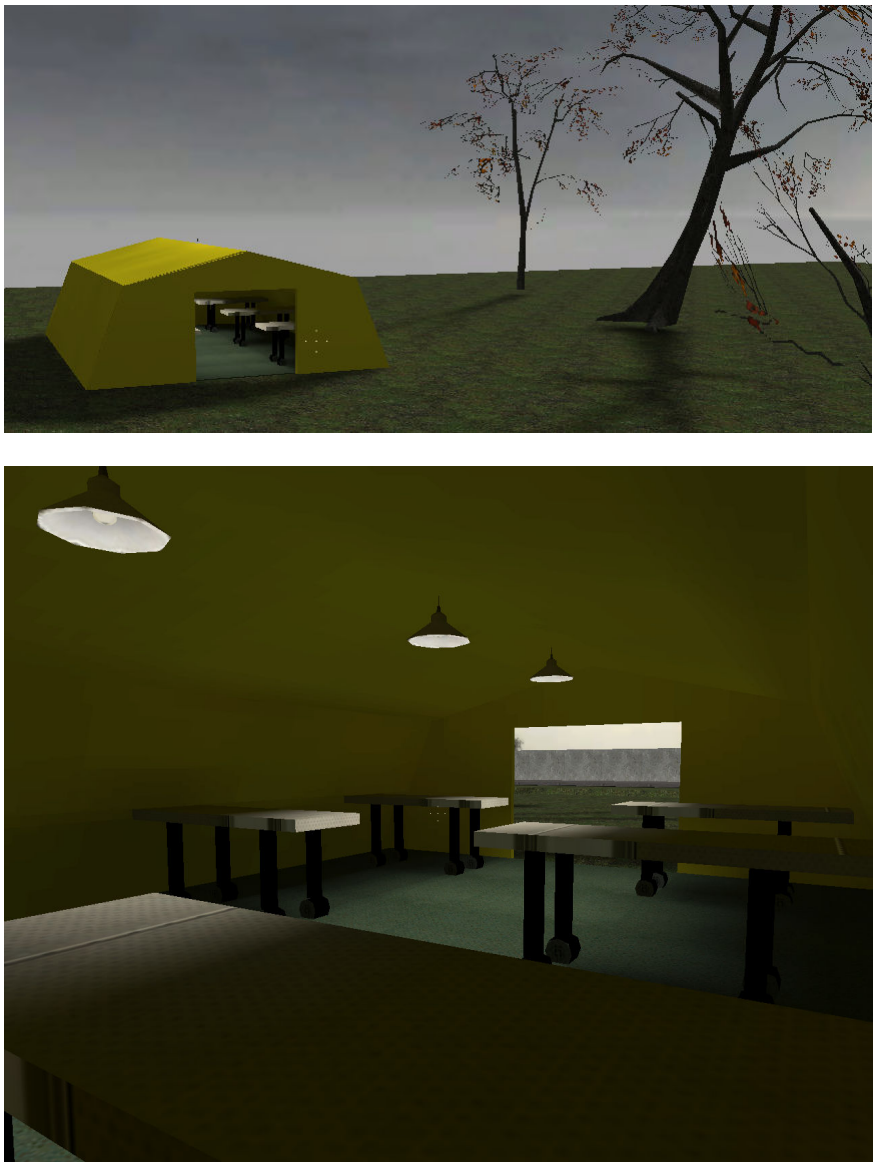


Fig. 12.5: Images from the empty SORT map that only needs to be filled with victim agents.

12.3.2 - SORT

For the SORT part of the game, only victims are needed, since a treatment area can very well be the same for every game. Therefore, a map is created that includes all necessary objects for a treatment area. Figure 12.5 displays this empty SORT map.

Since the victim agents are the only entities that need to be present, the SORT grammar is less complex. It actually represents the lower-left part of figure 12.4. Again, there are some defined rules as to how many victims there should be present given the user level. These follow the same number of victims as in the SIEVE part. Just like in the SIEVE grammar, the *CDesigner* class picks the victims at random from the victim database. For the complete SORT grammar, the reader is referred to appendix E.

Chapter 13 - Relations between types

By placing entities in the world at random, it is likely that the scenario isn't seen as a disaster area. The presence of certain entities may affect other entities in their position. To find a burned victim in the middle of a car crash is very unlikely. Therefore, the designer doesn't create victims with burns in the previous stated scenario. These relations between objects should be described in the grammars.

Spatial relations are also necessary within the automated scenario creator. A T1 victim that is too far away from the actual crash site might raise questions to the player. Buildings that are placed on roads aren't very realistic to the user and there are endless possibilities the expert doesn't want to see in the actual game.

13.1 - Distributions

Since the expert has created the most of the entity types, he/she is also responsible for the spatial relationships between those entities. Again, the system needs to understand the expert's input, so certain conventions are provided to let the process go as smooth as possible.

Both *CRegion* and *CAgent* search for an optimal location for their entity. This is done by calculating its satisfiability of neighbouring locations and compare this to the satisfiability of the current position.

Satisfiability gives an indication of how well the entity is placed in the world. A low satisfiability (reaching zero) indicates the entities isn't happy with its current position and is likely to move to another location, while a high satisfiability (reaching one) indicates the entity is placed at its (almost) optimal location and is likely to stay there.

A satisfiability distribution can be seen as a 3D probability distribution consisting of the position of the entity as its arguments (only x and y are taken into account) and the corresponding satisfiability as its result. Figure 13.1 displays an example of such a distribution. The expert can use these distributions to place certain entities at recommended locations. With this concept of distribution, spatial relations can be created.

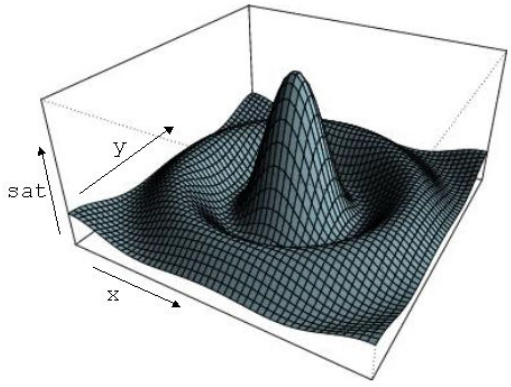


Fig. 13.1: An example of a satisfiability distribution.

In order for the expert to create such a distribution a slightly different approach has been taken to describe the distribution in for instance figure 13.1. The expert has the option to create a gray-scale image of the distributions. An example of such an image is depicted in figure 13.2. In this image, a pixel point (x,y) has a gray-scale colour representing the satisfiability. Black indicates a satisfiability of zero, while white indicates a satisfiability of 1. All gray-values in between represent a linear mapping of this satisfiability. These images are stored with an 8-bit resolution, meaning there are 256 tones the expert can choose from (i.e. 256 levels of satisfiability).

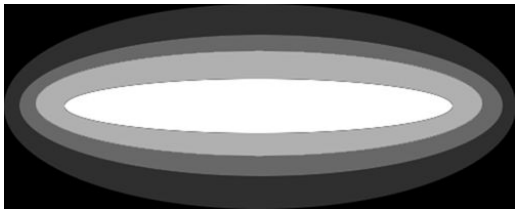


Fig. 13.2: Example image describing a satisfiability distribution.

With this approach, the expert can create his own distributions without getting confused on complex function descriptions. A simple image editor (like Photoshop or MS Paint) will suffice to create very complex distributions, as shown in figure 13.3. This figure illustrates the road scheme of the empty map (see figure 12.3). A downside of this approach is that information is stored in high-resolution images instead of a simple text file with function descriptions. When a lot of distributions are created, loading time will take longer since every distribution is loaded in before the game starts.

To create spatial relations by means of these distributions, the expert must make images for each relation he/she wants tot describe. The images must be read as distributions placed at the core of an

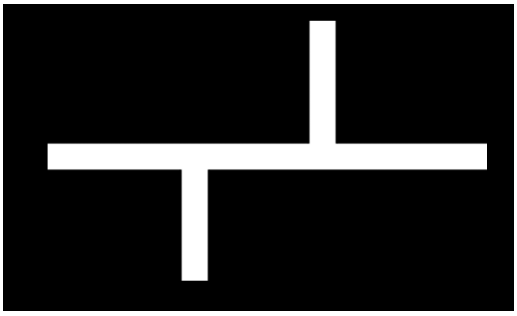


Fig. 13.3: Satisfiability distribution describing the spatial relation between disaster region and the world.

entity. The centre of the distribution represents the core of the affecting entity. Every pixel represents one Hammer unit in the game (this unit is approximately 0.75 inches in the real world, see [42] for details), although this can be scaled. As an example, figure 13.2 describes the spatial relation between T1 victims and the crash region. The core of the crash region is the centre of the distribution being completely white. The image itself has 500 pixels (i.e. 375 inches ≈ 9.252 meters) in width and 200 pixels (i.e. 150 inches ≈ 3.81 meters) in height. The distribution describes the satisfiability of the T1 victim agent relative to the position of the crash region core. The interpretation is, that the T1 victim is likely to move to the core of the region, which is in fact the desired outcome.

This example displayed a spatial relation between two defined entities. In order to create spatial relations between a defined entity and the virtual world, the *CDesigner* class has created a special region. This NIFV_REGION_TYPE_WORLD entity has its core at the absolute centre of the map (0, 0, 0). Figure 13.3 shows an example of a spatial relation between a region and the world. The disaster region always needs to relate to the world region, since this is the root entity to relate to (although multiple entities can provide a root of spatial relations). The figure shows that the disaster region must be placed on one of the roads in the empty map. This makes sense, because a large traffic accident usually doesn't happen off-road.

So far, the figures covered in this section only provided attraction to certain entities in the world. Repulsive distributions are of course also possible. An example can be that certain entities should not come close to each other. Especially when a victim comes close to the player's starting point. Figure 13.4 depicts this repulsive distribution. The player's starting position in the world is also a special entity that is created by the *CDesigner* class referred to the entity type NIFV_ENTITY_TYPE_PLAYER_STARTING_POS.

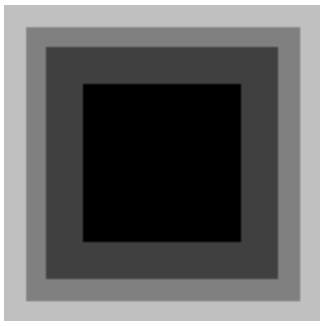


Fig. 13.4: Repulsive distribution describing the spatial relation between a victim and the player's starting position.

Once the expert has created all the images for the distributions, he/she needs to report them to the game. This is done via a `distributions.txt` file. Every distribution needs to be named, the path of the image needs to be given, the scale of the image and the outer value of the distribution must be entered. Two examples of such entries are given in figure 13.5. Note that the paths to the actual images aren't relative to the game directory. This has to do with the fact that the code library used for reading the images (image_basic, see [50]) does not provide this option. The `out_val` value is the satisfiability

```
"0"
{
	"name"		"repel_32x32"
	"path"		"C:\\Program Files\\Valve\\Steam\\SteamApps\\SourceMods\\
			NIFV\\scripts\\rulebase\\distributions\\repel_32x32.bmp"
	"out_val"	"1"
	"scale"		"1"
}
"7"
{
	"name"		"attract_disaster-world_traffic"
	"path"		"C:\\Program Files\\Valve\\Steam\\SteamApps\\SourceMods\\
			NIFV\\scripts\\rulebase\\distributions\\attract_disaster-
			world_traffic.bmp"
	"out_val"	"0"
	"scale"		"0.25"
}
```

Fig. 13.5: Example entries of the distributions.txt file.

that is given whenever the position of the affected entity isn't in range of the distribution image. For a repulsive distribution, this value should become 1, while an attractive distribution should have this value set to 0. To lower the size of certain images, a scale factor is provided. Especially with large images that describe distributions related to the world region, this factor becomes very helpful for managing the loading times of the game. A scale value of 0.25 means that the image is 4 times smaller than the original distribution should be. For a complete view of the distributions used in the game, the reader is referred to appendix E.

All distributions listed in the `distributions.txt` file are loaded in by the *CDistributionReader* class. The class creates and manages all images in *CDistribution* classes, also depicted in figure 12.1. An agent or region can, when allowed by *CPlacer*, ask the *CDistributionReader* class what the satisfiability is given a position relative to an affecting entity in the world.

13.2 - Relation table

The expert has only created and listed the distributions describing the spatial relations between certain entities. The system however doesn't know which distribution applies to what entities. For this problem, the expert must present a relation table. In this table, the expert declares which entity affects another entity by what distribution.

```
"R_DISASTER_TR"
{
	"NIFV_ENTITY_TYPE_PLAYER_STARTING_POS"	"repel_32x32"
	"NIFV_REGION_TYPE_WORLD"	"attract_disaster-world_traffic"
}
"R_CRASH_TR"
{
	"NIFV_ENTITY_TYPE_PLAYER_STARTING_POS"	"repel_32x32"
	"R_DISASTER_TR"		"attract_crash-disaster_traffic"
}
"A_VEH_CAR_TR"
{
	"R_CRASH_TR"		"attract_car-crash_traffic"
	"A_BUI_CORN_1_TR"	"repel_car-corn_building_traffic"
	"A_BUI_CORN_2_TR"	"repel_car-corn_building_traffic"
	"A_BUI_ROW_TR"		"repel_car-row_building_traffic"
}
"A_VIC_T1_TR"
{
	"R_CRASH_TR"		"attract_T1-crash_traffic"
	"A_VEH_CAR_TR"		"repel_240x240"
	"A_BUI_CORN_1_TR"	"repel_victim-corn_building_traffic"
	"A_BUI_CORN_2_TR"	"repel_victim-corn_building_traffic"
	"A_BUI_ROW_TR"		"repel_victim-row_building_traffic"
}
```

Fig. 13.6: Example entries of the relation_table.txt file. For example, the T1 victim agent is affected by the crash region as described in the attract_T1-crash_traffic distribution.

For each entity (agent or region) that can be affected by other entities an entry in this table must be created. Some of theses entries are shown in figure 13.6.

For example, the disaster region is only affected by the special entities created by *CDesigner*. It must stay away from the starting position of the player and it's attracted by the world region by the distribution called `attract_disaster-world_traffic`. The car agent has 4 affectors: the crash region and the three possible building agents in the world. The car is attracted by the crash region described in the `attract_car-crash_traffic` distribution. The car must avoid any collision with any of the buildings, so a repulsive distribution is linked between the car and the building agents.

With this relation table, the automated scenario creator has all the relevant information of which distribution to apply to what entities. The placing of these entities can begin.

Chapter 14 - Placing the scenario

Once the entities are defined by the expert and created by the *CDesigner* class, *CPlacer* tries to place all the entities according to the loaded distributions. For the SIEVE and SORT part, there are different methods to place all the entities. The SORT part only needs to place the victim agents onto an empty stretcher in the treatment area, whereas for the SIEVE a complete disaster area needs to be placed. In the following subsections the placement algorithm is explained for both parts of the game.

14.1 - SIEVE placement

The *CPlacer* class has the responsibility to let the process of placement go in a fashionable manner. Instead of placing all entities at the same time, the groups shown in table 12.1 are used to create some order. This will divide the load of work into manageable chunks of entities that shouldn't be moved whenever another group is being placed.

The placement algorithm starts with placement of the disaster region(s). After this group has found a suitable location in the virtual world the rest of the regions are placed. With these two groups placed, the regions are covered and the algorithm can move on by placing the agents. This goes according to similar group ordering: First the building agents (starting with A_BUI_), then the foliage (A_FOL_), vehicles (A_VEH_), victims (A_VIC_) and at last the debris agents (A_DEB_).

Every entity that needs to be placed must be categorized in one of the above-mentioned groups in order to be moved by *CPlacer*. Each entity is given a position at random in the world, within the borders defined by the expert. These are just two simple values providing the width and the depth of the placement area in the map. This area is always centred on the absolute centre of the map (point (0, 0, 0)). The placement algorithm is an iterative process moving regions and agents more and more to satisfiable locations.

Per group, the placement process is repeated until all entities in that group are on locations where the satisfiability is high enough. If the group satisfiability (represented by the average satisfiability of all the entities in the group) is lower than a certain rejection threshold (in this case 0.9), the entities will be re-placed (i.e. placed again), starting from their current position. If the group satisfiability overcomes the threshold, the algorithm moves on to the next group in line.

The placement algorithm is described in pseudo-code in figure 14.1 for the regions to be placed. The algorithm for the agent entities is similar, except for some name changes and the `AddAffectorAgents` addition to search for both regions and agents that can affect this agent.

In the figure there are 3 main methods described: `PlaceRegions`, `AddAffectorRegions` and `PlaceRegionGroup`. `PlaceRegions` is the function that positions all the regions in *CManager*. After this method is done, the `PlaceAgents` method is called to locate all the agents in the world. Whenever both methods are finished, the placement algorithm is finished and the player can start the game.

`PlaceRegions` sets up the placement algorithm for the separate groups (disaster regions and the other regions) by finding all the regions that belong to the group, store some placement information concerning the iterative process and calculating the satisfiabilities for the initial random position.

The expert provides the additional placement information in a text file called `placement_info.txt`. An example of an entry in this text file is given in figure 14.2. For each group, the expert can change the time spent on placing entities of that group by configuring 5 parameters. The `MAX_STEPS` indicate the number of iterations the placement algorithm makes before stopping placement of the group. `MIN_STEP_SIZE` and `MAX_STEP_SIZE` give the minimum and maximum distance an entity within the group can move per iteration. Finally `MIN_LOOK_DIST` and `MAX_LOOK_DIST` set the minimum and maximum distance an entity in the group can see to decide the satisfiability in the neighbourhood.

For every region in the current group, affectors need to be added. The relation table summarizes the entity types that can affect the current entity type in the group. The `AddAffectorRegions` (and the `AddAffectorAgents`) method uses this relation table and the manager to find all the created entities with the affecting types. These entities are added to the *CRegion* instance as a reference during the placement process.

Please note that the affectors themselves need to be placed before this group is placed. This implies that more affectors can be present when the placement algorithm is further in the group ordering. It also means that the group ordering was chosen with groups containing the less affected entities (e.g. regions and buildings) first. Debris agents are placed last, because certain debris must take into account practically every entity in the world (e.g. a tire must be placed close to a car, not in collision with a tree and/or victim). The bicycle in the traffic accident scenario is a debris agent. This has however less to do with the bicycle being affected by all other entities. Within the game, it is allowed to find a victim with a bicycle on top of him/her. The user can simply remove this object by picking it up and throw it away (with help from the use key). Therefore placing of the bicycles must be done after placement of the victims.

```
CPlacer::PlaceRegions()
{
	for all( region_groups reg_group in groups )
	{
		region_list = manager->FindRegions( reg_group )
		AddAffectorRegions()

		max_steps = GetMaxSteps( reg_group )
		min_step_size = GetMinStepSize( reg_group )
		max_step_size = GetMaxStepSize( reg_group )
		min_look_dist = GetMinLookDist( reg_group )
		max_look_dist = GetMaxLookDist( reg_group )

		for all( regions region in region_list )
		{
			region->CalculateSatisfiability( max_look_dist, border )
		}

		steps = 0
		sat = 0
		PlaceRegionGroup()
	}
}

CPlacer::AddAffectorRegions()
{
	for all( regions region in region_list )
	{
		affector_type = FindAffector( region->GetType() )
		affector_list = manager->FindRegions( affector_type )
		for all( affectors aff in affector_list )
		{
			region->AddAffector( aff )
		}
	}
}

CPlacer::PlaceRegionGroup()
{
	while( steps < max_steps && sat < desiredSat )
	{
		step_size = CalculateStepSize( steps, min_step_size, max_step_size )
		look_dist = CalculateLookDist( steps, min_look_dist, max_look_dist )

		sat = 2
		for all( regions region in region_list )
		{
			region->UpdatePosition( step_size )
			sat = min( region->CalculateSatisfiability( look_dist,
                                                        border), sat )
		}
		steps++
	}
}
```

Fig. 14.1: Pseudo-code describing the placement algorithm for the region groups.

The `PlaceRegionGroup` performs the actual placement process by moving the entities in the group and calculating their satisfiabilities. This iterative process is done until the maximum number of steps is reached, or the group satisfiability (now represented by the lowest satisfiability among the entities) is higher than the desired satisfiability (which is set in game to 0.97).

The pseudo-code for the methods in *CRegion* (`UpdatePosition` and `CalculateSatisfiability`) is displayed in figure 14.3.

```
"R_DISASTER"
{
	"MAX_STEPS"		"20"
	"MIN_STEP_SIZE"		"100"
	"MAX_STEP_SIZE"		"1500"
	"MIN_LOOK_DIST"		"100"
	"MAX_LOOK_DIST"		"1500"
}
```

Fig. 14.2: Example entry of the placement_info.txt. These parameters are used for the placement of the disaster region(s).

```
CRegion::UpdatePosition( step_size )
{
	dir = sat_matrix->ChooseBestDirection()
	Move( dir, step_size )
}

CRegion::CalculateSatisfiability( look_dist, border )
{
	sat_matrix->Reset()
	for all( entries neighbour in sat_matrix )
	{
		probe_points = CreateProbePoints( neighbour )
		for all( point p in probe_points )
		{
			for all( affectors aff in affector_list )
			{
				neighbour += CDistributionReader->
					CalculateSatisfiability( this->GetType(),
					aff->GetType(), aff.x-p.x, aff.y-p.y )
			}
			neighbour /= affector_list.count
		}
		neighbour /= probe_points.count
	}
	return sat_matrix->CurrentPosition()
}
```

Fig. 14.3: Pseudo-code describing the placement methods used in the CRegion class.

The `UpdatePosition` is a simple method for moving the entity in the direction with the most satisfiability. If multiple directions are present with the highest satisfiability, the method picks a direction at random. The satisfiability matrix (`sat_matrix` in the figure) is the important part in these methods. This is a 3x3 matrix and holds the satisfiabilities for going a certain direction. Fig. 14.4 shows the meaning of the satisfiability matrix in an example.

In this figure, the victim agents are placed. One of the victims is highlighted with its satisfiability matrix displayed in white. The highest satisfiability is in the middle of the orange ellipse, but the victim should not be placed in the vehicle agent (marked in red). The yellow lines are the possible directions the victim can move towards. The entries of the satisfiability matrix are the average satisfiabilities whenever the agent decides to move in the indicated direction. The centre entry of the matrix is the current satisfiability of the entity. When the `UpdatePosition` method is confronted with the matrix displayed in figure 14.4, the victim agent would be moved north because of the average satisfiability of 0.9 in that direction.

The `CalculateSatisfiability` method in the entity class has its task to update the values in the satisfiability matrix and to return the current satisfiability. The given looking distance (calculated by the placer) is the complete length of the yellow lines in figure 14.4. To only use one point in the calculation of the entire satisfiability for one direction doesn't seem the optimal approach. Especially when the looking distance is quite large, a satisfiable position can be overlooked. Therefore, some probe points are created along the line. In the current game, there are 5 points used for every direction, always ending at the point a looking distance away from the entity.

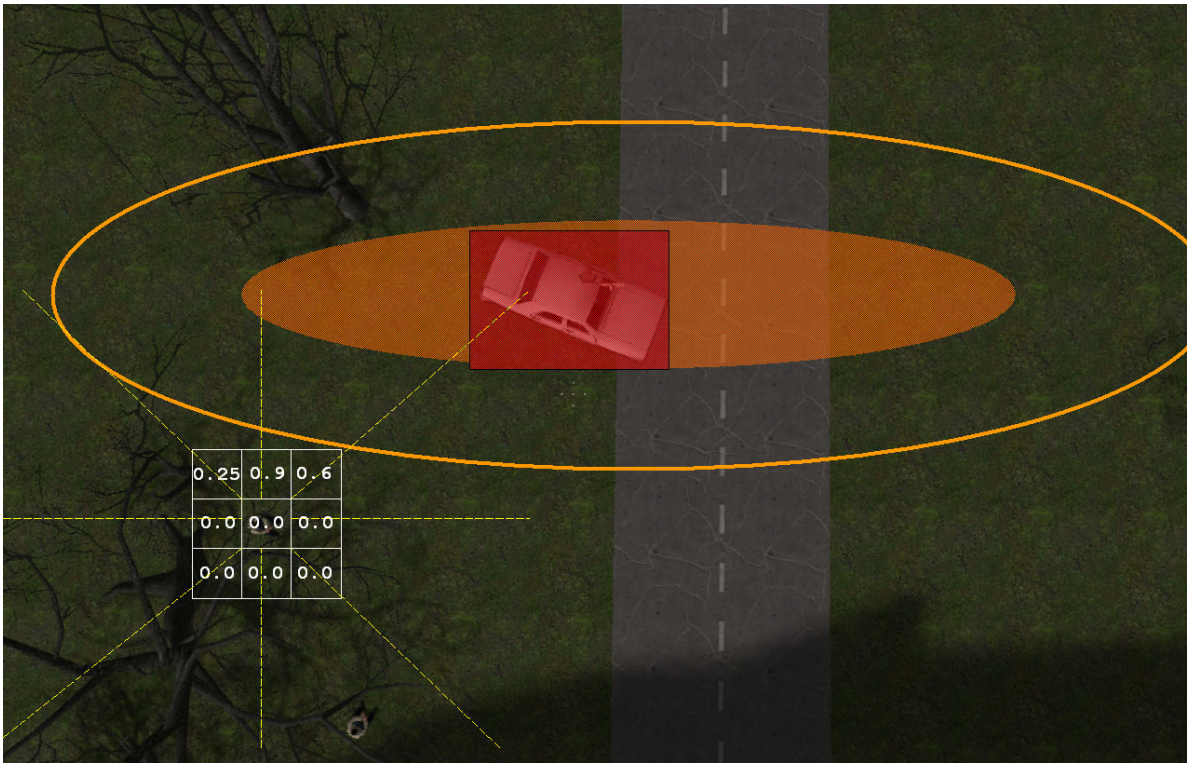


Fig. 14.4: Satisfiability matrix in action. Every entry gives the satisfiability of going towards the represented direction. The centre entry is the current satisfiability of the current position.

For every probe point the algorithm must check every affector of the entity to calculate the satisfiability of that point. Once these five points are calculated and the satisfiability is retrieved with help from *CDistributionReader*, the average satisfiability is taken and is entered in the corresponding entry of the satisfiability matrix. This means that the values of 0.9 in figure 14.4 does not indicate that the agent is actually receiving a satisfiability of 0.9 when moving towards the new direction. It only gives an indication of what direction is more satisfiable than others.

The placement algorithm described has many similarities with a steepest ascent hill-climbing search (see [11][48] for more information). In addition, the looking distance and step size used by the *CRegion* and *CAgent* class to calculate the satisfiabilities of neighbouring points and to move the actual entity towards the best direction are decreased over time. The ratio of the current iteration and the total number of allowed iterations is used to linearly decrease both values from its maximum to its minimum. With this addition, the entities start out with a large overview of the distributions and slowly refine their observations and actions. Where simulated annealing (see [11][74]) allows the entity to make more errors at the start of the iterative process by producing more random movement, this algorithm allows the entity to take larger steps at first to search the space for a suitable area to narrow its view.

After all groups are placed in a similar fashion as described above, the game is ready to be played by the user. Figure 14.5 shows some examples of the sieve_traffic map placed with the scenario entities.

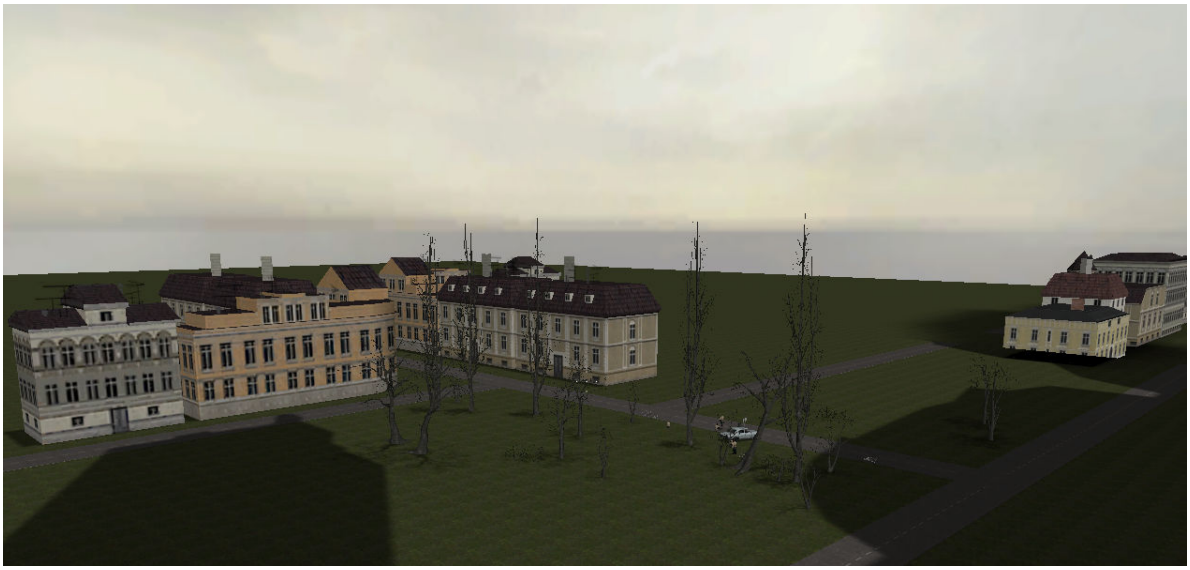


Fig. 14.5: Some screenshots of the SIEVE map with the traffic accident scenario.

14.2 - SORT placement

For the placement of victims in the SORT part of the game, a completely different approach is needed. Instead of letting the agents calculate the satisfiability, the level designer already has defined the locations of the victim agents.

If the *CPlacer* class finds out that the user is playing a SORT map, it starts looking for *CVictimLocation* entities in the world. These entities should have been placed by the level designer to indicate a spawn point for a victim. As can be seen in figure 12.5, victims should be placed lying down on a stretcher. For this map, there are 6 stretchers in total.

Once the placer has found the 6 *CVictimLocation* classes (located on top of each stretcher), the placer simply picks a random *CVictim* class from *CManager* and attaches this agent to the *CVictimLocation* class. If the locations are all occupied by a victim or all the victims are placed, *CPlacer* tells the *CScenarioGenerator* class to start the game. Figure 14.6 displays a SORT game played at level 2, starting out with 6 of the 10 victims placed on the stretchers.

What happens with the 4 victims that aren't placed on stretchers? They are not present in the virtual world until the ITS indicates the victims need to be updated. *CTriageInstructor* of the ITS has a connection with the scenario generator to present new victims in the treatment area. Whenever the user has triaged a victim, the instructor informs the scenario generator to update the SORT victims. The generator passes this command on to *CPlacer* via the `UpdateSORTVictims` method.

This method tries to find the *CVictimLocation* instance with the victim that was triaged by the player. Once it has found this victim it checks if the player can see the victim. If the user's sight is turned away from the triaged victim, The *CVictimLocation* class is told to release the current victim. The victim is removed from the virtual world and the *CPlacer* class picks one of the remaining victims in the manager to attach to the location. In this way the player doesn't notice the actual change of the victim and assumes the victim has been taken away while the user was triaging another victim.

This process of replacing the triaged victims doesn't need to take place in the first time the triage instructor informs the generator to update the victims. The instructor will keep informing the generator until all the victims in the world aren't triaged or all the victims have been triaged by the player. So if the player is treating another victim, but still sees the triaged victim, the placer waits patiently until the player has turned around.

If a level designer was to create a new SORT map, he/she only needs to take into consideration to place the *CVictimLocation* entities in the appropriate places. After that is done, the *CPlacer* class will place the victims created by *CDesigner* in the map and the player can perform his tasks in a new environment.

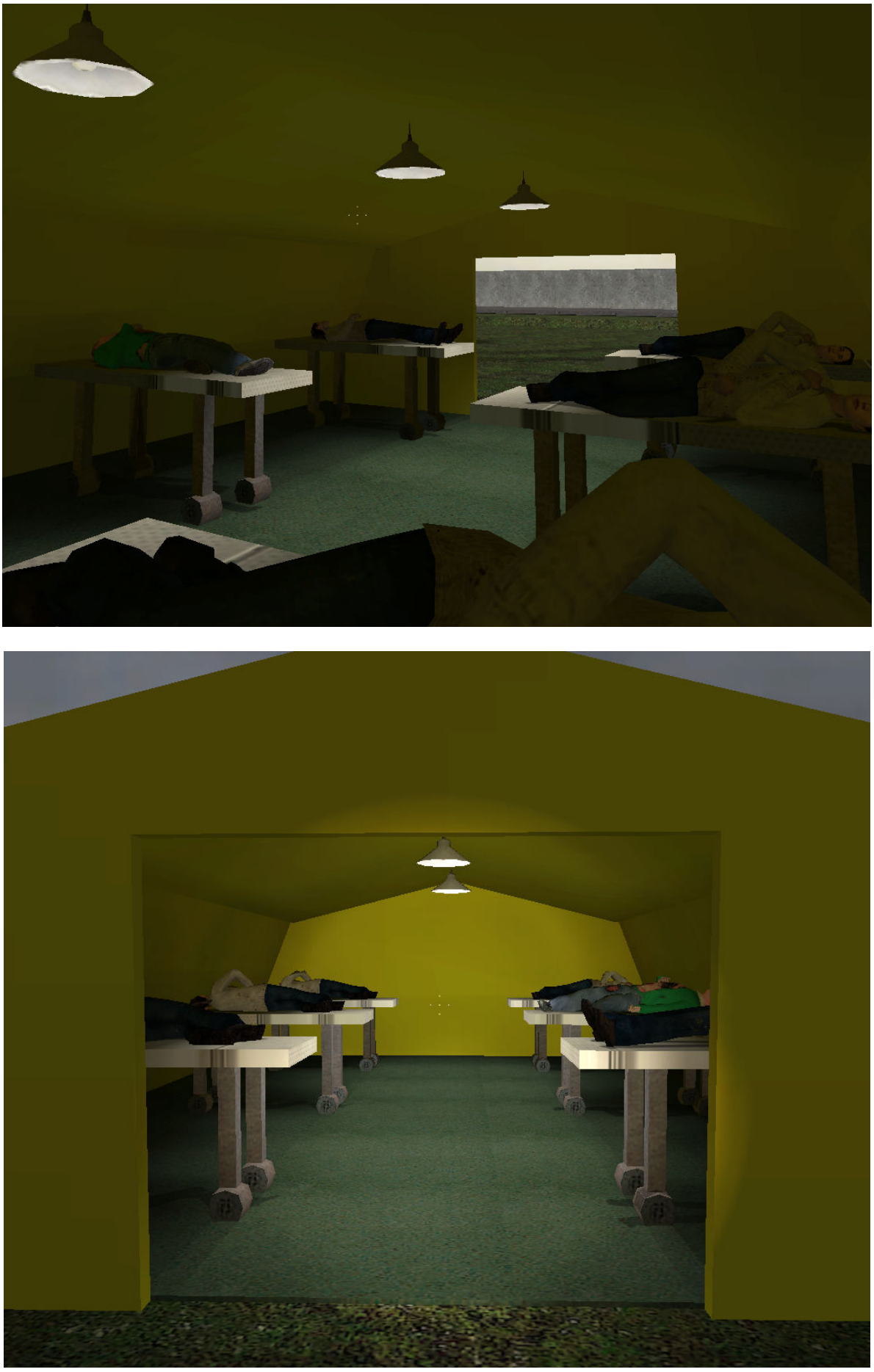


Fig. 14.6: Some screenshots of the SORT map.

Part V – Evaluation

Chapter 15 - Testing

In order to get a look at the results of the created serious game, testing needs to be done. However, testing whether the serious game has a significant effect on the learning rate of the user is very hard to measure. Especially when the game is teaching more than factual knowledge only.

Testing problems arise with this serious game since it is very hard to create a control group for the evaluation. Nowadays, medical personnel learn the triage procedure though a theory course. After this course, students can be trained by performing triage on several victims that are described on paper cards. This is a very abstract form of training, since only the factual and rule-based knowledge is trained. Another form of training that is presently used, is the simulation of a disaster. This is the complete opposite of the pen-and-paper training. Instead of training the factual knowledge of one medical worker, the focus lies more on the team's response to the situation. Communication between other first responders and controlling the situation is more of a point in these training simulations.

Another problem is in the evaluation of the medical workers. The rules and reasoning behind triage can be tested and evaluated. These evaluations can be compared between students who have played the serious game and who only have studied the theory. Only this is just a part of the overall task description the medical worker should be able to perform in a real disaster. Evaluation of the triage tasks within the disaster can only be done when a real disaster happens, since there isn't a proper simulation that mimics all the pressure and dangers realistically. A serious game can still be a step in the right direction.

However, it can still be very useful to let several persons play the implemented game to get a better insight in how to proceed. Even though the serious game is only in a pilot version, lacking important features like the graphical models of all victims, a small user study can check if the project is heading in the right direction.

15.1 - Method

Several co-workers at the NIFV were willing to participate in this preliminary test. 9 of them were invited to play the game and give a first impression. The test itself would last for approximately a half hour per participant.

Previous to the actual test, the partaker received an introductory document explaining the purpose of the test and the performance of triage. Since all the participants were Dutch, this introduction was also written in Dutch.

On the actual test day, every user was allowed to play the game for 20 minutes to get an impression of the playability of the game. Since most of the participants did not have any experience with gaming or the new triage method, an observant was present to help and direct the player when needed. The observant also noted experienced difficulties with the game to get a picture of the usability of the game. All the participants were introduced to the SIEVE and the SORT part of the game. At least 1 victim was triaged in both of these parts. The evaluation report in the game was also presented to all the players and all the different types of feedback were activated in-game.

After the 20-minute test run, the participant was asked to fill out a feedback form concerning the game. This questionnaire is described (in Dutch) in appendix F.

15.2 - Results

Unfortunately, 2 of the 9 invited participants couldn't make it to the scheduled appointment, leaving 7 users of this first evaluation. The overall first impression was that the game could very well be a promising educational tool for medical personnel. All participants of this preliminary test had fun playing this game and were positive about this way of training in general.

However, with practically every user study, several errors in the game (concerning both reasoning and usability) were found during the test. The triage tent in the SORT part of the game was to dark to operate. The flashlight brought some relieve, but it might be an improvement to make the environment lighter. At first participants couldn't hear the heartbeat of the victims, but with the addition of an external speaker the problem was solved.

A recurring usability error was the choices menu in the SIEVE and SORT (see figure 9.9 and 9.12 on page 65 and 66, respectively). It was difficult for the partakers to hold the <e> button on the keyboard and press the appropriate action button with the mouse. Whenever the player wanted to classify the victim to a certain triage class, the partaker thought he/she could click on the T1 or T2 miniature in the choices menu to classify. When the actual classification menu (see figure 9.10) popped up, some participants hesitated, but quickly pressed the appropriate button to actually classify the victim.

Practically all users had difficulties (some more than others) with the use of the mouse in combination with the keyboard to move the virtual player around. The use of two hands in the game was somewhat of a struggle, although 6 out of 7 participants thought the controls of the game were easy. Unfortunately, since many of the participants did not have any experience in gaming the actual control scheme could not be tested thoroughly. For PC gamers, these controls are standard for practically every game, and can be learned easily. For people without this experience of game controls, it could give extra problems, which may lead to an alternative device (e.g. a joystick or console controller). On the other hand, it might also be that the users, after playing some time, master the current controls quite easily, solving the problem mentioned before.

Most of the participants noticed the lack of graphical models, but this did not interfere with the actual process of triage. One participant noted that the disaster area in the SIEVE was a bit unrealistic, but he didn't know if this also affected the teaching of the triage topic. The display of vital functions was also received positively. Actually hearing a victim breathing (although some thought the breathing was a bit heavy), counting the number of breathes made the game a bit more realistic.

Regarding the evaluation of the system, the users agreed on the presentation of the feedback in-game. Most of them wanted feedback in a clear and short description per error made. Also the evaluation report was evaluated as OK, giving a clear representation of what the user had done. Although some of the errors reported were created by small bugs in the system, most of the participants understood what they did wrong.

To conclude, the 7 participants of this user survey saw this serious game as a good instructional tool for medical personnel. This form of training was encouraged by the partakers, also in other topics of disaster management like evacuation, decision-making, treatment of hazardous materials and management within a treatment area. This preliminary test showed that the created pilot of the serious game has been received very well and that further development of the serious game can be very interesting for future learning.

Chapter 16 - Conclusions & future work

16.1 - Conclusions

In this thesis, the design and creation of a serious game concerning triage has been discussed. The main goal of this serious game is to teach and train medical workers the new triage procedure being introduced in the Netherlands in 2009. This game can then function as an educational tool in a course concerning the new triage procedure. After a theory course, medical workers can put their knowledge to the test with this serious game.

A serious game is a (digital) game that has a different purpose than mere entertainment. By using game elements provided in regular games, serious games can provide a more entertaining way to educate or bring awareness to certain topics. With more and more interest in these kinds of applications, the serious game industry becomes an interesting field of commerce and research. Especially since this research area is still very young, a lot of work still needs to be done to find proper forms of evaluation and a good mixture between entertainment and education.

In order to provide direct feedback to the student playing the game, an Intelligent Tutoring System (ITS) was designed and created. This framework, in combination with the created game model, turns out to be a very good way to provide feedback in games. It creates a system that provides an environment for the player to try things out while being monitored and helped by a virtual instructor. Especially when the ITS framework is based on a modular structure, the framework can be used for multiple purposes in one game. In this serious game, the ITS is used to monitor the player's movement, find errors and provide basic feedback on the user's actions concerning triage and create a disaster scenario automatically.

The Valve Source engine is used to implement the serious game with the ITS framework. The Source engine is a game engine, providing only basic functionality (e.g. rendering from objects to screen, physics, sound, etc.) for the game to be created with. Only the game content has to be created by the developer, reducing the amount of development time and costs compared to the creation of a game from the ground up. This thesis compared several game engines with each other in the search to find the suitable engine for this project. Valve's Source engine, together with the Source SDK, turns out to be a very fitting candidate for the creation of serious games. Especially with a good hierarchical structure and a clear code base, the additional modules that need to be coded for this project can easily be merged with the engine and the corresponding game. There were however some problems during the development of the serious game. Especially when most of the tutorials found on the Internet are written by gamers instead of the development team at Valve, it's difficult to find a overall impression of how the game engine works. Small problems were however rapidly fixed with help from the developer's community. This community provided great insights in the engine with an answer present within 2 to 3 days.

The global ITS framework created for this serious game consists of the three common-used models in other ITSs: The expert model containing relevant information concerning the topic to be learned; the didactical model describing the different ways to engage in a conversation and how to best present the given information to the user; and the student model representing the knowledge of the player in the game. 4 additional modules were created that interact with the previous mentioned models in the ITS: The monitor observing the player while playing the game and reporting the user actions to the evaluator; the evaluator trying to find errors in the reported user actions and providing a score for each action; the instructor updating the student model and reasoning about what to do next; the generator manipulating the game to fit the instructor's goal.

With this global framework different implementations can be realized. In this project three implementations were created. The most important implementation is recording and providing feedback on the user's triage actions. Feedback is given both in the game itself as well as by means of an evaluation report after the game. For the human instructor, several text files are provided to discuss certain problems with the student after the game is played.

An additional feature that is included in the serious game is the automated creation of a disaster scenario. With this feature every disaster area will look slightly different, giving the player a constant challenge every level. For this project, one scenario of a traffic accident is described, while more

scenarios can easily be added to the game to provide more diversity. This approach has as main benefit that the creation of a disaster area is broken down in easy to understand rules. An expert can create a new disaster without having to plunge into code or other complex programs. However, experts no longer have total control over the environment the player is in. Objects can be placed in wrong areas (which sometimes also happens in this game), creating a world that is unrealistic. But when the disaster is placed correctly, the player can be more immersed in the game than with pre-defined levels, since victims are never in the same place. This consideration of randomness versus pre-defined locations in educational games should be investigated first before a conclusion can be made.

The implemented serious game was finally tested by 7 co-workers of the NIFV. All the participants had a positive impression of the game, indicating it was a promising educational tool for medical workers. Although there were some small problems (i.e. bugs) found, most of the participants had fun during the game and were interested in the further development of this serious game concerning triage.

16.2 - Future work

Since this master project was focused on the ITS within a serious game, the created pilot of the serious game NIFV – T1 can be improved with future work. The graphical representation of the game can be enhanced in the near future in order to make the game operational for training. Now, only one victim has its corresponding graphical model and the rest of the victims are represented by standard models of the Half-Life 2 game. By giving all the victims their corresponding model, the game will be more realistic for the player. The creation of realistic animations for those models (e.g. flexion and extension, walking towards the player, calling for help, etc.) is also good enhancement of the serious game. Other objects in the world can also be improved to create a more realistic effect in both the disaster as the treatment area. In addition, Non-Playable Characters simulating firemen or other medical workers can be created to run around in the virtual world to give a sense of urgency.

The ITS can further be improved by not only representing feedback in text, but also with images, spoken words or even a virtual instructor walking along with the user. To create a dynamic environment, the ITS can also be added with a generator actually affecting the virtual world. In this case, the ITS can affect the disaster area to meet with the skill level of the player. The addition of more scenarios to the game (e.g. collapse of a building, explosion, train accident, etc.) will also improve the serious game as a learning tool and will also provide more immersion in the game.

This serious game and its framework can also be used for research in several areas. The effects of several learning strategies can be tested with this framework as well as certain evaluation methods to be implemented. The results of automated creation of disaster scenarios can be evaluated with users playing the game as well as several immersion techniques to attract players in playing the game frequently.

With the base ITS framework and some parts of the game content created for this project, it is also possible to create a completely different serious game. Especially when remaining in the field of disaster management, this project can function as a building block for other educational tools that combine game elements and didactical strategies to teach a certain topic while playing a game.

Part VI - Appendices

Appendix A – Game engines

Email survey of game engines

	Replies	Pros	Cons	Dev. time	Dev. team
Delta3D	2 / 2	* full access to code * scalability to big projects * overall design	* lot of code needs to be written by developer, especially network functionality	12	5, plus 5 part-time
		* constant development	* steep learning curve	9	2
Quest3D	0 / 1	-	-	-	-
Source	1 / 3	* familiar * adequate	-	2,5	1, plus 2 mappers
Torque	1 / 1	* reasonable graphics * scripting * networking * community	* GUI builder lacks features * no AI * unicode problems	4	1, plus 2 experts
Unity	3 / 3	* writing for Mac	* Windows compatibility isn't perfect * video, debugger, profiler, Direct3D had to be added by developer	12	6
		* attentive developers * future growth * polished UI	-	15	6
		* good contact with engine developers	* lot of features lacking	5	-
Unreal 2	0 / 2	-	-	-	-

For this survey, developers of serious games were asked several questions concerning their use of the game engine they made use of to develop a 3D serious game with. Unfortunately, not all developers replied to the survey, causing missing results for the Unreal and Quest3D engine. Please note that the development time is represented in months. The development team represents the number of persons in the complete team creating the game.

Serious game developers were contacted according to the list on the following page. Here, a lot of serious games are displayed with the site to play this game. These serious games are all 3D instead of the widespread 2D Flash games found on the Internet. Also note that this list does not record all the 3D serious games ever made. It is just an indication of several serious games, their developer and the used game engine.

Game	Website	Developer	Engine	Engine Website
A Force More Powerful	http://www.afmpgame.com/	BreakAway Games	Gamebryo	http://www.emergent.net/index.php/homepage/products-and-services/gamebryo
Advanced Disaster Management Simulator	http://www.admstraining.com/	ETC Simulations	Own engine	http://www.admstraining.com/adms_company.php
America's Army	http://www.americasarmy.com/	Virtual Heroes	Unreal Tourn. 2003	http://www.unrealtechnology.com/html/homefold/home.shtml
Code3D	http://simopsstudios.com/	Sim Ops Studios	Own engine	http://panda3d.org/
Dangerous Waters	http://www.scs-dangerouswaters.com/	Sonalysts Combat Simulations	Own engine	
DARWARS Ambush! Convoy Simulator	http://ambush.darwars.net/	US Army	Real Virtuality	http://www.bistudio.com/inside/tech.html
DoomEd	http://www.desq.co.uk/doomed/	DESQ Digital Learning	Source	http://www.valvesoftware.com/
Dubai Police Serious Game	http://www.dcs.shef.ac.uk/~ahmed/	University of Sheffield	Torque	http://www.garagegames.com/
First to Fight	http://www.firsttofight.com/html/index.html	Destineer Studios	Own engine	http://www.destineerstudios.com/
Food Force	http://www.food-force.com/	United Nations	??	
Free Dive	??	BreakAway Games	Gamebryo	http://www.emergent.net/index.php/homepage/products-and-services/gamebryo
Full Spectrum Warrior	http://www.fullspectrumwarrior.com/	Pandemic Studios	??	
Global Conflicts: Palestine	http://www.seriousgames.dk/gc.html	Serious Games Interactive	Unity	http://unity3d.com/unity/index.html
GNN Visualization	??		Source	http://www.valvesoftware.com/
Harpoon	http://advancedgaming.biz/index.php?cevent=page.view&id=61&sub_id=61	Advanced Gaming Systems	Own engine	
Hazmat Hotzone	http://www.etc.cmu.edu/projects/hazmat/about.php	Entertainment Technology Center (CMU)	Panda3D	http://panda3d.org/
Making History	http://www.making-history.com/home.php	Muzzy Lane	SIGMA own engine	http://www.muzzylane.com/
Megane Alonzo Challenge	http://www.gingerstudios.com/index2.htm	Ginger Studios	Virtools	http://www.virtools.com/
Pulse!!	http://www.sp.tamucc.edu/pulse/	BreakAway	Source (?)	http://www.valvesoftware.com/
Serious Gordon	http://seriousgordon.blogspot.com/	Dublin Institute of Technology	Source	http://www.valvesoftware.com/
Ship Simulator / Rescue Sim	http://www.vstepoffice.nl/rescuesim/	Vstep	Quest3D	http://www.quest3d.com/
SimExam	http://www.simspace.es/index_ingles.htm	SimSpace	Delta3D	http://www.delta3d.org/index.php
Simport	http://www.simport.eu/	Tygron	Own engine	http://www.tygron.nl/
Straight Shooter	http://www.games2train.com/site/html/tutor2.html	Games2Train	??	
Tactical Languge & Culture Training System	http://www.tacticallanguage.com/tacticaliraqi/	Tactical Language Training LLC	Unreal Tourn. 2003	http://www.unrealtechnology.com/html/homefold/home.shtml
The Monkey Wrench Conspiracy	http://www.games2train.com/site/html/tutor.html	Games2Train	??	
Timez Attack	http://www.bigbrainz.com/index.php	Big Brainz	Unity	http://unity3d.com/unity/index.html
VBS1	http://www.virtualbattlespace.com/	Bohemia Interactive Studio	Real Virtuality	http://www.bistudio.com/inside/tech.html
VBS2	http://www.vbs2.com/site/index.html	Bohemia Interactive Studio	Real Virtuality 2	http://www.bistudio.com/inside/tech.html
Virthualis	http://www.virthualis.org/	Virthualis	Delta3D	http://www.delta3d.org/index.php
WolfQuest	http://www.wolfquest.org/	Eduweb	Unity	http://unity3d.com/unity/index.html
Yourself!Fitness	http://www.yourselffitness.com/	Pop Art	??	

Game engine comparison

	Gra	EC	MC	Ed	GUI	AI	Phy	Co	Doc	Too	So	Price
Delta3D	2	2	3	2	2	1	2	4	2	2	1	free
Quest3D	3	3	3	4	2	-	2	1	3	3	3	€ 1249
Source	4	4	4	4	4	3	4	4	3	4	4	free when distributed freely and only for Steam users
Torque	3	3	3	3	1	-	1	4	3	3	2	$ 150 / $ 749
Unity	4	3	4	4	3	-	3	1	3	3	2	$ 149 / $ 1099
Unreal 2	4	4	4	4	4	3	3	3	3	4	4	free when no game is made

Term	Meaning	Description
Gra	Graphics	Graphical features like light, shaders, etc.
EC	Entity Control	How entities can be placed and created.
MC	Model Control	Compatibility with different model programs, etc.
Ed	Editor	The world editor to create levels.
GUI	Graphical User Interface	How does the game support GUI's?
AI	Artificial Intelligence	Artificial Intelligence framework.
Phy	Physics	The physics and/or collision system.
Co	Compatibility with other code	How easy can other code libraries be used with the engine?
Doc	Documentation	Tutorials, forums, wiki's.
Too	Tools	Additional tools like scripting, animation tools.
So	Sound	Complexity of the sound system.

Grade	Description
-	Not available
1	Simple
2	Sufficient
3	Good
4	Very good

To compare the game engines used for serious games, a review of the sites has been performed (see [41][64][76][79][80][81]). These sites describe the features of the engine, which can be categorized in several factors. The second table describes these factors the game engines are compared by. The third table of this section explains the grade given to these factors.

An additional feature is compared, namely the price of distributing a produced game with the engine. The Delta3D engine is cost-free, meaning the engine can be downloaded freely and the developers do not have to pay any royalty fee for the distribution of a game. A developer needs to buy the Quest3D engine in order to create games with it. The minimum package costs 1249 euro. For both the Torque and Unity game engine, two prices are given. The left price is the costs for purchasing the engine for an independent game studio (i.e. small software developers that are not owned by a single publisher); the right price is the costs for commercial game studios and developers.

For both the Source and the Unreal 2 engine it is free to publish a created game, unless certain premises are met. These engines are licensed instead of bought (although both can be purchased in negotiation with the developer). For the Unreal 2 engine, the license states the engine can only be used freely for applications other than games. However, the question if serious games are also games remains unanswered in the license. For the publication of a game, the developer needs to license the engine for $ 350,000 plus a 3% royalty on all revenue from the game.

For using the Source engine, the developer needs to buy one of the published games by Valve. If this game is registered, the developer also has access to the Source SDK. Creating a mod for the game purchased is free of charge when the mod is published freely on the Internet. People who want to play the mod, need to have the original game the mod is based on. An example would be a modification of the game Half-Life 2. Both the developer and the future players need to purchase Half-Life 2 in order

to create and play the mod. Half-Life 2 can be purchased for € 15 in stores or on Steam [77]. If developers want to license the Source engine and the Source SDK for commercial distribution, they need to contact Valve in order to come to a certain price.

Both Quest3D and the Unity engine are interpreters, meaning the engine reads in 'scripted files' that provide the necessary functionality. While the developer is playing, these files can be altered without shutting down the game. While it is possible for a developer to create new 'scripted files', it is harder to attach an already existing application to such an engine. Therefore, `compatibility with other code` is ranked lower for these engines than for the compiler-based engines.

A big difference can be seen between Source and Unreal 2 and the other engines. This has to do with the fact that the Source and the Unreal 2 engine are created and used for the development of Commercial-Off-The-Shelf (COTS) games. These engines also provide an AI framework, since the games created with the engines use computer opponents that need to act intelligently in the virtual world.

Appendix B – Use case descriptions

This appendix describes the use cases depicted in figure 8.1 in more detail.

Log in

Use case: Log in.
Actor: The player of the game or the instructor.
Starting situation: The system starts and displays the main menu.

1. The actor can either press the button 'Log in' or he can run a new game or tutorial. The log in panel will then pop up.
2. The actor enters his name and presses enter.
3. In the case of pressing the 'Log in' button in the main menu, the actor returns to the main menu. In the case of the starting of a game, the game starts loading when the actor hits enter.

Play tutorial

Use case: Play tutorial.
Actor: The player of the game.
Starting situation: The system starts and displays the main menu.

1. The actor presses the button 'Play tutorial'.
2. A new panel pops up where the actor can select which tutorial to start.
3. The actor chooses the tutorial he wants to play.
4. In case the actor hasn't logged in, the log in panel pops up as described in the 'Log in' use case.
5. The user logs in when needed.
6. The system starts to run the tutorial.

In the diagram a few extensions are sketched to give an indication of what type of tutorials there could be. For instance, to get familiar with the general movement within the world the player can choose to follow the 'general movement' tutorial. Because this serious game could be used for different purposes, it is crucial to build up the tutorials (and games) as modular as possible. With this format, the game designer or even the expert can easily add new tutorials for additional features.

Play game

Use case: Play game.
Actor: The player of the game.
Starting situation: The system starts and displays the main menu.

1. The actor presses the button 'Play game'.
2. A new panel pops up where the actor can select the type of game fitting the occupation of the actor.
3. The actor chooses his appropriate occupation.
4. The panel displays the worlds that can be played for this occupation.
5. The actor selects the world he wants to play.
6. In case the actor hasn't logged in, the log in panel pops up as described in the 'Log in' use case.
7. The user logs in when needed.
8. The system starts to run the game.

There should be a possibility to create different game types for the different occupations there are when a calamity is active. A fire fighter and someone from the medical staff should be dealt different learning plans and different objectives in the game.

The world where the disaster is happening could however still be the same. The world should contain both information for creating a medical and a fire-fighting problem. The system can then filter that information for the particular player of the game.

Evaluate actions

Use case: Evaluate actions I.
Actor: The player of the game.
Starting situation: The actor is playing a tutorial or a game.

1. The actor presses the 'Escape' button.
2. The main menu comes to the foreground.
3. The actor presses 'Show evaluation'.
4. A panel pops up where global information is provided for the actor. The panel also has a section where specific information is given per object or objective.
5. The actor can scroll through the evaluation report and closes the panel.
6. The main menu comes up again.
7. The actor can choose to quit the game or can resume the game he's been playing.

This is an explanation of a global evaluation report of how the actor did during the playing of the game. Another form of evaluation that could take place is giving feedback to the user directly in game. This is one of the most important points within a serious game or Intelligent Tutoring System (ITS).

Also note that this use case only applies to the player of the game. The following use case is for the instructor in game.

Use case: Evaluate actions II.
Actor: The instructor.
Starting situation: The actor has logged in as instructor and the system displays the main menu.

1. The actor presses the 'Show evaluations' button.
2. A panel pops up with the list of players that have played the game.
3. The actor selects the player he wants more information from.
4. The panel displays the maps the player has been playing, including the tutorials. The panel also displays global information concerning the player.
5. The actor can select one of the maps to get an evaluation report of the player with respect to that map.
6. When done, the actor closes the panel and returns to the main menu.

Spectate

Use case: Spectate.
Actor: The instructor.
Starting situation: The actor has logged in as instructor and the system displays the main menu.

1. The actor presses the button 'Spectate'.
2. A panel pops up with the list of games that are currently being played.
3. The actor selects the game he wants to spectate.
4. The system starts the game, letting the actor join as spectator.

Make notes

Use case: Make notes.
Actor: The instructor.
Starting situation: The actor is spectating a particular game.

1. The actor selects a player the actor wants to write some remarks on.
2. A panel pops up with a standard text editor interface. When the actor has already noted something concerning this player, the panel shows the earlier made remarks.
3. The actor enters his remarks in this panel and closes when finished.
4. The actor can spectate the game again.

Create scenario

Use case: Create scenario I.
Actor: The instructor.
Starting situation: The actor has logged in as instructor and the system displays the main menu.

1. The actor presses the 'Create scenario' button.
2. A panel pops up with the list of maps that can be played by players.
3. The actor chooses one of the maps that he wants to created a scenario for.
4. The panel displays specific information about the map that can be altered to give a different beginning for the players.
5. The actor alters the information to his wishes and saves the new scenario.
6. The system stores the parameters and closes the panel.

The specific information that is talked about in this use case could be a variety of objects in the world. An example would be specific properties of a victim in the disaster area, like the starting location of the victim, whether he/she is injured, or how many victims there are in total when the player starts the game. In case of a fire-fighter mission, the actor could pre-define where a fire should break out and if an explosion should follow the spread of the fire.

All the parameters that can be altered are discussed in a later chapter concerning the global loop of the ITS.

Use case: Create scenario II.
Actor: The expert.
Starting situation: The actor has opened the specific text file where the scenarios are stored.

1. The actor alters the specific parameters of a map inside the text file.
2. Once the changes have been made, the actor saves the file.

With this approach, an expert can create or alter a pre-defined scenario without entering the system. The system should read in these files on start-up to use the latest changes in the scenarios. With this use case, an expert can give his opinion in a structured text that could be easier for him to manipulate than in game.

Create world

Use case: Create world.
Actor: The expert.
Starting situation: The actor has started the specific world editor of the Source SDK.

1. The actor creates a completely new world with entities and all.
2. The actor compiles the new map.
3. When no errors are made, the new map can be placed inside the games directory

Again, this use case allows the expert to create a map without entering the system. The system reads in the newly created map and players can play the map. With this functionality, the system can always be updated with new calamities so it can be an ever-changing experience for the player.

Insert knowledge

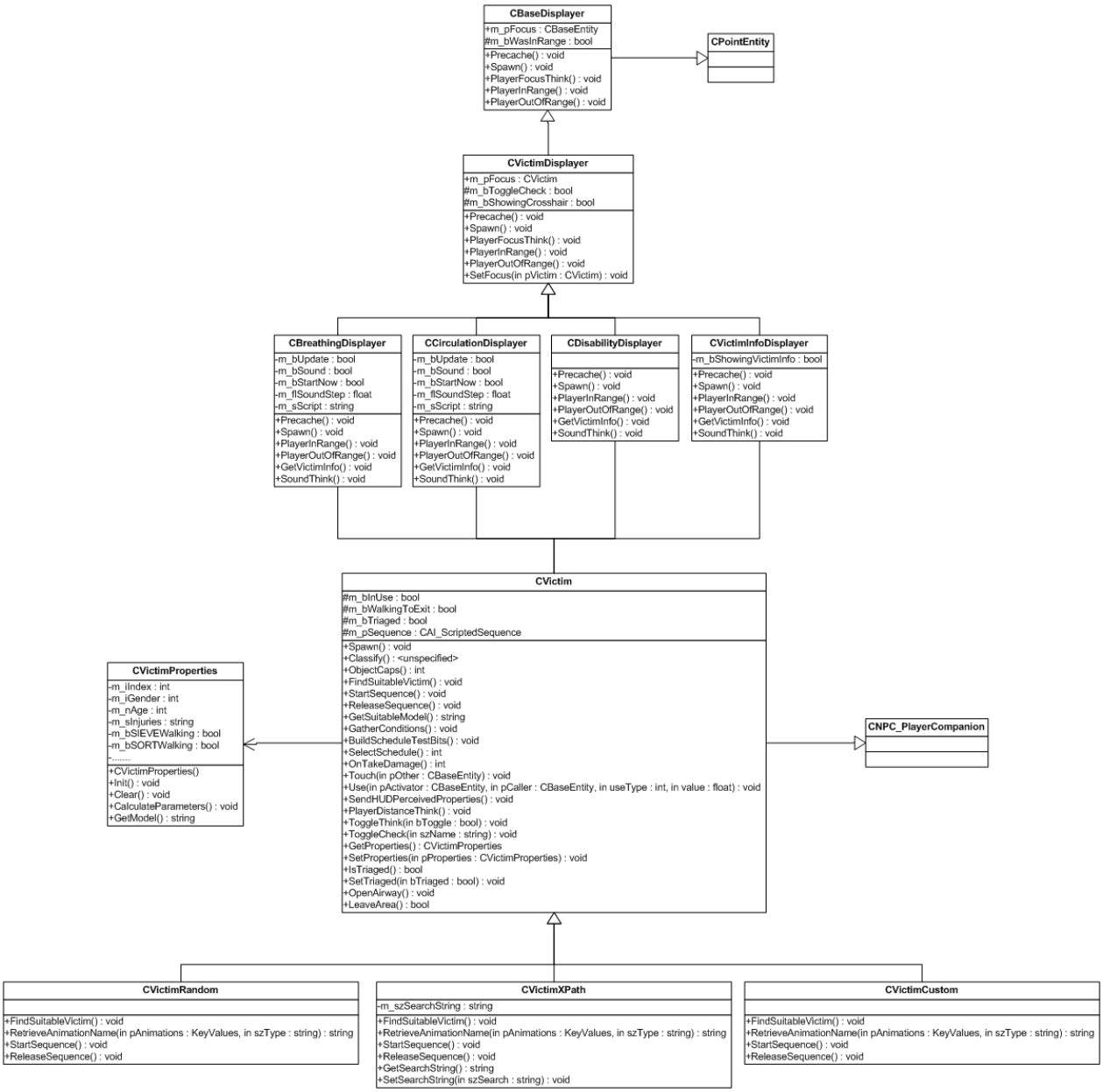
Use case: Insert knowledge.
Actor: The expert.
Starting situation: The actor has opened the specific text file where the knowledge that needs to be updated is located.

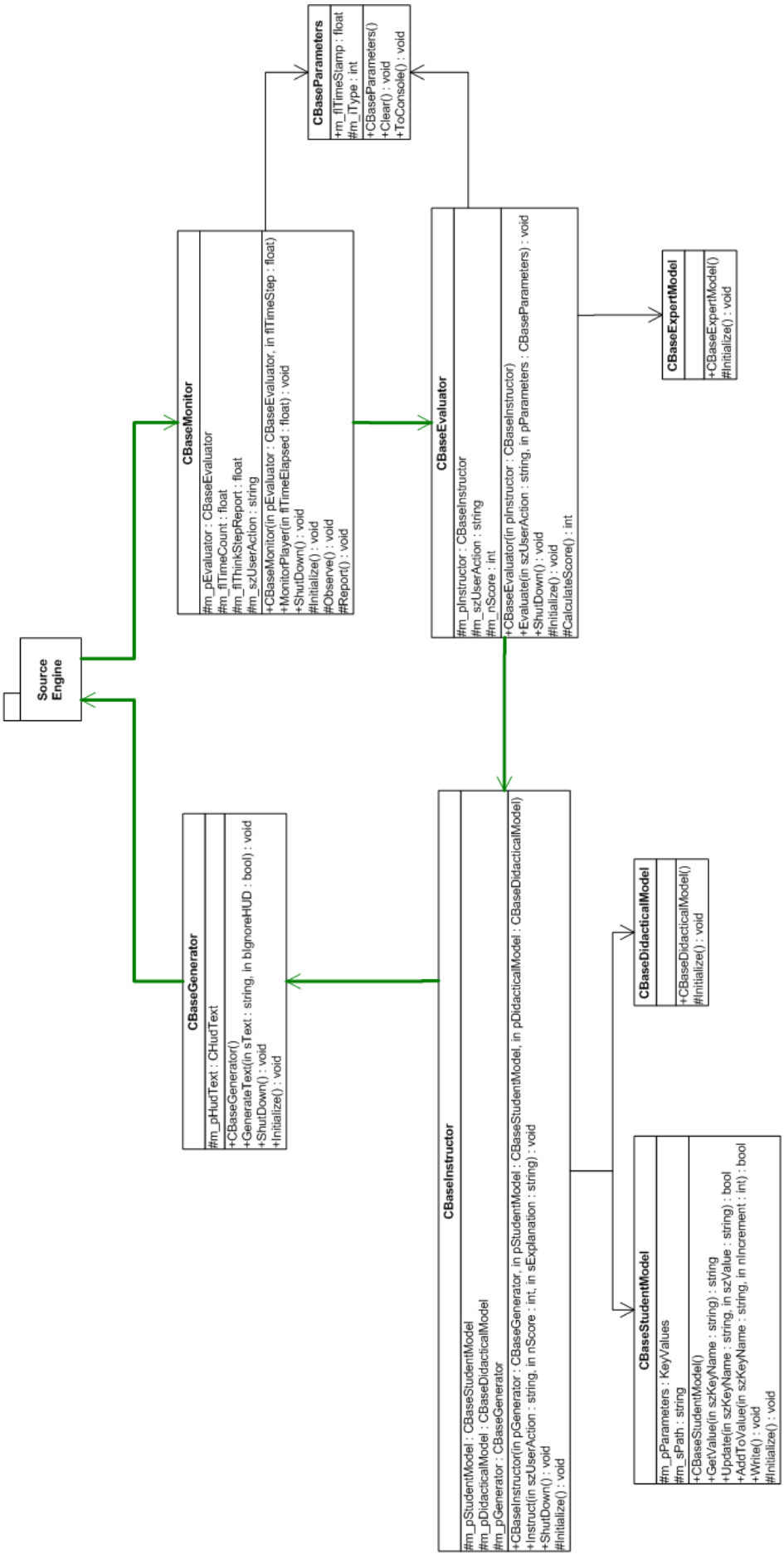
1. The actor alters specific rules or checklists in the text file.
2. Once the changes have been made, the actor saves the file.

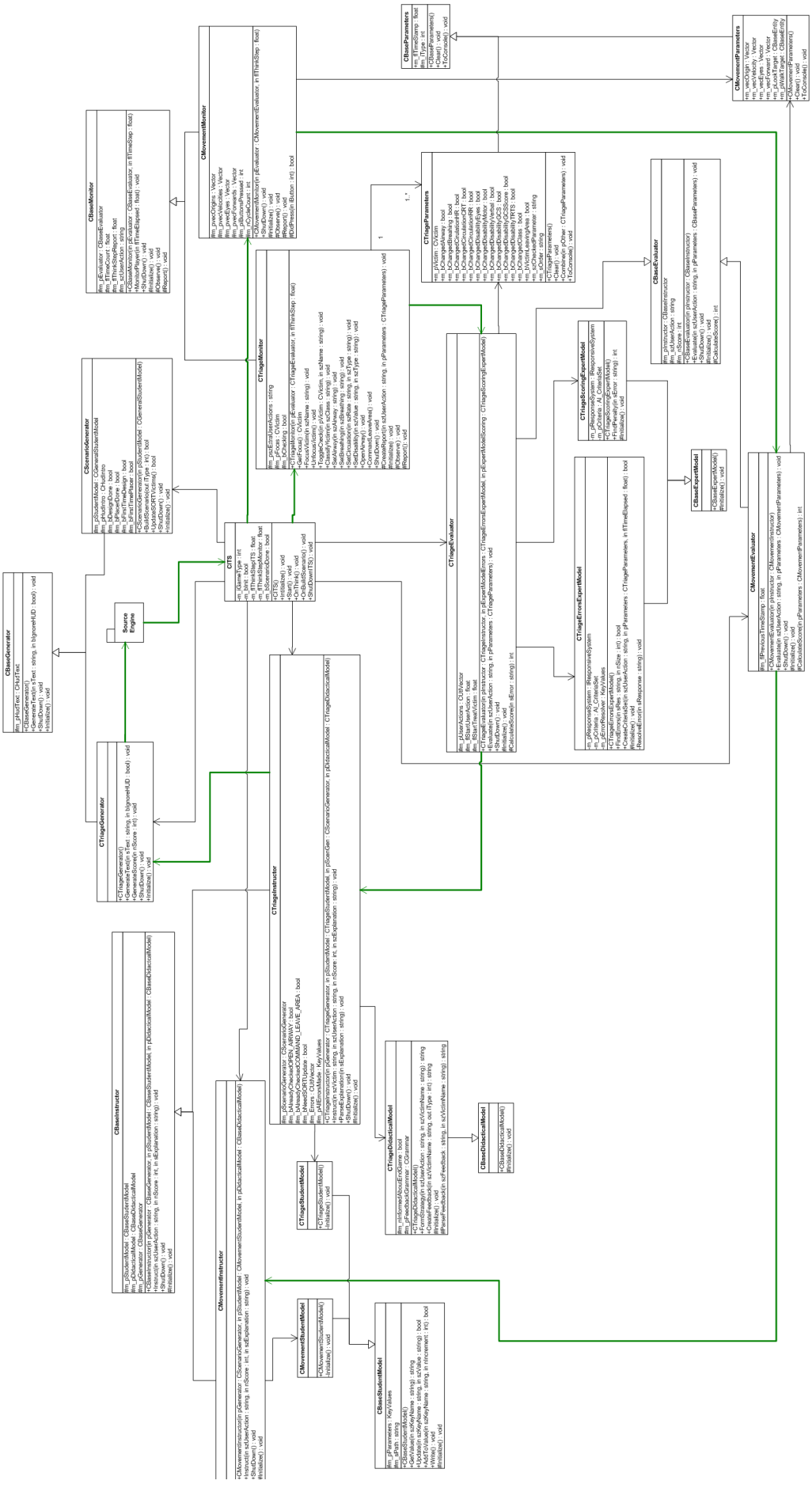
Whenever new knowledge should be used in the system, the expert should be able to easily add or change this knowledge without rebuilding the complete system. The knowledge should therefore be stored in text files that can easily be accessible by the persons who want to alter them. In this case, no programmer has to be called to change a small portion of rules or information in code.

Appendix C – UML diagrams

Complete UML diagrams with functions and variables.




CBaseParameters
+m_flTimeStamp : float
#m_iType : int
+CBaseParameters()
+Clear() : void
+ToConsole() : void
CBaseMonitor
#m_pEvaluator : CBaseEvaluator
#m_flTimeCount : float
#m_flThinkStepReport : float
#m_szUserAction : string
+CBaseMonitor(in pEvaluator : CBaseEvaluator, in flTimeStep : float)
+MonitorPlayer(in flTimeElapsed : float) : void
+ShutDown() : void
#Initialize() : void
#Observe() : void
#Report() : void
CBaseEvaluator
#m_pInstructor : CBaseInstructor
#m_szUserAction : string
#m_nScore : int
+CBaseEvaluator(in pInstructor : CBaseInstructor)
+Evaluate(in szUserAction : string, in pParameters : CBaseParameters) : void
+ShutDown() : void
#Initialize() : void
#CalculateScore() : int
CBaseExpertModel
+CBaseExpertModel()
#Initialize() : void
Source Engine
CBaseGenerator
#m_pHudText : CHudText
+CBaseGenerator()
+GenerateText(in sText : string, in bIgnoreHUD : bool) : void
+ShutDown() : void
#Initialize() : void
CBaseInstructor
#m_pStudentModel : CBaseStudentModel
#m_pDidacticalModel : CBaseDidacticalModel
#m_pGenerator : CBaseGenerator
+CBaseInstructor(in pGenerator : CBaseGenerator, in pStudentModel : CBaseStudentModel, in pDidacticalModel : CBaseDidacticalModel)
+Instruct(in szUserAction : string, in nScore : int, in sExplanation : string) : void
+ShutDown() : void
#Initialize() : void
CBaseDidacticalModel
+CBaseDidacticalModel()
#Initialize() : void
CBaseStudentModel
#m_pParameters : KeyValues
#m_sPath : string
+CBaseStudentModel()
+GetValue(in szKeyName : string) : string
+Update(in szKeyName : string, in szValue : string) : bool
+AddToValue(in szKeyName : string, in nIncrement : int) : bool
+Write() : void
#Initialize() : void




CDistribution
CDistributionReader
CLogger
CRecorder
CAgent
CRegion
CDesigner
CManager
CPlacer
CVictimLocation
CBaseGenerator
CScenarioGenerator
CGrammar

Appendix D – Expert models

expert_triage.txt

```
//=========================================================================
// Expert - Triage criteria
//=========================================================================
criterion "PlayingSIEVE" "game_type" "SIEVE" required
criterion "PlayingSORT" "game_type" "SORT" required
criterion "PlayingTraining" "game_type" "TRAINING" required

criterion "EditingCard" "user_action" "EDIT_CARD" required
criterion "CheckingVictim" "user_action" "CHECK_VICTIM" required

criterion "AirwayChanged" "airway_changed" "1" required
criterion "BreathingChanged" "breathing_changed" "1" required
criterion "CirculationHRChanged" "circulation_hr_changed" "1" required
criterion "CirculationCRTChanged" "circulation_crt_changed" "1" required
criterion "CirculationRRChanged" "circulation_rr_changed" "1" required
criterion "DisabilityEyesChanged" "disability_eyes_changed" "1" required
criterion "DisabilityMotorChanged" "disability_motor_changed" "1" required
criterion "DisabilityVerbalChanged" "disability_verbal_changed" "1" required
criterion "DisabilityGCSChanged" "disability_gcs_changed" "1" required
criterion "DisabilityGCSScoreChanged" "disability_gcs_score_changed" "1" required
criterion "DisabilityTRTSChanged" "disability_trts_changed" "1" required
criterion "DisabilityAVPUChanged" "disability_avpu_changed" "1" required
criterion "NotCheckedChecking" "parameter_checked_changed" "1" required

criterion "ObstructedAirway" "airway_actual" "0" required
criterion "CanOpenAirway" "can_open_airway" "1" required
criterion "IncorrectAirway" "airway_actual" "!=key:airway_perceived" required
criterion "BreathingTooHigh" "breathing_actual_max" "<key:breathing_perceived" required
criterion "BreathingTooLow" "breathing_actual_min" ">key:breathing_perceived" required
criterion "CirculationHRTooHigh" "circulation_hr_actual_max" "<key:circulation_hr_perceived"
required
criterion "CirculationHRTooLow" "circulation_hr_actual_min" ">key:circulation_hr_perceived"
required
criterion "IncorrectCirculationCRT" "circulation_crt_actual" "!=key:circulation_crt_perceived"
required
criterion "IncorrectCirculationRR" "circulation_rr_actual" "!=key:circulation_rr_perceived"
required
criterion "IncorrectDisabilityEyes" "disability_eyes_actual" "!=key:disability_eyes_perceived"
required
criterion "IncorrectDisabilityMotor" "disability_motor_actual"
"!=key:disability_motor_perceived" required
criterion "IncorrectDisabilityVerbal" "disability_verbal_actual"
"!=key:disability_verbal_perceived" required
criterion "IncorrectDisabilityGCS" "disability_gcs_actual" "!=key:disability_gcs_perceived"
required
criterion "IncorrectDisabilityGCSScore" "disability_gcs_score_actual"
"!=key:disability_gcs_score_perceived" required
criterion "IncorrectDisabilityTRTS" "disability_trts_actual" "!=key:disability_trts_perceived"
required
criterion "IncorrectDisabilityAVPU" "disability_avpu_actual" "!=key:disability_avpu_perceived"
required

criterion "CheckBreathing" "parameter_checked" "Breathing" required
criterion "CheckCirculation" "parameter_checked" "Circulation" required
criterion "CheckDisability" "parameter_checked" "Disability" required
criterion "NotCheckingDisability" "parameter_checked" "!=Disability" required

criterion "TimeElapsed5-15" "time_elapsed" ">5,<=15" required
criterion "TimeElapsed15-20" "time_elapsed" ">15,<=20" required
criterion "TimeElapsed20-30" "time_elapsed" ">20,<=30" required
criterion "TimeElapsed30-35" "time_elapsed" ">30,<=35" required
criterion "TimeElapsed>35" "time_elapsed" ">35" required
//=========================================================================
// Expert - Triage Errors
//=========================================================================
response "ERROR_AIRWAY"
{
        print "ERROR_AIRWAY"
}
```

```
response "ERROR_CAN_OPEN_AIRWAY"
{
	print "ERROR_CAN_OPEN_AIRWAY"
}
response "ERROR_BREATHING_TOO_HIGH"
{
	print "ERROR_BREATHING_TOO_HIGH"
}
response "ERROR_BREATHING_TOO_LOW"
{
	print "ERROR_BREATHING_TOO_LOW"
}
response "ERROR_CIRCULATION_HR_TOO_HIGH"
{
	print "ERROR_CIRCULATION_HR_TOO_HIGH"
}
response "ERROR_CIRCULATION_HR_TOO_LOW"
{
	print "ERROR_CIRCULATION_HR_TOO_LOW"
}
response "ERROR_CIRCULATION_CRT"
{
	print "ERROR_CIRCULATION_CRT"
}
response "ERROR_CIRCULATION_RR"
{
	print "ERROR_CIRCULATION_RR"
}
response "ERROR_DISABILITY_EYES"
{
	print "ERROR_DISABILITY_EYES"
}
response "ERROR_DISABILITY_MOTOR"
{
	print "ERROR_DISABILITY_MOTOR"
}
response "ERROR_DISABILITY_VERBAL"
{
	print "ERROR_DISABILITY_VERBAL"
}
response "ERROR_DISABILITY_GCS"
{
	print "ERROR_DISABILITY_GCS"
}
response "ERROR_DISABILITY_GCS_SCORE"
{
	print "ERROR_DISABILITY_GCS_SCORE"
}
response "ERROR_DISABILITY_TRTS"
{
	print "ERROR_DISABILITY_TRTS"
}
response "ERROR_DISABILITY_AVPU"
{
	print "ERROR_DISABILITY_AVPU"
}
response "ERROR_CHECKING_TOO_LONG"
{
	print "ERROR_CHECKING_TOO_LONG"
}
response "WARNING_CHECKING_30_SECONDS"
{
	print "WARNING_CHECKING_30_SECONDS"
}
//=============================================================================
// Expert - Triage Rules EDIT_CARD
//=============================================================================
rule "AirwayIncorrect"
{
	criteria	EditingCard AirwayChanged IncorrectAirway
	response	ERROR_AIRWAY
}
rule "DidNotOpenAirway"
{
	criteria	EditingCard AirwayChanged ObstructedAirway CanOpenAirway
	response	ERROR_CAN_OPEN_AIRWAY
}
```

```
rule "BreathingTooHigh"
{
	criteria	EditingCard BreathingChanged BreathingTooHigh
	response	ERROR_BREATHING_TOO_HIGH
}
rule "BreathingTooLow"
{
	criteria	EditingCard BreathingChanged BreathingTooLow
	response	ERROR_BREATHING_TOO_LOW
}
rule "CirculationHRTooHigh"
{
	criteria	EditingCard CirculationHRChanged CirculationHRTooHigh
	response	ERROR_CIRCULATION_HR_TOO_HIGH
}
rule "CirculationHRTooLow"
{
	criteria	EditingCard CirculationHRChanged CirculationHRTooLow
	response	ERROR_CIRCULATION_HR_TOO_LOW
}
rule "CirculationCRTIncorrect"
{
	criteria	EditingCard CirculationCRTChanged IncorrectCirculationCRT
	response	ERROR_CIRCULATION_CRT
}
rule "CirculationRRIncorrect"
{
	criteria	PlayingSORT EditingCard CirculationRRChanged IncorrectCirculationRR
	response	ERROR_CIRCULATION_RR
}
rule "DisabilityEyesIncorrect"
{
	criteria	PlayingSORT EditingCard DisabilityEyesChanged IncorrectDisabilityEyes
	response	ERROR_DISABILITY_EYES
}
rule "DisabilityMotorIncorrect"
{
	criteria	PlayingSORT EditingCard DisabilityMotorChanged IncorrectDisabilityMotor
	response	ERROR_DISABILITY_MOTOR
}
rule "DisabilityVerbalIncorrect"
{
	criteria	PlayingSORT EditingCard DisabilityVerbalChanged
			IncorrectDisabilityVerbal
	response	ERROR_DISABILITY_VERBAL
}
rule "DisabilityGCSIncorrect"
{
	criteria	PlayingSORT EditingCard DisabilityGCSChanged IncorrectDisabilityGCS
	response	ERROR_DISABILITY_GCS
}
rule "DisabilityGCSScoreIncorrect"
{
	criteria	PlayingSORT EditingCard DisabilityGCSScoreChanged
			IncorrectDisabilityGCSScore
	response	ERROR_DISABILITY_GCS_SCORE
}
rule "DisabilityTRTSIncorrect"
{
	criteria	PlayingSORT EditingCard DisabilityTRTSChanged IncorrectDisabilityTRTS
	response	ERROR_DISABILITY_TRTS
}
rule "DisabilityAVPUIncorrect"
{
	criteria	PlayingSORT EditingCard DisabilityAVPUChanged IncorrectDisabilityAVPU
	response	ERROR_DISABILITY_AVPU
}
//=============================================================================
// Expert - Triage Rules CHECK_VICTIM
//=============================================================================
rule "CheckingIncorrect"
{
	criteria	NotCheckedChecking CheckingVictim PlayingSIEVE NotCheckingDisability
			TimeElapsed>35
	response	ERROR_CHECKING_TOO_LONG
}
```

```
rule "CheckingIncorrect2"
{
	criteria	NotCheckedChecking CheckingVictim PlayingSORT TimeElapsed>35
	response	ERROR_CHECKING_TOO_LONG
}
rule "Checking30Seconds"
{
	criteria	NotCheckedChecking CheckingVictim PlayingSIEVE NotCheckingDisability
			TimeElapsed30-35
	response	WARNING_CHECKING_30_SECONDS
}
rule "Checking30Seconds2"
{
	criteria	NotCheckedChecking CheckingVictim PlayingSORT TimeElapsed30-35
	response	WARNING_CHECKING_30_SECONDS
}
rule "Checking30Seconds3"
{
	criteria	NotCheckedChecking CheckingVictim PlayingSIEVE NotCheckingDisability
			TimeElapsed20-30
	response	WARNING_CHECKING_30_SECONDS
}
rule "Checking30Seconds4"
{
	criteria	NotCheckedChecking CheckingVictim PlayingSORT TimeElapsed20-30
	response	WARNING_CHECKING_30_SECONDS
}

#include "rulebase/expert_triage_reasoning.txt"
```

expert_triage_reasoning.txt

```
// Triage classes
enumeration TriageClass
{
	"DECEASED"	"0"
	"T1"		"1"
	"T2"		"2"
	"T3"		"3"
	"T3_WOUNDED"	"4"
}
//=============================================================================
// Expert - Triage reasoning criteria
//=============================================================================
criterion "GivingCard" "user_action" "GIVE_CARD" required
criterion "LeavingVictim" "user_action" "LEAVE_VICTIM" required

criterion "NotCheckedReasoning" "reasoning_changed" "1" required
criterion "NotCheckedLeavingVictim" "leaving_victim_changed" "1" required

criterion "TriageClassChanged" "triage_class_changed" "1" required
criterion "IncorrectTriageClass" "triage_class_actual" "!=key:triage_class_perceived" required

criterion "IsWalking" "walking" "1" required
criterion "IsNotWalking" "walking" "0" required

criterion "IsBreathing" "airway_perceived" "1" required
criterion "IsNotBreathing" "airway_perceived" "0" required

criterion "Breathing10-29" "breathing_perceived" ">=10,<30" required
criterion "Breathing<10" "breathing_perceived" "<10" required
criterion "Breathing>=30" "breathing_perceived" ">=30" required

criterion "CirculationHR>=120" "circulation_hr_perceived" ">=120" required
criterion "CirculationHR<120" "circulation_hr_perceived" "<120" required
criterion "CirculationHRNotZero" "circulation_hr_perceived" "!=0" required
criterion "CirculationCRT>=2" "circulation_crt_perceived" ">=2" required
criterion "CirculationCRT<2" "circulation_crt_perceived" "<2" required
criterion "CirculationCRTNotZero" "circulation_crt_perceived" "!=0" required

criterion "IsDeceased" "triage_class_perceived" "0" required
criterion "IsT1" "triage_class_perceived" "1" required
criterion "IsT2" "triage_class_perceived" "2" required
criterion "IsT3" "triage_class_perceived" "3" required
criterion "IsT3Wounded" "triage_class_perceived" "4" required
criterion "IsNotDeceased" "triage_class_perceived" "!=0" required
```

```
criterion "IsNotT1" "triage_class_perceived" "!=1" required
criterion "IsNotT2" "triage_class_perceived" "!=2" required
criterion "IsNotT3" "triage_class_perceived" "!=3" required
criterion "IsNotT3Wounded" "triage_class_perceived" "!=4" required

criterion "MathErrorGCS" "disability_gcs_should_perceive" "!=key:disability_gcs_perceived"
weight 3 required
criterion "MathErrorGCSScore" "disability_gcs_score_should_perceive"
"!=key:disability_gcs_score_perceived" weight 2 required
criterion "MathErrorTRTS" "disability_trts_should_perceive" "!=key:disability_trts_perceived"
required

criterion "IsCheckBreathing1" "order_1" "CHECK_VICTIM_Breathing" required
criterion "IsCheckBreathing3" "order_3" "CHECK_VICTIM_Breathing" required

criterion "IsNotCheckBreathing1" "order_1" "!=CHECK_VICTIM_Breathing" required
criterion "IsNotCheckBreathing2" "order_2" "!=CHECK_VICTIM_Breathing" required
criterion "IsNotCheckBreathing3" "order_3" "!=CHECK_VICTIM_Breathing" required
criterion "IsNotCheckBreathing4" "order_4" "!=CHECK_VICTIM_Breathing" required
criterion "IsNotCheckBreathing5" "order_5" "!=CHECK_VICTIM_Breathing" required
criterion "IsNotCheckBreathing6" "order_6" "!=CHECK_VICTIM_Breathing" required
criterion "IsNotCheckBreathing7" "order_7" "!=CHECK_VICTIM_Breathing" required
criterion "IsNotCheckBreathing8" "order_8" "!=CHECK_VICTIM_Breathing" weight 5 required

criterion "IsCheckCirculation2" "order_2" "CHECK_VICTIM_Circulation" required
criterion "IsCheckCirculation4" "order_4" "CHECK_VICTIM_Circulation" required

criterion "IsNotCheckCirculation1" "order_1" "!=CHECK_VICTIM_Circulation" required
criterion "IsNotCheckCirculation2" "order_2" "!=CHECK_VICTIM_Circulation" required
criterion "IsNotCheckCirculation3" "order_3" "!=CHECK_VICTIM_Circulation" required
criterion "IsNotCheckCirculation4" "order_4" "!=CHECK_VICTIM_Circulation" required
criterion "IsNotCheckCirculation5" "order_5" "!=CHECK_VICTIM_Circulation" required
criterion "IsNotCheckCirculation6" "order_6" "!=CHECK_VICTIM_Circulation" required
criterion "IsNotCheckCirculation7" "order_7" "!=CHECK_VICTIM_Circulation" required
criterion "IsNotCheckCirculation8" "order_8" "!=CHECK_VICTIM_Circulation" weight 3 required

criterion "IsCheckDisability3" "order_3" "CHECK_VICTIM_Disability" required

criterion "IsNotCheckDisability1" "order_1" "!=CHECK_VICTIM_Disability" required
criterion "IsNotCheckDisability2" "order_2" "!=CHECK_VICTIM_Disability" required
criterion "IsNotCheckDisability3" "order_3" "!=CHECK_VICTIM_Disability" required
criterion "IsNotCheckDisability4" "order_4" "!=CHECK_VICTIM_Disability" required
criterion "IsNotCheckDisability5" "order_5" "!=CHECK_VICTIM_Disability" required
criterion "IsNotCheckDisability6" "order_6" "!=CHECK_VICTIM_Disability" required
criterion "IsNotCheckDisability7" "order_7" "!=CHECK_VICTIM_Disability" required
criterion "IsNotCheckDisability8" "order_8" "!=CHECK_VICTIM_Disability" required

criterion "IsNotEditCard3" "order_3" "!=EDIT_CARD" required
criterion "IsNotEditCard4" "order_4" "!=EDIT_CARD" required
criterion "IsNotEditCard5" "order_5" "!=EDIT_CARD" required

criterion "IsLeaveVictim1" "order_1" "LEAVE_VICTIM" required
criterion "IsNotLeaveVictim1" "order_1" "!=LEAVE_VICTIM" required

criterion "IsOpenAirway1" "order_1" "OPEN_AIRWAY" required
criterion "IsOpenAirway2" "order_2" "OPEN_AIRWAY" required
criterion "IsNotOpenAirway2" "order_2" "!=OPEN_AIRWAY" required
//=============================================================================
// Expert - Triage Reasoning Errors
//=============================================================================
response "ERROR_TRIAGE_CLASS"
{
	print "ERROR_TRIAGE_CLASS"
}
response "ERROR_REASONING_NOT_T3"
{
	print "ERROR_REASONING_NOT_T3"
}
response "ERROR_REASONING_T3"
{
	print "ERROR_REASONING_T3"
}
response "ERROR_REASONING_NOT_DECEASED"
{
	print "ERROR_REASONING_NOT_DECEASED"
}
```

```
response "ERROR_REASONING_DECEASED"
{
	print "ERROR_REASONING_DECEASED"
}
response "ERROR_REASONING_T1_STEP_B"
{
	print "ERROR_REASONING_T1_STEP_B"
}
response "ERROR_REASONING_T1_STEP_C"
{
	print "ERROR_REASONING_T1_STEP_C"
}
response "ERROR_REASONING_T2"
{
	print "ERROR_REASONING_T2"
}
response "ERROR_MATH_GCS"
{
	print "ERROR_MATH_GCS"
}
response "ERROR_MATH_GCS_SCORE"
{
	print "ERROR_MATH_GCS_SCORE"
}
response "ERROR_MATH_TRTS"
{
	print "ERROR_MATH_TRTS"
}
response "ERROR_NO_CHECK_BREATHING"
{
	print "ERROR_NO_CHECK_BREATHING"
}
response "ERROR_NO_CHECK_CIRCULATION"
{
	print "ERROR_NO_CHECK_CIRCULATION"
}
response "ERROR_NO_CHECK_DISABILITY"
{
	print "ERROR_NO_CHECK_DISABILITY"
}
response "ERROR_INCORRECT_ORDER"
{
	print "ERROR_INCORRECT_ORDER"
}
//=============================================================================
// Expert - Triage Rules GIVE_CARD
//=============================================================================
rule "TriageClassIncorrect"
{
	criteria	GivingCard TriageClassChanged IncorrectTriageClass
	response	ERROR_TRIAGE_CLASS
}
rule "SieveNoT3"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsNotWalking IsT3
	response	ERROR_REASONING_NOT_T3
}
rule "SieveNoT3Wounded"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsNotWalking IsT3Wounded
	response	ERROR_REASONING_NOT_T3
}
rule "SieveIncorrectT3"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsWalking IsNotT3
			IsNotT3Wounded
	response	ERROR_REASONING_T3
}
rule "SIEVENoDeceasedA"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsNotWalking IsBreathing
			IsDeceased
	response	ERROR_REASONING_NOT_DECEASED
}
```

```
rule "SIEVEIncorrectDeceasedA"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsNotWalking IsNotT3
			IsNotT3Wounded IsNotBreathing IsNotDeceased
	response	ERROR_REASONING_DECEASED
}
rule "SIEVEIncorrectT1B"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsNotWalking IsNotT3
			IsNotT3Wounded IsBreathing IsNotDeceased Breathing<10 IsNotT1
	response	ERROR_REASONING_T1_STEP_B
}
rule "SIEVEIncorrectT1B2"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsNotWalking IsNotT3
			IsNotT3Wounded IsBreathing IsNotDeceased Breathing>=30 IsNotT1
	response	ERROR_REASONING_T1_STEP_B
}
rule "SIEVEIncorrectT1C"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsNotWalking IsNotT3
			IsNotT3Wounded IsBreathing IsNotDeceased Breathing10-29
			CirculationHR>=120 IsNotT1
	response	ERROR_REASONING_T1_STEP_C
}
rule "SIEVEIncorrectT1C2"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsNotWalking IsNotT3
			IsNotT3Wounded IsBreathing IsNotDeceased Breathing10-29
			CirculationCRT>=2 IsNotT1
	response	ERROR_REASONING_T1_STEP_C
}
rule "SIEVEIncorrectT2C"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsNotWalking IsNotT3
			IsNotT3Wounded IsBreathing IsNotDeceased Breathing10-29
			CirculationHR<120 CirculationHRNotZero IsNotT2
	response	ERROR_REASONING_T2
}
rule "SIEVEIncorrectT2C2"
{
	criteria	NotCheckedReasoning PlayingSIEVE GivingCard IsNotWalking IsNotT3
			IsNotT3Wounded IsBreathing IsNotDeceased Breathing10-29
			CirculationCRT<2 CirculationCRTNotZero IsNotT2
	response	ERROR_REASONING_T2
}
rule "SORTMathErrorGCS"
{
	criteria	NotCheckedReasoning PlayingSORT EditingCard MathErrorGCS
	response	ERROR_MATH_GCS
}
rule "SORTMathErrorGCSScore"
{
	criteria	NotCheckedReasoning PlayingSORT EditingCard MathErrorGCSScore
	response	ERROR_MATH_GCS_SCORE
}
rule "SORTMathErrorTRTS"
{
	criteria	NotCheckedReasoning PlayingSORT EditingCard MathErrorTRTS
	response	ERROR_MATH_TRTS
}
//=============================================================================
// Expert - Triage Rules LEAVE_VICTIM
//=============================================================================
rule "NotCheckBreathing"
{
	criteria	NotCheckedLeavingVictim LeavingVictim IsNotCheckBreathing1
			IsNotCheckBreathing2 IsNotCheckBreathing3 IsNotCheckBreathing4
			IsNotCheckBreathing5 IsNotCheckBreathing6 IsNotCheckBreathing7
			IsNotCheckBreathing8 IsNotLeaveVictim1
	response	ERROR_NO_CHECK_BREATHING
}
```

```
rule "NotCheckCirculation"
{
	criteria	NotCheckedLeavingVictim LeavingVictim IsNotCheckCirculation1
			IsNotCheckCirculation2 IsNotCheckCirculation3 IsNotCheckCirculation4
			IsNotCheckCirculation5 IsNotCheckCirculation6 IsNotCheckCirculation7
			IsNotCheckCirculation8 IsNotLeaveVictim1
	response	ERROR_NO_CHECK_CIRCULATION
}
rule "NotCheckDisability"
{
	criteria	PlayingSORT NotCheckedLeavingVictim LeavingVictim PlayingSORT
			IsNotCheckDisability1 IsNotCheckDisability2 IsNotCheckDisability3
			IsNotCheckDisability4 IsNotCheckDisability5 IsNotCheckDisability6
			IsNotCheckDisability7 IsNotCheckDisability8 IsNotLeaveVictim1
	response	ERROR_NO_CHECK_DISABILITY
}
rule "IncorrectOrder1"
{
	criteria	NotCheckedLeavingVictim LeavingVictim IsNotCheckBreathing1
			IsNotLeaveVictim1
	response	ERROR_INCORRECT_ORDER
}
rule "IncorrectOrder2"
{
	criteria	NotCheckedLeavingVictim LeavingVictim IsCheckBreathing1
			IsNotCheckCirculation2 IsNotOpenAirway2
	response	ERROR_INCORRECT_ORDER
}
rule "IncorrectOrder3"
{
	criteria	NotCheckedLeavingVictim LeavingVictim PlayingSIEVE IsCheckBreathing1
			IsCheckCirculation2 IsNotEditCard3
	response	ERROR_INCORRECT_ORDER
}
rule "IncorrectOrder4"
{
	criteria	NotCheckedLeavingVictim LeavingVictim PlayingSIEVE IsCheckBreathing1
			IsOpenAirway2 IsNotCheckBreathing3
	response	ERROR_INCORRECT_ORDER
}
rule "IncorrectOrder5"
{
	criteria	NotCheckedLeavingVictim LeavingVictim PlayingSIEVE IsCheckBreathing1
			IsOpenAirway2 IsCheckBreathing3 IsNotCheckCirculation4
	response	ERROR_INCORRECT_ORDER
}
rule "IncorrectOrder6"
{
	criteria	NotCheckedLeavingVictim LeavingVictim PlayingSIEVE IsCheckBreathing1
			IsOpenAirway2 IsCheckBreathing3 IsCheckCirculation4 IsNotEditCard5
	response	ERROR_INCORRECT_ORDER
}
rule "IncorrectOrder7"
{
	criteria	NotCheckedLeavingVictim LeavingVictim PlayingSORT IsCheckBreathing1
			IsCheckCirculation2 IsNotCheckDisability3
	response	ERROR_INCORRECT_ORDER
}
rule "IncorrectOrder8"
{
	criteria	NotCheckedLeavingVictim LeavingVictim PlayingSORT IsCheckBreathing1
			IsCheckCirculation2 IsCheckDisability3 IsNotEditCard4
	response	ERROR_INCORRECT_ORDER
}
```

expert_triage_scoring.txt

```
//=============================================================================
// Expert - Triage scoring criteria
//=============================================================================
criterion "NoError"                  "error" "ERROR_NONE" required
criterion "Airway"                   "error" "ERROR_AIRWAY" required
criterion "CanOpenAirway"            "error" "ERROR_CAN_OPEN_AIRWAY" required
criterion "BreathingTooHigh"         "error" "ERROR_BREATHING_TOO_HIGH" required
criterion "BreathingTooLow"          "error" "ERROR_BREATHING_TOO_LOW" required
criterion "CirculationHRTooHigh"     "error" "ERROR_CIRCULATION_HR_TOO_HIGH" required
criterion "CirculationHRTooLow"      "error" "ERROR_CIRCULATION_HR_TOO_LOW" required
```

```
criterion "CirculationCRT"           "error" "ERROR_CIRCULATION_CRT" required
criterion "CirculationRR"            "error" "ERROR_CIRCULATION_RR" required
criterion "DisabilityEyes"           "error" "ERROR_DISABILITY_EYES" required
criterion "DisabilityMotor"          "error" "ERROR_DISABILITY_MOTOR" required
criterion "DisabilityVerbal"         "error" "ERROR_DISABILITY_VERBAL" required
criterion "DisabilityGCS"            "error" "ERROR_DISABILITY_GCS" required
criterion "DisabilityGCSScore"       "error" "ERROR_DISABILITY_GCS_SCORE" required
criterion "DisabilityTRTS"           "error" "ERROR_DISABILITY_TRTS" required
criterion "DisabilityAVPU"           "error" "ERROR_DISABILITY_AVPU" required
criterion "TriageClass"              "error" "ERROR_TRIAGE_CLASS" required
criterion "CheckingTooLong"          "error" "ERROR_CHECKING_TOO_LONG" required
criterion "Checking30Seconds"        "error" "WARNING_CHECKING_30_SECONDS" required
criterion "ReasoningNotT3"           "error" "ERROR_REASONING_NOT_T3" required
criterion "ReasoningT3"              "error" "ERROR_REASONING_T3" required
criterion "ReasoningNotDeceased"     "error" "ERROR_REASONING_NOT_DECEASED" required
criterion "ReasoningDeceased"        "error" "ERROR_REASONING_DECEASED" required
criterion "ReasoningT1StepB"         "error" "ERROR_REASONING_T1_STEP_B" required
criterion "ReasoningT1StepC"         "error" "ERROR_REASONING_T1_STEP_C" required
criterion "ReasoningT2"              "error" "ERROR_REASONING_T2" required
criterion "MathGCS"                  "error" "ERROR_MATH_GCS" required
criterion "MathGCSScore"             "error" "ERROR_MATH_GCS_SCORE" required
criterion "MathTRTS"                 "error" "ERROR_MATH_TRTS" required
criterion "OpenAirway1"              "error" "WARNING_OPEN_AIRWAY_1" required
criterion "NotCheckBreathing"        "error" "ERROR_NO_CHECK_BREATHING" required
criterion "NotCheckCirculation"      "error" "ERROR_NO_CHECK_CIRCULATION" required
criterion "NotCheckDisability"       "error" "ERROR_NO_CHECK_DISABILITY" required
criterion "IncorrectOrder"           "error" "ERROR_INCORRECT_ORDER" required
//=============================================================================
// Expert - Triage scoring penalties
//=============================================================================
response "PENALTY_0"
{
       print "PENALTY_0"
}
response "PENALTY_5"
{
       print "PENALTY_5"
}
response "PENALTY_10"
{
       print "PENALTY_10"
}
response "PENALTY_15"
{
       print "PENALTY_15"
}
//=============================================================================
// Expert - Triage scoring Rules
//=============================================================================
rule "NoError"
{
       criteria       NoError
       response       PENALTY_0
}
rule "Airway"
{
       criteria       Airway
       response       PENALTY_10
}
rule "CanOpenAirway"
{
       criteria       CanOpenAirway
       response       PENALTY_15
}
rule "BreathingTooHigh"
{
       criteria       BreathingTooHigh
       response       PENALTY_10
}
rule "BreathingTooLow"
{
       criteria       BreathingTooLow
       response       PENALTY_10
}
```

```
rule "CirculationHRTooHigh"
{
	criteria	CirculationHRTooHigh
	response	PENALTY_10
}
rule "CirculationHRTooLow"
{
	criteria	CirculationHRTooLow
	response	PENALTY_10
}
rule "CirculationCRT"
{
	criteria	CirculationCRT
	response	PENALTY_10
}
rule "CirculationRR"
{
	criteria	CirculationRR
	response	PENALTY_10
}
rule "DisabilityEyes"
{
	criteria	DisabilityEyes
	response	PENALTY_10
}
rule "DisabilityMotor"
{
	criteria	DisabilityMotor
	response	PENALTY_10
}
rule "DisabilityVerbal"
{
	criteria	DisabilityVerbal
	response	PENALTY_10
}
rule "DisabilityGCS"
{
	criteria	DisabilityGCS
	response	PENALTY_10
}
rule "DisabilityGCSScore"
{
	criteria	DisabilityGCSScore
	response	PENALTY_10
}
rule "DisabilityTRTS"
{
	criteria	DisabilityTRTS
	response	PENALTY_10
}
rule "DisabilityAVPU"
{
	criteria	DisabilityAVPU
	response	PENALTY_10
}
rule "TriageClass"
{
	criteria	TriageClass
	response	PENALTY_15
}
rule "CheckingTooLong"
{
	criteria	CheckingTooLong
	response	PENALTY_10
}
rule "Checking30Seconds"
{
	criteria	Checking30Seconds
	response	PENALTY_5
}
rule "ReasoningNotT3"
{
	criteria	ReasoningNotT3
	response	PENALTY_10
}
```

```
rule "ReasoningT3"
{
	criteria	ReasoningT3
	response	PENALTY_10
}
rule "ReasoningNotDeceased"
{
	criteria	ReasoningNotDeceased
	response	PENALTY_10
}
rule "ReasoningDeceased"
{
	criteria	ReasoningDeceased
	response	PENALTY_10
}
rule "ReasoningT1StepB"
{
	criteria	ReasoningT1StepB
	response	PENALTY_10
}
rule "ReasoningT1StepC"
{
	criteria	ReasoningT1StepC
	response	PENALTY_10
}
rule "ReasoningT2"
{
	criteria	ReasoningT2
	response	PENALTY_10
}
rule "MathGCS"
{
	criteria	MathGCS
	response	PENALTY_5
}
rule "MathGCSScore"
{
	criteria	MathGCSScore
	response	PENALTY_5
}
rule "MathTRTS"
{
	criteria	MathTRTS
	response	PENALTY_5
}
rule "OpenAirway1"
{
	criteria	OpenAirway1
	response	PENALTY_5
}
rule "NotCheckBreathing"
{
	criteria	NotCheckBreathing
	response	PENALTY_5
}
rule "NotCheckCirculation"
{
	criteria	NotCheckCirculation
	response	PENALTY_5
}
rule "NotCheckDisability"
{
	criteria	NotCheckDisability
	response	PENALTY_5
}
rule "IncorrectOrder"
{
	criteria	IncorrectOrder
	response	PENALTY_5
}
```

error_resolver.txt

```
"resolve_error"
{
	"ERROR_NONE"				""
	"ERROR_AIRWAY"				"airway_changed"
	"ERROR_CAN_OPEN_AIRWAY"			"airway_changed"
	"ERROR_BREATHING_TOO_HIGH"		"breathing_changed"
	"ERROR_BREATHING_TOO_LOW"		"breathing_changed"
	"ERROR_CIRCULATION_HR_TOO_HIGH"		"circulation_hr_changed"
	"ERROR_CIRCULATION_HR_TOO_LOW"		"circulation_hr_changed"
	"ERROR_CIRCULATION_CRT"			"circulation_crt_changed"
	"ERROR_CIRCULATION_RR"			"circulation_rr_changed"
	"ERROR_DISABILITY_EYES"			"disability_eyes_changed"
	"ERROR_DISABILITY_MOTOR"		"disability_motor_changed"
	"ERROR_DISABILITY_VERBAL"		"disability_verbal_changed"
	"ERROR_DISABILITY_GCS"			"disability_gcs_changed"
	"ERROR_DISABILITY_GCS_SCORE"		"disability_gcs_score_changed"
	"ERROR_DISABILITY_TRTS"			"disability_trts_changed"
	"ERROR_DISABILITY_AVPU"			"disability_avpu_changed"
	"ERROR_TRIAGE_CLASS"			"triage_class_changed"
	"ERROR_CHECKING_TOO_LONG"		"parameter_checked_changed"
	"WARNING_CHECKING_30_SECONDS"		"parameter_checked_changed"
	"ERROR_REASONING_NOT_T3"		"reasoning_changed"
	"ERROR_REASONING_T3"			"reasoning_changed"
	"ERROR_REASONING_NOT_DECEASED"		"reasoning_changed"
	"ERROR_REASONING_DECEASED"		"reasoning_changed"
	"ERROR_REASONING_T1_STEP_B"		"reasoning_changed"
	"ERROR_REASONING_T1_STEP_C"		"reasoning_changed"
	"ERROR_REASONING_T2"			"reasoning_changed"
	"ERROR_MATH_GCS"			"reasoning_changed"
	"ERROR_MATH_GCS_SCORE"			"reasoning_changed"
	"ERROR_MATH_TRTS"			"reasoning_changed"
	"WARNING_OPEN_AIRWAY_1"			"leaving_victim_changed"
	"ERROR_NO_CHECK_BREATHING"		"leaving_victim_changed"
	"ERROR_NO_CHECK_CIRCULATION"		"leaving_victim_changed"
	"ERROR_NO_CHECK_DISABILITY"		"leaving_victim_changed"
	"ERROR_INCORRECT_ORDER"			"leaving_victim_changed"
}
```

Appendix E – Placement files

models.txt

```
"models"
{
	"A_BUI_CORN_1_TR"
	{
		"count" "1"
		"0"		"models/props_buildings/row_corner_1_fullscale.mdl"
		"entity"	"prop_dynamic_override"
		"solid" "SOLID_VPHYSICS"
		"offset"	"-56"
	}
	"A_BUI_CORN_2_TR"
	{
		"count" "1"
		"0"		"models/props_buildings/row_corner_1_fullscale.mdl"
		"entity"	"prop_dynamic_override"
		"solid" "SOLID_VPHYSICS"
		"offset"	"-56"
		"rotation"	"180"
	}
	"A_BUI_ROW_TR"
	{
		"count" "1"
		"0"		"models/props_buildings/row_church_fullscale.mdl"
		"entity"	"prop_dynamic_override"
		"solid" "SOLID_VPHYSICS"
		"offset"	"-56"
	}
	"A_FOL_TR"
	{
		"count" "8"
		"0"		"models/props_foliage/tree_deciduous_01a-lod.mdl"
		"1"		"models/props_foliage/tree_deciduous_01a.mdl"
		"2"		"models/props_foliage/tree_deciduous_02a.mdl"
		"3"		"models/props_foliage/tree_deciduous_03a.mdl"
		"4"		"models/props_foliage/tree_deciduous_03b.mdl"
		"5"		"models/props_foliage/tree_poplar_01.mdl"
		"6"		"models/props_foliage/bramble001a.mdl"
		"7"		"models/props_foliage/shrub_01a.mdl"
		//"6"		"models/props_foliage/oak_tree01.mdl"
		"entity"	"prop_dynamic_override"
		"solid" "SOLID_VPHYSICS"
		"offset"	"-56"
		"rotation"	"-1"
	}
	"A_VEH_CAR_TR"
	{
		"count" "4"
		"0"		"models/props_vehicles/car002a_physics.mdl"
		"1"		"models/props_vehicles/car003a_physics.mdl"
		"2"		"models/props_vehicles/car004a_physics.mdl"
		"3"		"models/props_vehicles/car005a_physics.mdl"
		"entity"	"prop_physics"
		"solid" "SOLID_VPHYSICS"
		"offset"	"-15"
		"rotation"	"-1"
	}
	"A_DEB_BICYCLE_TR"
	{
		"count" "1"
		"0"		"models/props_junk/bicycle01a.mdl"
		"entity"	"prop_physics"
		"solid" "SOLID_VPHYSICS"
		"offset"	"0"
		"rotation"	"-1"
	}
	"NIFV_VICTIMS"
	{
		"-1"		"models/victims/victim_blank_male.mdl"
		"16"		"models/victims/victim_16.mdl"
	}
```

```
}
```

grammar_sieve.txt

```
//============================================================================
// Grammar - criteria
//============================================================================
criterion "variableStart" "variable" "Start" required
criterion "variableDisaster" "variable" "Disaster" required
criterion "variableCrash" "variable" "Crash" required
criterion "variableEnvironment" "variable" "Environment" required
criterion "variableVictims" "variable" "Victims" required
criterion "variableVictim" "variable" "Victim" required
criterion "variableVictimx5" "variable" "Victimx5" required
criterion "variableVictimx10" "variable" "Victimx10" required
criterion "variableVictimx15" "variable" "Victimx15" required
criterion "variableVictimx20" "variable" "Victimx20" required
criterion "variableVictimx30" "variable" "Victimx30" required
criterion "variableVehicles" "variable" "Vehicles" required
criterion "variableBuildings" "variable" "Buildings" required
criterion "variableFoliage" "variable" "Foliage" required

criterion "mapTraffic" "map_name" "sieve_traffic" required

criterion "UserLevel1" "user_level" "1" required
criterion "UserLevel2" "user_level" "2" required
criterion "UserLevel3" "user_level" "3" required
criterion "UserLevel4" "user_level" "4" required
criterion "UserLevel5" "user_level" "5" required

criterion "UserLevel<=2" "user_level" "<=2" required
criterion "UserLevel>2" "user_level" ">2" required

criterion "Level1" "level" "1" required

criterion "Chance<50" "random" "<0.5" required
criterion "Chance>=50" "random" ">=0.5" required
criterion "Chance<83" "random" "<0.83" required
criterion "Chance>=83" "random" ">=0.83" required
criterion "Chance<98" "random" "<0.98" required
criterion "Chance>=98" "random" ">=0.98" required
//============================================================================
// Grammar - used for now
//
// Possible leave symbols:
// R_DISASTER_TR              disaster region (sieve_traffic)
// R_CRASH_TR                 center of a crash (sieve_traffic)
// A_BUI_CORN_1_TR            corner building (sieve_traffic)
// A_BUI_CORN_2_TR            another corner building (sieve_traffic)
// A_BUI_ROW_TR               row building (sieve_traffic)
// A_FOL_TR                   tree & bush (sieve_traffic)
// A_VEH_CAR_TR               regular car (sieve_traffic)
// A_DEB_BICYCLE_TR           bicycle (sieve_traffic)
// A_VIC_DECEASED_TR  deceased victim (sieve_traffic)
// A_VIC_T1_TR        T1 victim (sieve_traffic)
// A_VIC_T2_TR        T2 victim (sieve_traffic)
// A_VIC_T3_WOUNDED_TR T3 wounded victim (sieve_traffic)
// REM                        prints ""
//============================================================================
response "Buildings"
{
	print "<A_BUI_CORN_1_TR><A_BUI_ROW_TR>"
	print "<A_BUI_CORN_1_TR><A_BUI_CORN_2_TR>"
}
response "T1-Victim"
{
	print "<A_VIC_T1_TR><A_DEB_BICYCLE_TR>"
}
response "T2-Victim"
{
	print "<A_VIC_T2_TR><A_DEB_BICYCLE_TR>"
}
response "T3W-Victim"
{
	print "<A_VIC_T3_WOUNDED_TR><A_DEB_BICYCLE_TR>"
}
```

```
response "D-Victim"
{
	print "<A_VIC_DECEASED_TR>"
}
response "Victimx5"
{
	print "{Victim}{Victim}{Victim}{Victim}{Victim}"
}
response "Victimx10"
{
	print "{Victimx5}{Victimx5}"
}
response "Victimx15"
{
	print "{Victimx10}{Victimx5}"
}
response "Victimx20"
{
	print "{Victimx10}{Victimx10}"
}
response "Victimx30"
{
	print "{Victimx20}{Victimx10}"
}
response "Car"
{
	print "<A_VEH_CAR_TR>"
}
response "Cars"
{
	print "<A_VEH_CAR_TR><A_VEH_CAR_TR>"
}
response "Foliage"
{
	print
"<A_FOL_TR><A_FOL_TR><A_FOL_TR><A_FOL_TR><A_FOL_TR><A_FOL_TR><A_FOL_TR><A_FOL_TR><A_FOL_TR><A_
FOL_TR><A_FOL_TR><A_FOL_TR>"
}
response "Disaster"
{
	print "<R_DISASTER_TR>{Crash}{Environment}"
}
response "Crash"
{
	print "<R_CRASH_TR>{Victims}{Vehicles}"
}
response "Environment"
{
	print "{Buildings}{Foliage}"
	print "{Buildings}{Foliage}{Foliage}"
}
response "Start"
{
	print "{Disaster}"
}

//=============================================================================
// Grammar - rules
//=============================================================================
rule "Start"
{
	criteria	mapTraffic variableStart Level1
	response	Start
}
rule "Disaster"
{
	criteria	mapTraffic variableDisaster
	response	Disaster
}
rule "Crash"
{
	criteria	mapTraffic variableCrash
	response	Crash
}
```

```
rule "Victims"
{
	criteria	mapTraffic variableVictims UserLevel1
	response	Victimx5
}
rule "Victims2"
{
	criteria	mapTraffic variableVictims UserLevel2
	response	Victimx10
}
rule "Victims3"
{
	criteria	mapTraffic variableVictims UserLevel3
	response	Victimx15
}
rule "Victims4"
{
	criteria	mapTraffic variableVictims UserLevel4
	response	Victimx20
}
rule "Victims5"
{
	criteria	mapTraffic variableVictims UserLevel5
	response	Victimx30
}
rule "Victimx5"
{
	criteria	mapTraffic variableVictimx5
	response	Victimx5
}
rule "Victimx10"
{
	criteria	mapTraffic variableVictimx10
	response	Victimx10
}
rule "Victimx15"
{
	criteria	mapTraffic variableVictimx15
	response	Victimx15
}
rule "Victimx20"
{
	criteria	mapTraffic variableVictimx20
	response	Victimx20
}
rule "Victimx30"
{
	criteria	mapTraffic variableVictimx30
	response	Victimx30
}
rule "Victim"
{
	criteria	mapTraffic variableVictim Chance<50
	response	T3W-Victim
}
rule "Victim2"
{
	criteria	mapTraffic variableVictim Chance>=50 Chance<83
	response	T2-Victim
}
rule "Victim3"
{
	criteria	mapTraffic variableVictim Chance>=83 Chance<98
	response	T1-Victim
}
rule "Victim4"
{
	criteria	mapTraffic variableVictim Chance>=98
	response	D-Victim
}
rule "Vehicles"
{
	criteria	mapTraffic variableVehicles UserLevel>2
	response	Cars
}
```

```
rule "Vehicles2"
{
	criteria	mapTraffic variableVehicles UserLevel<=2
	response	Car
}
rule "Environment"
{
	criteria	mapTraffic variableEnvironment
	response	Environment
}
rule "Buildings"
{
	criteria	mapTraffic variableBuildings
	response	Buildings
}
rule "Foliage"
{
	criteria	mapTraffic variableFoliage
	response	Foliage
}
```

grammar_sort.txt

```
//=============================================================================
// Grammar - criteria
//=============================================================================
criterion "variableStart" "variable" "Start" required
criterion "variableDummy" "variable" "Dummy" required
criterion "variableVictim" "variable" "Victim" required

criterion "Level1" "level" "1" required
criterion "NotLevel1" "level" "!=1" required
criterion "LevelLast" "level_text" "last" required
criterion "NotLevelLast" "level_text" "!=last" required

criterion "UserLevel1" "user_level" "1" required
criterion "UserLevel2" "user_level" "2" required
criterion "UserLevel3" "user_level" "3" required
criterion "UserLevel4" "user_level" "4" required
criterion "UserLevel5" "user_level" "5" required

criterion "Chance<=65" "random" "<=0.65" required
criterion "Chance>65" "random" ">0.65" required
criterion "Chance<=95" "random" "<=0.95" required
criterion "Chance>95" "random" ">0.95" required
//=============================================================================
// Grammar - responses
//
// Possible leave symbols:
// A_VIC_DECEASED             deceased victim
// A_VIC_T1                   T1 victim
// A_VIC_T2                   T2 victim
// A_VIC_T3                   T3 victim
// A_VIC_T3_WOUNDED           T3 wounded victim
// REM                        prints ""
//=============================================================================
response "Start1"
{
	print "{Victim}{Victim}{Victim}{Victim}{Victim}"
}
response "Start2"
{
	print
"{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}"
}
response "Start3"
{
	print
"{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Vict
im}{Victim}{Victim}{Victim}"
}
response "Start4"
{
	print
"{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Vict
im}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}"
}
```

```
response "Start5"
{
	print
"{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Vict
im}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}{Vi
ctim}{Victim}{Victim}{Victim}{Victim}{Victim}{Victim}"
}
response "VictimT1"
{
	print "<A_VIC_T1>"
}
response "VictimT2"
{
	print "<A_VIC_T2>"
}
response "VictimT3_wounded"
{
	print "<A_VIC_T3_WOUNDED>"
}
//================================================================================
// Grammar - rules
//================================================================================
rule "Start1"
{
	criteria	variableStart Level1 UserLevel1
	response	Start1
}
rule "Start2"
{
	criteria	variableStart Level1 UserLevel2
	response	Start2
}
rule "Start3"
{
	criteria	variableStart Level1 UserLevel3
	response	Start3
}
rule "Start4"
{
	criteria	variableStart Level1 UserLevel4
	response	Start4
}
rule "Start5"
{
	criteria	variableStart Level1 UserLevel5
	response	Start5
}
rule "VictimT1"
{
	criteria	variableVictim Chance<=65
	response	VictimT1
}
rule "VictimT2"
{
	criteria	variableVictim Chance>65 Chance<=95
	response	VictimT2
}
rule "VictimT3_wounded"
{
	criteria	variableVictim Chance>95
	response	VictimT3_wounded
}
```

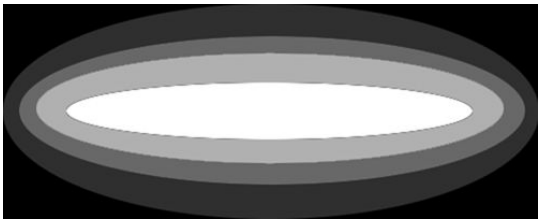
relation_table.txt

```
"relations"
{
	"R_DISASTER_TR"
	{
		"NIFV_ENTITY_TYPE_PLAYER_STARTING_POS"	"repel_32x32"
		"NIFV_REGION_TYPE_WORLD"	"attract_disaster-world_traffic"
	}
	"R_CRASH_TR"
	{
		"NIFV_ENTITY_TYPE_PLAYER_STARTING_POS"	"repel_32x32"
		"R_DISASTER_TR"	"attract_crash-disaster_traffic"
	}
```

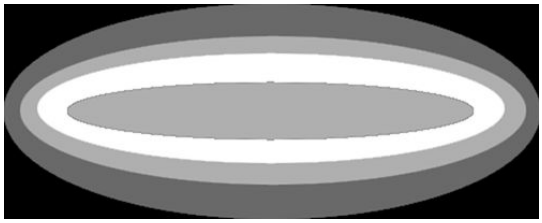
```
        "A_BUI_CORN_1_TR"
        {
                "NIFV_REGION_TYPE_WORLD"        "attract_building_corner1-world_traffic"
                "A_BUI_ROW_TR"                  "repel_corn_building-row_building_traffic"
        }
        "A_BUI_CORN_2_TR"
        {
                "NIFV_REGION_TYPE_WORLD"        "attract_building_corner2-world_traffic"
        }
        "A_BUI_ROW_TR"
        {
                "NIFV_REGION_TYPE_WORLD"        "attract_building_row-world_traffic"
        }
        "A_FOL_TR"
        {
                "NIFV_REGION_TYPE_WORLD"        "attract_foliage-world_traffic"
        }
        "A_VEH_CAR_TR"
        {
                "R_CRASH_TR"                    "attract_car-crash_traffic"
                "A_BUI_CORN_1_TR"               "repel_car-corn_building_traffic"
                "A_BUI_CORN_2_TR"               "repel_car-corn_building_traffic"
                "A_BUI_ROW_TR"                  "repel_car-row_building_traffic"
        }
        "A_DEB_BICYCLE_TR"
        {
                "R_CRASH_TR"                    "attract_bicycle-crash_traffic"
                "A_VEH_CAR_TR"                  "repel_240x240"
                "A_BUI_CORN_1_TR"               "repel_victim-corn_building_traffic"
                "A_BUI_CORN_2_TR"               "repel_victim-corn_building_traffic"
                "A_BUI_ROW_TR"                  "repel_victim-row_building_traffic"
        }
        "A_VIC_DECEASED_TR"
        {
                "R_CRASH_TR"                    "attract_D-crash_traffic"
                "A_VEH_CAR_TR"                  "repel_240x240"
                "A_BUI_CORN_1_TR"               "repel_victim-corn_building_traffic"
                "A_BUI_CORN_2_TR"               "repel_victim-corn_building_traffic"
                "A_BUI_ROW_TR"                  "repel_victim-row_building_traffic"
        }
        "A_VIC_T1_TR"
        {
                "R_CRASH_TR"                    "attract_T1-crash_traffic"
                "A_VEH_CAR_TR"                  "repel_240x240"
                "A_BUI_CORN_1_TR"               "repel_victim-corn_building_traffic"
                "A_BUI_CORN_2_TR"               "repel_victim-corn_building_traffic"
                "A_BUI_ROW_TR"                  "repel_victim-row_building_traffic"
        }
        "A_VIC_T2_TR"
        {
                "R_CRASH_TR"                    "attract_T2-crash_traffic"
                "A_VEH_CAR_TR"                  "repel_240x240"
                "A_BUI_CORN_1_TR"               "repel_victim-corn_building_traffic"
                "A_BUI_CORN_2_TR"               "repel_victim-corn_building_traffic"
                "A_BUI_ROW_TR"                  "repel_victim-row_building_traffic"

        }
        "A_VIC_T3_WOUNDED_TR"
        {
                "R_CRASH_TR"                    "attract_T3W-crash_traffic"
                "A_VEH_CAR_TR"                  "repel_240x240"
                "A_BUI_CORN_1_TR"               "repel_victim-corn_building_traffic"
                "A_BUI_CORN_2_TR"               "repel_victim-corn_building_traffic"
                "A_BUI_ROW_TR"                  "repel_victim-row_building_traffic"
        }
}
```

distributions



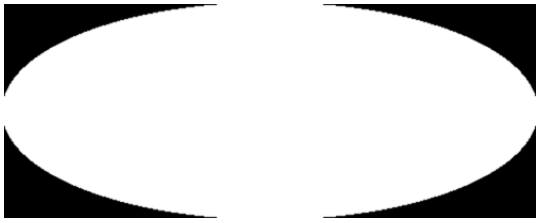
attract_T1-crash_traffic



attract_T2-crash_traffic

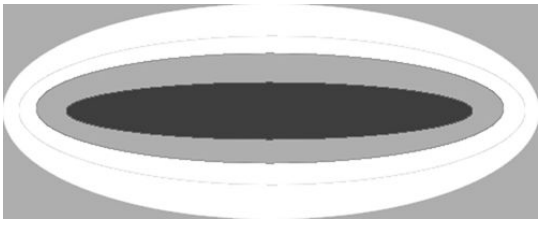
attract_building_corner2-world_traffic

attract_building_row-world_traffic

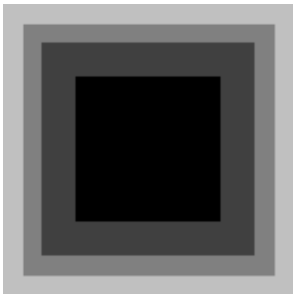


attract_bicycle-crash_traffic

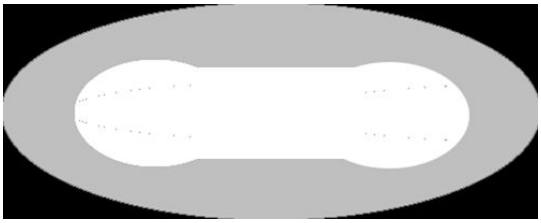
attract_building_corner1-world_traffic



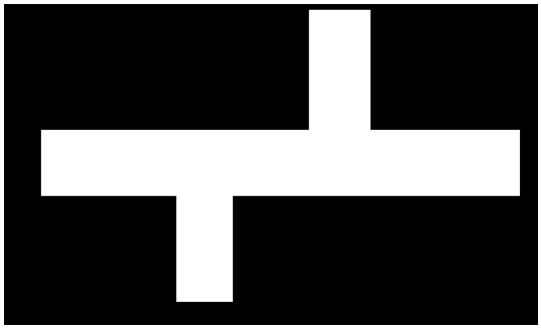
attract_T3W-crash_traffic



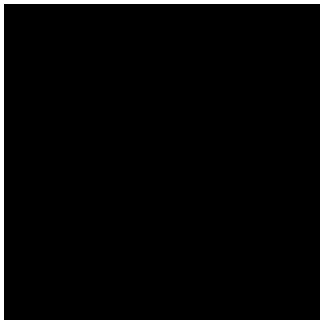
repel_32x32



attract_car-crash_traffic

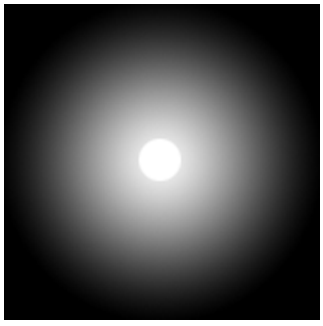


attract_car-streets_traffic

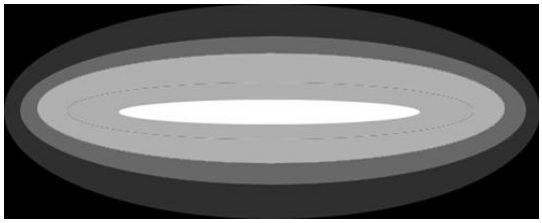


repel_240x240

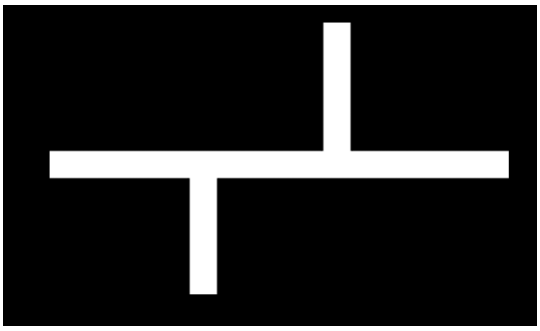
repel_car-corn_building_traffic



attract_crash-disaster_traffic



attract_D-crash_traffic



attract_disaster-world_traffic

attract_foliage-world_traffic

repel_car-row_building_traffic

repel_corn_building-row_building_traffic

repel_victim-corn_building_traffic

repel_victim-row_building_traffic

placement_info.txt

```
"placement_info"
{
	"R_DISASTER"
	{
		"MAX_STEPS"		"20"
		"MIN_STEP_SIZE"		"100"
		"MAX_STEP_SIZE"		"1500"
		"MIN_LOOK_DIST"		"100"
		"MAX_LOOK_DIST"		"1500"
	}
	"R_REST"
	{
		"MAX_STEPS"		"30"
		"MIN_STEP_SIZE"		"30"
		"MAX_STEP_SIZE"		"2500"
		"MIN_LOOK_DIST"		"30"
		"MAX_LOOK_DIST"		"2500"
	}
	"A_BUILDINGS"
	{
		"MAX_STEPS"		"200"
		"MIN_STEP_SIZE"		"10"
		"MAX_STEP_SIZE"		"3000"
		"MIN_LOOK_DIST"		"10"
		"MAX_LOOK_DIST"		"3000"
	}
	"A_FOLIAGE"
	{
		"MAX_STEPS"		"20"
		"MIN_STEP_SIZE"		"30"
		"MAX_STEP_SIZE"		"1500"
		"MIN_LOOK_DIST"		"30"
		"MAX_LOOK_DIST"		"1500"
	}
	"A_VEHICLES"
	{
		"MAX_STEPS"		"150"
		"MIN_STEP_SIZE"		"30"
		"MAX_STEP_SIZE"		"4000"
		"MIN_LOOK_DIST"		"30"
		"MAX_LOOK_DIST"		"4000"
	}
	"A_VICTIMS"
	{
		"MAX_STEPS"		"300"
		"MIN_STEP_SIZE"		"10"
		"MAX_STEP_SIZE"		"4000"
		"MIN_LOOK_DIST"		"10"
		"MAX_LOOK_DIST"		"4000"
	}
	"A_DEBRIS"
	{
		"MAX_STEPS"		"200"
		"MIN_STEP_SIZE"		"10"
		"MAX_STEP_SIZE"		"4000"
		"MIN_LOOK_DIST"		"10"
		"MAX_LOOK_DIST"		"4000"
	}
}
```

Appendix F – Questionnaire

Ter afsluiting van deze gebruikerstest wordt u gevraagd om deze 18 vragen te beantwoorden. Aangezien deze antwoorden gebruikt gaan worden voor eventuele verbeteringen, zou ik het zeer op prijs stellen als u zo volledig mogelijk antwoordt. De gegeven antwoorden worden strikt vertrouwelijk behandeld, maar zouden kunnen worden gebruikt in mijn afstudeerscriptie om een en ander toe te lichten.

Man / Vrouw **Leeftijd**: ...

1. Welke functie bekleedt u?

2. Wat is uw globale indruk van het spel?

3. Heeft u het idee dat u veel geleerd heeft over triage?

4. Zou het spel volgens u kunnen bijdragen aan een betere training voor ambulance personeel? Waarom?

5. Vond u het leuk om dit spel te spelen?

6. Waar stoorde u zich het meest aan tijdens het spelen?

7. **a)** Vond u de besturing van het spel makkelijk?
   b) Wat was voor u het moeilijkste om door te krijgen?

a)

b)

8. **a)** Wat vond u van de representatie van de slachtoffers?
   b) Wat zou hier verbeterd kunnen worden?

a)

b)

9. Kon u de vitale functies (ademhaling, hartslag, etc.) goed vinden bij het slachtoffer?

10. **a)** Wat vond u van de representatie van de vitale functies (geluidsfragment van ademhaling, tekst bij bloeddruk)?
    b) Wat zou hier verbeterd kunnen worden?

a)

b)

11. Waren de menu's tijdens het spelen van het spel (keuze menu naar de andere menu's, triage kaart, classificatie menu, etc.) duidelijk en begrijpelijk?

12. **a)** Wat vond u van het rampgebied in de SIEVE?
    b) Wat zou hier verbeterd kunnen worden?

a)

b)

13. **a)** Wat vond u van de triage tent in de SORT?
    b) Wat zou hier verbeterd kunnen worden?

a)

b)

14. **a)** Heeft u feedback gekregen van het systeem? Zo ja, begreep u wat u fout had gedaan nadat het systeem dit aangaf?
    b) Welke vorm van feedback zou uw voorkeur hebben (tekstueel, via afbeeldingen, geluidsfragementen, etc. maar ook kort, meer uitgebreid, etc.)?

a)

b)

15. Wat vond u van het evaluatierapport aan het eind van het spel?

16. Zou deze methode van training (de serious game) u aanspreken? Waarom?

17. Ziet u mogelijkheden om deze vorm van training (een serious game) op andere gebieden behalve triage toe te passen? Zo ja, welke gebieden?

18. Heeft u nog verdere opmerkingen? Zo ja, welke?

Hartelijk dank voor uw medewerking!!!

Martijn Hendriks, 19-05-2008

Appendix G – Paper

Abbreviations

AI	Artificial Intelligence
NIFV	Nederlands Instituut Fysieke Veiligheid
ITS	Intelligent Tutoring System
START	Simple Triage And Rapid Treatment
MIMMS	Major Incident Medical Management and Support
MCI	Mass Casualty Incident
MCIMS	Mass Casualty Incident Management System
MMT	Mobiel Medisch Team
OvD-G	Officier van Dienst Geneeskundig
HR	Heart Rate
CRT	Capillary Refill Time
RR	Riva-Rocci
ABCD	Airway Breathing Circulation Disability
SIGMA	Snel Inzetbare Groep ter Medische Assistentie
NIBP	Non-Invasive Blood Pressure
GCS	Glasgow Coma Scale
TRTS	Triage Revised Trauma Score
EMV-score	Eyes, Motor, Verbal score
LOTUS	Landelijke Opleiding Tot Uitbeelding van Slachtoffers
FPS	First-Person Shooter
HUD	Heads Up Display
NPC	Non-Playable Character
UML	Unified Modeling Language
SIEVE	Primary triage inside the disaster region
SORT	Secondary triage inside the treatment area

List of Figures

List of Tables

References

Books

[1] Bergeson B. *Developing Serious Games*; Hingham, Massachusetts: Charles River Media; 2006.

[2] Van Den Brink G, Lindsen F & Uffink T. *Leerboek intensive-care-verpleegkunde*; Maarssen: Elsevier Gezondheidszorg; 2003.

[3] Gee JP. *What Video Games have to Teach Us about Learning and Literacy*; New York, New York: Palgrave Macmillan; 2003.

[4] Hartman JAM, Lichtveld RA, De Vries GMJ & Ten Wolde WLM. *Landelijk Protocol Ambulancezorg 6*; Zwolle: Stichting Landelijke Ambulance & Meldkamer Protocollen (LAMP); 2005.

[5] Hartman JAM, Lichtveld RA, De Vries GMJ & Ten Wolde WLM. *Toelichting Landelijk Protocol Ambulancezorg 6*; Zwolle: Stichting Landelijke Ambulance & Meldkamer Protocollen (LAMP); 2005.

[6] Hustinx P, Meeuwis D & Hermans R. *Geneeskundig management bij grootschalige incidenten (Major Incident Medical Management and Support) Studieboek*; Utrecht: de Tijdstroom; 2004.

[7] Hustinx P, Meeuwis D & Hermans R. *Geneeskundig management bij grootschalige incidenten (Major Incident Medical Management and Support) Operationele handleiding*; Utrecht: de Tijdstroom; 2004.

[8] De Lange JJ & Bierens JJLM. *Vital functies onder rampenomstandigheden*; Amsterdam: VU Uitgeverij; 2002.

[9] Michael D & Chen S. *Serious Games, Games that Educate, Train and Inform*; Boston, Massachusetts: Thomson Course Technology PRT; 2006.

[10] Prensky M. *Digital Game-Based Learning*; St. Paul, Minnesota: Paragon House; 2001.

[11] Russell S & Norvig P. *Artificial Intelligence: A Modern Approach*; Upper Saddle River, New Jersey: Pearson Education Inc.; 2003.

Articles

[12] Anderson JR, Conrad FG & Corbett AT. *Skill Acquisition and the LISP Tutor* : http://act-r.psy.cmu.edu/papers/127/SkillAcq.Lisp.Tut.pdf, in Cognitive Science 13, 467-505; 1989; accessed 2008-06-04.

[13] AVD, SAVE, NIvU & Nibra. *Leidraad Operationele Prestaties*, 2001.

[14] BBC News. *Video games 'stimulate learning'*: http://news.bbc.co.uk/1/hi/education/1879019.stm, 2002; accessed 2008-06-04.

[15] BBC News. *Video games 'valid learning tools'*: http://news.bbc.co.uk/2/hi/uk_news/education/730440.stm, 2000; accessed 2008-06-04.

[16] Blow J. *Game Development: Harder Than You Think*: http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=114; accessed 2008-06-04.

[17] Brown E. & Cairns P. *A Grounded Investigation of Game Immersion*: http://portal.acm.org/citation.cfm?doid=986048, in CHI '04 Vienna, Austria 2004; accessed 2008-06-04.

[18] Carney K. *The Ins and Outs of User-Generated Game Content*: http://www.clickz.com/showPage.html?page=3628747, 2008; accessed 2008-06-04.

[19] Carr B. *Wusor II: A Computer Aided Instruction Program With Student Modelling Capabilities*: http://hdl.handle.net/1721.1/6277, in AI Memos (1959 – 2004); 1977; accessed 2008-06-04.

[20] Cheng K. & Cairns P.A. *Behaviour, Realism and Immersion in Games*: http://portal.acm.org/citation.cfm?id=1056894, in CHI '05 Portlan, Oregon, USA 2005; accessed 2008-06-04.

[21] DeMaria Rusel: *Postcard for SGS 2005: Healthcare and Forestry – Half-Life 2: Meet Serious Games Modding*: http://www.gamasutra.com/view/feature/2450/postcard_from_sgs_2005_healthcare_.php, 2005; accessed 2008-06-04.

[22] Entertainment Software Association. *2007 sales, demographic and usage data: Essential Facts about the computer and video game industry*: http://www.theesa.com/facts/pdfs/ESA_EF_2007.pdf, 2007; accessed 2008-06-04.

[23] Goodale G. *In case of emergency, play video game*: http://www.usatoday.com/tech/news/2005-06-06-emergency-games_x.htm, The Christian Science Monitor 2005; accessed 2008-06-04.

[24] De Koning K, Bredeweg B, Breuker J & Wielinga B, *Model-based reasoning about learner behaviour*, in Artificial Intelligence 117 (2000) 173-229; 2000.

[25] Lewis M & Jacobson J. *Game Engines In Scientific Research*: http://citeseer.ist.psu.edu/lewis02game.html, in Communications of the ACM Vol. 45 No. 1 2002; accessed 2008-06-04.

[26] Mac Namee B, Rooney P, Lindstrom P, Ritchie A, Boylan F & Burke G. *Serious Gordon: Using Serious Games To Teach Food Safety in the Kitchen*: http://www.comp.dit.ie/bmacnamee/papers/SeriousGordon.pdf, in Proceedings of the 9th International Conference on Computer Games: AI, Animation, Mobile, Educational & Serious Games CGAMES06 2006; accessed 2008-06-04.

[27] Matsuda N & VanLehn K. *Advanced Geometry Tutor: An intelligent tutor that teaches proof-writing with construction*: http://www.cs.cmu.edu/~mazda/Doc/AIED05/Matsuda05-toAppear.pdf, in Proceedings of the 12th International Conference on Artificial Intelligence in Education; 2005; accessed 2008-06-04.

[28] McGrath D & Hill D. *UnrealTriage: A Game-Based Simulation for Emergency Response*: http://www.ists.dartmouth.edu/library/58.pdf, The Huntsville Simulation Conference 2004; accessed 2008-06-04.

[29] Nikken P. *Computerspellen in het gezin*; Hilversum: Nederlands Instituut voor de Classificatie van Audiovisuele Media (NICAM): http://www.kijkwijzer.nl/upload/download_pc/2.pdf, 2003; accessed 2008-06-04.

[30] Schiesel S. *On Maneuvers With the Army's Game Squad*: http://query.nytimes.com/gst/fullpage.html?res=9C02E2DB133AF934A25751C0A9639C8B63&sec=&spon=, New Tork Times 2005; accessed 2008-06-04.

[31] Susi T, Johannesson M & Backlund P. *Serious Games – An Overview*: http://www.his.se/upload/19354/HS-%20IKI%20-TR-07-001.pdf, 2007; accessed 2008-06-04.

[32] Thompson C. *Saving the World, One Video Game at a Time*: http://www.nytimes.com/2006/07/23/arts/23thom.html, New York Times 2006; accessed 2008-06-04.

[33] Valve. *Source Brochure*: http://source.valvesoftware.com/SourceBrochure.pdf, accessed 2008-06-04.

Websites

[34] *America's Army*: http://www.americasarmy.com/, accessed 2008-06-04.

[35] *Cafébrand Volendam*: http://nl.wikipedia.org/wiki/Caf%C3%A9brand_Volendam, accessed 2008-06-04. **Dutch**

[36] *Counter-Strike: Source*: http://www.steamgames.com/v/index.php?area=game&AppId=240, accessed 2008-06-04.

[37] *Dance Dance Revolution Community*: http://www.ddronlinecommunity.com/, accessed 2008-06-04.

[38] *Dance Dance Revolution Movie*: http://www.youtube.com/watch?v=vybXe9JbJhY, accessed 2008-06-04.

[39] *Darfur is Dying*: http://www.addictingclips.com/Content.aspx?key=20D387810E09B89D&refCode=&brand=ag, accessed 2008-06-04.

[40] *Day of Defeat: Source*: http://www.dayofdefeatmod.com/, accessed 2008-06-04.

[41] *Delta3D – Open source gaming & simulation engine*: http://www.delta3d.org/index.php, accessed 2008-06-04.

[42] *Dimensions in the Source engine*: http://developer.valvesoftware.com/wiki/Unit, accessed 2008-06-04.

[43] *DoomEd*: http://www.desq.co.uk/doomed/index.aspx, accessed 2008-06-04.

[44] *Game Developers Conference*: http://www.gdconf.com/, accessed 2008-06-04.

[45] *Global Conflicts: Palestine*: http://www.globalconflicts.eu/, accessed 2008-06-04.

[46] *Half-Life 2: Riot Act*: http://uoz.multiplayer.it/ra/, accessed 2008-06-04.

[47] *Hazmat: Hotzone*: http://www.etc.cmu.edu/projects/hazmat_2005/index.php, accessed 2008-06-04.

[48] *Hill climbing*: http://en.wikipedia.org/wiki/Hill_climbing, accessed 2008-06-04.

[49] *Hungarian notation*: http://en.wikipedia.org/wiki/Hungarian_notation, accessed 2008-06-04.

[50] *Image Class Library*: http://www.kwasan.kyoto-u.ac.jp/~kamio/archive/image/, accessed 2008-06-04.

[51] *Interlopers*: http://www.interlopers.net/, accessed 2008-06-04.

[52] *LISP Tutor Jr.*: http://reed.cs.depaul.edu/peterwh/Tools/lisptutor.html, accessed 2008-06-04.

[53] *List of game engines*: http://en.wikipedia.org/wiki/List_of_game_engines, accessed 2008-06-04.

[54] *Mass Casualty Incident Management System*: http://www.tsgassociates.co.uk/index.htm, accessed 2008-06-04.

[55] *MINERVA*: http://www.hylobatidae.org/minerva/index.shtml, accessed 2008-06-04.

[56] *Mod DB – Half-Life 2 Tutorials*: http://www.moddb.com/tutorials?filter=t&kw=&subtype=def&meta=def&game=61, accessed 2008-06-04.

[57] *Nederlands Instituut Fysieke Veiligheid*: http://www.nifv.nl/, accessed 2008-06-04. **Dutch**. English version: http://www.nifv.nl/web/show/id=46027.

[58] *Nederlands Triage Systeem*: http://www.nederlandstriagesysteem.nl/, accessed 2008-06-04. **Dutch**

[59] *Newsgaming.com*: http://www.newsgaming.com/index.htm, accessed 2008-06-04.

[60] *NIFV – Virtueel oefenen*: http://www.nifv.nl/web/show/id=117248, accessed 2008-06-03. **Dutch**

[61] *The Orange Box*: http://orange.half-life2.com/, accessed 2008-06-04.

[62] *Portal*: http://www.steampowered.com/v/index.php?area=game&AppId=400&cc=NL&genre=, accessed 2008-06-04.

[63] *QuakeWorld*: http://www.quakeworld.net/, accessed 2008-06-04.

[64] *Quest3D*: http://www.quest3d.com/, accessed 2008-06-04.

[65] *Schipholbrand*: http://nl.wikipedia.org/wiki/Brand_in_cellencomplex_Schiphol, accessed 2008-06-04. **Dutch**

[66] *September 12th*: http://www.newsgaming.com/games/index12.htm, accessed 2008-06-04.

[67] *Serious Games Initiative:* http://www.seriousgames.org/, accessed 2008-06-04.

[68] *Serious Games Summit*: http://www.seriousgamessummit.com/, accessed 2008-06-04.

[69] *Serious Gordon Blog*: http://seriousgordon.blogspot.com/, accessed 2008-06-04.

[70] *Serious Gordon Movie*: http://www.youtube.com/watch?v=zwKY6nbG6gU&feature=related, accessed 2008-06-04.

[71] *The Ship*: http://www.theshiponline.com/, accessed 2008-06-04.

[72] *Sim Ops Studios*: http://www.simopsstudios.com/, accessed 2008-06-04.

[73] *Simple Triage And Rapid Treatment*: http://www.start-triage.com/, accessed 2008-06-04.

[74] *Simulated Annealing*: http://en.wikipedia.org/wiki/Simulated_annealing, accessed 2008-06-04.

[75] *SiN Episodes: Emergence*: http://storefront.steampowered.com/v2/index.php?area=game&AppId=1300, accessed 2008-06-04.

[76] *Source Engine Features*: http://developer.valvesoftware.com/wiki/Source_Engine_Features, accessed 2008-06-04.

[77] *Steam*: http://www.steampowered.com/v/index.php, accessed 2008-06-04.

[78] *Steam Tool Discussions*: http://forums.steampowered.com/forums/forumdisplay.php?f=191, accessed 2008-06-04.

[79] *Torque Game Engine*: http://www.garagegames.com/, accessed 2008-06-04.

[80] *Unity*: http://unity3d.com/unity/index.html, accessed 2008-06-04.

[81] *Unreal Engine*: http://www.unrealtechnology.com/, accessed 2008-06-04.

[82] *Valve*: http://www.valvesoftware.com/, accessed 2008-06-04.
[83] *Valve Developer Community*: http://developer.valvesoftware.com/wiki/Main_Page, accessed 2008-06-04.
[84] *Valve-ERC*: http://www.chatbear.com/board.plm?b=668&v=flatold, accessed 2008-06-4.
[85] *Vampire: the Masquerade: Bloodlines*: http://www.vampirebloodlines.com/, accessed 2008-06-04.
[86] *Vuurwerkramp Enschede*: http://nl.wikipedia.org/wiki/Vuurwerkramp_Enschede, accessed 2008-06-04. **Dutch**
[87] *Wumpus World*: http://www.cogsci.rpi.edu/Otter/Wumpus/, accessed 2008-06-04.
[88] *XML Path Language (XPath):* http://www.w3.org/TR/xpath, accessed 2008-06-04.