TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

MASTERARBEIT

Integrating Electronic Institutions with 3D Virtual Worlds

Ausgeführt am Institut für

Softwaretechnik und Interaktive Systeme
der Technischen Universität Wien

unter der Anleitung von
ao. Univ. Prof. Dr. Dieter Merkl
und
Dipl.–Ing. Dr. Helmut Berger

durch

Ingo Seidel

Wehlistrasse 195/17
A–1020 Wien

Wien, am 19. März 2007

Datum

Unterschrift

Zusammenfassung

Diese Arbeit ist im Zuge eines Forschungsprojektes, welches sich mit der Umsetzung einer 3D e-Tourismus Umgebung beschäftigt, entstanden. Das Ziel dieses Projekts ist die Entwicklung eines Instruments, um die komplexen Interaktionsmuster von Anbietern und Konsumenten im e-Tourismus zu unterstützen. Diese Anbieter und Konsumenten sind TeilnehmerInnen in einer heterogenen Gesellschaft von Menschen und Software-Agenten, die gemeinschaftlich in einer 3D Umgebung basierend auf einem Multi Agenten System zusammenleben. Im konkreten besteht das System aus drei Ebenen: einer 3D Visualisierungs-, einer Middleware- und einer Multi Agenten System Ebene. Diese Arbeit beschäftigt sich mit der praktischen Umsetzung eines solchen Systems. Der Fokus ist dabei auf das Design und die Erstellung der Middleware Komponente, sowie auf die Visualisierung in der 3D Welt gelegt. Die Middleware verbindet das Multi Agenten System mit der 3D Welt und leitet Nachrichten in beiden Richtungen weiter. Sie garantiert damit ein konsistentes Verhältnis zwischen diesen beiden Komponenten, indem jeder Zustandswechsel in der einen Komponente in der anderen Komponente propagiert wird. Die 3D Welt dient einerseits als User Interface für Benutzer und wird andererseits für die Visualisierung von Softwareagenten verwendet. Die Steuerung dieser Agenten, sowie die Interaktionsmechanismen mit dem Benutzer werden ebenfalls in dieser Arbeit vorgestellt.

Abstract

This master thesis is embedded within a research project that has the principal goal of developing an instrument to support the complex interaction patterns of providers and consumers in an e-Tourism setting. In particular, these providers and consumers, either humans or software agents, are members of a heterogeneous society cohabiting in a multi-agent based 3D virtual environment. Conceptually speaking, the environment is designed according to a three-layered architecture comprising a Multi Agent System layer, a middleware layer and a 3D visualization layer. The major contribution of this master thesis lies in the design and implementation of the middleware connecting the two other layers. The middleware mediates the communication between the Multi Agent System layer and the 3D visualization layer and guarantees a consistent relationship between these components. The 3D virtual world serves as user interface for human users and is used to visualize the actions of agents in the Multi Agent System. Thus, it becomes possible for users to interact with software agents in an immersive 3D environment. We utilize a game engine for the creation of the 3D virtual world. The framework specific functionalities of the 3D virtual world such as agent control or user interaction are also established in this thesis.

Contents

1 Introduction

Tourism is an important economic sector of each country and is the leading market in business to costumer (B2C) commerce (Werthner and Ricci, 2004). The World Travel & Tourism Council (WTTC) publishes forecasts for the development of the tourism sector and according to their forecast for 2007, 10.4% of the worldwide gross domestic product (GDP) is obtained in the tourism domain. In Austria the figures are above this value with tourism contributing 16.7% to the GDP. This illustrates the importance of tourism for the Austrian economy.

An important aspect for tourism is the Internet and its online booking systems. 30 years ago Airline companies started to use reservation systems to control the booking of airplanes. In the USA, for example, there are four major reservation systems and in the year 2000, 98% of all airline reservations where booked using these systems. Travel agents, tour operators, hotels or car rentals use such reservation systems as well (Gratzer et al., 2004). In the same work it is pointed out that the Internet is going to change tourism business. Online markets provide new opportunities and threats for tourism providers. They evaluated those opportunities for several tourism players by conducting an expert survey. Some experts had to estimate the value of the Internet for tourism providers on a scale from 1 to 7 (where 1 represents strong opportunities and 7 represents strong threat). The overall score of 2.39 demonstrates the great importance of the Internet in this sector. Airlines, IT&T companies, hotel chains and hotels with more than 50 beds had the best scores, whereas travel agents and tour operates did not profit that much from these new technologies (with values of 4.7 and 3.6 respectively).

So far we have only been highlighting the importance of the Internet for the suppliers. The individual tourist is also effected by this new trend. Over the last years a vast number of online booking platforms have emerged and millions of tourists are using these services every day (UsageStats). A survey which compared online booking with booking at traditional travel agents gives insights on the booking behavior of tourists (Bogdanovych et al., 2006). The survey showed that people are using online booking services preferably to book domestic trips whereas international trips are still more likely to be booked with a travel agent. Note that these results were obtained in Australia and may not apply to each country. The main advantages of travel agents are their expertise, the social interaction when making difficult decisions and the help when making impulse decisions. Also people have more trust in travel agents than in Web sites, but those figures may change in the future. Interestingly, tourists also distrust unknown travel agents as they have a fear of being pushed towards more expensive products. In contrast online booking systems are more convenient, have lower response times and provide answers to inquiries in an environment familiar to users.

Besides the actual booking process, the Internet is increasingly used by customers as a source of valuable information. A tourism product has some special properties that require a good knowledge of the product prior to purchase. Usually it is not possible for the tourist to experience the product in advance - it is a confidence product (Gratzer et al., 2004). Traditionally the impression of a destination is based on high quality photos in travel catalogs and information from the travel agent. In addition, information gathering on the Internet has become more and more important in the last years. There is an ever increasing number of Web sites where offers can be compared, there are forums, newsgroups and Wikis where all kinds of topics are discussed and there are blogs where travelers report on their experiences. The interested tourist can get a very good picture of his travel destination up front and even specific concerns can be discussed and cleared in discussion forums. When relying on the Internet as information source, the quality of this information is a major issue.

To get insights on the quality of information posted on forums, travel blogs, etc., Schwabe and Prestipino (2005) compared the information quality of such online tourism communities against traditional guidebooks. They investigated the information quality of virtual communities from the viewpoint of the members, by asking travel related questions and assessing the quality of the answer. They compared a traditional guide book (Rough Guide) with an online forum by asking 18 questions and the answers where evaluated by four judges. The quality factors were *timeliness*, *completeness*, *structure* and *personalization*. The authors analyzed which medium has the best performance for each factor. They found out that online communities have more timely information since new content can be added at any time. In general they have an update cycle of a few weeks - the time that passes between the visit of the location and the return. In contrast, a guide book has an update cycle of several months or even years. The quantity of information that can be stored electronically in a forum is higher than in a guide book. Thus, information requests can be answered more completely. Concerning the structure, the guide book performs better because the outline of a book is better structured than the threads in an information forum or the entries in a blog. Another important aspect of online communities is, that a person can ask any question she wants to be clarified and, therefore, personal requests and personalization can be better served in online communities. This leads to the conclusion that information provided on the Internet can be of high quality and can even be superior to traditional information sources.

The next step after gathering information on the different travel locations is to book the trip. There are lots of online booking platforms on the Internet. Most offer a conservative user interface where customers need to specify preferences via drop down lists, option buttons or selection lists. Such interfaces neglect the social needs of customers (Preece and Maloney-Krichmar, 2003).

To overcome these disadvantages, new interface metaphors have been developed. There are a number of different approaches in this area covering a wide variety of new interaction mechanisms. One such approach that, however, relies on a traditional interface element, a textbox, tries to address this issue by means of natural language interaction. Berger et al. (2004) developed a system where users can formulate a query in natural language. The advantages of this approach are the same as with online discussion forums - users can express more complex queries when using their natural language and will get more appropriate results for their search. Furthermore, they do not have to learn a special query language or to structure their request in a way the computer understands.

A field trial was carried out to test the system. Within a period of 10 days 1333 queries where posted. The first interesting aspect hereby is that the average query length was much higher than in traditional search engines. This shows that users post more complex and detailed queries when they are able to use the natural language. The next result concerns the complexity of sentence constructs. Although users are not limited in the complexity of their requests, most of the queries were simple in reference to language constructs. The acceptance of the interface was evaluated with a usability study. Most of the participants considered the language interface more comfortable in contrast to standard interfaces. They explicitly stated their preference for this kind of interface and they were more satisfied with the obtained results.

Another approach is to employ virtual 3D worlds as user interfaces. They resemble the natural environment, allowing users to quickly become acquainted with the interface metaphors. Users navigate in the same manner as they do in the real world. They walk around with their avatar (Damer, 1997), enter a house through a door and are restricted in their movement by the shape of the world. Furthermore, social interaction is implicitly addressed in virtual 3D worlds. Users see each other in the world, can walk towards other users and can start talking with them.

It is especially interesting how such virtual 3D places evolved (Castronova, 2005). In contrast to other areas their advancement was driven by th commercial market than by research labs. Research mainly concentrated on virtual reality - researchers tried to rebuild the real world as accurate as possible. The user should not be able to differentiate the simulated environment from the real world. This required tremendous amounts of computing power and special hardware equipment. The user is equipped with goggles displaying the environment onto her complete field of vision. Wired gloves are used to transmit the user's motions into the virtual space. While the research concentrated on developing these sensory input devices, the commercial market approached the problem from another direction. The focus was laid on the software rather than the hardware and on the community rather than the individual. The first applications in this domain were 3D computer games. Although the computer graphics were simple and could not resemble the real world, users were immersed through the content. In the 1980s and 1990s 3D computer games began to attract millions of people - the 3D shooter Doom, released in 1993, was sold 1 million times and was estimated to have been installed on more than 10 million computers (DoomStats). This success continues until now. Many 3D games have become popular selling millions of copies. Some examples are Half Life (HL), Quake (Quake) or Myst (Myst).

Similar to the tourism sector, the Internet had a big influence on 3D computer games. In the 1990s online 3D virtual worlds such as The Realm Online (RO) or Ultima Online (UO) were born. Castronova (2005) investigates the economic and social impacts of online 3D games. Based on a conservative view he estimates the number of people who regularly spend time in online 3D games to be about 10 million. His research efforts are focused on the economic markets in such online 3D games. He shows that a value-and-demand market, based on item trade, exists and estimates the annual trade to be US$ 1 Billion. Although Castronova concentrates on 3D games, there are non game worlds as well. One such world that became extremely popular at the beginning of 2007 was Second Life (SL). In contrast to online 3D games, the content in Second Life is user generated. There exists no story line or goal like in traditional games. Users entertain themselves by means of content creation or simply socialize with others. This approach is widely accepted as there are about 2 million users in Second Life as of March 2007 (SLUsage).

Summarizing the above, a list of facts that motivate this research project is obtained:

- Tourism is an important economic factor throughout the whole world.
- Information technology and the Internet are of great importance in the tourism domain.
- Online communities are used to exchange up-to-date and personalized travel related information.
- New interface metaphors are needed and accepted by the users.
- 3D worlds are popular and provide familiar and intuitive environments for users.
- Online 3D worlds are virtual market places.

The overarching goal of the project "A 3D e-Tourism environment" is to provide a platform for tourism providers and consumers that supports the complex interaction patterns between those parties. It is anticipated to achieve this goal by providing an instrument to foster the development of a sustainable tourism community in an online 3D virtual world. Users should feel comfortable in this environment, should be able to socialize and should have access to a wide variety of information. Tourism providers should be able to present their products to many people interested in tourism. This materializes in three sub goals:

- Provide a 3D e-Tourism environment for providers and consumers that enables versatile interaction between participants including the trade of tourism products.

- Provide a 3D e-Tourism environment that becomes a community facilitator to create and establish a lively and sustainable community involving both, providers and consumers.
- Provide a 3D e-Tourism environment that is information-rich and multimedia-based to offer transparent and unified access to disparate information sources.

To achieve this, the application of a 3D game engine for the creation of an immersive 3D virtual world is proposed. To enrich the environment with information and to allow versatile interaction between participants, agent technology will be used. According to Woolridge (2001), agent based solutions should be employed whenever the information is spread across several sources. This applies to the tourism sector since information is spread over the Internet and stored in databases on different organizational levels (regional, national, international). Agents and humans are the participants in the 3D virtual world. They work together to cooperatively achieve their goals.

The first major contribution of this master thesis lies in the design of the middleware connecting the 3D game engine with the Multi Agent System. This includes considerations regarding the design of the middleware, protocol definitions, as well as the conceptualization of the middleware architecture. The second major contribution is the actual implementation of the middleware based on the architecture presented herein. In order to test the functionality of the system and, thus, to verify the smooth interplay of the components, a prototypical 3D visualization has been designed and implemented.

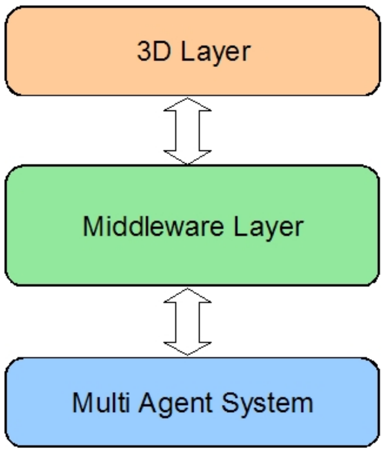


Figure 1: Architecture.

In general, the architecture of the system consists of three layers (cf. Figure 1). The bottom layer contains a Multi Agent System controlling the interactions between software agents. The top layer contains the 3D virtual world used to visualize the actions of agents in the Multi Agent System and to provide an interface for human users. The middleware connects the bottom layer with the top layer. This connection materializes in two directions. Events happening in the Multi Agent System are represented in the

3D virtual world. Such events include the visualization of agents as avatars, the movement of these avatars and the communication between users and agent avatars. The 3D virtual world displays a view of the Multi Agent System. Users, participating in the 3D virtual world, must act according to the rules of the Multi Agent System. Therefore the direction, from the 3D virtual world to the Multi Agent System, is used to verify all user actions with the Multi Agent System. The middleware forwards events and actions in both directions and guarantees a consistent relationship between the two systems.

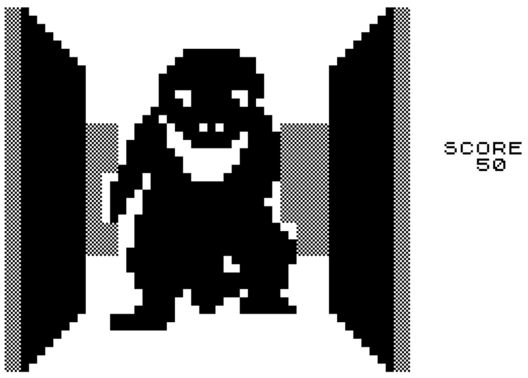
The remainder of this thesis is organized as follows. In Chapter 2 we introduce works that are related to our framework. Starting with 3D virtual worlds we move on to Multi Agent Systems and applications in this area and finally describe research efforts that aim at connecting 3D virtual worlds with Multi Agent Systems. Then in Chapter 3 the project setting and background are presented. This is followed by an introduction of Electronic Institutions that are employed as the Multi Agent System and the Torque Game Engine which is used for the visualization. Chapter 4 gives insights on the architecture of the middleware that connects the Multi Agent System with the 3D virtual world. The message protocols are introduced and the relationship between the Multi Agent System and the 3D virtual world is highlighted. The implementation of the Middleware is presented in Chapter 5. The interplay of the components is illustrated by means of a prototypical showcase in Chapter 6. Finally, in Chapter 7, we summarize the conducted work and provide an overview of future work.

2 Related Works

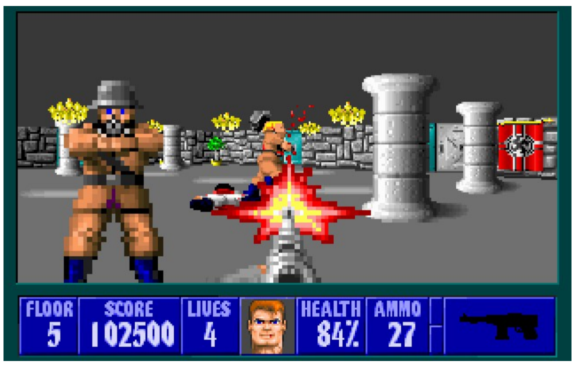
2.1 Short History Of 3D Computer Games

The earliest sources for 3D computer games date back to the year 1974, when a space simulation called Spasim was created (Spasim). It was developed for an educational network named PLATO and included network support. It could be played by up to 32 players simultaneously. Although it is not certain that Spasim was the first 3D game ever, it is most certainly the first 3D multi player game. The author of Spasim, Jim Bowery, offers a reward of 500 US$ for any documentation of an earlier 3D game. Since Spasim was developed for a particular domain, only a few people had access to this game.

A game having had high impact on computer games was the maze game 3D Monster Maze (3DMaze). It was released in 1981 on the ZX81 and the home computer and reached more people than Spasim. The game is played from the first person perspective and the objective is to find the exit of a maze without getting eaten by a Tyrannosaurus Rex, see Figure 2.



(a) 3D Monster Maze.

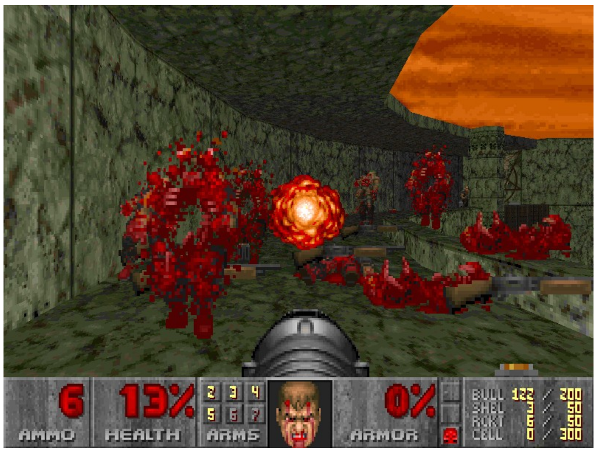


(b) Wolfenstein 3D.

Figure 2: First examples of 3D computer games.

Another 11 years later the next landmark in 3D computer gaming was reached. Wolfenstein 3D was released by id Software in 1992 (WS3D). In Wolfenstein 3D the player is an American soldier in World War Two, trying to escape from a Nazi stronghold (Figure 2). The player was able to move freely in the game world but was restricted to one axis (left/right movement, the player could not look up and down). Wolfenstein 3D was the first game that used textures and all objects were drawn with the *billboarding* technique. A billboard is a 2D image that always faces the player, giving the impression of acting in a 3 dimensional environment. These innovations, despite its plot, made the game visually appealing and contributed to the great success of Wolfenstein 3D.

One year later, in 1993, id Software published the game Doom (Doom) which enabled the player to look in any direction, since rotation around a second axis (up/down) was introduced. Moreover, the resolution of all objects was increased, floor and ceiling were textured and the concept of height was established, i.e. players were able to jump on platforms or to fall into death pits. This time the player is a space marine and has to fight against zombies and creatures from hell which came into existence due to a failed teleportation experiment, see Figure 3. Doom was a great success and was downloaded about 10 million times (DoomStats). Another feature which made Doom the reference product for the next several years was its network functionality. It was the first game that introduced multiplayer features such as deathmatch and connected thousands of people all around the world.



(a) Doom.

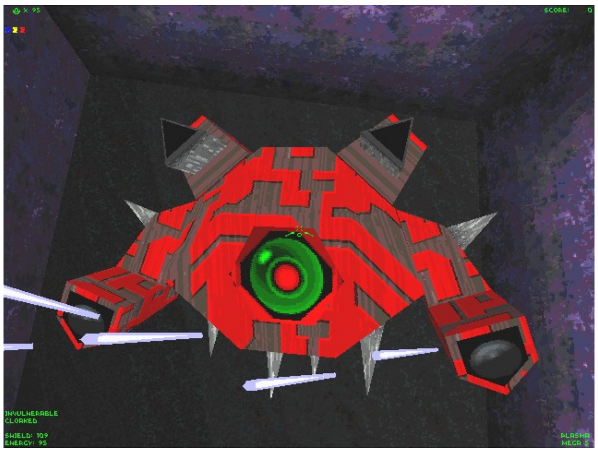

(b) Descent.

Figure 3: 3D computer games in the 1990s.

In the following years all newly released games were compared against Doom. The term “doom clone” was coined in newsgroups as a synonym for 3D games until the term “first person shooter” was established. Many games were released in the 1990s that became popular and introduced new features and new ideas. One of the most outstanding among them was Descent (Descent). In this game the player controls a space ship allowing movement around all three axes (Figure 3). Furthermore, the game provides a fully 3D polygonal graphics engine. Due to its steep learning curve, Descent became not as popular as other games, but contributed to the technical advancement of 3D games.

2.2 Game Engines Used As 3D Worlds

One of the first examples of a game engine being used for a serious task was the work of Chao (2001). He used the Doom engine (its source code was released in 1997) as an interface for process management in a Unix system. Chao’s intention was to explore new interface metaphors that provide a more intuitive access to computers for non-technical people. The desktop metaphor, for example, stems from the times where computers

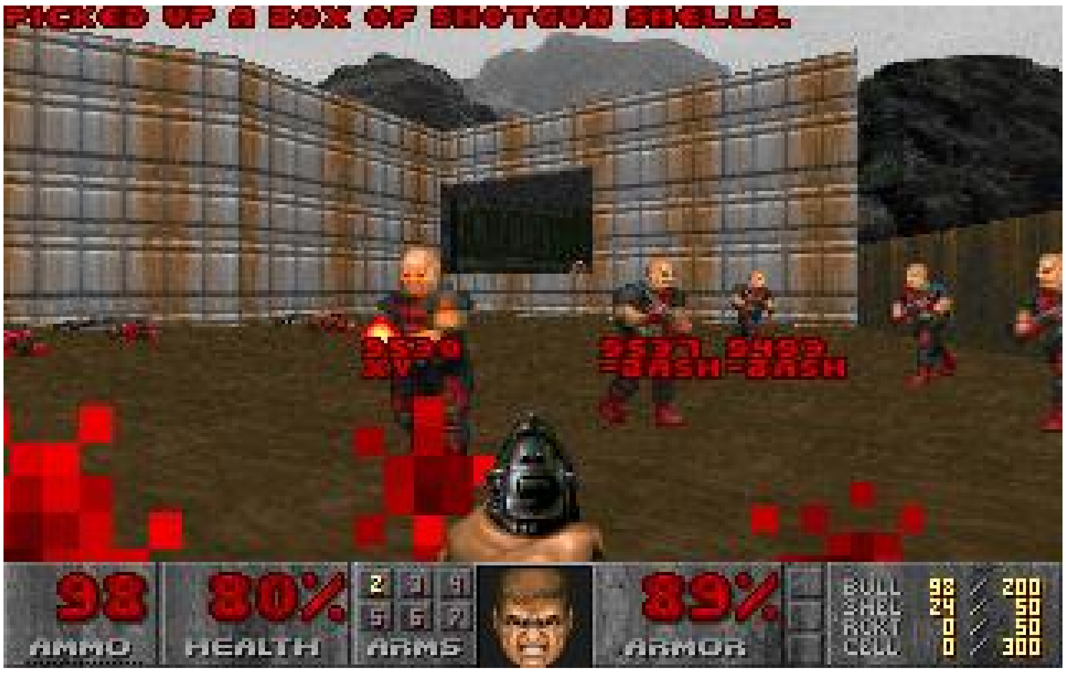


Figure 4: PSDoom. Every "enemy" process has its id and name displayed in front of him (Chao, 2001).

where mainly used by business people to do business work. In this context the desktop metaphor worked well because those people spent most of their time at desks and were familiar with this environment. Other people such as children or non office workers need other metaphors that make the interfaces more appropriate for them. Chao suggests to use the computer world's and popular culture vocabulary to access ordinary people more easily. Especially children are now growing up with computer games and are used to this type of interface. Another advantage of such interfaces is their playful character. Instead of having forms with buttons, game interfaces are simply more appealing and make work more fun.

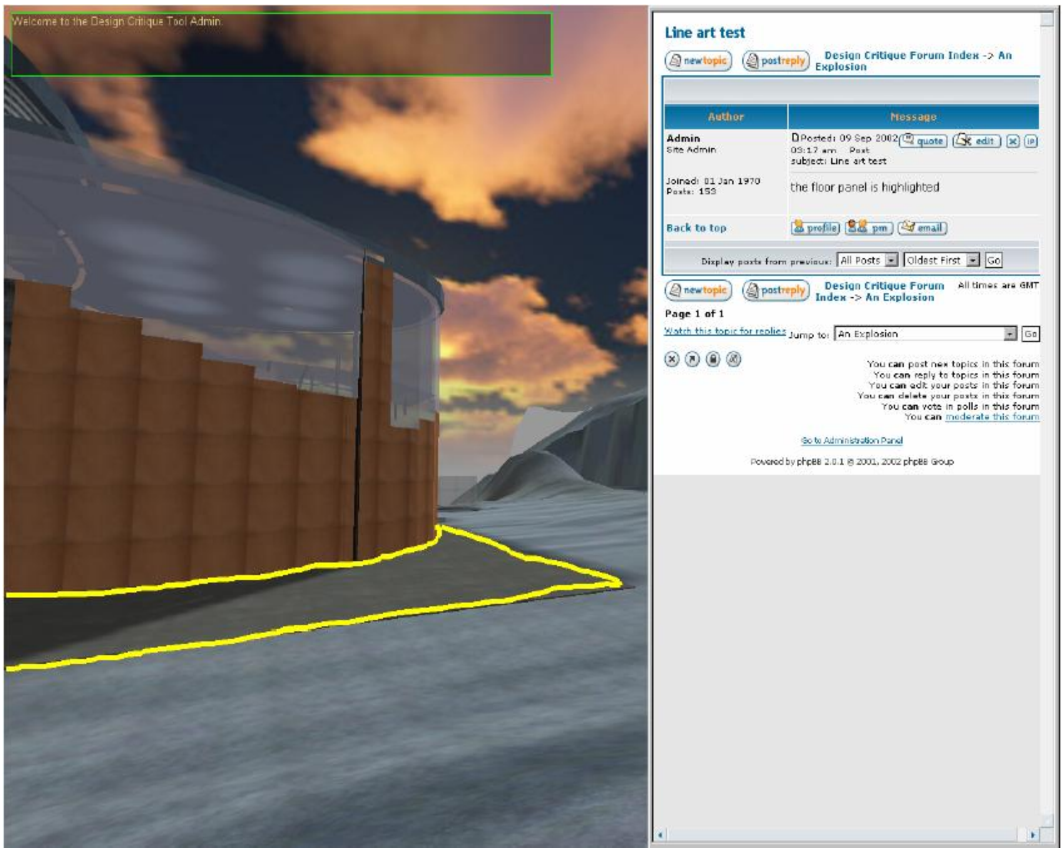
The program, Chao developed, is called PSDoom and works as follows. Each process currently running on the system is represented as a monster in a dungeon. The system administrator may enter the dungeon and start shooting at processes. This is depicted in Figure 4. When a monster is hit, the priority of this process is lowered. If the monster is killed its corresponding process is killed respectively. This may seem violent but reflects the common terminology in Unix process management where "daemons" (processes running in the background) are "killed" (shut down) with the `kill` command on the command line. Another reason to use Doom was the fact that it was a popular game among system administrators at that time.

The benefits of PSDoom can be summarized as follows: The system load is expressed by the number of monsters in the room. The user is not omnipotent anymore, it takes time to kill all processes because every single monster has to be defeated. The power of the user can be regulated by giving him different weapons. Otherwise the program is also very unpredictable because monsters can attack each other. This can be seen as advantage because crowded systems would regulate themselves. In the original version, however, each process had the same power. Thus, important processes could be killed resulting in system failure. Furthermore, processes were not aware of self-preservation and kept attacking until all enemies were eliminated. Chao suggested improvements by giving important processes more power or let them only attack others when needed. Apparently those enhancements have never been implemented, because there is no information on this issue available on the Web page (PSDoom). The most recent version of the software dates back to the year 2000.

Using open source game engines to realize virtual 3D environments has another major advantage - they are affordable compared to commercial game engines. Today, a vast selection of different game engines for every budget exists. There are open source engines that are entirely free, there are some engines that cost moderate amounts of money and there are high-end game engines which are hardly affordable. These costs were one of the driving motives for our next example, a design critique tool for architects (Moloney et al., 2003).

In the field of architecture, computer aided design (CAD) software is mainly used for designing and building architectural models. Such software systems offer accurate and precise representations of 3D models but are not designed for model presentation or quick alteration of the model. This is the point where Collaborative Virtual Environments (CVE) come into play. According to the authors, the two major advantages of CVE systems over CAD systems are the support of an iterative working approach and a better instrument for design critique. Such systems are iterative in a sense that students can change their models and directly perceive these changes in a real context. In the process of reviewing a model, reviewers can experience the model from any desired angle and can, therefore, get a better impression of the work. Game engines are perfectly suited for these tasks since these systems offer multi-player support (multiple reviewers), realistic and real-time rendering (good impression of the model) and network support (different geographical locations of reviewers and students). The authors have chosen the Torque game engine from GarageGames because of its low cost and the great freedom of development.

Next we will discuss this program and its features in more detail. Architectural models are presented to reviewers by means of creating pre-recorded tours or by performing live tours. Pre-recorded tours can be compiled by the designer and may then be watched by the reviewers at any point in time. The functionality of creating demos is an implicit feature of the Torque game engine. Pragmatically speaking, live tours resemble the core functionality of a game engine. Another feature is the possibility to comment the model. Reviewers can place a comment at any desired spot, can mark areas and can record the angle and position when they were writing the comment. This makes it easier for the designer and other reviewers to understand the critique and they can react to it more accurate. The comments are stored in a forum structure which can also be accessed outside the 3D world and may further encourage discussion. Other functionalities that are incorporated in the program are a sun placement tool (for precise lighting conditions) and functions to bundle, select and download designs. A screenshot of the system is depicted in Figure 5.



Line art test

Design Critique Forum Index -> An Explosion

Author	Message
Admin Site Admin Joined: 01 Jan 1970 Posts: 153	Posted: 09 Sep 2002 03:17 am Post subject: Line art test the floor panel is highlighted
Back to top	

Display posts from previous: All Posts Oldest First

Design Critique Forum Index -> An Explosion All times are GMT

Page 1 of 1

Watch this topic for replies Jump to: An Explosion

You can post new topics in this forum
You can reply to topics in this forum
You can edit your posts in this forum
You can delete your posts in this forum
You can vote in polls in this forum
You can moderate this forum

Go to Administration Panel

Powered by phpBB 2.0.1 © 2001, 2002 phpBB Group

Figure 5: The Design Critique tool in action. On the left you can see a marked area, on the right is the discussion forum (Moloney et al., 2003).

Another example for the use of game engines as working environments is the work of Kot et al. (2005). Their motivation is to use a 3D game engine for improved data visualization in the context of source code comprehension. They provide a tool that allows developers to quickly perceive the structure of source code and the relation between different source code files. To this end, source code files are visualized as 3D objects in a 3D virtual world. The size of the object is determined by the size of the source code file, i.e. the more lines of code the larger the object. To identify different files, the file names are displayed above the visualized objects. There is no restriction on the placement of the file object in the 3D world.

The 3D environment contains an accumulation of 3D file objects that can be explored by the user. The user is able to pick these objects up and carry them around. If the user walks into an object, the view changes to a 2D editor displaying the content of the file. Invocations of functions and variables are displayed as hyperlinks in this editor. If a user clicks on one of these links, the editor scrolls to the definition of the symbol. If the symbol is defined in a different file, the editor is closed and the user gets slid towards the file in question. Each file the user recently opened is saved in a history. Similar to the navigation in a Web browser this history can be traversed back and forth.

The intended audience of this program are unexperienced developers and employees at a company that need to be introduced to the code base of a project. A member of the team will then give a guided tour that can be attended by the newcomers. For this reason a user can synchronize his view with the view of another person. Furthermore, the guide has the possibility to mark certain areas in the code. This usage scenario defines the most important properties that the game engine needs to offer. This includes multiuser support, stability and reliability. The authors have chosen the Quake 3 game engine because it is well-tested, provides the required functionality and offers the possibility to extend this functionality. In contrast to the Torque engine, the source code of the Quake 3 engine was not publicly available for the implementation. This led to problems related to debugging and the message protocol used by the engine could not be altered.

They evaluated the tool with voluntary participants in the course of the development. This evaluation identified some problems like overlapping labels or users dropping files onto other users but gives no information on the performance of the program in a real world setting.

2.3 Agents and Multi Agent Systems

Several definitions of software agents and Multi Agent Systems exist. However, some principles and properties found wide acceptance in the research community. According to Woolridge (2001) "an agent is a software component that is *situated* in some environment and is capable of *autonomous action* in this environment in order to meet its design objectives". Multi Agent Systems, abbreviated as MAS, provide an infrastructure for multiple agents. One such MAS system, called RETSINA (Reusable Environment for Task Structured Intelligent Network Agents) was developed by Sycara et al. (2003). In the same paper the authors also propose a general MAS structure. We will use their definition to present the general components of a Multi Agent System. Another, more comprehensive overview of the needs and definitions of MAS systems can be found in Gasser (2001).

2.3.1 Overview of MAS components

Figure 6 shows the MAS infrastructure as proposed in Sycara et al. (2003). The infrastructure is based on a hierarchy where the upper layers depend on the functionalities implemented in the lower layers. In order to allow heterogeneous agents to participate in the MAS, the infrastructure does not make any assumptions on the problem solving be-

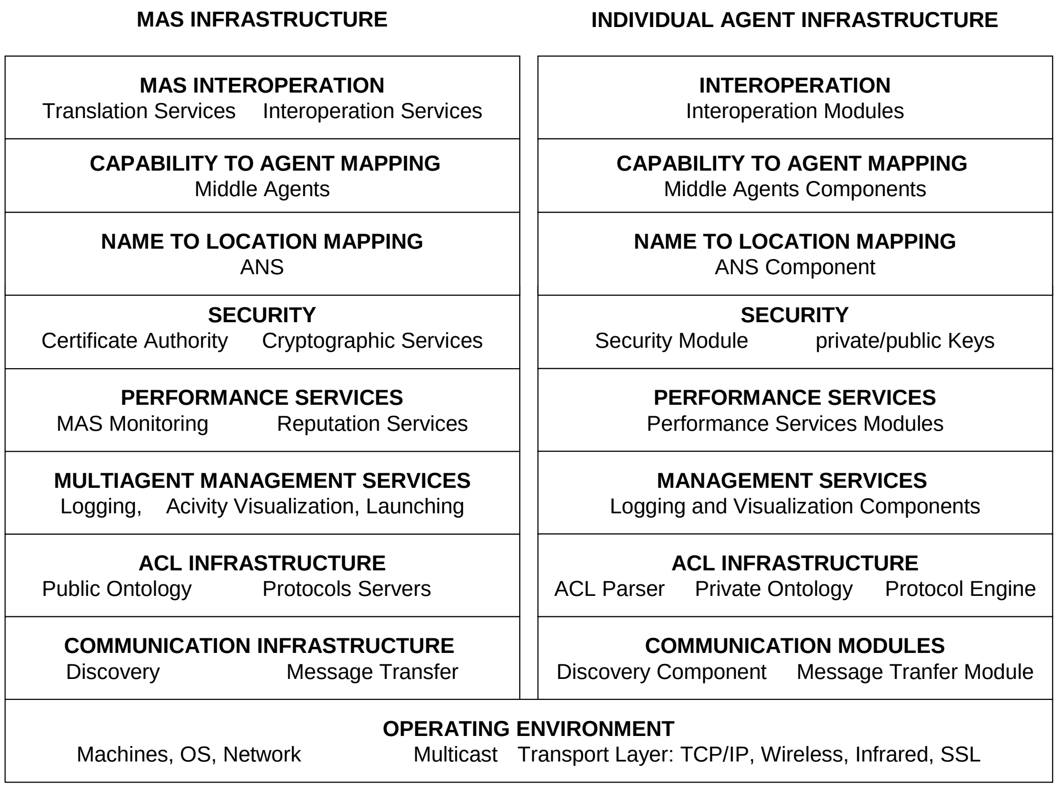


Figure 6: MAS components as proposed in Sycara et al. (2003)

havior of the agent. However, agents participating in the MAS must implement functions for the interaction with the MAS services. This is illustrated on the right side of Figure 6.

The bottom layer constitutes the physical operating environment. Both models, MAS and agent, may run on the same host. So, this layer stretches over both infrastructures. The *Communication Infrastructure* provides services for the message transportation between agents. This layer defines supported transportation media (wireless, wired) and is provided by both infrastructures separately. In order to enable agent communication, a common language has to be chosen. The *ACL* (Agent Communication Languages) layer describes the syntactic form and the semantic interpretation of messages. Agents participating in the system need to be able to compose and process such messages. The *Multiagent Management Services* are used to observe and record the execution of the system. Services that help to configure and start the infrastructure are also contained at that level. The *Performance Service* layer provides facilities to measure the performance of individual agents. This measurement is used to judge the reliability and integrity of agents. *Security Services* are needed to provide trust in the heterogeneous society of agents. Those services include authentication and certification mechanisms. Agents in the MAS must be able to locate each other. Such services are provided on the *Location Mapping* layer. Naming services abstract the physical location of agents enabling agents to communicate seamless across machine and network boundaries. The next layer, *Capability Mapping*, also provides facilities to locate agents. On this level, however, agents are located by their functionalities or capabilities rather than on their location. The last layer, *Interoperation*, is used for the interaction with other MAS architectures.

2.3.2 MAS Examples

The first example is situated in the domain of aircraft maintenance. The process of maintaining and repairing aircrafts is a complicated task. In the U.S. Army this process follows a standard procedure, covering expert consulting and information gathering. When a mechanic detects a discrepancy at an airplane, he must consult an engineer to decide the required repair procedure. The mechanic fills in a 202a form and may attach additional graphics and sketches that help to illustrate the problem. The engineer receives the form and suggests a repair procedure based on several information sources. He may consult historical repair data, manuals or experts at another air base. Most of this data is not available in electronic form and can thus not be searched easily. Remote experts have to be contacted via voice mail or fax. Their answers are usually delayed for hours or days, prolonging the repair process even more. After the engineer has decided which repair procedure to follow, he fills in a 202b form. This form is passed on to the mechanic who carries out the repair.

The work of Shehory et al. (1999) aims to overcome the problems in this procedure with Multi Agent Systems. Agents are used to assist engineers in the task of information retrieval. They use the RETSINA infrastructure and propose three different agent types to solve the problem:

- The *form agent* analyzes the 202a form received from the mechanic and characterizes the problem. It searches for relevant information and presents the filtered results in a comprehensive manner to the engineer.
- The *history agent* receives information requests and searches the historical data for problem relevant information.
- The *manual agent* is similar to the history agent. Upon request it searches the manuals for problem relevant data.

The organization of the Multi Agent System is depicted in Figure 7. Mechanics are equipped with a wearable computer and when a problem is encountered, they fill in an electronic 202a form. They may further attach audio or video content to illustrate the problem. The form is forwarded to a form agent, whereas multiple mechanics may access the same form agent. The form agent queries history and manual agents at different service centers. Those agents have access to historical information, stored as electronic 202 forms, and manual information, stored in databases. The replied information is merged by the form agent and presented to the engineer. Based on this information the engineer decides upon the repair procedure and fills in an electronic 202b form which is then sent to the wearable computer of the mechanic.

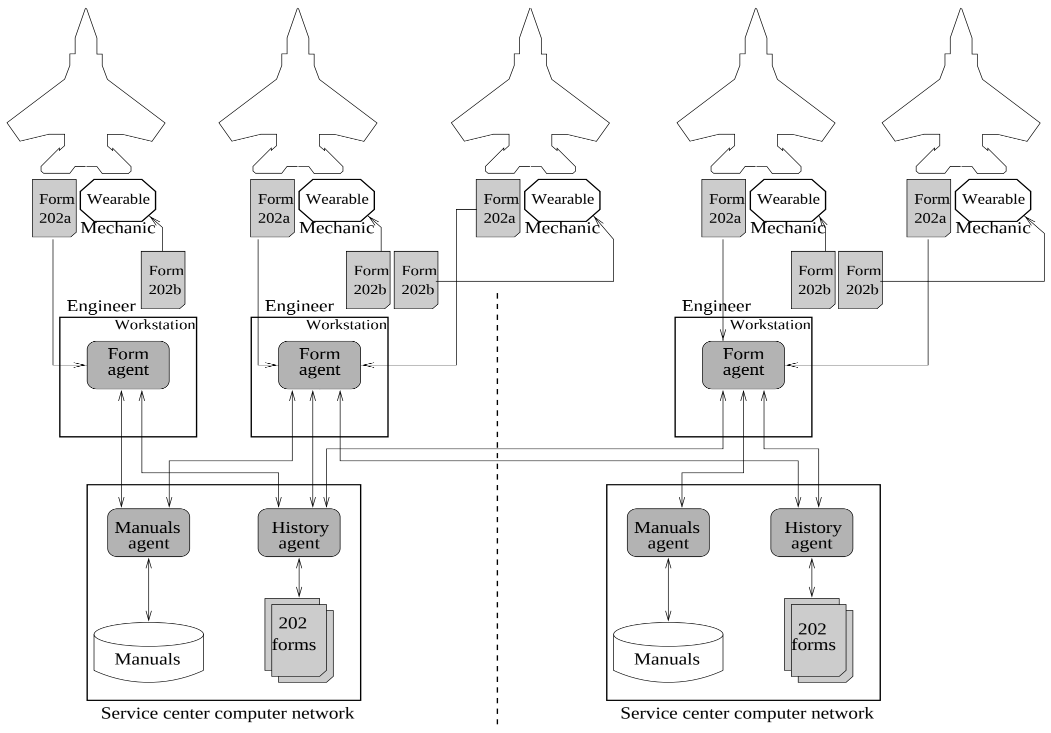


Figure 7: Organization of the MAS for Aircraft Maintenance

This system provides advantages such as automated retrieval of relevant information. Historical forms and manuals are stored electronically making an index based search possible. This increases the search efficiency and the retrieved information is more accurate and complete. The average repair time is reduced, due to the better organization and structure of the data.

The second example deals with coalition creation among customers when purchasing products. A coalition is a group of customers interested in buying the same product. Depending on the number of customers purchasing this particular product, a supplier will usually grant a discount. A supplier has an incentive to sell his products at wholesale, because above some quantity his utility will be larger than selling the same amount of products at retail. A customer also has an incentive to buy his products in a coalition, because he will get the product at a lower price. However, the formation and management of the coalition requires additional expenses. A coalition is, therefore, only viable if the utility of buying wholesale is greater than the expenses of forming a coalition. When forming coalitions two different protocol types can be distinguished: pre negotiation and post negotiation. In a pre negotiation protocol, the number of participants is determined beforehand and the coalition leader collects the money of all customers. The leader then negotiates a contract with the supplier and distributes the products to the customers. In a post negotiation protocol, the coalition leader estimates the size of interested customers and buys the products from the supplier at wholesale. The leader then advertises the products and customers can join the coalition. The products are then sold to the

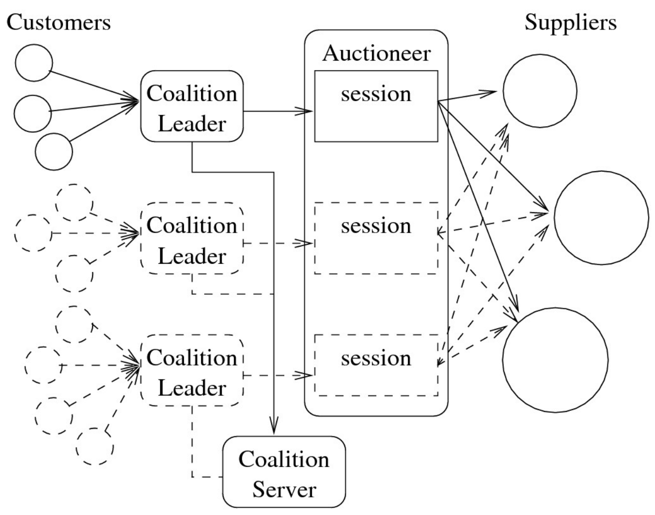


Figure 8: MAS structure of the coalition example (Tsvetovat and Sycara, 2000).

customers at the advertised price.

Tsvetovat and Sycara (2000) use an agent system to implement such coalition formations in the domain of book purchasing. An overview of the system can be seen in Figure 8. It consists of coalition leader agents, an auctioneer agent, a coalition server and supplier agents. The coalition server is used by the coalition leaders to advertise their coalitions. Customers can search and join the available coalitions through a Web based interface. They implemented a pre negotiation protocol which works as follows. The coalition leader specifies the book to be bought and submits a request for bids to the auctioneer agent. The auctioneer informs the supplier agents of the request. The supplier agents query information agents and compute a bid price for different quantities of the item in question. The bid is then submitted to the auctioneer agent who collects all bids that are submitted within a specified time period. After the auction time has expired, the auctioneer reveals the bids to the coalition leader. The coalition leader then chooses one supplier and informs the winner of the tentative accept. The leader opens the coalition for a specified amount of time and new members can join the coalition within this interval. After the time period has finished, the final price of the product is determined (depending on the number of participants). The coalition leader notifies the supplier of the order size and provides a delivery date. The supplier then executes the transaction.

2.4 Integrating Agents in 3D Virtual Worlds

In the following Section we describe approaches that aim at combining agents with 3D virtual worlds. We start off with an approach where much of the agent logic is incorporated in the 3D environment and which has been developed by Smith et al. (2003). The main motivation of their work was the introduction of dynamic behavior into 3D virtual worlds. According to the authors, most worlds are largely static and objects are used to trigger pre-programmed behavior. Agents are supposed to enrich the world and act according to their goals and the current state of the world. The framework consists of a society of agents. As illustrated in Figure 9, each agent can perceive the world through sensors and is able to change the world or to issue messages through effectors.

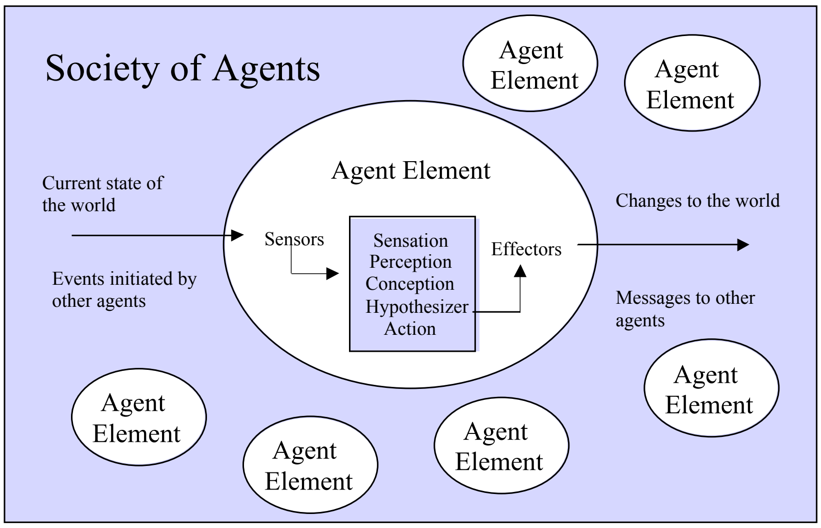


Figure 9: The society of agents as proposed in Smith et al. (2003).

The behavior of an agent is determined by its internal model that is composed of several layers. Messages are perceived by the agent through its sensors and are processed by the *Sensation* process. The output of this process is sense-data that can directly be used by the agent. The agent is able to react to this raw data with its reflexive behavior. If the agent does not react on this level, the data is further processed by the *Perception* process. This process recognizes patterns as percepts and the agent can react to it with its reactive behavior. The percepts can then be further processed by the *Conception* process that assigns further meaning to each percept. Finally, the *Hypothesizer* uses the processed percepts to reason about the current state and the goals of the agent. It proposes an *Action* that will assist the agent to achieve its goals. This behavior is called reflective behavior.

In order to enable agents to communicate with each other, a common language has to be defined. In this particular case, the Contract Net negotiation protocol was chosen (Smith, 1981). The Contract Net protocol is based on proposals and each agent having a request sends out a proposal to other agents. If an agent believes that it can fulfill the proposal, it replies with another proposal. The initiator then collects all proposals, chooses the best proposal and makes a contract with the winner about his proposal.

As example, a conference room consisting of wall agents and one room agent is presented. The room agent is responsible for providing an appropriate room size. If too many avatars enter the room, the room agent will sense this and will try to expand the room. The room agent sends a proposal to the wall agents requesting to increase the size of the wall. One wall agent may then get a contract and start to expand itself. This wall agent, in turn, sends proposals to the other wall agents that they should also grow in size. Since the agents are autonomous and act on their behalf, some wall agents may get larger but others may stay the same and leave a gap behind. This clearly is an undesired situation, but the authors do not present any solution to this problem - they just state that it can happen. Future work will be needed to resolve such inconsistencies.

The conference room was implemented in Active Worlds (AW). Active Worlds is a virtual 3D world reality platform. It offers a SDK that can be used to interface the world by controlling the movement and behavior of bots. The agents can be written in any language and only the sensors and effectors themselves are exposed to the virtual environment. One downside of their approach is the strict architecture they enforce. When using their framework you have to stick to the communication language they use and the agents will have to implement effectors and sensors. The internal agent model could vary from their proposed model as this information is encapsulated behind the effector/sensor interface. Nevertheless you cannot switch agent systems on the fly and have to adapt your agents according to their rules.

The next project abstracts the Multi Agent System and provides a uniform interface to the 3D world. The project called GameBots (GameBots) was developed by Adobbati et al. (2001). The intention of their work was to create a new multi agent research test bed that is not tailored to a specific kind of task in a fixed environment and supports human testing and interaction. They proposed GameBots as a 3D test bed that supports humans-as-agents, is easily customizable (due to a scripting language) and supports multiple environments and tasks. Their criteria for a suitable game engine included client/server model, large user base (i.e. well tested, reliable code) and easy modification. They decided to use the Unreal Tournament (UT) engine and implemented GameBots as a module that communicates with the UT server and bot clients, see Figure 10.

The module works bidirectional. It provides sensory information to the bot clients and controls the bots in the game. Based on the information a client receives, it computes the appropriate actions and issues the commands back to the module that, in return, causes the bot to move, jump, shoot, etc.

The communication between the client and the module is defined by a simple protocol. The initial handshake determines the type of game played at this server and the conditions for a win. After that, sensory information is sent to each client in a specified interval (10 times per seconds in the current implementation). These messages contain information about the game status and agent specific information such as viewpoint of the agent or state of the agent. For example, a map can contain a set of navigation points and each message will tell the agent, which point is reachable from its current position. This information can then be used to navigate the map. The second type of messages are messages that refer to immediate events and are triggered when the bot hits a wall or hears a noise.

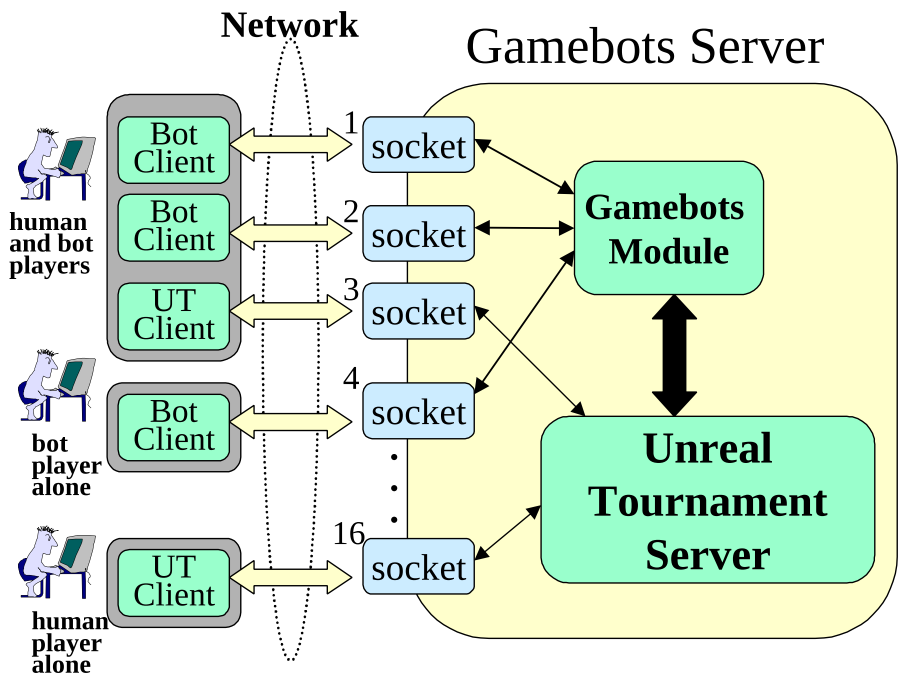


Figure 10: The GameBots architecture (Adobbati et al., 2001)

In the other direction (client to module) bot action commands are sent. Such commands tell the bot in the 3D world what to do and include the following: STOP - stops all movement and action, RUNTO - run towards a navigation point or a specified location (the command is only executed if the target is in sight), JUMP - causes the bot to jump.

GameBots also provides monitoring facilities. A logging tool allows the developer to log the events that are happening in the 3D virtual world. There are visualization tools that offer bird's eye view of the map including real-time movement of the bots. Furthermore, all recorded actions can be replayed and studied at a later point in time. The advantage of their approach is the loose coupling between the agent system and the 3D world. Of course the agent must be able to deal with the bot interaction protocol but there is no restriction on the communication between the agents.

Although the GameBots project seems not be to developed any further[1] there are other projects that use the GameBots module. In the following we will present one work that is based upon GameBots. The project is called UTSAF and was developed by Manojlovich et al. (2003). UTSAF stands for Unreal Tournament Semi Automated Force and is a framework that connects military simulations with a 3D visualization. As the name implies they use the Semi Automated Force (SAF) environment for the simulations and the Unreal Tournament (UT) engine to visualize the simulation space. As there have been several problems in creating visualizations for such distributed interactive simulations (DIS) in the past, the goal of their work was to create a framework that scales

[1]According to the Web page (GameBots) the last news stem from 2002.

well on large and heterogeneous simulation environments. To this end, they propose a new approach in which they use an agent system as the mediating component between the two layers. The architecture of UTSAF is depicted in Figure 11.

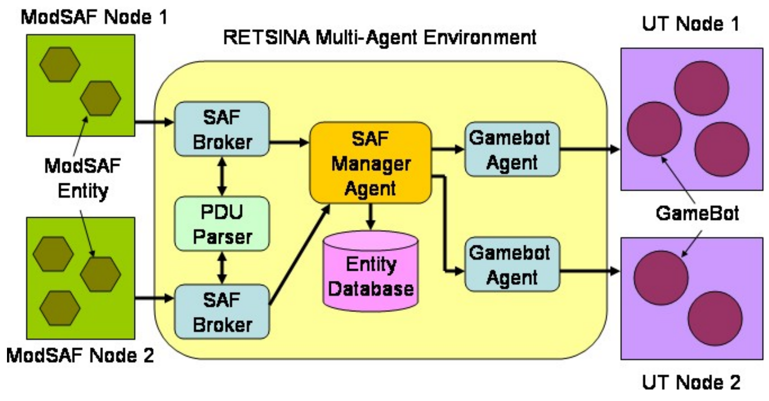


Figure 11: The UTSAF architecture (Manojlovich et al., 2003).

Several agents are used to establish a connection between the entities in the SAF environment and the bots in the UT world. Each of these agents has a specific task that will be summarized in the following. The SAF environment uses a standardized protocol (DIS protocol) for the communication between the entities in the environment. The task of the *SAF Broker* agent is to listen to the communication on one such node and to parse the messages it observers. The traffic is encoded in Protocol Data Units (PDU) that can be parsed with the help of the PDU parser. The heart of the architecture is the *SAF Manager* agent that manages the traffic between SAF Broker and GameBot agents. The database is used to update the status of entities in the SAF environment and changes are forwarded to the GameBot agents. The *GameBot* agents are used to control the GameBots in the 3D virtual world. Every time they receive an update via the SAF Manager they update the location and status of the corresponding GameBot in the Unreal Tournament.

3 Background and Foundations

The work reported herein is embedded within a research project that aims at the development of a radically new environment for e-tourism applications (Berger et al., 2007). In a nutshell, 3D virtual worlds will provide a familiar and intuitive environment for both, tourism suppliers and consumers. The system is intended to become a community facilitator allowing users to socialize with others and to establish new contacts. The environment is enriched with information by means of agent technology. Agents and humans are the participants in the 3D virtual world and work together to cooperatively achieve their goals. This can be summarized as follows:

- Provide a 3D e-Tourism environment for providers and consumers that enables versatile interaction between participants including the trade of tourism products.
- Provide a 3D e-Tourism environment that becomes a community facilitator to create and establish a lively and sustainable community involving both, providers and consumers.
- Provide a 3D e-Tourism environment that is information-rich and multimedia-based to offer transparent and unified access to disparate information sources.

These goals will materialize as an integrated, game-like e-Business application where each participant is impersonated as an avatar. Users are able to interact with each other in a familiar environment and have access to a wide variety of information. With this environment it will become possible to study the human-agent relationship and researchers will be able to get insights on the needs and requirements of human users concerning agent technology. The application of agent technology is especially promising in the tourism domain, since such information is usually spread over the Internet and agents are able to aid users in the collection of such information.

In this 3D e-Tourism environment two types of participants need to be considered: humans and agents. Agents are either autonomous or controlled by a human user. In the latter case, the couple human/agent is represented as an avatar in the 3D virtual world. The visualization of autonomous agents depends on their task and can range from the visualization as avatar to the visualization as information monitor. Humans and agents learn from each other and work together to collaboratively achieve certain goals. The user delegates tasks such as information gathering or product purchasing to the agent and learns from the agent which rules and restrictions apply in the environment.

These agents participate in a Multi Agent System (cf. Figure 12). The connection between the Multi Agent System and its visualization is realized in a causal manner. The term causality or causal connection refers to the connection between a system and the representation of this system. We will use the definition of Maes and Nardi (1988), where a system and its representation are causally connected when the following two criteria are met:

- Whenever the representation of a system is changed, the system itself has to change as well.
- Whenever the system evolves, its representation has to be modified in order to maintain a consistent relationship.

In our case the Multi Agent System is the system and the 3D virtual world is the representation of this system. The middleware causally connects both components. This connection, including the couple human/agent, is visualized in Figure 12. In this Figure, a user named *Elaine* is logged into the 3D virtual world. She is represented as an avatar and is the principal of an agent in the Multi Agent System. All actions Elaine is performing in the 3D virtual world are verified by the Multi Agent System. In this example Elaine is standing in front of a door and wants to enter the room behind it. The middleware forwards the request to her agent that checks whether this action is permitted in the current state of the Multi Agent System. The agent sends back a reply message that is forwarded to Elaine via the middleware. Depending on the state of the Multi Agent System, the door is opened or remains closed.

However, in the course of implementation we realized that we will not be able to enforce causality in this strict manner. The problem lies in the fact that we are using the Multi Agent System like a black box and cannot change its state arbitrarily. We can run into a situation in which an action has been permitted in the Multi Agent System, but at the time the action is executed, it is no longer permitted. Suppose that in the last example Elaine's action was permitted and the door had been opened. It takes several seconds until Elaine enters the room. According to the causality definition, Elaine's agent must also execute this action in the Multi Agent System. Meanwhile, however, the state of the Multi Agent System has changed and the agent is no longer permitted to perform the action. Since we are using the Multi Agent System like a black box, we cannot alter the state of the system and are not able to enforce causality in this direction.

We solve this problem in the following way. The components of the 3D virtual world, which are used to interact with the framework, are implemented by us. We therefore have complete control of the state in the 3D virtual world. Thus, it is possible to change the state of the representation whenever the system state changes. Since Elaine's agent could not perform the action in the Multi Agent System, the current state of the system has to be enforced in the 3D virtual world. This is achieved by teleporting Elaine's avatar to the room where she came from.

Although we cannot provide a causal connection in both directions, we still achieve a consistent relation between the two components. The causal connection between the Multi Agent System and the 3D virtual world is used to resolve any inconsistencies. Whenever

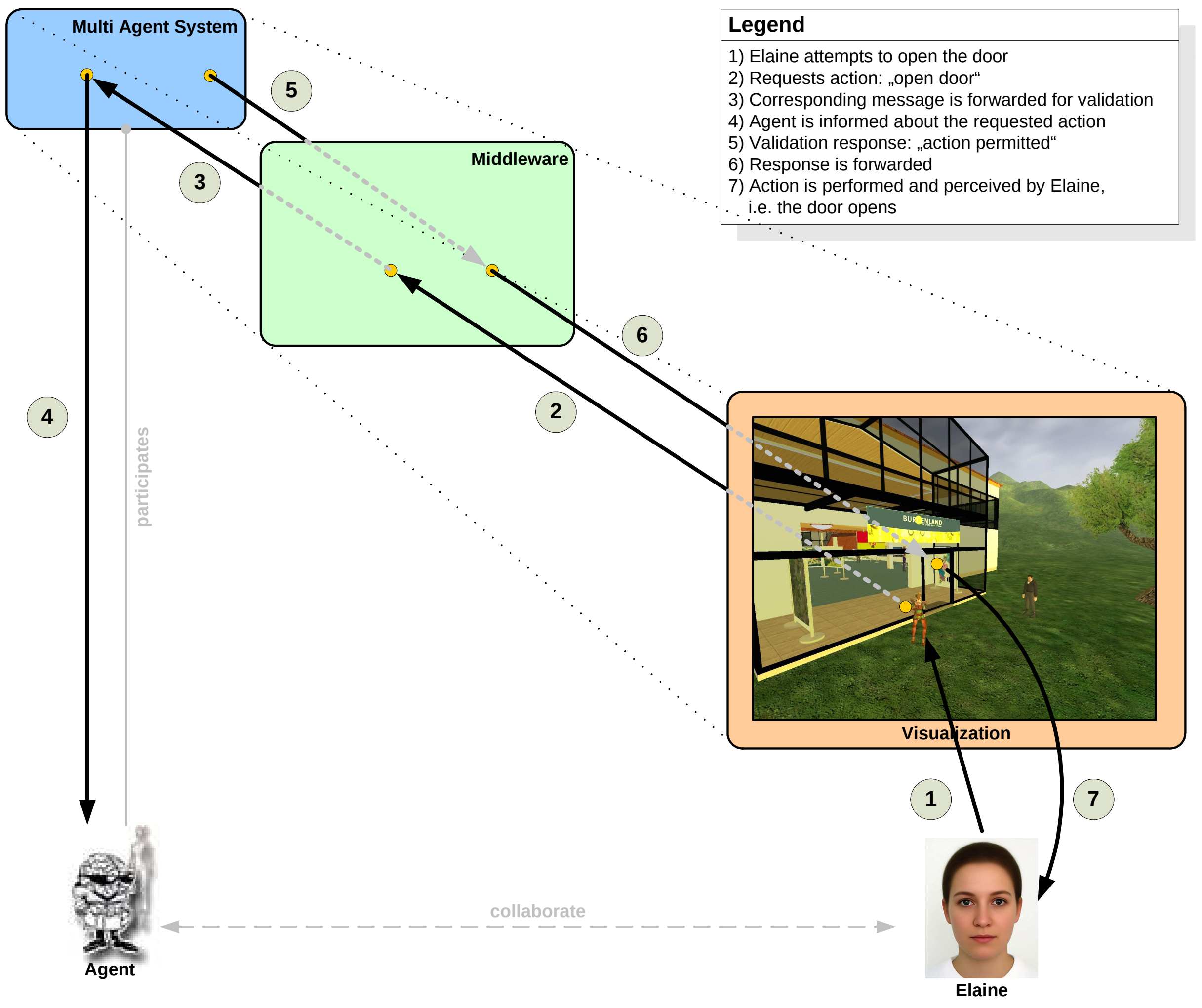

Multi Agent System
Middleware
Visualization
Legend
1) Elaine attempts to open the door
2) Requests action: „open door“
3) Corresponding message is forwarded for validation
4) Agent is informed about the requested action
5) Validation response: „action permitted“
6) Response is forwarded
7) Action is performed and perceived by Elaine, i.e. the door opens
1
2
3
4
5
6
7
participates
collaborate
Agent
Elaine

Figure 12: The Human-Agent relationship.

the system and its representation are in different states, the 3D virtual world is modified to maintain the consistent relationship.

Another important aspect of this project is the establishment of a lively tourism community. In order to realize this vision, our system has to provide community and social interaction features. A 3D virtual world implicitly addresses some of these requirements. Users are aware of each other, they see what other users are doing and perceive the environment as a community place. Furthermore, such environments have to provide mechanisms that enable users to communicate and encourage social interactions (Smith et al., 2003). To this end, we provide communication facilities such as chat tools, instant messaging or voice communication. To encourage users in social interactions we plan to implement reward mechanisms and try to establish a social status within the community. Users should be able to express themselves and their social status should be visible to other users. In Second Life (SL), for example, users can alter any aspect of their appearance. These modifications range from the seamless alteration of the body shape, to the adjustment of the eyelash size. A great deal of the success of Second Life is based on this feature. There are clothing stores, tattoo studios and barber shops where the appearance of avatars can be perfected. Users are frequenting these locations, spend real money and invest lots of time into the looks of their avatar.

Besides the implementation of such an avatar customization tool, the 3D e-Tourism environment provides avatar representation codes. Such a code defines visual clues for the representation of agents. It specifies how agents are visualized and if their visualization must contain certain elements. For example, seller agents may be required to wear a special outfit such that they can easily be distinguished from other avatars. The visual representation of an agent is not restricted to an avatar object, any 3D entity can be used. An information agent might be visualized as screen presenting pictures and videos to the users.

To realize the proposed system, we have to choose concrete components for the visualization and the Multi Agent System layer. Electronic Institutions are a proven multi agent methodology and provide a system for the interaction of agents (Esteva et al., 2004). They define which actions each agent is permitted to perform and control the correct behavior of the agents. Like in other works on agent technology, the research efforts for Electronic Institutions are placed on the software side and the society of agents. We follow the approach of Berger et al. (2007) who propose 3D Electronic Institutions as a combination of Electronic Institutions and 3D virtual worlds. While retaining the advantages of both systems, 3D Electronic Institutions open new dimensions allowing to study humans and agents from a new perspective. A game engine is used to provide an immersive 3D virtual world. We chose the Torque game engine for this task, because it is a well tested, industry proven and popular engine. Electronic Institutions and the Torque game engine are presented in more detail in the next two Sections.

3.1 Electronic Institutions Introduction

The concept of Electronic Institutions was developed at the Artificial Intelligence Research Institute (IIIA) in Spain. The main concept of Electronic Institutions is the application of real life institutions for Multi Agent Systems. In our life we have to deal with institutional processes all the time. Those processes define how we are supposed to behave and what we are allowed to do. Consider, for example, a traditional auction in an auction house like Sotheby's. An auctioneer tries to sell one good at a time and a group of buyers can bid for the product. Every new bid must be higher than the previous bid. If no new bids are put in, the auctioneer announces the person who stated the last bid as the winner. According to the institutional rules, this person is then obliged to buy the product. If the buyer cancels the purchase after this point, she has violated the rules of the institution and has to pay a fine.

An Electronic Institution defines rules and norms for the interaction and behavior of agents. It specifies the actions an agent can perform in the current state of the institution and how the agent can move in the institution. The structure of an Electronic Institution can be described along three dimensions:

- Conventions on Language, the *Dialogical Framework*. Electronic Institutions are open systems in which heterogeneous agents are participating. The Dialogical Framework defines the organizational structure of the agent society and is composed of agent roles. A role identifies the permitted actions of an agent. To facilitate information exchange between heterogeneous agents, the Dialogical Framework also defines a common language. The vocabulary of this language is specified in an ontology.
- Conventions on activities, the *Performative Structure*. Agent communication in Electronic Institutions is defined through scenes. A scene contains a scene protocol specifying the messages that may be sent between agents. Those scenes are inter connected with each other through transitions, forming the Performative Structure. A transition is restricted to a set of agent roles, identifying the possible movements of an agent in the Performative Structure.
- Conventions on behavior, the *Norms*. The Norms can be split into inter-scene and intra-scene roles. The inter-scene norms are essentially the scene protocols and describe which role is permitted to send which message to which agents in a scene. The intra-scene norms specify the movement of agents between scenes.

3.1.1 Specification of Electronic Institutions

This Section presents the different elements of an Electronic Institutions and how they are specified in more detail.

Dialogical Framework

One aspect of institutions is the fact that every participant adopts a specific role. In the auction example, for example, the participants adopted either the buyer or the auctioneer role. Agents are the players in an Electronic Institution and each agent adopts a role when acting in an Electronic Institution. Basically roles are the main elements for the communication and movement in an Electronic Institution. All oncoming specifications are based on the type of role the agent currently inhabits.

It is possible to switch roles over time, so an auctioneer might be a buyer in another auction. It is even possible for an agent to play several roles at the same time. For this reason relationships between roles need to be defined. Roles can be specified being mutually exclusive, meaning that a single agent cannot play those roles at the same time. This would be the case in the auction example. A person cannot be an auctioneer and a buyer at the same time, since this would violate the concept of an auction (the person could purchase the product at any price, because she decides who will get it). Moreover there is a distinction between internal and external roles. Internal roles are responsible for the execution of the Electronic Institution and can only be adopted by internal agents. External agents, participating in the system, can only adopt external roles. In the auction example, the auctioneer role is an internal role whereas the buyer role is an external role. External agents are not allowed to play the auctioneer role.

In order to allow heterogeneous agents to communicate with each other, it is necessary to provide a common language. The messages that can be exchanged between agents are based on their current role type. Within the Dialogical Framework an ontology, describing the vocabulary that can be used by the agents, is defined. The communication language is defined by expressions of the following form:

$$(i, (\alpha_i, \rho_i), (\beta), \pi, \tau)$$

where:

- i is an illocutionary particle.
- α_i, ρ_i is a term which can be either an agent variable or an agent identifier.
- β represents the addressee(s) of the message which can be an agent or a group of agents.
- π is an expression of the ontology.
- τ is a term which can be either a time variable or a time constant.

The definition of this expression is related to speech act theory. The variables have the following meaning. The illocutionary particle defines which type this expression has - examples of illocutionary particles are: *request*, *inform*, *failure*. The pair (α_i, ρ_i) describes the sender of this message. α_i can be an agent variable or an agent identifier and ρ_i can be a role variable or a role identifier. The same applies for the β identifier. This is the receiver of the message and can also be described as a pair (α_j, ρ_j). This is the case if the message is addressed to a single receiver. The message can also be addressed to all the agents of a specific role. In this case β is only of the form ρ_j. Furthermore the message can be addressed to all agents. Then β is of the form *all*. If such a language expression contains unbound variables it is called an *illocution scheme*, otherwise it is called *illocution*.

Scenes

Scenes are the heart of each Electronic Institution. They define communication patterns between roles that allow agents to interact with each other. A scene is composed of several states that are interconnected with directed edges. This state automaton describes the different conversations that may take place in the scene. Each scene has an initial and a final state, describing in which states a new scene instance can be generated and closed. It is possible to instantiate multiple instances of the same scene. Scenes are usually created and closed by internal agents. We can draw an analogy with the auction example: the creation of a scene refers to the start of the auction and when the auction has finished the scene is closed. For each scene state it is possible to define the roles that may enter and exit the scene in the current state. Usually roles, eligible to participate in a scene, are able to enter it through the initial state and can exit it through the final state. The enter and exit schemes in other states depend on the specification of the scene.

Communication in a scene is defined by the edges that connect the single states. Each edge is labeled with illocution schemes defining the possible messages that can be said. In a scene state all illocution schemes on outgoing edges might be said in that state. Since an illocution scheme may also restrict the sender and receiver role of the illocution, an agent can only utter those illocutions that match its role. When uttering an illocution, the illocution is matched against the illocution scheme and unbound variables are replaced with concrete values. If a variable is prefixed with a '?', this means that the variable is unbound and can be bound to any new value. If the variable is prefixed with a '!', this means that the variable is bound and has to be replaced with the last value of this variable. Whenever an illocution is uttered, the scene evolves and the state of the scene changes. This is how the communication takes place. Agents utter illocutions and the scene changes from one state to another.

The Performative Structure

The Performative Structure describes how the scenes are interconnected and is a specification of the composition of the institution. Considering the auction example the Performative Structure describes the auction house, where the auctions take place and who is allowed to enter which auction.

The Performative Structure can be specified by means of a graph. In this graph each node represents a scene that is interconnected through transitions. A transition is special node called transition node. Several outgoing scene edges can end in a transition node and the outgoing edges of the transition nodes lead to other scene nodes. Those edges are labeled with role identifiers, specifying which role can enter/exit a scene over this edge. A transition node describes how agents can move between the scenes. There are two types of transitions in general: *and* and *or* transitions. Whenever an agent leaves a scene over an outgoing edge, it enters the subsequent transition. If this transition is of type *or*, the agent can enter any scene that is reachable from this transition for its role type. If the transition is of type *and*, the agent has to enter all scenes connected to this transition.

Each incoming edge is labeled with one of the following: *1*, *some*, *all* and *new*. The first type, a *1-edge*, expresses that the agent has to enter exactly one instance of the target scene. The *some* label defines that the agent can enter any subset of instances. When the edge is labeled with the *all* symbol the agent must enter all current instances of the scene. The *new* label describes that a new instance of the scene must be generated. Those *new* edges are usually restricted to internal agents who use them to create new scene instances.

3.1.2 An Auction Example

In order to get a better understanding of these concepts, we now examine a concrete Electronic Institution. This institution is described in (EIDE) and defines an auction house where agents can buy and sell items using auction mechanisms.

Role structure

In Figure 13 the different roles of this example are depicted. The different colors of the roles refer to internal and external roles. The institution is run by internal staff agents who may also adopt the auctioneer role.

External agents can inhabit the guest role which is mutually exclusive with the internal roles - no agent can adopt an internal and an external role at the same time (ssd stands for static separation of duty). Agents inhabiting the guest role must also have two properties, namely an *admitted* and a *credit* property. The admitted property identifies if they are permitted to attend an auction and the credit property defines the credit limit of the agent. An external agent can further act as a buyer or as a seller in an auction. Again those two roles mutually exclude each other.

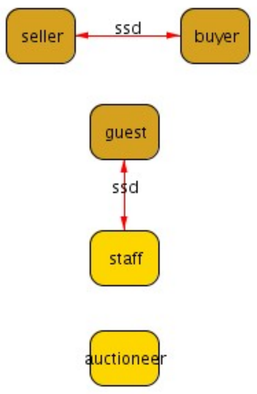


Figure 13: Role structure.

Ontology

We will only present some of the possible functions and data types of the ontology that are relevant for the example.

- *item*. This is a data type used to describe the items in the auction. It simply contains the name of the item and a textual description of this object.
- *login*. Agents can use this function to log in the institution. The function has two parameters the name and the e-mail address of the agent.

- *accept.* If a login request has been accepted by the internal agents, the agent in question is informed via this accept function. This function also informs the agent of its credit limit.
- *close.* This function is used by the internal agents to end the execution of a scene.

Scenes

The Performative Structure of the auction house contains several scenes. The relationship between scenes and the Performative Structure is detailed in the next section. As an example, the Admission scene is presented in Figure 14. This scene defines a login procedure allowing agents to participate in the auction house. The core of a scene is a state machine. The Admission scene contains four states whereof W0 is the initial state and W3 is the final state. The numbers along the connecting edges correspond to the messages detailed in Table 1.

Label	**Message**	**Actions**
1	request (?x guest) (?y staff) (login ?user ?email)	

2	inform (!y staff) (!x guest) (accept ?credit)	x.admitted = true
		x.credit = !credit

3	failure (!y staff) (!x guest) (login !user !email)	x.admitted = false
4	request (?x guest) (!y staff) (login ?user ?email)	
5	inform (!y staff) (all guest) (close)	
6	inform (?y staff) (all guest) (close)	

Table 1: Messages of the Admission scene.

The signature of each message is

Illocutionary Particle (*Sender Role*) (*Receiver Role*) (*Function FunctionParameters*)

The variables x and y refer to the agents involved in the conversation. The '?' preceding the variable identifies unbound variables, not being bound to a specific agent, whereas the '!' identifies variables that are already bound to a particular agent. Consider, for example, the messages 1 and 2. Each agent playing the guest role (?x) can send a login request to a staff agent. After the staff agent has evaluated the request, it returns the accept message to the particular agent (!x) that sent the request.

The Admission scene might be accessed by guest and staff agents via state W0. Guest agents are permitted to leave the scene in this state, whereas staff agents can only leave the scene in the final state W3. In state W0 two possible messages can be said. The staff agent can close the scene by issuing a close message (label 6) to all guests that are currently participating in the scene. The scene will then evolve to the final state W3 and all agents have to leave the scene because its execution has ended. The second message that can be said in state W0 is a login request. The guest agent can make a login request

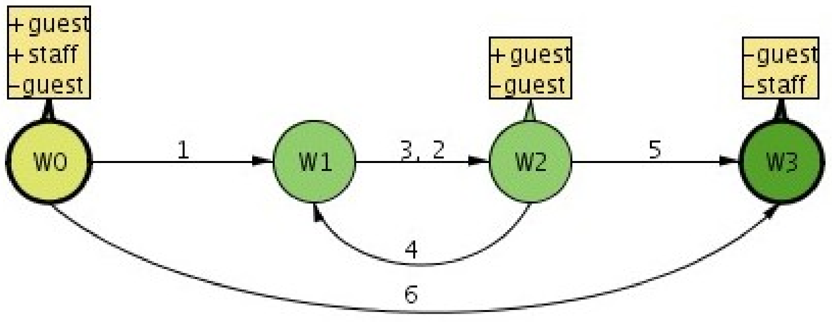


Figure 14: The Admission scene of the Auction House.

(label 1) and the scene evolves to state W1 in which the request is processed by the staff agent. If the login request is negatively evaluated by the staff agent, a failure message (label 3) containing the original login function request is returned. If the request was successful, the staff agent replies with an accept message (label 2). This message sets the property admitted=true and credit to the value specified in the accept function. In either case the scene evolves to state W2. In this state the guest agent may leave the scene and new guest agents may enter the scene. Guest agents can state new login requests by traversing the edge with label 4. The scene then alternates between the states W2 and W1 until the staff agent closes the scene with a close message as indicated by the edge labeled 5.

Performative Structure

The Performative Structure of the institution is shown in Figure 15. Each Electronic Institution has an Initial scene where agents may enter it. The Final scene is used to leave the Electronic Institution. These scenes are marked with a special coloring. All other scenes are blue in color.

After entering the institution through the Initial scene, the agents automatically access the first transition. This transition is of type *and*. An agent must access all following scenes for his role type. So, the staff agent is obliged to create new instances of the Admission, the ItemRegister and the AuctionInfo scenes. A guest agent can only enter the Admission scene. The scene protocol of this scene was presented in the previous section. After registering itself in the institution, the guest agent may move to the ItemRegister or the AuctionInfo scene. In the ItemRegister scene it is possible to register an item for an auction and in the AuctionInfo scene an agent may request information on current auctions. Note that the agent must change its role from guest to seller or buyer respectively. After registering an item, the staff agent will open a new Auction scene and changes its role to auctioneer. There may exist several Auction scene instances at the same time. An agent, inhabiting the buyer role in the AuctionInfo scene, may enter the Auction scene.

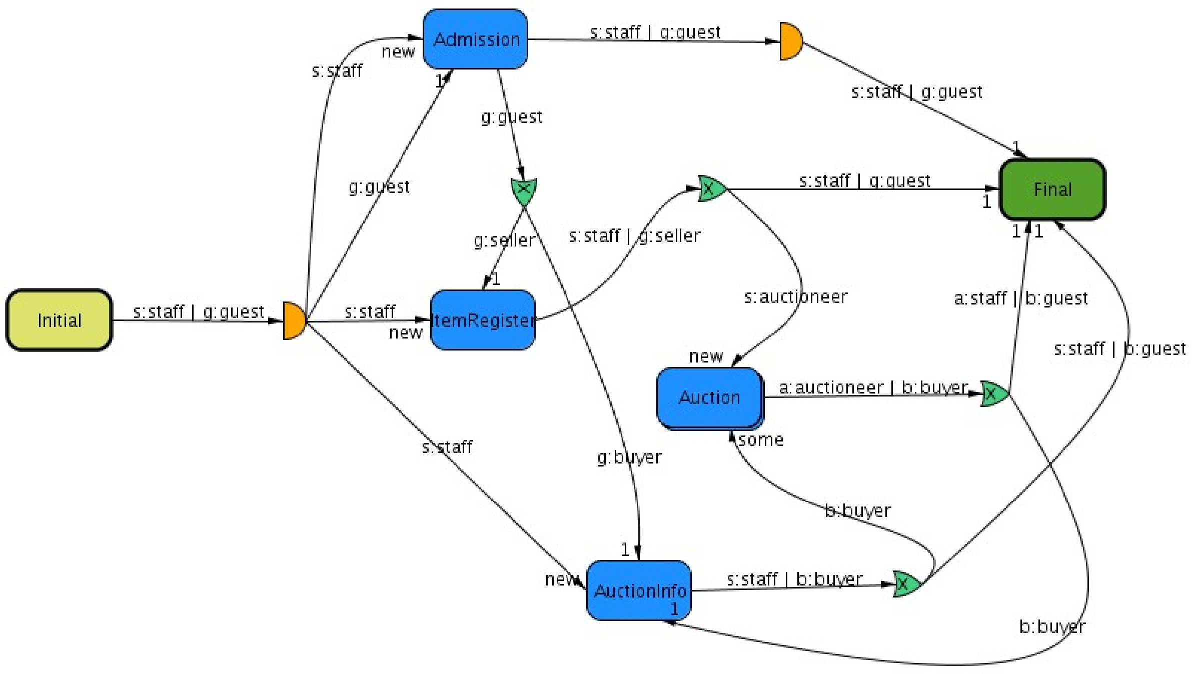


Figure 15: The Performative Structure of the auction house.

This agent may enter several instances of this scene at the same time. The Auction scene is used to execute an auction. The protocol of the scene is not fixed, thus, several types of auctions can be held in this scene. The buyer can then circle between the Auction and the AuctionInfo scenes. Agents leave the institution via the Final scene.

3.1.3 The Electronic Institutions Development Environment

To support the development of Electronic Institutions, a framework was developed by researchers at IIIA. This framework introduces programs that allow designers and programmers to specify, implement, verify and run Electronic Institutions (Arcos et al., 2005). It is called Electronic Institutions Development Environment (EIDE) and is freely available (GNU license) from the project Web page (EIDE). It includes the following tools.

- **Islander** is used to create specifications for Electronic Institutions (Esteva et al., 2002). It consists of a graphical editor where the entities of a structure can be arranged and connected. All images of the auction house example (cf. Section 3.1.2) are specifications made with Islander. Furthermore, specifications can be statically verified, helping the designer to avoid errors. The output of Islander is a XML document containing specifications of all components of an Electronic Institution.

- **Ameli** is the runtime environment allowing the execution of Electronic Institutions. It is started with a set of Electronic Institution specifications. Agents can then join the system and participate in Electronic Institutions. The behavior and movement of all agents is controlled by Ameli.

- The **Agent Builder** simplifies the development of agents. Similar to Islander, it contains a graphical editor where agents and their attributes can be specified. This specification is stored in XML and transformed to source code by another component of the Agent Builder. The generated source code can then be refined by the developer.

- **Simdei**. Since Electronic Institutions are open systems where heterogeneous agents can participate, the dynamics of the system can hardly be estimated at design time. Simdei (Simulation and Dynamics of Electronic Institutions) is used to simulate the dynamic behavior of Electronic Institutions.

The tool that is incorporated in our framework is the Ameli runtime environment. Electronic Institutions that are executed in Ameli are organized by a specific structure. Each *Electronic Institution* runs within the context of a *Federation* which in turn runs in the context of a *Platform*. Ameli executes exactly one Platform. A Platform can contain an arbitrary number of Federations and a Federation can contain an arbitrary number of Electronic Institutions.

3.2 Game Engine Introduction

The choice for the game engine was supported by a software evaluation report (Seidel, 2005). In this work four different 3D virtual worlds were evaluated - two game engines, the Q Engine (Q) and the Torque Engine (Torque), one 3D virtual browser, Active Worlds (AW) and one 3D API, Java 3D (Java3D). As a result the Torque game engine was selected.

3.2.1 The Torque Game Engine

The Torque game engine is an open source game engine developed by GarageGames (Torque). The feature list includes seamless indoor/outdoor rendering, animation support, a lighting engine, powerful editors, a scripting language and an award-winning network code. Furthermore, the complete source code is provided, making any desired modifications possible. The engine can be purchased as Indie license or as Commercial license, depending on the yearly income of the developer or his company. In both versions the royalty free licensing model allows developers to distribute and publish their games without further costs.

Torque applications can be developed on three different levels, depending on how much control the developer needs:

- The first level is used for content creation. Several in-game editors allow the designer to position objects in the 3D world and to modify the landscape. The World Editor is used to place, resize and name 3D objects. The Terrain Builder and Terrain Editor are used to generate, shape and texture the terrain. The GUI Editor is used to design the graphical user interface of the game.

- The behavior of a game can be programmed using the Torque Scripting Language (TSL). It is a dynamic language and based on function definitions that can be grouped in namespaces. The source code is compiled on-demand to a binary code which is interpreted during the execution of the application. The Torque package includes many script functions that facilitate the development and even functions in the engine code can be called from the scripts. On the contrary, the dynamic behavior and the poor structuring make it hard to develop complex constructs with the scripting language.
- The game engine itself is written in C++. Since Torque is distributed under an open source policy, this source code is accessible. This allows the developer to alter the game engine according to his own's wishes. Furthermore new features can be implemented and compiled into the game engine executable.

4 The Middleware Architecture

An overview of the architecture of the 3D e-Tourism environment is depicted in Figure 16. Each layer is named after the concrete application running on this layer. The enclosing rectangles define self-contained execution environments in which each program is running. Those environments may run on the same computer or can be distributed across several machines. The communication between the layers is based on the TCP protocol. The components within each layer listen to network traffic and send messages on predefined ports.

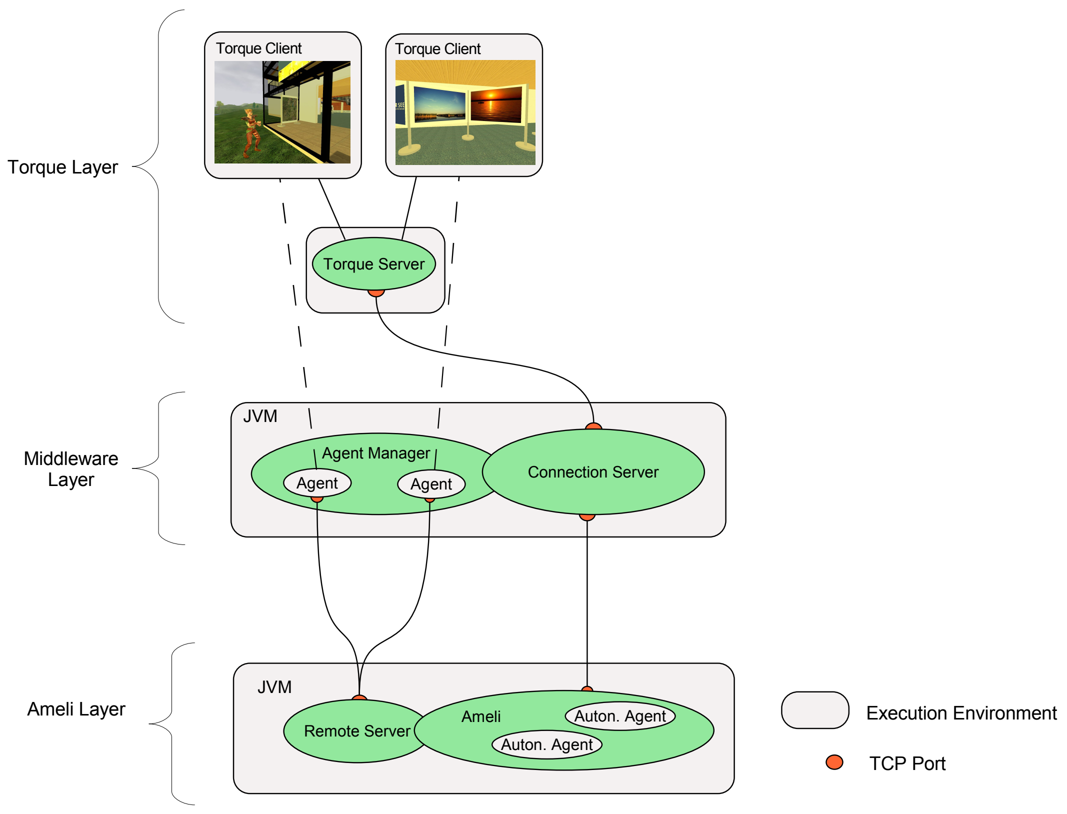


Figure 16: Overview of the framework architecture.

The bottom layer contains Ameli, the multi agent runtime environment. The TCP port in the Ameli component is used to send all events to the middleware. Another component of the Ameli system is emphasized, because it is also responsible for the message exchange with the middleware. The Remote Server allows external agents to participate in the Ameli system. A predefined communication protocol (which is specified in the EIDE package) is used to facilitate the communication with external agents. Agents state action requests to the Remote Server which are then validated in the Ameli system. Depending on the current state of the Ameli system, the server sends back a reply message to the agent.

The middleware, henceforth referred to as Connection Server or CS, connects these two layers. The connection to the Ameli system is used to listen to events that happen in this system. The connection with the Torque layer is used to exchange messages with the Torque system. The Agent Manager component of the Connection Server is highlighted because it also establishes a connection with the Ameli layer. Remember that each user is the principal of one agent (cf. Section 3). The Connection Server controls these agents and is responsible to transfer the user actions in the 3D virtual world to actions in the Ameli system. The agents act as external agents and communicate with the Ameli system through the Remote Server.

Torque is running on the top layer which is further split into a sever and client components. The server is running as a dedicated server that displays the 3D virtual world, controls the visualization and observers the actions of the users. The server guarantees the consistent relationship with the Ameli system and changes the state of the 3D virtual world if necessary. The connection between the Torque server and the Connection Sever is used to exchange messages between these two layers.

The Torque client is running on the computer of the user. It is used to connect to one of several servers (there may be more than one server visualizing different Ameli systems). The server and client components run in different execution environments and can also be split across several machines (the connection is based on TCP).

4.1 Information Mapping

In this Section we outline the connection of an Electronic Institution with a 3D virtual world and deploy the protocols that are needed to achieve this connection. We start off with a conceptual mapping of the Performative Structure of the Electronic Institution. The first part of a mapping specification is the generation of a floor plan for the Performative Structure. A straight forward approach is to map scenes on rooms and transitions on doors (Berger et al., 2007). A simple example is illustrated in Figures 17 and 18. The Electronic Institution described in Figure 17 is a simple institution that is placed in the tourism domain. The Electronic Institution depicted in Figure 17 resembles a simple tourism scenario and is taken from Berger et al. (2007):

- The *Traveler's Lounge*: A meeting scene where travelers can meet and discuss topics of interest.
- In the *Travel Advisory* a tourist can get professional information on destinations she is interested in.
- The *Travel Agency* can be used to gather information on different offers and to finally book a trip.

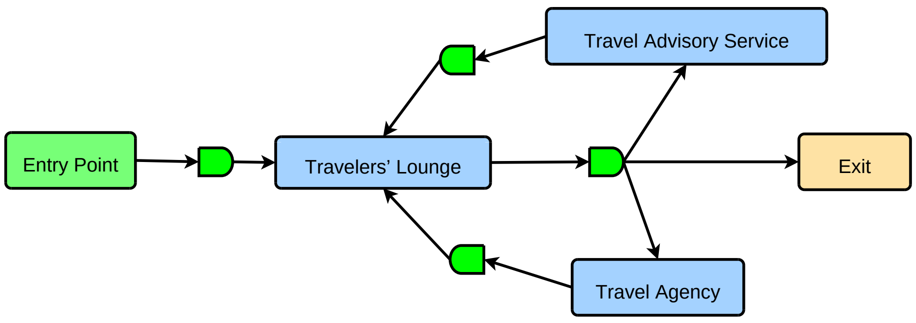


Figure 17: Performative Structure of a Travel Electronic Institution

For the generation of the floor plan we employ a straight forward and intuitive mapping. The Traveler's Lounge scene is mapped onto one room, the Travel Advisory scene is mapped onto another room and the Travel Agency scene is also mapped onto a room. Similar to the specification in the Performative Structure, the Travel Electronic Institution is entered and exited through the Traveler's Lounge. The Entry and Exit scenes are mapped onto the doors of the Traveler's Lounge. The Travel Agency and the Travel Advisory are accessible from the Traveler's Lounge. The transitions between the scenes are mapped onto the doors connecting those rooms. In the remainder of this thesis we will refer to the definition of this institution and its floor plan.

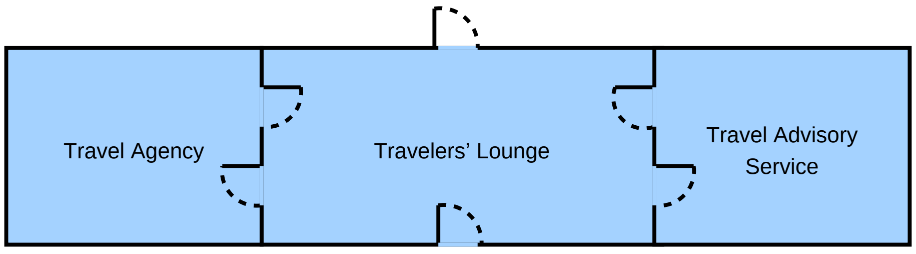


Figure 18: A possible floor plan for the Travel Electronic Institution.

So far we have been talking about the mapping of single Electronic Institutions. Remember that Ameli is organized according to three different levels - Platform, Federation, Electronic Institution (cf. Section 3.1.3). These organizational levels must also be mapped onto the 3D virtual world. We deployed the following approach. A Federation is mapped onto one 3D virtual world. The Electronic Institutions within this Federation are visualized as buildings in the 3D virtual world. Therefore, each 3D virtual world visualizes exactly one Federation and (possibly) multiple Electronic Institutions. The Platform concept is not mapped onto the 3D virtual world. Autonomous agents participating in the Ameli system are only visualized if they are acting on the Federation or Electronic Institution level.

The implementation of the 3D virtual world is based on a client/server model (cf. Section 4). The server is running as a dedicated server and visualizes exactly one Federation. The Federation to be visualized is specified at startup of the server and cannot be changed during runtime. The server “knows” how the Federation and its Electronic Institutions are mapped onto the 3D virtual world and guarantees a consistent relationship with the Ameli system. A user who is running the Torque client on his computer can query all accessible servers over the network (LAN or Internet). When the user has decided to which server she would like to connect, a connection will be established. The user agent automatically passes the Platform level and enters the Federation in the Ameli system. The user is spawned as a new avatar in the 3D virtual world and can act on the Federation and Electronic Institution level by entering and leaving buildings. When the user disconnects from the server, his agent leaves the Ameli system.

4.1.1 Formalizing the Mapping

The mapping concepts of the previous Section are formalized by means of a specification. This specification is used to define which scene is mapped onto which room and which transition is mapped onto which door. Furthermore, this specification provides additional features allowing the developer to specify the EI-3D virtual world relationship in more detail. Thus it is possible

- to map more than one scene onto one room. One scene will then be the primary scene and the other scenes are called “virtual”. The user can then participate in different scenes without leaving the room. It would, for example, be possible to map the Traveler’s Lounge and the Travel Advisory scene to the same room. The Traveler’s Lounge could be the primary scene and the Travel Advisory scene could be visualized as information desk in this room.

- to map more than one transition onto one door and to also map scenes onto doors. These entities are henceforth referred to as “subsumed entities” - a transition or scene is subsumed in a door. The choice for the scene/door mapping has one vital reason: the initial and final scenes (that must be specified for each Electronic Institution) are automatically passed. When an agent enters an Electronic Institution it automatically passes through the initial scene. The same happens when it exits the Electronic Institution through the final scene. As we cannot abandon those scenes and it makes no sense to model them as a room, we decided to allow a multi mapping onto doors. The initial scene will then be mapped onto the entrance door of the Electronic Institution building, and the final scene onto the exit door.

- to define visual clues for an agent role. With this approach agents with the same role will look the same way, making it easier for users to distinguish the responsibilities of them. This is just the same as in real life, where, for example, a seller can be identified by his dress or a name tag.

All these properties are covered in a mapping file that consists of the elements specified in the Tables 2, 3 and 4. The first part is the agent mapping that allows developers to specify the role name and the 3D shape that will be used for agents with this role. It is also possible to visualize a role in a static manner (information screen). In this case the room in which the object is placed must also be specified.

Element Name	**Element Description**
Role Name	The name of the role as specified in Ameli
Avatar Path	The path to the avatar model for this role
Static	True if this role will be statically visualized, otherwise false
Room Name	The room in which this role will be visualized (only needed if the Static field is true)

Table 2: Agent Mapping.

The room mapping contains the name of the room in the 3D world and all scenes that are mapped onto this room[2]. Each scene is specified by its name, its mapping “mode” (virtual or non virtual) and the maximum number of agents that are allowed to enter this scene (this is mainly a hint for the designer when considering the room size). Note that at least one scene in each room has to be non virtual.

Element Name		**Element Description**
Room Name		The name of the room in the 3D world
Scenes		All scenes that will be visualized in this room
	Scene Name	The name of the scene
	Virtual	True if the scene is a virtual scene, otherwise false
	Max Users	The maximum number of users that can participate in this scene

Table 3: Room Mapping.

Element Name		**Element Description**
Door Name		The name of the door in the 3D world
First Room		The first room this door connects
Second Room		The second room this door connects
Subsumed Entities		Entities that are subsumed in this door
	Name	The name of the entity
	Type	The type of the entity (either transition or scene)

Table 4: Door Mapping.

[2]It is assumed that the 3D virtual world supports the creation of named areas. In Torque, for example, there are Trigger areas which are invisible for the user and report whenever an avatar enter or leaves the area. Such areas are used to specify the extent of the room in a Torque world.

The door mapping consists of a door name (the name used in the 3D world), the rooms this door is connecting and the entities (scenes and transitions) that are subsumed in this door. Each entity is specified by its name and its type.

4.1.2 Design Considerations

In the course of developing this framework we had to make one essential design decision: where should the mapping of information be placed? There are two possible locations where this information can be placed - either in the Connection Server or in the 3D virtual world. Both locations have their pros and cons. When we put this information into the Connection Server, no MAS specific properties are exposed to the 3D virtual world. From the viewpoint of the 3D virtual world it would then be possible to switch agent systems without changing the code on the 3D side. However, the disadvantage is, that the middleware must have knowledge of *all* 3D virtual worlds. It must know the structure and layout of each building in each virtual world. The middleware becomes a fat component that must be synchronized with the 3D layer. Every time a 3D virtual world is changed or a new one is created, the middleware and the mapping information in the middleware must be adjusted. Clearly, the system does not scale very well and may become a bottleneck.

These are mainly the reasons why we did not put the mapping information in the middleware. We decided to implement the mapping on the 3D side. The advantages of this approach are:

- *Better separation of duty.* The mapping information contains attributes of the 3D virtual world and the middleware should not have knowledge of this information - 3D information is placed on the 3D side.
- *Thin middleware component.* Most of the world logic is incorporated in the 3D virtual worlds making the middleware smaller and more flexible.
- *The system scales better.* When new 3D virtual worlds are added, the middleware only has to process some more messages. The computationally intensive tasks (mapping computation) are split across the single 3D virtual worlds.

4.2 Messaging

From an implementation viewpoint the middleware runs in accordance with the Multi Agent System. This means that each running middleware instance mediates the events and messages of one particular Multi Agent System instance. We therefore have a tight coupling between the middleware and the Multi Agent System. The connection between those two layers is implementation specific and cannot be specified in a general message protocol. Thus, we only provide a brief overview in Section 4.2.1, the full details can be found in Section 5.

On the other hand the middleware and the 3D layer are loosely coupled. Therefore we need a general message protocol that is not bound to one particular 3D virtual world. The message protocol is used to synchronize the state between the Multi Agent System and the 3D virtual world and only contains messages that are relevant for the 3D virtual world and the Multi Agent System. The Multi Agent System does not have to know about all actions that happen in the 3D virtual world and the 3D virtual world is not interested in all events that happen in the Multi Agent System. The middleware forwards only those messages that are relevant for the respective layer. This message protocol is introduced in the Sections 4.2.2 to 4.2.5.

4.2.1 Connection Server and Ameli

There are two different connections between those two layers (cf. Figure 16). There is the connection which is used to transmit the Ameli events and there is the connection used by external agents. In the first connection messages are only sent in one direction, namely from the Ameli system to the Connection Server. The possibility to monitor all Ameli events is a feature of the Ameli system. The user can define a format (XML, text) and a medium (file, TCP socket, console) to which the events are written. We use this function to send all events to a specified TCP port. The Connection Server listens on this port for new events and processes them. The detailed mechanism is presented in Chapter 5.2.

The second connection between the Remote Server and the Connection Server is used by the external agents. These agents must be able to understand the predefined message protocol used for the interaction with the Remote Server. We reused one component of the Ameli system for this purpose. The EIDE package defines a so called *DummyAgent* package which is a graphical interface for human users. With the help of this interface a user can participate in the Ameli system. The user controls the movement of one agent and can engage in scene protocols. We use parts of the *DummyAgent* code to control the user agents in the Connection Server. We extracted the GUI functionality of the original *DummyAgent* and reuse its internal message structure and message handling procedures. The functionality of these components includes the parsing, construction and handling of messages that are exchanged with the Remote Server. The remaining code was rewritten to provide an interface for the Connection Server. The new *DummyAgent* can therefore be controlled from the source code. More information on this aspect can be found in Section 5.4.

4.2.2 The Message Protocol

In general, we aim at a loose coupling between the middleware and the 3D virtual world. It should be possible to replace the 3D virtual world relatively easy and we therefore abstract all necessary messages in a common message protocol. The message protocol can be grouped along two dimensions: The direction and the message type. Messages are either sent from the Connection Server to the 3D virtual world or from the 3D virtual world to the Connection Server. When distinguishing the messages by their types, each message can be placed in one of the following groups:

- **Status Messages:** Status messages are used to query Ameli about the current state of the system. These messages do not change the state of Ameli nor the 3D virtual world.
- **Ameli Action Messages:** These messages are sent by the Connection Server to inform the 3D virtual world that something changed in the Ameli system.
- **3D Action Messages:** These messages are used by the 3D virtual world to inform Ameli that an action in the took place in the 3D virtual world.
- **Error Messages:** This type of messages refers to a failure that occurred.

Every message contains a header and a content section. The header is used like an address field to correctly deliver the message to the receiver. Since Ameli's internal structure is organized in three levels: Platform, Federation and Electronic Institution level, every message header contains a Platform, Federation and Electronic Institution field. The message is then handled by the middleware and the 3D virtual world according to this information. Some messages do not require Platform, Federation and Electronic Institution information. The header fields in those messages are all empty. Such a message would, for example, be an `askPlatforms` message, which is used by the 3D virtual world to query the middleware which Platforms are connected. In a valid message header the fields are filled according to one of four possible combinations, listed in Table 5.

Platform Field	Federation Field	EI Field	
empty	*empty*	*empty*	Messages on a higher level than the Platform level like `enterPlatform`
nonempty	*empty*	*empty*	Messages on the Platform level like `enterFederation`
nonempty	*nonempty*	*empty*	Messages on the Federation level like `enterEI`
nonempty	*nonempty*	*nonempty*	Messages on the Electronic Institution level like `enterScene`

Table 5: Possible field assignments in the message header.

We will now present some messages of each category by example. This is done in the next two Sections 4.2.3 and 4.2.4. The complete message protocol, including all possible messages, can be found in Appendix A. Note that the header information is omitted in the message descriptions since the same information is contained in every message and this would only clutter the message description. Therefore in all message tables only the content of each message is listed.

4.2.3 Communication Patterns - Autonomous Agents

As previously mentioned, we distinguish two types of agents: autonomous agents and user controlled agents. Autonomous agents operate in Ameli on their behalf and are not controlled by a human through the 3D virtual world. This makes the agent autonomous from the perspective of the 3D virtual world. However, it is possible that a human is connected to Ameli via a different interface and not the 3D virtual world. In that particular case, the agent controlled by the human is seen as an autonomous agent by the 3D virtual world. The 3D virtual world is responsible for the visualization of this agent but is not used to control the agent.

Message Name	Parameters		Purpose
`enterFederation`	`id`	The avatar identifier	An agent entered a Federation, visualize it in the 3D virtual world.
`enterScene`	`id`	The avatar identifier	The avatar with identifier `id` moved to a scene in Ameli. Move the avatar to the mapped room.
	`name`	The name of the scene	

Table 6: Ameli Action Messages

The task of the 3D virtual world is to visualize the actions of the autonomous agents. To this end, the message protocol contains messages that are used to inform the 3D virtual world about the movement of agents (cf. Table 6). Whenever an agent enters a Federation, the `enterFederation` message is sent to the 3D virtual world. The 3D virtual world is now responsible to spawn a new avatar for this agent. If the agent moves within the Electronic Institution, the 3D virtual world has to move the avatar correspondingly. The other message, `enterScene`, states that an agent entered a scene in an Electronic Institution. The 3D virtual world must move the corresponding avatar to the mapped room. It receives an `enterScene` message, looks up in which room the scene is visualized and moves the avatar to this room. In which way the movement is carried out is decided by the 3D virtual world. It is possible to simply teleport the avatar to the room or the avatar may walk along a predefined path to the destination.

4.2.4 Communication Patterns - 3D User Controlled Agents

Communication initiated by the 3D virtual world is more complicated. The 3D virtual world makes a request and receives a response for this request. Consider the status messages in Table 7. The 3D virtual world is able to visualize several Electronic Institutions. In order to identify which Electronic Institutions are connected through the middleware, the 3D virtual world sends an `askEInstituions` message to the Connection Server. The request is validated by Ameli and the middleware responds with an `informEInstitutions` message.

Message Name	Parameters		Purpose
`askEInstiutions`	`none`		Used to determine which Electronic Institutions are available.
`informEInstitutions`	`names`	A list of available Electronic Institutions	Used to retrieve available Electronic Institutions.

Table 7: Status Messages

Message Name	Parameters		Purpose
`tryEnterScene`	`id`	The avatar identifier	The avatar with identifier `id` moved to a room in the 3D virtual world. Its agent must move to the corresponding scene in the Electronic Institution.
	`name`	The name of the scene the avatar moved in	
`informEnterScene`	`id`	The avatar identifier	Informs the 3D virtual world whether the agent was successful in entering a scene.
	`name`	The name of the scene	
	`successful`	Boolean Variable indicating whether the avatar's agent entered the scene	
	`reason`	The reason if it could not be entered	

Table 8: 3D Action Messages

All actions that are performed by the user must be validated on the Multi Agent System level. The messages in the 3D Action Messages Table (Table 8) are used for this purpose. Suppose a user has entered a room in the 3D virtual world. His corresponding agent must enter a scene in the Ameli system. First, the 3D virtual world checks which scene is visualized in this room and sends a `tryEnterScene` message to the Connection Server. The message has the prefix `try` since it cannot be guaranteed that the Ameli system is in a state where the agent is permitted to enter the scene. Then, the middleware validates the request with Ameli. If the agent is permitted to enter the scene, the 3D virtual world is informed through an `informEnterScene` message. In this case, the `successful` parameter of the message is set true. If the agent is not permitted to enter the scene, the `successful` parameter is set false and the `error` field indicates the reason.

4.2.5 Specification of the Message Protocol

XML was used for the implementation of the message protocol (XML). It is widely recognized as a data exchange language and many programming languages provide native support for it. Another advantage of XML is XMLSchema (XMLSchema). It is used to formalize the structure and content of an XML document. This has two major implications:

- We can use this schema to verify all messages that are sent/received by the middleware. This guarantees that the middleware will always send valid messages and that only valid messages are processed.
- The schema definition defines an exact contract between the middleware and the 3D virtual world. It is exactly defined how a message has to look. This provides guidance for a 3D world developer on how to represent each message.

The schema definition is exemplified by means of the `askEInstituions` and `informEInstitutions` messages (cf. Table 8). These are used by the 3D virtual world to query the middleware about connected Electronic Institutions. The schema definition is shown in Listing 1.

Listing 1: XML schema definition for askEInstitutions and informEInstitutions messages

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified">

  <!-- Possible Direction Types (which way the message is intended to
      travel) -->
  <xsd:simpleType name="DirectionType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="cs-3d"/>
      <xsd:enumeration value="3d-cs"/>
    </xsd:restriction>
  </xsd:simpleType>

  <!-- The header of a CS3DMessage contains the Platform, Federation
      and Electronic Institution -->
```

```
<xsd:complexType name="HeaderElement">
  <xsd:sequence>
  <xsd:element name="platform" type="xsd:string"/>
  <xsd:element name="federation" type="xsd:string"/>
  <xsd:element name="ei" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<!-- askEInstitutions Messages are used for querying the accessible
     Electronic Institutions in a Federation -->
<xsd:complexType name="askEInstitutionsType">
  <xsd:attribute fixed="3d-cs" name="direction" type="DirectionType
     "/>
</xsd:complexType>

<!-- informEInstitutions Messages are sent in reply to
    askEInstitutions messages and include the connected Electronic
    Institutions in the queried Federation -->
<xsd:complexType name="informEInstitutionsType">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" name="ei" type="xsd:string"/
       >
  </xsd:sequence>
  <xsd:attribute fixed="cs-3d" name="direction" type="DirectionType
     "/>
</xsd:complexType>


<!-- The CS3DMessage contains a header and a content field with the
     possible message contents -->
<xsd:complexType name="CS3DMessageType">
    <xsd:sequence>
      <xsd:element name="header" type="HeaderElement"/>
      <xsd:element name="content">
        <xsd:complexType>
        <xsd:choice maxOccurs="1" minOccurs="1">
          <xsd:element name="askEInstitutions" type="
             askEInstitutionsType"/>
          <xsd:element name="informEInstitutions" type="
             informEInstitutionsType"/>
        </xsd:choice>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  <xsd:attribute name="in-reply-to" type="xsd:integer" use="
     optional"/>
  <xsd:attribute name="id" type="xsd:integer" use="required"/>
</xsd:complexType>

<!-- The actual definition of CS3DMessage -->
<xsd:element name="CS3DMessage" type="CS3DMessageType"/>
```

```
</xsd:schema>
```

The `CS3DMessageType` defines the complete message protocol. It has two integer attributes `id` and `in-reply-to` respectively. Every message must contain an `id` attribute and the Connection Server is responsible to generate unique message numbers for all connected 3D virtual worlds. The `in-reply-to` field is optional. It can be used in conversations where messages are exchanged based on a request/reply pattern. The `in-reply-to` attribute can be set in the reply message and refers to the message number of the request message.

The body of the message is split into a header and a content section. The header section contains three elements indicating the context (Platform, Federation and Electronic Institution) in which this message is sent. The content section contains all applicable messages that can be sent. It is restricted by a choice element with one occurrence, describing that exactly one message type is present in a message. The `askEInstitutions` message does not contain any content and only has one attribute - the direction. The `DirectionType` identifies the direction a message is supposed to be sent (either from the CS to the 3D virtual world or vice versa) and can be used as meta information when processing messages. The `informEInstitutions` message contains a list of Electronic Institutions that are accessible in the current Platform and Federation respectively. The body is made up of an unbound list of `ei` elements that contain the names of the Electronic Institutions. Note that the values of the direction attribute are different in both messages.

Instances of these message types (`askEInstitutions` and `informEInstitutions`) are shown in Listing 2. Note how the message context is specified in the header section. Furthermore the `in-reply-to` information only makes sense in the `informEInstitutions` message, where the previous message is referenced. However, the `in-reply-to` field is not restricted to message types and will be ignored by the Connection Server in those messages that do not need this information.

Listing 2: Instances of the askEInstitutions and informEInstitutions message

```
<CS3DMessage id="29">
  <header>
    <platform>platform-malls</platform>
    <federation>federation-travel</federation>
    <ei/>
  </header>
  <content>
    <askEInstitutions direction="3d-cs"/>
  </content>
</CS3DMessage>
```

```
<CS3DMessage in-reply-to="29" id="30">
  <header>
    <platform>platform-malls</platform>
    <federation>federation-travel</federation>
    <ei/>
  </header>
  <content>
    <informEInstitutions direction="cs-3d">
      <ei>ei-itchy-feet</ei>
      <ei>ei-neckermann</ei>
    </informEInstitutions>
  </content>
</CS3DMessage>
```

5 Implementation

In this chapter the implementation of the middleware components is described. In Sections 5.1 to 5.4 the implementation of the Connection Server is described. The implementation of the Torque related components is presented in Sections 5.5 - 5.7. In these Sections the class diagrams are tailored to the respective components, only the relevant methods and fields are shown.

5.1 The Connection Server

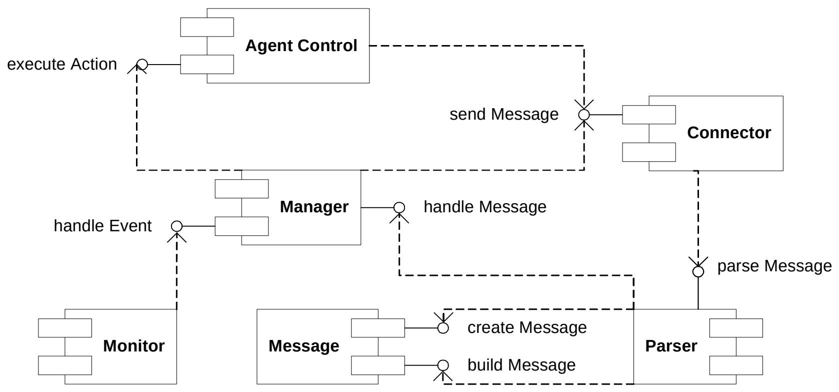


Figure 19: Component diagram of the Connection Server.

The Connection Server is implemented in Java. The different components of the Connection Server are shown in Figure 19. The *Manager* component is the heart of the application and has two major purposes. First, events received from the Ameli infrastructure (monitored by the *Monitor* component) are handled by the *Manager* component and forwarded to the 3D virtual world through the *Connector* component. Second, messages from the 3D virtual world are handled by the *manager* component and forwarded to the agents that are controlled by the *Agent Control* component. Incoming messages are received by the *Connector* component and parsed by the *Parser* component. The *Parser* uses the *Message* component to instantiate new message objects and to fill them with values received in the incoming stream. When the message has been parsed, it is forwarded to the *Manager* component.

5.2 MAS and Connection Server

The Connection Server must be informed of all events that occur in Ameli and has to decide how to handle each event. More precisely, the Connection Server applies one of the following approaches:

- *Forward the event.* The 3D virtual world must be informed of this event, so it has to be forwarded to the 3D layer. This includes, for example, the movement events of autonomous agents. The 3D virtual world is notified by means of the messages in the Ameli Action Message Table (cf. Appendix A, Table 10). These include `enterFederation`, `enterScene` or `enterTransition` messages.

- *Only use it internally.* The event has no relevance for the 3D virtual world, but the Connection Server needs to update its internal data structures. The event is processed by the Connection Server and not forwarded. This includes events on the Platform level. Since the 3D virtual world resembles a Federation (see Section 3.1.3), it does not have to know about events on this level. Examples of such events are `EnteredAgentPlatformEvent` or `StartedPlatformEvent`. The Connection Server, however, has to process these events to update its internal data structures.

- *Ignore it.* If an event just contains information that is not needed by the Connection Server nor the 3D virtual world, the event is simply ignored. This, for example, would be a `FederatedPlatformEvent`, which occurs shortly after a `StartedPlatformEvent` and contains no additional information.

5.2.1 Monitoring

The Ameli infrastructure offers the possibility to monitor events that occur in the MAS. These events can be written to a file, can be sent over a network socket or can be written to the console. Furthermore, the output format of events can be adjusted, namely to a simple text format or an XML format. In order to avoid the development of an XML or text parser, we implemented our own formatter. The standard event formatters (text and XML) can be easily substituted with other formatters. So, a formatter that serializes[3] each event and sends it to the specified channel was developed. Serialization can be easily achieved in Java, by implementing the Serializable interface. Any object, that implements this interface, can then be written on an `ObjectOutputStream` and retrieved via the `ObjectInputStream`. Since Ameli and the Connection Server are connected via TCP, we use a network socket to transmit events. Every time an event occurs in Ameli, it is serialized and is sent over the TCP connection to the Connection Server where it is deserialized and finally handled accordingly.

This mechanism resembles a very tight coupling between these two components. The Connection Server must possess knowledge of the implementation of each event. It must have access to the event class structure of the Ameli package and has to import all the necessary events. However, this tight coupling has already been introduced in the conceptual design, when we decided to use the Ameli system. No matter how we implement the Ameli event handling in the Connection Server, this always requires knowledge of the Ameli system.

[3]Serialization converts an object into a persistent form that can be stored and later retrieved by deserializing it.

5.2.2 Event handling

When retrieving a serialized object via an `ObjectInputStream`, we do not have any information on the class of this object. Java has to support serialization for all possible Java classes and, therefore, the read method of the `ObjectInputStream` can only return the most common class. Since all classes in Java are sub classes of the class `Object`, the `read` method of the `ObjectInputStream` returns this type.

How is it now possible to distinguish between different event types? The obvious solution is to query the object type and to type cast the object to a more specific type. This, however, is a dirty solution since such tasks should be delegated to the compiler (dynamic method dispatch). Meyers (1998) puts it this way: "Anytime you find yourself writing code of the form 'If the object is of type T1 do something and if it is of type T2 do something else,' slap yourself". It would be preferable to avoid this mechanism but in our situation we had no other choice. Over all, we are bound to it because we are using serialization. But there are other reasons as well that justify our decision:

- The interface to retrieve serialized objects is fixed to the common `Object` type. As long as we are using serialization we clearly have to use casts because the interface enforces it.

- Meyers gives an exception for his above statement: you should use downcasting in class hierarchies if the definition of these classes is beyond your scope of control. This applies in our case as we cannot change the class structure of Ameli events.

- In the Gang of Four (GoF) book a design pattern named *Chain of Responsibility* is presented (Gamma et al., 1995). This pattern can be used to pass a request along several handlers, giving each handler the chance to handle the request. To support an open range of requests they suggest to use a single handler function that accepts a request code as parameter. Depending on the request code, the request object is then cast to its type and handled. The only requirement is, that sender and receiver must agree on the type of request. This exactly fits our situation. The sender (Ameli) and the receiver (Connection Server) agree on the type of requests (Ameli events) and the Connection Server checks the event type, casts the event and forwards it.

- The last point represents not directly a reason for this design pattern, but should definitely be mentioned. The Ameli framework makes extensive use of this pattern. Events are passed between components in their most common form. Every time the runtime information of an object is needed, its type is tested and the object is cast to this type.

The Implementation

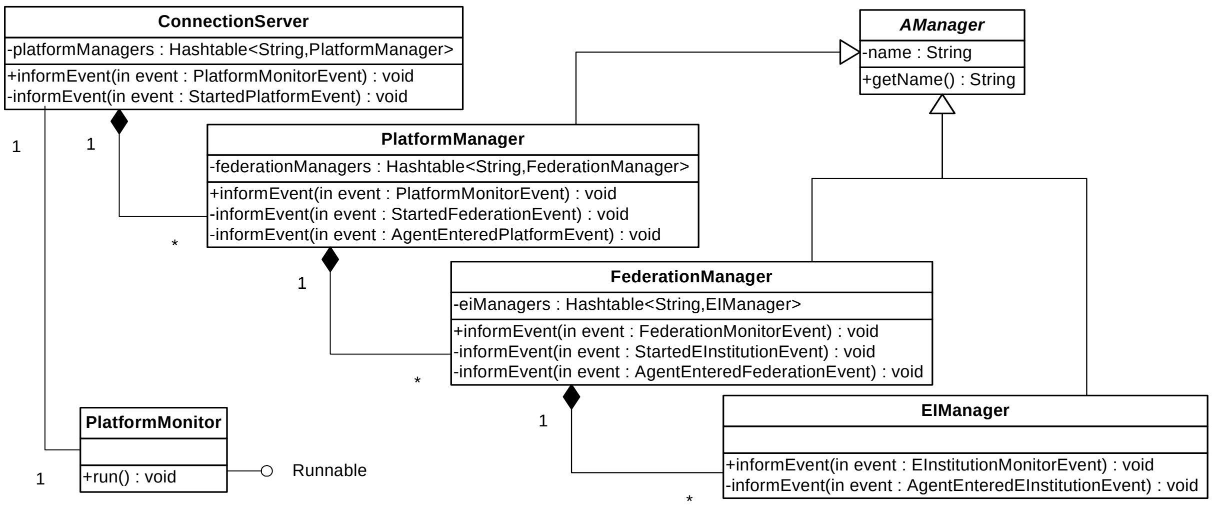

ConnectionServer
-platformManagers : Hashtable<String,PlatformManager>
+informEvent(in event : PlatformMonitorEvent) : void
-informEvent(in event : StartedPlatformEvent) : void
AManager
-name : String
+getName() : String
PlatformManager
-federationManagers : Hashtable<String,FederationManager>
+informEvent(in event : PlatformMonitorEvent) : void
-informEvent(in event : StartedFederationEvent) : void
-informEvent(in event : AgentEnteredPlatformEvent) : void
FederationManager
-eiManagers : Hashtable<String,EIManager>
+informEvent(in event : FederationMonitorEvent) : void
-informEvent(in event : StartedEInstitutionEvent) : void
-informEvent(in event : AgentEnteredFederationEvent) : void
EIManager
+informEvent(in event : EInstitutionMonitorEvent) : void
-informEvent(in event : AgentEnteredEInstitutionEvent) : void
PlatformMonitor
+run() : void
Runnable
1
1
*
1
*
1
*
1

Figure 20: Event handling in the Connection Server.

The class structure of the event handling component of the Connection Server is depicted in Figure 20. In Section 3.1 it was pointed out that Ameli is organized according to three levels - Platform, Federation and Electronic Institution level. The classes in the Connection Server are also organized according to this scheme. For every different level there are managers that handle the connection on this level. For each started Platform a `PlatformManager` is launched, for each started Federation a `FederationManager` is launched and for each started Electronic Institution an `EIManager` is launched. The level above the Platform managers is represented by the main control class of the Connection Server, the `ConnectionServer`[4] class. The event reception is done by the `PlatformMonitor` class. The Ameli system is configured in such a way that it sends all Platform events to a specified host and port. The `PlatformMonitor` listens for connections on the same host and port. When the Ameli infrastructure is started it establishes a connection with the `PlatformMonitor`. The `PlatformMonitor` receives all events as serialized objects, deserializes them and passes them on to the `ConnectionServer` class.

The event is then examined by each manager. First the event type is checked. If the event can be handled by the manager it is cast to its runtime type and handled by the manager. Otherwise the Platform, Federation or Electronic Institution information of the event is examined. If there exists a manager on the next level for this Platform,Federation or Electronic Institution, the event is forwarded to this manager. Otherwise the event cannot be handled and a warning message[5] is issued.

[4]Henceforth the term Connection Server refers to the whole middleware component, whereas the term ConnectionServer refers to the concrete class in the implementation.

[5]As stated before, the Connection Server must not be able to process all events. Some events may be ignored. This is why we only issue a warning and do not raise an error.

The following example is used to clarify this mechanism. Suppose an `EnteredFederation` event occurs in Ameli. All this happens in the Platform "ShoppingMall" and the Federation that was entered is called "TouristShops". The `ConnectionServer` receives the event, realizes that it cannot handle it, searches for a link to a `PlatformManager` named "ShoppingMall" and forwards the event to this manager. The `PlatformManager` does the same and forwards the event to the `FederationManager` named "TouristShops". Finally the `FederationManager` handles the event.

The implementation of this message handling process is based on the *Chain of Responsibility* pattern in the GoF book (Gamma et al., 1995). The process is exemplified by the Federation level event handling in the Platform manager (cf. Figure 20). The `instanceof` operator of the Java language is used to query the runtime type of the event. All events that can be handled by the `PlatformManager` are cast to their type and handled by overloaded methods in the `PlatformManager`. The `FederationMonitorEvent` is especially interesting, because the `PlatformManager` cannot directly pass all these events to the Federation managers. If the event is of type `StartedFederationEvent`, the `PlatformManager` has to handle this event because it has to spawn a new `FederationManager` for the new Federation. The `RegisteredFederationEvent` can be ignored because it contains no additional information than the `StartedFederationEvent`. All other `FederationMonitorEvents` are passed on to the `FederationManager` in question.

5.3 Connection Server Message Structure

The connection between the Connection Server and Torque is based on the message protocol that was specified in Section4.2.5. The message protocol allows to abstract the connected 3D virtual world. Communication between the Connection Server and Torque is realized by means of a TCP connection.

The message structures are basically the same in both systems (Connection Server and Torque). The Torque message structure, however, is somehow a mirrored version of the Connection Server message structure. For this reason we do not cover both structures in one Section and present them separately. The Torque message structure is presented in Section 5.7. The class hierarchy of the Connection Server is displayed in Figure 21. The abstract class `AMessage` is at the top of the hierarchy. It contains attributes and functions that all messages have in common. The `MessageContext` is a simple container class that essentially stores the header information of each message. The name "context" was chosen because the message header identifies the context (Platform, Federation or Electronic Institution) in which this message is placed.

The next two child classes differentiate the messages by communication direction. The `A3DCSMessage` is the base class for all messages that are received from the 3D virtual world

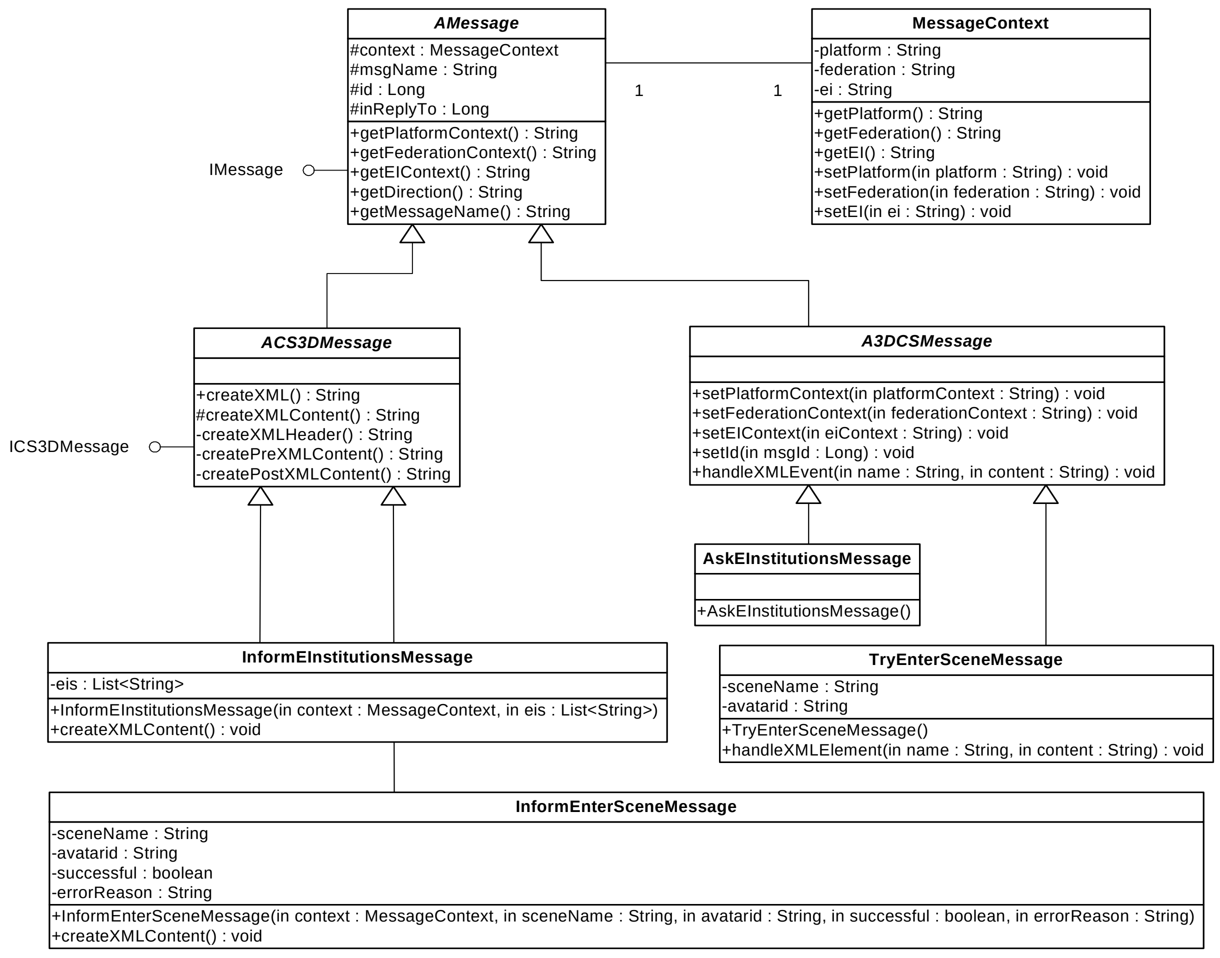

AMessage
#context : MessageContext
#msgName : String
#id : Long
#inReplyTo : Long
+getPlatformContext() : String
+getFederationContext() : String
+getEIContext() : String
+getDirection() : String
+getMessageName() : String
IMessage
MessageContext
-platform : String
-federation : String
-ei : String
+getPlatform() : String
+getFederation() : String
+getEI() : String
+setPlatform(in platform : String) : void
+setFederation(in federation : String) : void
+setEI(in ei : String) : void
1
1
ACS3DMessage
+createXML() : String
#createXMLContent() : String
-createXMLHeader() : String
-createPreXMLContent() : String
-createPostXMLContent() : String
ICS3DMessage
A3DCSMessage
+setPlatformContext(in platformContext : String) : void
+setFederationContext(in federationContext : String) : void
+setEIContext(in eiContext : String) : void
+setId(in msgId : Long) : void
+handleXMLEvent(in name : String, in content : String) : void
AskEInstitutionsMessage
+AskEInstitutionsMessage()
InformEInstitutionsMessage
-eis : List<String>
+InformEInstitutionsMessage(in context : MessageContext, in eis : List<String>)
+createXMLContent() : void
TryEnterSceneMessage
-sceneName : String
-avatarid : String
+TryEnterSceneMessage()
+handleXMLElement(in name : String, in content : String) : void
InformEnterSceneMessage
-sceneName : String
-avatarid : String
-successful : boolean
-errorReason : String
+InformEnterSceneMessage(in context : MessageContext, in sceneName : String, in avatarid : String, in successful : boolean, in errorReason : String)
+createXMLContent() : void

Figure 21: Message structure in the Connection Server.

and the `ACS3DMessage` is the base class for all messages that are sent by the Connection Server to the 3D virtual world. As can be seen in the diagram, their interfaces are completely different because they are processed in different ways.

`A3DCSMessages` are received from the 3D virtual world and need to be parsed by the Connection Server. The detailed procedure for this task is presented in the next section. During the parsing, message objects are constructed and their attributes have to be filled with values from the XML stream. For this reason the `A3DCSMessage` class contains setter methods for setting the context and id information. Note that those setters are only relevant for this message type and do not need to be placed in the `ACS3DMessage` class. The `handleXMLElement` method is used to handle message specific XML elements. It can be overwritten by subclasses for the implementation of individual parsing procedures. The parameters of the method contain the name of the element and its content. Consider, for example, a message of type `tryEnterScene`[6] which includes an avatar identifier and a scene name. In the implementation the `TryEnterSceneMessage` overrides the

[6]tryEnterScene messages are used whenever an avatar in the 3D virtual world attempts to enter a room (that is mapped onto a scene). The message contains the identifier of this avatar and the scene the avatar wants to enter.

`handleXMLElement` method and can set its internal variables `sceneName` and `avatarid` to the values that are specified in the avtarid and scene XML elements. Note that it is not required for a message to override this method. The `AskEInstitutionsMessage` does not contain any content. It must therefore not implement any specific element handling procedures and does not need to override the `handleXMLElement` method.

The other message type, `ACS3DMessage`, represents messages that are sent by the Connection Server. The Connection Server must, therefore, be capable of constructing an XML string for these messages. The public interface contains just the `createXML` method, which creates and returns the XML string of this message. It can be used by the sender class to get the XML code that needs to be sent. The only part of an XML message that is specified by subclasses of `ACS3DMessage` is the content of the message. For this reason the `createXMLContent` is declared protected and can be overwritten by subclasses.

Consider, for example, the `InformEInstitutionsMessage` which contains a list of currently available Electronic Institutions. In order to transfer this information into the XML string, the class overwrites the `createXMLContent` method where it creates a String containing this information. So, each message class is responsible for the creation of the correct XML content string. The other message in Figure 21, `InformEnterSceneMessage`, contains four internal variables. These variables are used to store information that is sent via this message. In the overwritten `createXMLContent` method it is specified how this information is mapped onto the XML string.

5.3.1 Message Parsing

Parsing XML data with Java is straight forward. Sun provides the Java API for XML processing (JAXP) that defines an API specification for the parsing and processing of XML documents. The API is widespread and many different implementations exist. It is, therefore, possible to change the parser without modifying the client code. Nevertheless, the designer of the XML data processing structure must decide between two different modes of operation that are supported by JAXP. These two modes are called SAX and DOM and are reviewed in the following.

The SAX API is one of the most complete and correct XML APIs so far. It is based on an event driven approach - every time an element is encountered, a callback function for processing the element is called. This makes SAX very fast and memory efficient since no data structures for storing the XML document are needed. SAX is suitable for streaming applications where the elements of an XML document are processed one after another. If the application needs to access different sections of the XML document, SAX is the wrong choice.

The Document Object Model is contrary to the SAX approach. As the name implies this API builds a complete model of the XML document. The document is stored in a tree structure where the single XML elements are represented as nodes of this tree. The structure can then be traversed by the client. With this approach arbitrary sections of a document can be accessed and the document can be modified. In contrast to SAX, these features require higher memory usage and longer processing times. Every time a new XML document is encountered the tree must be build internally and kept in memory.

The Implementation

In our application the choice for the right parser was fairly easy. We receive the XML data via a network stream and it is sufficient to process one element after another (we do not need arbitrary access to the message). So, the SAX API is the right choice for the Connection Server. As stated above the SAX API is event driven and provides a callback interface that is used to implement application specific callback handlers.

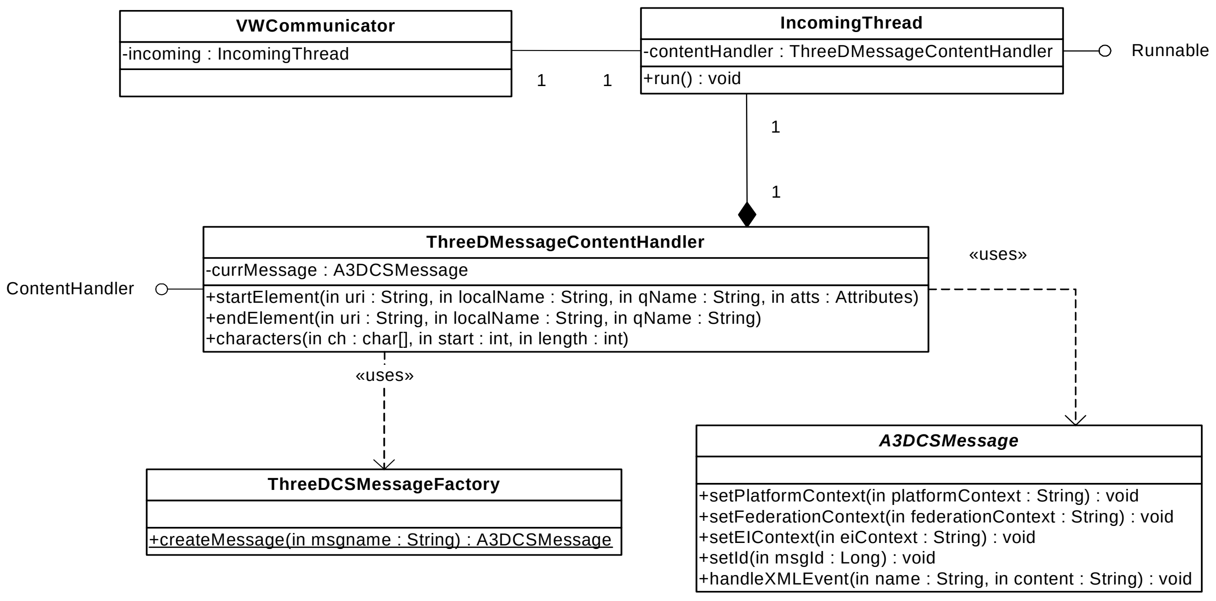

VWCommunicator
-incoming : IncomingThread
1
1
IncomingThread
-contentHandler : ThreeDMessageContentHandler
+run() : void
Runnable
1
1
ThreeDMessageContentHandler
-currMessage : A3DCSMessage
+startElement(in uri : String, in localName : String, in qName : String, in atts : Attributes)
+endElement(in uri : String, in localName : String, in qName : String)
+characters(in ch : char[], in start : int, in length : int)
ContentHandler
«uses»
«uses»
ThreeDCSMessageFactory
+createMessage(in msgname : String) : A3DCSMessage
A3DCSMessage
+setPlatformContext(in platformContext : String) : void
+setFederationContext(in federationContext : String) : void
+setEIContext(in eiContext : String) : void
+setId(in msgId : Long) : void
+handleXMLEvent(in name : String, in content : String) : void

Figure 22: Message construction in the Connection Server.

The callback interface is implemented in the `ThreeDMessageContentHandler` shown in Figure 22. The content handler is connected to the incoming XML stream by means of the `IncomingThread` class. Depending on the current information in the XML stream, callback methods are called in the `ThreeDMessageContentHandler`. The whole parsing procedure is implemented in three callback methods. As the name implies, the `startElement` method is called each time a new element is encountered, the `endElement` is called when the end of an element is reached and the `characters` callback is used to process data in the elements. In our case this works as follows: when a new message is received a new message object is instantiated. The several fields of the object are set using the setter methods specified in the `A3DCSMessage` class. When the content section of the message is reached, the overloaded `handleXMLContent` method is called and message specific attributes are set. When the end of the message is reached, the message object is forwarded to the message handler (which will be explained in the next section). During this procedure the `startElement` and `endElement` methods are used to switch the different parser states and to call the appropriate setters. The `characters` method simply records the data in the elements.

One question remains in this mechanism - how are the different message instances generated? As we are using Java we can take advantage of a very powerful feature, namely *Reflection*. Reflection allows a computer program to observe its own state and behavior and to access different parts of the control structure. So, it is possible to discover information about the fields, methods and constructors of loaded classes. We are interested in the access to constructors during runtime, since this gives us the possibility to create object instances based on some parameters. We can instantiate any class based on a string representation of the classes name.

The `createMessage` method of the `ThreeDCSMessageFactory` is used by the message parser to create new message instances. The message name provided as parameter is simply the name of the message in the XML string. We have decided to name the Java message classes according to the names in the protocol definition. Since Java class names have to start with an upper case letter, the naming in the protocol and in the implementation differ in the first letter. For this reason we have to adapt the received XML message name. To instantiate an object we then have to get the class information for this object. This class information is loaded at the startup of the Java virtual machine and can be accessed through the class `Class`. A call to the `newInstance` method returns an instance of the respective message class. Since we use the same construction procedure for all message classes, we have to cast the returned type to the `A3DCSMessage` class (as this represents the base class that all implementing message classes must be a subclass of).

5.3.2 Message Handling

Now that we have seen how the messages are constructed, the Connection Server needs to handle these messages and take proper actions. We could incorporate the same mechanism as we did with the events (Section 5.2) but this approach is not the best one. Our main goal for the message handling component was to avoid this pattern and to find a better solution.

The main problem in developing a better a solution than the casting approach is the interface of the message creator. We have seen in Figure 22 that the `createMessage` method returns the super class of all concrete 3DCS message classes, namely the `A3DCSMessage` class. This has two reasons:

- We do not want to adapt the message factory each time we add a new message type. This implies that we can only return the common super class, which must be inherited by all message classes.

- The content handler (`ThreeDMessageContentHandler`) should not need any information of the message classes. We do not want to modify this class every time we create new message classes and it is simply not necessary that the parsing procedure has access to the message specific functions. The parser only needs those methods that are defined in the abstract class `A3DCSMessage`.

After the message parsing has finished we run into the following situation: The parser forwards the message to the message handlers but those message handlers (in our case the managers) want to access the individual fields and methods of each message class (the managers have information of the message class implementations). Since the message parser "only knows" of the `A3DCSMessage`, it can only forward this type. In order to provide access for the managers to the runtime types, one solution would be the type checking and casting approach we wanted to avoid. We will now show how we could avoid this situation with a different design.

This solution does not rely on runtime object type checking and casting and is based on the *Visitor* pattern (Gamma et al., 1995). The design can be seen in Figure 23. It is essentially a combination of the last two class diagrams of Figures 20 and 21. Again only the relevant methods and fields are displayed.

With the help of the Visitor pattern we delay the casting procedure to each concrete message class implementation. This is done with the `accept` method in the `A3DCSMessage` class. This method is declared abstract, meaning that every sub class must implement this method. The parameter of the method is the `IMessageHandler` interface which defines message handling methods and can be compared with the `IVisitor` interface of the Visitor pattern. The code in the `accept` method is the same for every message class:

```java
public void accept(IMessageHandler handler) {
  handler.handleMessage(this);
}
```

With these methods we can get the runtime type of the message object without casting. The `accept` method can be called on objects with type `A3DCSMessage` and due to polymorphism, the appropriate method is then called in the respective message object. The object then hands itself to the `IMessageHandler` interface.

In our implementation the message handling is statically defined. We know at compile time which message types are handled by which manager type. What we do not know is which manager instance will handle which concrete message instance. This information is only available at runtime. Consider, for example, the `TryEnterSceneMessage` which must be handled by the `EIManager` class. This information is available at compile time, but we do not know in advance which `EIManager` instance will handle which

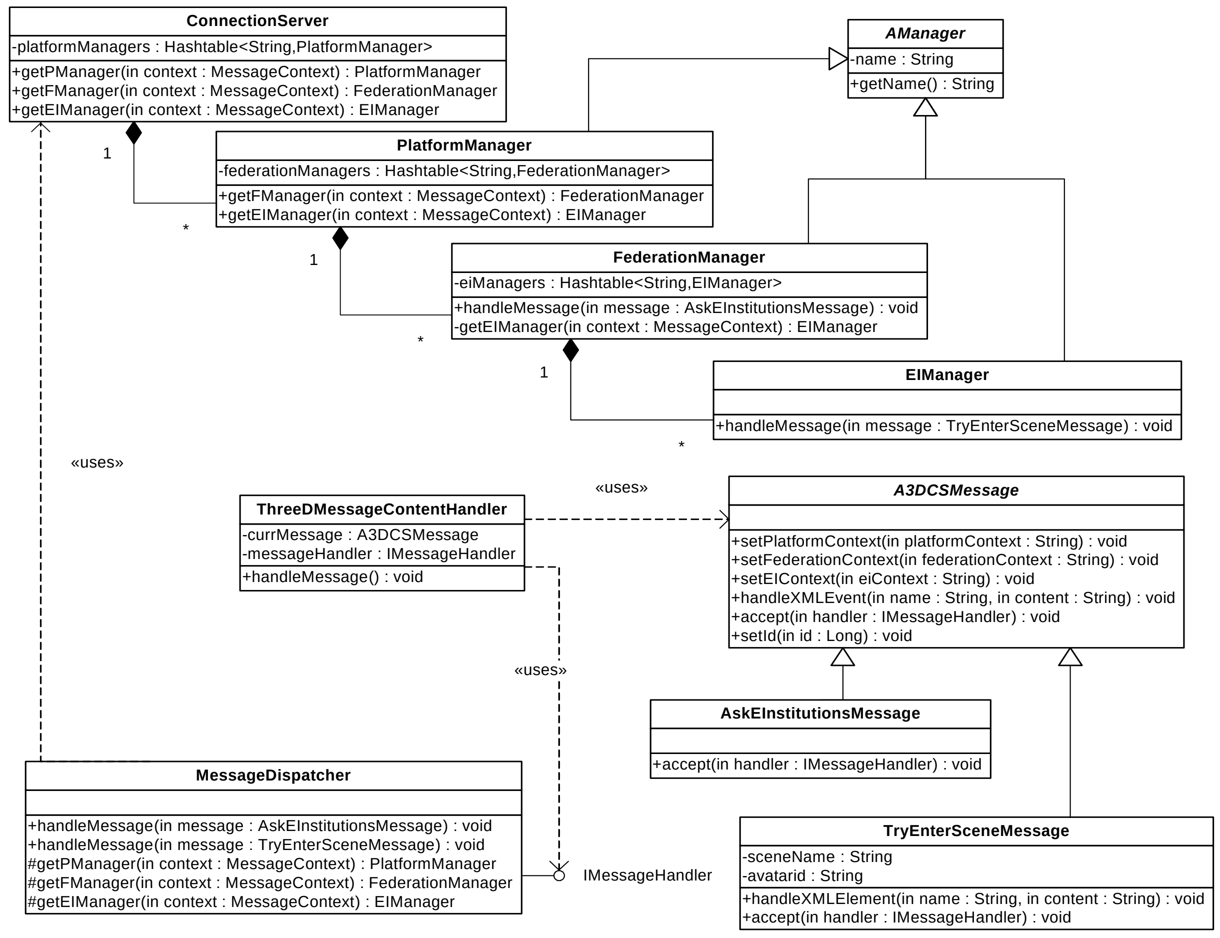

ConnectionServer
-platformManagers : Hashtable<String,PlatformManager>
+getPManager(in context : MessageContext) : PlatformManager
+getFManager(in context : MessageContext) : FederationManager
+getEIManager(in context : MessageContext) : EIManager
AManager
-name : String
+getName() : String
PlatformManager
-federationManagers : Hashtable<String,FederationManager>
+getFManager(in context : MessageContext) : FederationManager
+getEIManager(in context : MessageContext) : EIManager
FederationManager
-eiManagers : Hashtable<String,EIManager>
+handleMessage(in message : AskEInstitutionsMessage) : void
-getEIManager(in context : MessageContext) : EIManager
EIManager
+handleMessage(in message : TryEnterSceneMessage) : void
1
*
«uses»
ThreeDMessageContentHandler
-currMessage : A3DCSMessage
-messageHandler : IMessageHandler
+handleMessage() : void
A3DCSMessage
+setPlatformContext(in platformContext : String) : void
+setFederationContext(in federationContext : String) : void
+setEIContext(in eiContext : String) : void
+handleXMLEvent(in name : String, in content : String) : void
+accept(in handler : IMessageHandler) : void
+setId(in id : Long) : void
AskEInstitutionsMessage
+accept(in handler : IMessageHandler) : void
MessageDispatcher
+handleMessage(in message : AskEInstitutionsMessage) : void
+handleMessage(in message : TryEnterSceneMessage) : void
#getPManager(in context : MessageContext) : PlatformManager
#getFManager(in context : MessageContext) : FederationManager
#getEIManager(in context : MessageContext) : EIManager
IMessageHandler
TryEnterSceneMessage
-sceneName : String
-avatarid : String
+handleXMLElement(in name : String, in content : String) : void
+accept(in handler : IMessageHandler) : void

Figure 23: Message handling in the Connection Server.

`TryEnterSceneMessage` instance during runtime. This information is contained in the message context and only available at runtime.

For this reason we have developed a `MessageDispatcher` that implements the `IMessageHandler` interface and dispatches the messages to the concrete managers. The `MessageDispatcher` contains a handling method for each message type which is called by the message objects in the `accept` method. The `MessageDispatcher` then needs to pass on the message to the concrete manager. It must therefore have knowledge of all running managers. In the current implementation the manager hierarchy is queried each time a message needs to be dispatched. This is done with the `getPManager`, `getFManager` and `getEIManager` methods that return a `PlatformManager`, a `FederationManager` and an `EIManager` respectively. Those methods are implemented in the `ConnectionServer` and in the managers. Depending on the message type, the dispatcher calls one of these methods to get the appropriate manager instance (this can be done due to the compile time knowledge of the message type ⇔ manager type relation). The appropriate manager is identified through the message context and returned to the `MessageDispatcher`. The message is then forwarded to this manager.

Finally, we want to summarize the advantages of this approach: we do not need to rely on type checking (`instanceof`) and we do not need to cast objects anymore. Furthermore, we obtain compile time safety when adding new message types. This is achieved in the following way:

- When we add a new message type we have to implement the `accept` method (compile time error otherwise).
- When we implement the `accept` method correctly we have to extend the `IMessage-Handler` interface (compile time error otherwise).
- When we extend the interface we have to adapt the `MessageDispatcher` (compile time error otherwise).
- When we implement the handling method correctly we must implement the handling method in the appropriate manager type also (compile time error otherwise).

In the casting solution this is not guaranteed, since we could forget to add the handler code inside the `informEvent` methods. This error can only be detected at runtime and may lead to confusion when testing the program.

5.3.3 Message Transmission

Subclasses of the `ACS3DMessage` class are used to inform the 3D virtual world of state changes in Ameli. Whenever an Ameli event is monitored in the Connection Server, it is forwarded to the respective manager and handled by it (cf. Section 5.2.2). If the 3D virtual world needs to be informed of this event, the manager constructs a new message object and sends it to 3D virtual world. Consider, for example, the `EnteredAgentFederationEvent`. Such an event occurs when an agent entered a Federation in Ameli. If the respective agent is an autonomous agent, the Connection Server needs to construct and send an `EnterFederationMessage` to the 3D virtual world. The action is then visualized in the 3D virtual world by spawning a new avatar for this agent.

The message transmission is achieved with the help of the `VWCommunicator` class, see Figure 24. The `VWCommunicator` provides an interface for the transmission of `ACS3DMes-sages`. The `createXML` method of the `ACS3DMessage` is used by the `VWCommunicator` to retrieve the XML string that needs to be sent to the 3D virtual world. Figure 24 shows that a `VWCommunicator` is either associated with the `ConnectionServer` class or with one `FederationManager`. This is due to the fact that one 3D virtual world visualizes one Federation (cf. Section 4.1). When a new 3D virtual world is launched, it establishes a connection with the Connection Server via the `VWCommunicator` class. As long as the 3D virtual world has not decided which Federation it will visualize, the respective `VWCommunicator` is associated with the `ConnectionServer` class. The `launchedVirtualWorldMessage` (cf.

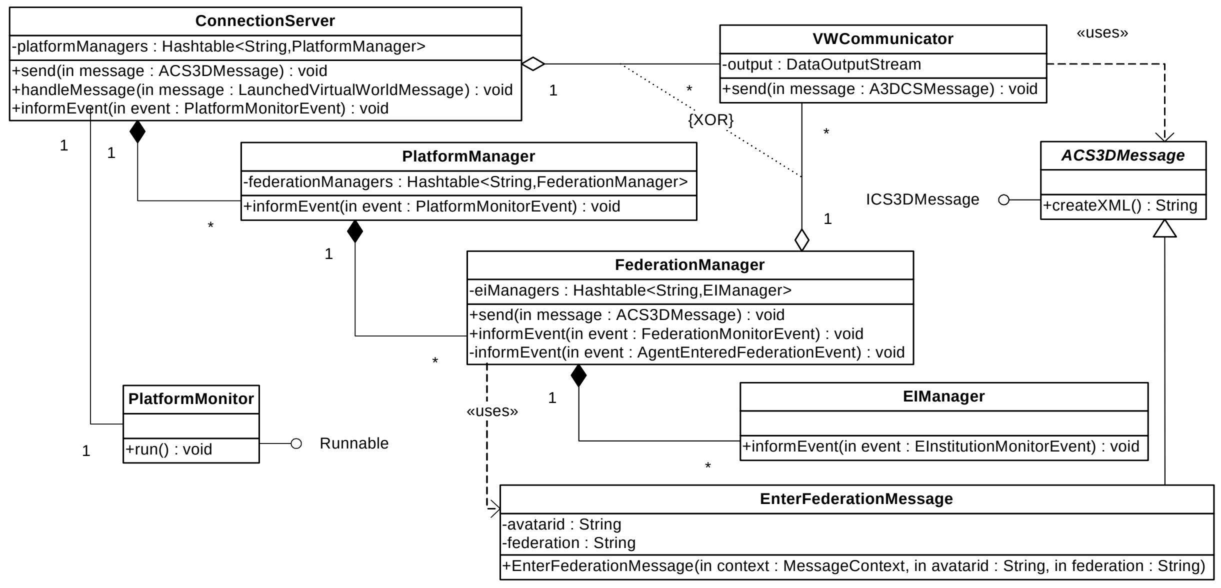

ConnectionServer
-platformManagers : Hashtable<String,PlatformManager>
+send(in message : ACS3DMessage) : void
+handleMessage(in message : LaunchedVirtualWorldMessage) : void
+informEvent(in event : PlatformMonitorEvent) : void
VWCommunicator
-output : DataOutputStream
+send(in message : A3DCSMessage) : void
«uses»
{XOR}
ACS3DMessage
+createXML() : String
ICS3DMessage
PlatformManager
-federationManagers : Hashtable<String,FederationManager>
+informEvent(in event : PlatformMonitorEvent) : void
FederationManager
-eiManagers : Hashtable<String,EIManager>
+send(in message : ACS3DMessage) : void
+informEvent(in event : FederationMonitorEvent) : void
-informEvent(in event : AgentEnteredFederationEvent) : void
PlatformMonitor
+run() : void
Runnable
«uses»
EIManager
+informEvent(in event : EInstitutionMonitorEvent) : void
EnterFederationMessage
-avatarid : String
-federation : String
+EnterFederationMessage(in context : MessageContext, in avatarid : String, in federation : String)
1
*
1
1
*
1
*
1
*
1
1
*

Figure 24: Message Transmission in the Connection Server.

Appendix A, Table 9) is used to inform the Connection Server that a 3D virtual world is now visualizing a particular Federation. When such an event arrives in the Connection Server, the `VWCommunicator` is handed to the respective `FederationManager`. From this moment on, the message transmission to this 3D virtual world is executed through the `FederationManager`. The `EIManagers` within this `FederationManager` use the `send` method to forward Electronic Institution level messages to the 3D virtual world.

5.4 Agent Control in the Connection Server

Human users participating in the framework via the 3D virtual world must be represented as agents in the Ameli system. The *Agent Control* component defines an interface for the creation of new agents and the control of these agents. It is based on the *DummyAgent* package from the EIDE environment (cf. Section 4.2.1) and is used by the *Manager* component. In Figure 25 the relationship between the classes is shown.

The `AgentManager` provides the interface for the control of the agents and is used by all manager types. Whenever a new user connects to the 3D virtual world a `TryEnterPlatformMessage` is triggered and sent to the Connection Server. The message is handled by the `ConnectionServer` class which calls the `launchNewAgent` and `enterPlatform` methods. The `PlatforManager` and `FederationManager` call the `enterFederation` and `enterEI` methods when a `tryEnterFederation` or `tryEnterEI` message is received. The `EIManager` calls the `enterScene` method whenever a `TryEnterScene` message arrives.

The `DummyAgentController` is the interface to the reused parts of the *DummyAgent* package. The `AgentManager` uses this interface to forward the requests (stated by the managers) to the particular agents. The `AgentManager` essentially hides the agent infor-

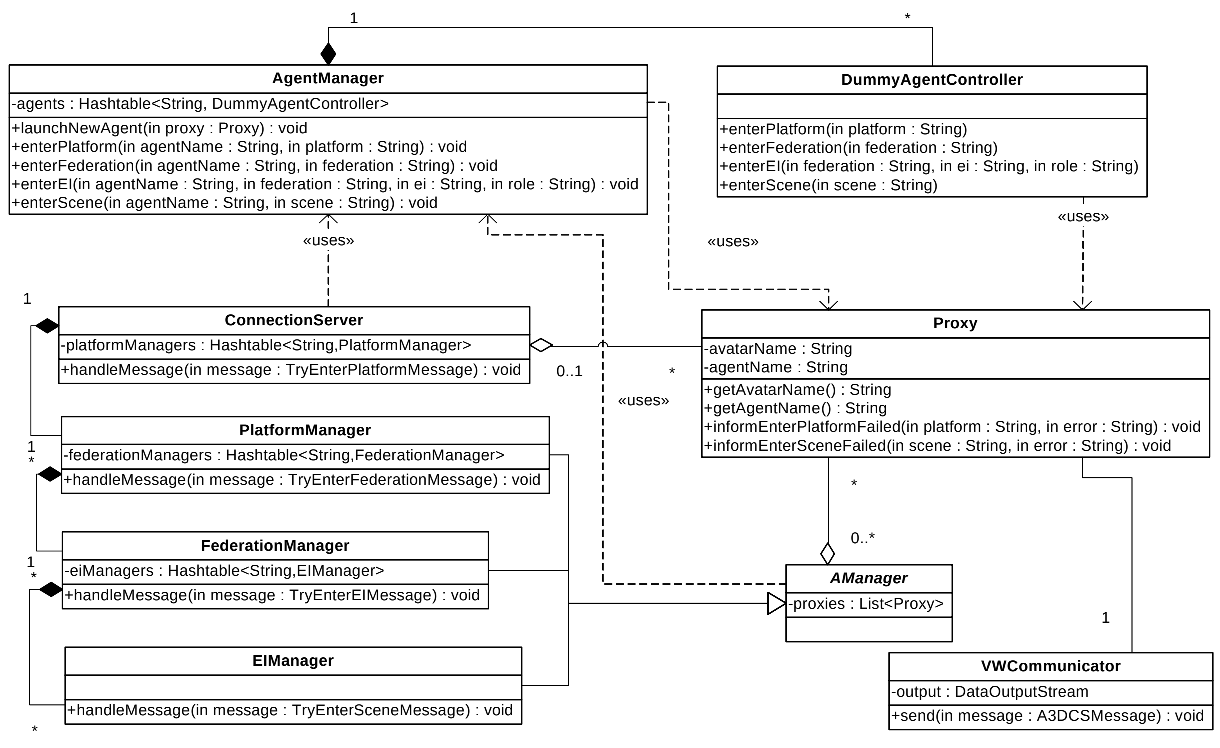

AgentManager
-agents : Hashtable<String, DummyAgentController>
+launchNewAgent(in proxy : Proxy) : void
+enterPlatform(in agentName : String, in platform : String) : void
+enterFederation(in agentName : String, in federation : String) : void
+enterEI(in agentName : String, in federation : String, in ei : String, in role : String) : void
+enterScene(in agentName : String, in scene : String) : void
DummyAgentController
+enterPlatform(in platform : String)
+enterFederation(in federation : String)
+enterEI(in federation : String, in ei : String, in role : String)
+enterScene(in scene : String)
«uses»
ConnectionServer
-platformManagers : Hashtable<String,PlatformManager>
+handleMessage(in message : TryEnterPlatformMessage) : void
PlatformManager
-federationManagers : Hashtable<String,FederationManager>
+handleMessage(in message : TryEnterFederationMessage) : void
FederationManager
-eiManagers : Hashtable<String,EIManager>
+handleMessage(in message : TryEnterEIMessage) : void
EIManager
+handleMessage(in message : TryEnterSceneMessage) : void
Proxy
-avatarName : String
-agentName : String
+getAvatarName() : String
+getAgentName() : String
+informEnterPlatformFailed(in platform : String, in error : String) : void
+informEnterSceneFailed(in scene : String, in error : String) : void
AManager
-proxies : List<Proxy>
VWCommunicator
-output : DataOutputStream
+send(in message : A3DCSMessage) : void
1
*
0..1
0..*

Figure 25: Agent Control in the Connection Server.

mation from the `Platform-`, `Federation-` and `EIManagers`. As can be seen by the function signatures, the managers only provide the name of the agent and the `AgentManager` dispatches the requests to the particular agents.

The `Proxy` class facilitates the mapping between avatars and agents. The managers use this class to store the avatar/agent couples that are currently acting in the Platform, Federation or Electronic Institution in question. Furthermore, the `Proxy` also severs another purpose. In the original version all Ameli events were intended to be processed over the *Monitor* component. During the course of implementation we realized that not all events are sent to this monitor. Some events, such as failed enter attempts, are only sent to the particular agent that stated the request. Therefore, the *Agent Control* component must also be able to send messages to the 3D virtual world. To this end, the `Proxy` class provides functions for the transmission of messages. The `DummyAgentController` uses these functions to inform the 3D virtual world of failed requests.

5.5 The Game Engine

In Section 3.2.1 the Torque Game Engine was introduced. Remember that the editors are used to create the content of the 3D virtual world, the scripting language is used for simple tasks and more complex structures and behavior are implemented in C++. A Torque application is usually split into the game engine executable (the compiled C++ code) and a script language code base. The examples on the Web page are based on this structure (Torque). The script language code base and the executable are strongly coupled: the script code invokes functions in the game engine executable and the C++ code invokes functions that are defined in the script code.

The script code is compiled and evaluated at runtime. This makes script code development fast compared to the development in C++ (where the code has to be compiled and linked after modification). The main disadvantage is the dynamic binding of the scripting language. Variable types are not specified in the code and all type errors are detected at runtime when the code is processed. Furthermore, the scripting language does not provide object oriented features. In contrast, C++ offers compile time safety and provides powerful language constructs. Thus, most of our code is implemented in C++.

Similar to the official examples, our implementation is split into a C++ code base (which is compiled into the game engine executable) and a script code base (which is evaluated during runtime). The 3D virtual world runs on a dedicated server and users connect to it through clients that are executed on their machines (cf. Section 3.2.1).

The dedicated server is responsible for the message exchange with the Connection Server and guarantees consistency in terms of visualizing all the Ameli system. The code on the client side is responsible for the interaction with the users. The dedicated server and some functions on the client side are implemented in C++. The remaining code on the client side is implemented in Torque Script. The components of the C++ code base are displayed in Figure 26.

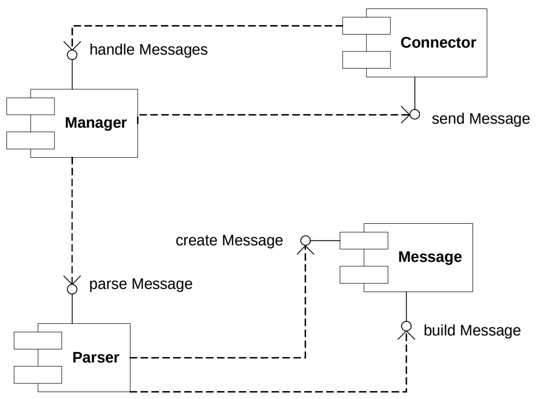


Figure 26: Torque Components of the C++ code base.

The *Message* component is similar to its counterpart in the Connection Server. It provides a message structure to represent messages that are exchanged with the Connection Server. Two interfaces, one for the creation of new message instances and one for the message building are provided. The *Parser* component uses these two interfaces to instantiate new message objects and to fill the objects with values received in the XML stream. The *Manager* component represents the main control structure on the Torque side. It is responsible for the message exchange with the Connection Server. Incoming messages are received through the *Connector* component, parsed by the *Parser* component and handled by the *Manager* component. Outgoing messages are constructed by the *Manager* component and sent to the Connection Server through the *Connector* component. Moreover, the *Manager* component also controls the state of the 3D virtual world, moves avatars in the world and verifies the movement of the users.

5.6 Game Engine Managers

The *Manager* component is split into two different types of managers, see Figure 27. The `Manager` is responsible for messages exchanged on the Platform and Federation level and provides an interface for the script code base. The `EIManager` defines an abstract base class for the implementation of Electronic Institution managers. For each visualized Electronic Institution one such manager must be implemented. This class is then responsible for the messages that are sent within this Electronic Institution and for the actions that are performed in the visualized building. In Figure 27 there exists one concrete manager implementation, named `TravelManager`, which handles all messages concerning the Travel Electronic Institution (cf. Section 4.1). This manager is responsible for the consistent relationship of the Travel Electronic Institution with the Ameli system.

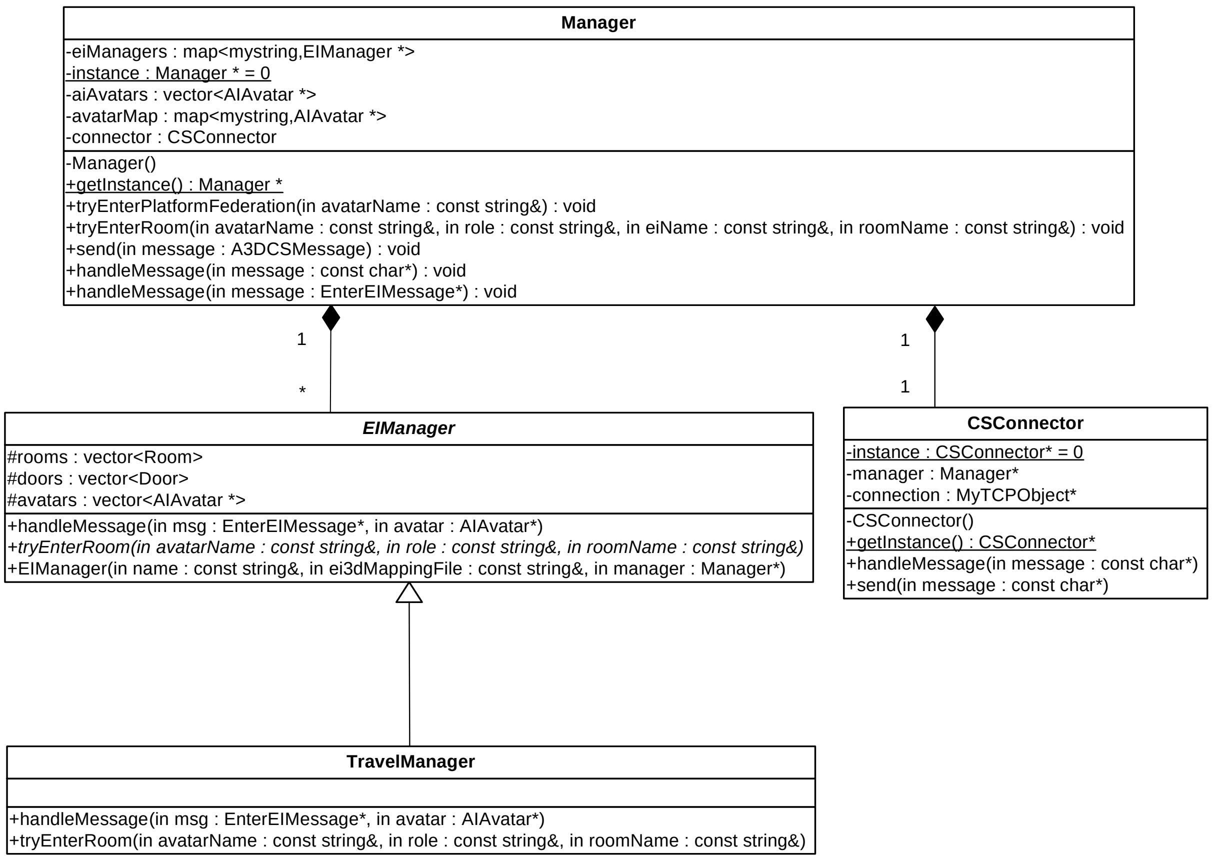

Manager
-eiManagers : map<mystring,EIManager *>
-instance : Manager * = 0
-aiAvatars : vector<AIAvatar *>
-avatarMap : map<mystring,AIAvatar *>
-connector : CSConnector
-Manager()
+getInstance() : Manager *
+tryEnterPlatformFederation(in avatarName : const string&) : void
+tryEnterRoom(in avatarName : const string&, in role : const string&, in eiName : const string&, in roomName : const string&) : void
+send(in message : A3DCSMessage) : void
+handleMessage(in message : const char*) : void
+handleMessage(in message : EnterEIMessage*) : void
1
*
1
1
EIManager
#rooms : vector<Room>
#doors : vector<Door>
#avatars : vector<AIAvatar *>
+handleMessage(in msg : EnterEIMessage*, in avatar : AIAvatar*)
+tryEnterRoom(in avatarName : const string&, in role : const string&, in roomName : const string&)
+EIManager(in name : const string&, in ei3dMappingFile : const string&, in manager : Manager*)
CSConnector
-instance : CSConnector* = 0
-manager : Manager*
-connection : MyTCPObject*
-CSConnector()
+getInstance() : CSConnector*
+handleMessage(in message : const char*)
+send(in message : const char*)
TravelManager
+handleMessage(in msg : EnterEIMessage*, in avatar : AIAvatar*)
+tryEnterRoom(in avatarName : const string&, in role : const string&, in roomName : const string&)

Figure 27: Torque Managers.

When an autonomous agent enters the Ameli system, the `Manager` spawns a new avatar in the 3D virtual world. It further controls the movement of this avatar in the 3D virtual world. The script code, running on the client side, uses the `Manager` to state requests to the Ameli system. Consider, for example, a user trying to connect to a 3D virtual world. The script code contacts the dedicated server and the `tryEnterPlatformFederation` method is called. The `Manager` states the request to the middleware. The reply is processed by the `Manager` which in turn informs the client by calling a function in the script code. If the user's agent was allowed to enter the Ameli system on both levels, the user is actually logged in the 3D virtual world.

Actions that happen on the Electronic Institution level, such as entering a scene, are forwarded to the appropriate Electronic Institution manager. Consider, for example, a user that is standing inside a building that represents the Travel Electronic Institution. The user wants to enter a room and presses the appropriate key that triggers this action. The request is posted to the `TravelManager` (via the interface defined in the `EIManager`) and handled in the overloaded `tryEnterRoom` function. Then the request is forwarded to the middleware and the response is again processed by the `TravelManager`. Message reception and transmission is achieved through the `CSConnector` class used by the `Manager`. The concrete managers use the `Manager` to send messages to the middleware. Incoming messages are also received by the `Manager` object and forwarded to the appropriate manager implementation.

5.7 Game Engine Message Structure

The message structure in Torque is based on the same model as the message structure in the Connection Server (cf. Section 5.3 and Figure 21). As can be seen in Figure 28, the mechanism remains the same, but this time the `A3DCSMessage` provides the interface for message construction and the `ACS3DMessage` provides the interface for message building. Another difference is the representation of the message context. In the Connection Server the message context is encapsulated in its own class (class `MessageContext` in Figure 21). In the Torque code this information is directly encoded into the `AMessage` class. The functionality remains the same, but the signatures of the message class constructors differ.

5.7.1 Message Parsing

Conceptually, the message parsing is accomplished similar to the procedure used in the Connection Server (cf. Section 5.3.1). The actual process is depicted in Figure 29. The `Manager` receives new messages through the `CSConnector` and uses the `CS3DMessagePar-ser` to parse these messages. The `MessageFactory` creates new message objects and the interface of the `ACS3DMessage` is used by the parser to build the messages. There are two main differences compared to the process in the Connection Server which are presented in the following.

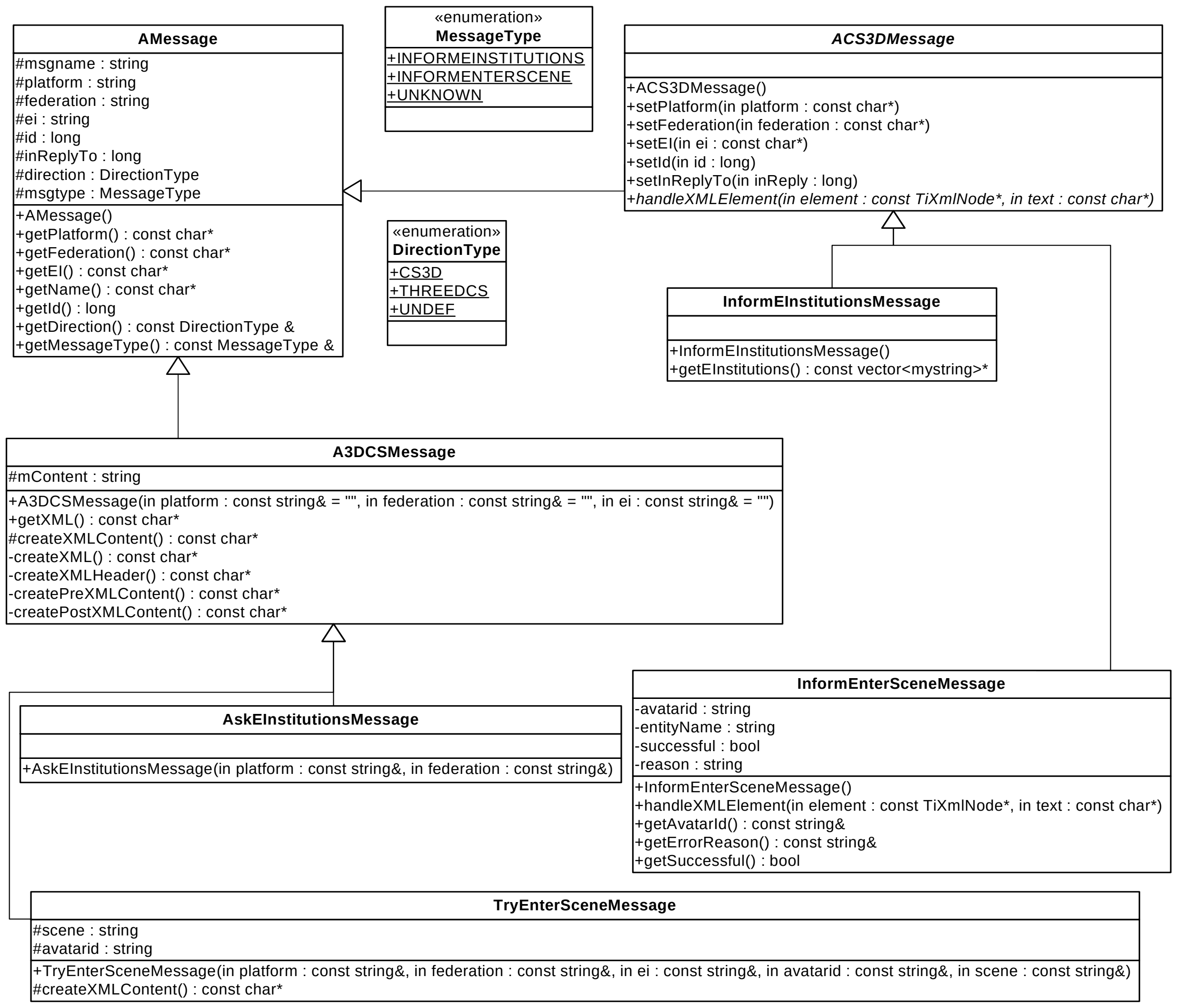

AMessage
#msgname : string
#platform : string
#federation : string
#ei : string
#id : long
#inReplyTo : long
#direction : DirectionType
#msgtype : MessageType
+AMessage()
+getPlatform() : const char*
+getFederation() : const char*
+getEI() : const char*
+getName() : const char*
+getId() : long
+getDirection() : const DirectionType &
+getMessageType() : const MessageType &
«enumeration» MessageType
+INFORMEINSTITUTIONS
+INFORMENTERSCENE
+UNKNOWN
«enumeration» DirectionType
+CS3D
+THREEDCS
+UNDEF
ACS3DMessage
+ACS3DMessage()
+setPlatform(in platform : const char*)
+setFederation(in federation : const char*)
+setEI(in ei : const char*)
+setId(in id : long)
+setInReplyTo(in inReply : long)
+handleXMLElement(in element : const TiXmlNode*, in text : const char*)
InformEInstitutionsMessage
+InformEInstitutionsMessage()
+getEInstitutions() : const vector<mystring>*
A3DCSMessage
#mContent : string
+A3DCSMessage(in platform : const string& = "", in federation : const string& = "", in ei : const string& = "")
+getXML() : const char*
#createXMLContent() : const char*
-createXML() : const char*
-createXMLHeader() : const char*
-createPreXMLContent() : const char*
-createPostXMLContent() : const char*
AskEInstitutionsMessage
+AskEInstitutionsMessage(in platform : const string&, in federation : const string&)
InformEnterSceneMessage
-avatarid : string
-entityName : string
-successful : bool
-reason : string
+InformEnterSceneMessage()
+handleXMLElement(in element : const TiXmlNode*, in text : const char*)
+getAvatarId() : const string&
+getErrorReason() : const string&
+getSuccessful() : bool
TryEnterSceneMessage
#scene : string
#avatarid : string
+TryEnterSceneMessage(in platform : const string&, in federation : const string&, in ei : const string&, in avatarid : const string&, in scene : const string&)
#createXMLContent() : const char*

Figure 28: Message structure on the Torque level.

In the Connection Server message objects are instantiated with the reflection mechanism of the Java language. The C++ language does not provide such a feature and we solved this task with the Factory pattern (Gamma et al., 1995). A straightforward implementation assigns each object type a unique ID. A factory method is then used to instantiate new objects based on the ID provided as parameter. This is exemplified in Listing 3.

Listing 3: A straight forward implementation of the Factory pattern

```
enum ObjectType { SPRODUCT; CPRODUCT }

Product
Factory::create(ObjectType id)
{
  switch(id)
  {
  case SPRODUCT:
    return new SimpleProduct();
    break;
  case CPRODUCT:
    return new ComplexProduct();
    break;
  default:
    // error unknown object type
  }
}
```

Alexandrescu (2001) points out the disadvantages of this approach: All knowledge about product classes is stored in one source file. This leads to compile time dependencies because all header files of all products must be included. Furthermore, it is difficult to extend the factory, because every time a new product class is created, a new unique ID must be assigned and the switch statement must be extended.

Alexandrescu provides an implementation of the Factory pattern that overcomes these problems. Instead of having a single switch statement for the creation of new objects, this code is deferred to the implementation of each product class. Each product class implements a method for the creation of new product objects and registers this method in the factory. When the factory creates a new product object, it dispatches the creation method of this product type and calls it.

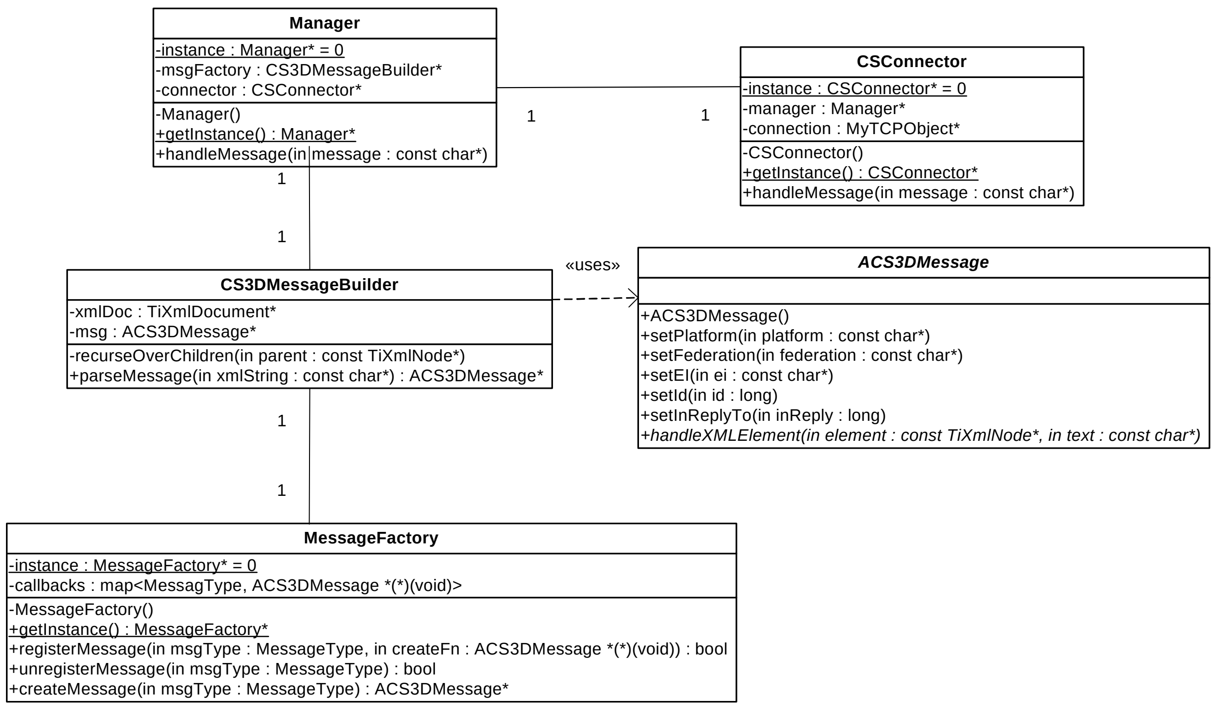

Manager
-instance : Manager* = 0
-msgFactory : CS3DMessageBuilder*
-connector : CSConnector*
-Manager()
+getInstance() : Manager*
+handleMessage(in message : const char*)
1
1
CSConnector
-instance : CSConnector* = 0
-manager : Manager*
-connection : MyTCPObject*
-CSConnector()
+getInstance() : CSConnector*
+handleMessage(in message : const char*)
1
1
«uses»
CS3DMessageBuilder
-xmlDoc : TiXmlDocument*
-msg : ACS3DMessage*
-recurseOverChildren(in parent : const TiXmlNode*)
+parseMessage(in xmlString : const char*) : ACS3DMessage*
ACS3DMessage
+ACS3DMessage()
+setPlatform(in platform : const char*)
+setFederation(in federation : const char*)
+setEI(in ei : const char*)
+setId(in id : long)
+setInReplyTo(in inReply : long)
+handleXMLElement(in element : const TiXmlNode*, in text : const char*)
1
1
MessageFactory
-instance : MessageFactory* = 0
-callbacks : map<MessagType, ACS3DMessage *(*)(void)>
-MessageFactory()
+getInstance() : MessageFactory*
+registerMessage(in msgType : MessageType, in createFn : ACS3DMessage *(*)(void)) : bool
+unregisterMessage(in msgType : MessageType) : bool
+createMessage(in msgType : MessageType) : ACS3DMessage*

Figure 29: Message parsing on the Torque level.

We use this approach for the creation of new message objects. The `MessageFactory` is used to register new creation methods and to call these methods. Every message class (subclasses of `ACS3DMessage`) defines a function for the creation of new message objects of this type. At startup this function is registered in the `MessageFactory` via the `registerMessage` method. The registration adds a new function pointer for this `MessageType` to the `callbacks` map. New message objects can then be created by calling the `createMessage` function. Using the callback map, the appropriate creation function for the passed `MessageType` is called.

The second difference is the XML processor. In the Connection Server we use the SAX API to process XML documents. On the Torque side we use the TinyXml library for XML parsing (TXML). It is a simple, small and minimal C++ XML parser that can be easily integrated into other programs. Similar to the DOM API, the TinyXml library represents each XML document in a tree like structure. This structure is stored in C++ objects and can be traversed by the client code. As shown in Figure 29 we use the `TiXmlNode` class to traverse the XML document and to retrieve the element information.

5.7.2 Message Handling

The message handling procedure is completely different to the approach used in the Connection Server. The approach is similar to the Ameli event handling in Section 5.2.2 - message objects are distinguished based on their type and cast to the appropriate runtime type.

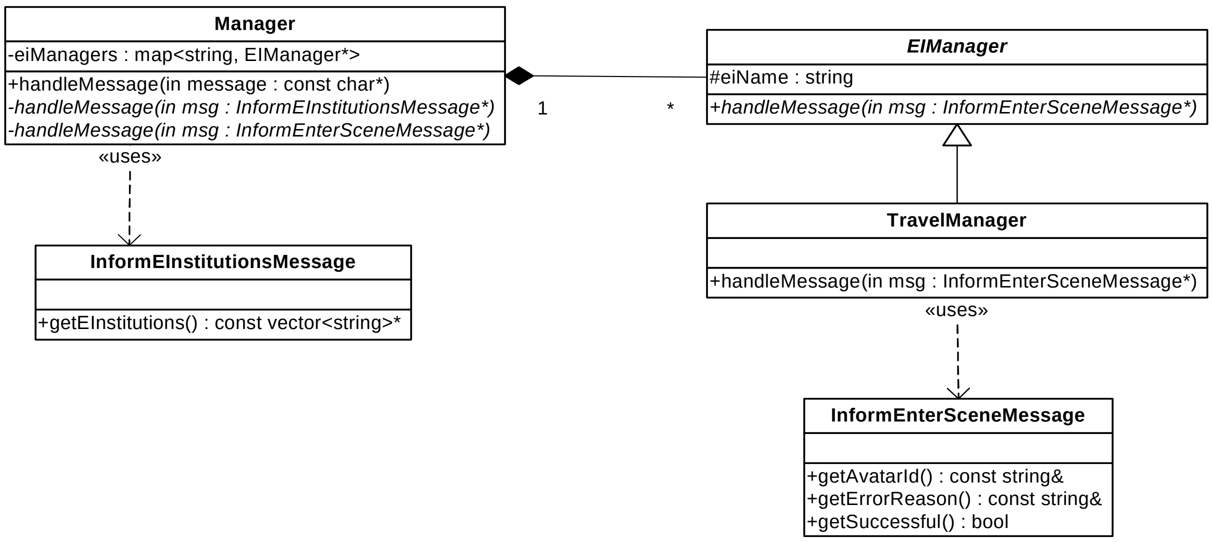

Manager
-eiManagers : map<string, EIManager*>
+handleMessage(in message : const char*)
-handleMessage(in msg : InformEInstitutionsMessage*)
-handleMessage(in msg : InformEnterSceneMessage*)
«uses»
InformEInstitutionsMessage
+getEInstitutions() : const vector<string>*
1
*
EIManager
#eiName : string
+handleMessage(in msg : InformEnterSceneMessage*)
TravelManager
+handleMessage(in msg : InformEnterSceneMessage*)
«uses»
InformEnterSceneMessage
+getAvatarId() : const string&
+getErrorReason() : const string&
+getSuccessful() : bool

Figure 30: Message handling on the Torque side.

The process of message handling in the current implementation is displayed in Figure 30. Messages are handled as follows. First, the `Manager` creates a new message object using the `CS3DMessageBuilder` (cf. Figure 29). Then, the `handleMessage` method detects the object type at runtime via the `MessageType` information. The message object is cast to its runtime type and the appropriate overloaded `handleMessage` method is called. In the case of the `InformEInstitutionsMessage` the `Manager` handles the message itself (the contained information is stored in an internal data structure). The

`InformEnterSceneMessage` (sent in the context of an Electronic Institution) cannot be handled by the `Manager` and is forwarded to the appropriate `EIManager`. If a user tries to enter a scene in the Travel Electronic Institution a `TryEnterSceneMessage` is sent to the Connection Server. The Connection Server replies with an `InformEnterInstitutionMessage` which is handled by the `TravelManager`. Finally, the `TravelManager` informs the user of the outcome of the request by calling a function in the script code on the client side.

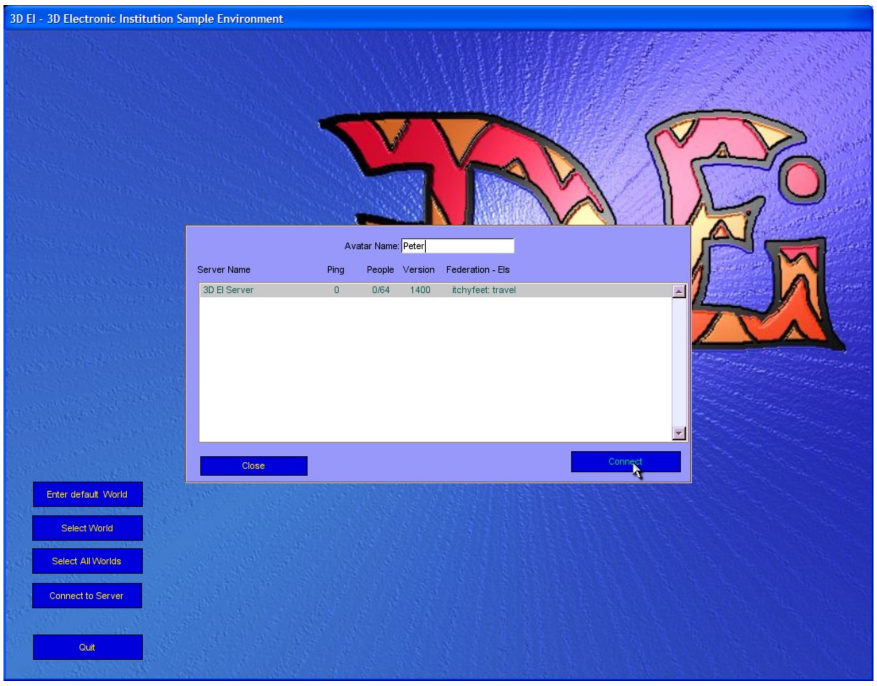
5.8 Testing

In order to assess the reliability of the system, we employed a couple of tests. In the context of the Connection Server, these tests are realized as Unit tests. We only implemented tests for the *Parser* and the *Message* component, because writing tests for the other components would have been too time consuming - testing the core components (*Manager*, *Agent Control*) requires to simulate the Ameli system and the 3D virtual world. The functionality of those components was verified during an estimated 100 hours of system execution.

The functionality of the *Parser* and the *Message* component was tested through functional unit tests. The message construction is tested by comparing the produced XML code with predefined XML messages. The *Parser* component is tested in the same way: predefined XML messages act as input to the parser and the attributes of the constructed messages are compared with the values in the XML messages.

Integrating unit tests in the Torque code was not possible as there exists no built in Unit testing framework. However, we used an approach that is encouraged by the Torque developer community. To verify the correct functionality of the code, *assert* statements are inserted into it. The *assert* statement allows the developer to compare the runtime value of a variable with a predefined value. If those values are unequal, an error is raised. We extensively use assertions in the *Parser* component and in the XML processing routines. All incoming messages must correspond to the proposed message structure and the correct structure of XML configuration files is also verified. In the other components assertions are mainly used to check the proper initialization of data structures or the values of function parameters. In analogy to the Connection Server, the functionality of the Torque components was tested through execution of the system.

6 Verifying the Interplay of the Components

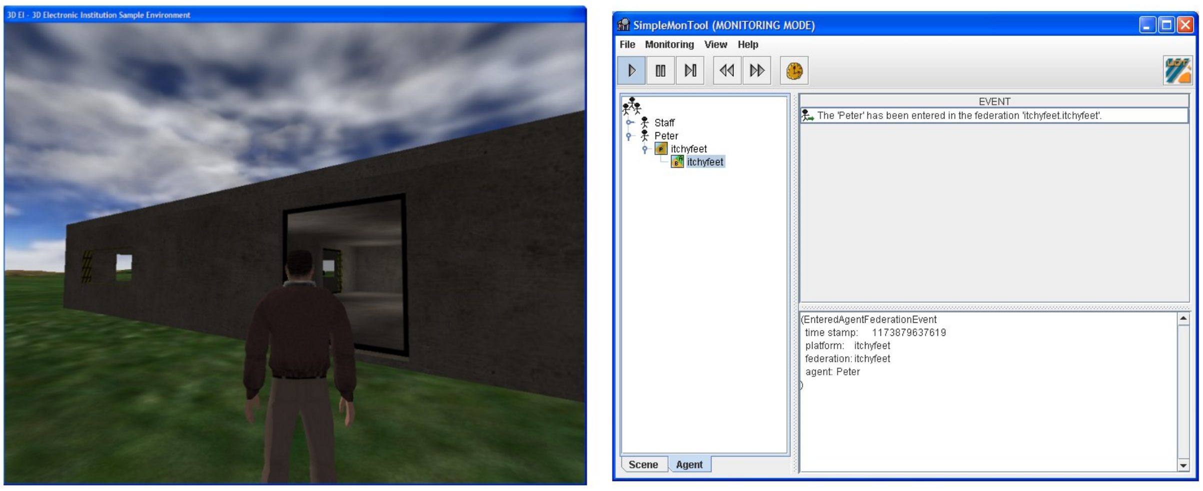


Avatar Name: Peter

Server Name	Ping	People	Version	Federation - Els
3D El Server	0	0/64	1400	itchyfeet: travel

Figure 31: The User Interface of the Torque client.

In order to test the functionality of the system and, thus, to verify the smooth interplay of the components, a prototypical 3D visualization has been designed and implemented. Figure 31 depicts the graphical user interface that is displayed after starting the Torque client. The user is able to query available servers and to connect to them. In this particular case there is one server available. This server visualizes the Travel Electronic Institution that was presented in Section 4.1. The user can choose a login name and is able to connect to this server by clicking on the “Connect” button.

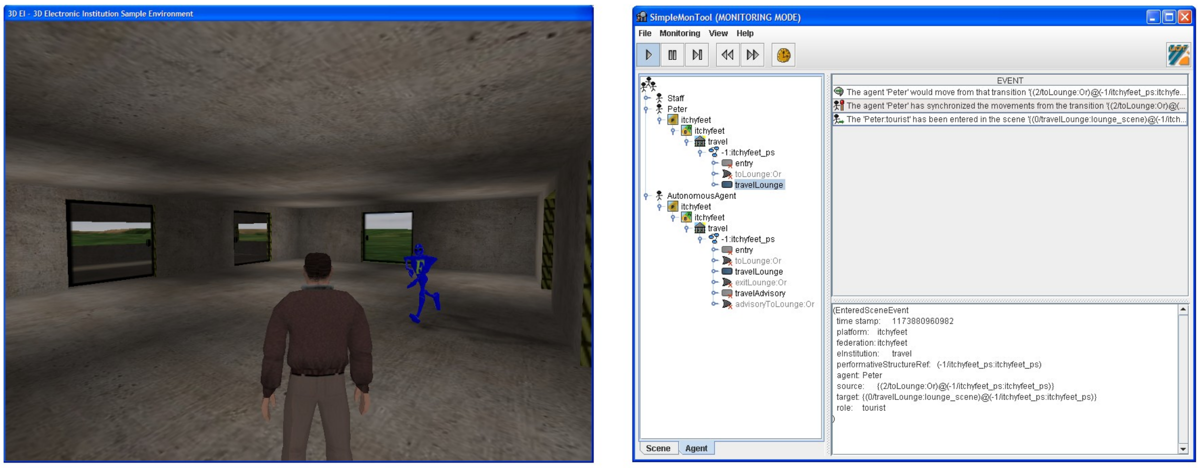

3D EI - 3D Electronic Institution Sample Environment
SimpleMonTool (MONITORING MODE)
File
Monitoring
View
Help
Staff
Peter
itchyfeet
itchyfeet
EVENT
Scene
Agent

The 'Peter' has been entered in the federation 'itchyfeet.itchyfeet'.

```
(EnteredAgentFederationEvent
 time stamp:    1173879637619
 platform:   itchyfeet
 federation: itchyfeet
 agent: Peter
)
```

Figure 32: The user entered the Federation.

After the user has clicked the "Connect" button, a series of events are happening in the background. The Torque Server contacts the middleware which spawns a new agent. The agent then tries to enter the Ameli system on the Platform and Federation level. If the agent has successfully entered the Federation level, the user is allowed to enter the 3D virtual world. Otherwise the user will not be able to participate in this world. Figure 32 shows the screen of the user after he successfully entered the 3D virtual world. The right side of this figure displays an agent monitoring tool of the Ameli system. The tool shows which agents are currently playing in the system. As can be seen in this screenshot, a new agent, representing the user, has been entered in the "itchyfeet" Federation.


3D EI - 3D Electronic Institution Sample Environment
SimpleMonTool (MONITORING MODE)
File
Monitoring
View
Help
Scene
Agent

```
Staff
Peter
  itchyfeet
    itchyfeet
      travel
        -1:itchyfeet_ps
          entry
          toLounge:Or
          travelLounge
AutonomousAgent
  itchyfeet
    itchyfeet
      travel
        -1:itchyfeet_ps
          entry
          toLounge:Or
          travelLounge
          exitLounge:Or
          travelAdvisory
          advisoryToLounge:Or
```

```
EVENT
The agent 'Peter' would move from that transition '{(2/toLounge:Or)@(-1/itchyfeet_ps:itchyfe...
The agent 'Peter' has synchronized the movements from the transition '{(2/toLounge:Or)@(...
The 'Peter:tourist' has been entered in the scene '{(0/travelLounge:lounge_scene)@(-1/itch...
```

```
(EnteredSceneEvent
 time stamp:    1173880960982
 platform:   itchyfeet
 federation: itchyfeet
 eInstitution:     travel
 performativeStructureRef:   (-1/itchyfeet_ps:itchyfeet_ps)
 agent: Peter
 source:     {(2/toLounge:Or)@(-1/itchyfeet_ps:itchyfeet_ps)}
 target: {(0/travelLounge:lounge_scene)@(-1/itchyfeet_ps:itchyfeet_ps)}
 role:    tourist
)
```

Figure 33: The user entered the Travel Electronic Institution.

The user is now standing in front of a building. This building is used to visualize the Travel Electronic Institution and consists of three rooms that are arranged according to the floor plan in Figure 18. The user can enter the Travel Electronic Institution and the Traveler's Lounge through the door. If he states the request to open the door, a series of events is triggered. The user's agent tries to enter the Electronic Institution, the Initial scene, the first transition and the Traveler's Lounge scene. The user is only allowed to enter the room if all those entities could successfully be entered by his agent. This is the case in the next screenshot, displayed in Figure 33. The agent entered the Traveler's Lounge scene in the Travel Electronic Institution and the user is standing in the mapped room in the 3D virtual world. The doors on the left and right side lead to the other two scenes Travel Advisory and Travel Agency respectively.

In the same Figure an autonomous agent can also be seen. This agent entered the Ameli system via another interface and is now playing in the Electronic Institution. The monitoring tool shows that the agent entered the Travel Electronic Institution, moved to the Travel Advisory and then came back to the Traveler's Lounge. The 3D virtual world visualizes this agent with a blue avatar and is responsible for the movement of this avatar in the 3D world.

7 Conclusion & Future Work

This master thesis is embedded within a research project that has the principal goal of developing an instrument to support the complex interaction patterns of providers and consumers in an e-Tourism setting. In particular, these providers and consumers, either humans or software agents, are members of a heterogeneous society cohabiting in a multi-agent based 3D virtual environment. Conceptually speaking, the environment is designed according to a three-layered architecture comprising a Multi Agent System layer, a middleware layer and a 3D visualization layer.

The first major contribution of this master thesis was the design of the middleware connecting the 3D game engine with the Multi Agent System. This included considerations regarding the design of the middleware, protocol definitions, as well as the conceptualization of the middleware architecture. The second major contribution was the actual implementation of the middleware and the design and prototypical implementation of a 3D virtual world. We were able to test the functionality of the middleware with this 3D virtual world and could verify the smooth interplay of the components. The system behaved as expected and the interplay between the Multi Agent System and the 3D virtual world could directly be observed. The correct visualization of agents could be verified in the 3D virtual world and the correct representation of users in the Multi Agent System was verified with the help of monitoring tools.

The future work can be split into conceptual design considerations and implementation improvements. Causality, for example, is one of these conceptual design considerations. In the current version we only guarantee a consistent relationship, but do not enforce causality in its strict manner. At the moment, inconsistencies are resolved by teleporting avatars to other locations. Although this approach might be annoying to users, we think that such situations will not occur frequently in practice. This claim, however, needs to be investigated by testing the system in a real world setting. Another approach would be to aim at the implementation of a causal connection which can be achieved by modifying the Ameli system.

The next design consideration deals with the coupling between the 3D virtual world and the Multi Agent System. Currently, these components are strongly coupled - the 3D virtual world has knowledge of the organization of the Multi Agent System. The message protocol reflects this relationship and messages such as `tryEnterScene` or `enterTransition` must be understood by the 3D virtual world. In Section 4.1 we mentioned that this situation is caused by the placement of the mapping information. In future research we aim at abstracting this message protocol, in order to make the 3D virtual world independent from the used Multi Agent System. With this approach it will be possible to switch Multi Agent Systems.

Considering the implementation, future work is also required. First of all the system needs to be tested more thoroughly. At present, the system was only executed by a few people who did not test the functionality. We will define testing procedures and test cases that are executed by testers who never run the system before. These tests will reveal hidden errors and will detect further requirements that are desired by the users.

The message handling in the Torque component, which is currently based on type checking, will be improved in future versions. We plan to implement the same message handling procedure that is used in the Connection Server.

In Section 5.4 we pointed out that not all events are sent to the Connection Server over the *Monitor* component. As we discovered this problem during the implementation, additional functions had to be added and the event processing was split up into several components. Some events are processed by the *Manager* component, others by the *Agent Control* component. This processing behavior needs to be harmonized in future versions.

A The Message Protocol

Message Name	Parameters		Purpose
`askPlatforms` `askFederations` `askEInstitutions`	None		Ask which Platforms, Federations or Electronic Institutions are available
`informPlatforms` `informFederations` `informEInstitutions`	`names`	A list of available Platforms, Federations or Electronic Institutions	Inform which Platforms, Federations or Electronic Institutions are available
`askEnterTransition` `askEnterScene`	`id`	The avatar identifier	Ask if the avatar with identifier `id` is allowed to enter a transition or scene.
	`name`	The name of the transition or scene	
`informEnterTransition` `informEnterScene`	`id`	The avatar identifier	The avatar with identifier `id` is allowed/not allowed to enter a transition or scene.
	`transition`	The name of the transition or scene	
	`allowed`	Boolean (true/false) indicating weather the agent is allowed to enter	
	`reason`	The reason if it is not allowed to enter	
`askPossibleMessages`	`id`	The avatar identifier	Ask which messages may be uttered by the avatar with identifier `id` in a scene.
	`scene`	The name of the scene in which the avatar resides	
`informPossibleMessages`	`id`	The avatar identifier	Inform which messages may be uttered by the avatar with identifier `id`.
	`scene`	The name of the scene	
	`messages`	The messages that may be uttered by the avatar	
`launchedVirtualWorld`	`federation`	Which federation the virtual world is visualizing	Inform the Connection Server that a 3D virtual world is now visualizing a particular Federation.

Table 9: Status Messages

Message Name	Parameters		Purpose
`enterFederation` `enterEI`	`id`	The avatar identifier	An agent entered a Federation or Electronic Institution, visualize it in the 3D virtual world.
`enterTransition` `enterScene`	`id`	The avatar identifier	The avatar with identifier `id` must move to a transition or scene in the 3D virtual world.
	`name`	The name of the transition or scene	
`joinToScene`	`ids`	The avatar identifiers	The specified avatars with identifiers in list `ids` must move together to a scene in the 3D virtual world.
	`scene`	The name of the scene	
`sayMessage`	`message`	The message to be said	A scene message was uttered in the Electronic Institution and the 3D virtual world must visualize it.

`exitFederation`	`id`	The avatar identifier	An agent exited a Federation, visualize it in the 3D virtual world.
	`federation`	The name of the Federation	

Table 10: Ameli Action Messages

Message Name	Parameters		Purpose
`tryEnterTransition` `tryEnterScene`	`id`	The avatar identifier	The avatar with identifier `id` wants to move to a transition or scene.
	`transition`	The name of the transition	
`informEnterTransition` `informEnterScene`	`id`	The avatar identifier	Inform the 3D virtual world whether the avatar could successfully enter the transition or scene.
	`name`	The name of the transition or scene	
	`successful`	Boolean Variable indicating whether the avatar entered the transition or scene	
	`reason`	The reason if it could not be entered	
`trySayMessage`	`id`	The avatar identifier	The avatar with identifier `id` wants to say a message in a scene. Inform the Electronic Institution of it.
	`scene`	The name of the scene	
	`message`	The message that was said	
`informMessage`	`id`	The avatar identifier	Inform the 3D world whether the avatar's agent uttered the message in the Electronic Institution or not.
	`scene`	The name of the scene	
	`message`	The message that was said	
	`successful`	Boolean variable indicating whether the message could be uttered	
	`reason`	The reason if it was not possible to say this message	
`tryEnterPlatform` `tryEnterFederation` `tryEnterEI`	`id`	The avatar identifier	The avatar with identifier `id` wants to enter a Platform, Federation or Electronic Institution.
	`name`	The name of the Platform, Federation or Electronic Institution	
`informEnterPlatform` `informEnterFederation` `informEnterEI`	`id`	The avatar identifier	Inform the 3D world whether the avatar's agent could successfully enter a Platform, Federation or Electronic Institution.
	`name`	The name of the Platform, Federation or Electronic Institution	
	`successful`	Boolean Variable indicating whether the avatar's agent could be entered	
	`reason`	The reason if it could not be entered	
`exitPlatform` `exitFederation` `exitEI`	`id`	The avatar identifier	Tell Ameli that the avatar with ID `id` did exit a Platform, Federation or Electronic Institution.
	`name`	The name of the Platform, Federation or Electronic Institution	

Table 11: 3D Action Messages

Message Name	Parameters		Purpose
`invalidMessage`	`message`	The invalid message	Whenever an invalid message is sent by the 3D virtual world, this is the reply.
	`reason`	The reason why this message is invalid	

Table 12: Error Messages

List of Figures

List of Tables

References

3DMaze. `http://en.wikipedia.org/wiki/3d_monster_maze`

Adobbati, R., Marshall, A. N., Scholer, A., Tejada, S., Kaminka, G. A., Schaffer, S. and Sollitto, C. (2001). Gamebots: a 3d virtual world test-bed for multi-agent research, *Proceedings of the Second International Workshop on Infrastructure, MAS and MAS Scalability*, Montreal, Canada.

Alexandrescu, A. (2001). *Modern C++ design: Generic Programming and Design Patterns Applied*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Arcos, J. L., Esteva, M., Noriega, P., Rodrguez-Aguilar, J. A. and Sierra, C. (2005). An integrated developing environment for electronic institutions, *Software Agent based Applications. Platforms and Development Kits*, Birkhäuser Publisher, pp. 121–142.

AW. `http://www.activeworlds.com/`

Berger, H., Dittenbach, M. and Merkl, D. (2004). User-oriented evaluation of a natural language tourism information system, *Information Technology and Tourism* **6**(3): 167–180.

Berger, H., Dittenbach, M., Merkl, D., Bogdanovych, A., Simoff, S. and Sierra, C. (2007). Opening new dimensions for e-tourism, *Virtual Reality.* Accepted for publication.

Bogdanovych, A., Berger, H., Simoff, S. and Sierra, C. (2006). Travel agents vs. online booking: Tackling the shortcomings of nowadays online tourism portals, *Proceedings of the 13th International Conference on Information Technologies in Tourism (ENTER'06)*, Springer, Lausanne, Switzerland, pp. 418–428.

Castronova, E. (2005). *Synthetic Worlds: The Business and Culture of Online Games*, University Of Chicago Press.

Chao, D. L. (2001). Doom as an interface for process management, *Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, ACM Press, New York, pp. 152–157.

Damer, B. (1997). *Avatars: Exploring and Building Virtual Worlds on the Internet*, Peachpit Press, Berkeley, CA, USA.

Descent. `http://en.wikipedia.org/wiki/descent_(computer_game)`

Doom. `http://www.idsoftware.com/games/doom/`

DoomStats. `http://en.wikipedia.org/wiki/doom`

EIDE. `http://e-institutor.iiia.csic.es`

Esteva, M., de la Cruz, D. and Sierra, C. (2002). Islander: an electronic institutions editor, *Proceedings of the first International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, ACM Press, New York, NY, USA, pp. 1045–1052.

Esteva, M., Rosell, B., Rodriguez-Aguilar, J. A. and Arcos, J. L. (2004). Ameli: An agent-based middleware for electronic institutions, *Proceedings of the 3rd International Conference on Autonomous Agents and Multi Agent Systems*, Vol. 01, IEEE Computer Society, Los Alamitos, CA, USA, pp. 236–243.

GameBots. `http://www.planetunreal.com/gamebots/`

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series.

Gasser, L. (2001). Mas infrastructure: Definitions, needs and prospects, *Revised Papers from the International Workshop on Infrastructure for Multi-Agent Systems*, Springer-Verlag, London, UK, pp. 1–11.

Gratzer, M., Werthner, H. and Winiwarter, W. (2004). Electronic business in tourism, *International Journal of Electronic Business* **2**(5): 450–459.

HL. `http://www.half-life.com/`

IIIA. `http://www.iiia.csic.es/`

Java3D. `http://java3d.dev.java.net/`

JAXP. `http://java.sun.com/xml/downloads/jaxp.html`

Kot, B., Wuensche, B., Grundy, J. and Hosking, J. (2005). Information visualisation utilising 3d computer game engines case study: a source code comprehension tool, *Proceedings of the 6th ACM SIGCHI New Zealand chapter's International Conference on Computer-Human Interaction (CHINZ'05)*, ACM Press, New York, NY, USA, pp. 53–60.

Maes, P. and Nardi, D. (eds) (1988). *Meta-Level Architectures and Reflection*, Elsevier Science Inc., New York, NY, USA.

Manojlovich, J., Prasithsangaree, P., Hughes, S., Chen, J. and Lewis, M. (2003). Utsaf: A multiagent-based framework for supporting military-based distributed interactive simulations in 3d virtual environments, *Proceedings of the 35th Conference on Winter Simulation (WSC'03)*, New Orleans, LA, pp. 960–968.

Meyers, S. (1998). *Effective C++ (2nd ed.): 50 specific ways to improve your programs and designs*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Moloney, J., Amor, R., Furness, J. and Moores, B. (2003). Design critique inside a multi-player game engine, *Proceedings of the CIB W78 Conference on IT in Construction*, Waiheke Island, New Zealand, pp. 255–262.

Myst. `http://www.mystworlds.com/us/`

Preece, J. and Maloney-Krichmar, D. (2003). Online communities: focusing on sociability and usability, *The Human-Computer Interaction Handbook: Fundamentals, evolving Technologies and emerging Applications*, Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, pp. 596–620.

PSDoom. `http://www.cs.unm.edu/~dlchao/flake/doom`

Q. `http://qdn.qubesoft.com/`

Quake. `http://www.quake.com/`

RO. `http://www.realmserver.com/`

Schwabe, G. and Prestipino, M. (2005). How tourism communities can change travel information quality, *Proceedings of the 13th European Conference on Information Systems, Information Systems in a Rapidly Changing Economy, (ECIS'05)*, Regensburg, Germany.

Seidel, I. (2005). Evaluating game engines for the use with 3d electronic institutions, *Technical report*, Institute for Software Technology and Interactive Systems, Vienna University of Technology, Vienna.

Shehory, O., Sycara, K., Sukthankar, G. and Mukherjee, V. (1999). Agent aided aircraft maintenance, *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS '99)*, ACM Press, New York, NY, USA, pp. 306–312.

SL. `http://www.secondlife.com`

SLUsage. `http://s3.amazonaws.com/static-secondlife-com/_files/xls/sl_virtual_economy_metrics_02-02-07.xls`

Smith, G., Maher, M. and Gero, J. (2003). Designing 3d virtual worlds as a society of agents, *Proceedings of the 10th International Conference on Computer Aided Architectural Design Futures (CAADFutures03)*, Tainan, Taiwan, pp. 105–114.

Smith, R. G. (1981). The contract net protocol: High-level communication and control in a distributed problem solver, *IEEE Transactions on Computers* **C-29**(12): 1104–1113.

Spasim. `http://www.geocities.com/jim_bowery/spasim.html`

Sycara, K., Paolucci, M., Velsen, M. V. and Giampapa, J. A. (2003). The retsina mas infrastructure, *Autonomous Agents and Multi-Agent Systems* **7**(1/2): 29–48.

Torque. `http://www.garagegames.com`

Tsvetovat, M. and Sycara, K. (2000). Customer coalitions in the electronic marketplace, *Proceedings of the Fourth International Conference on Autonomous Agents*, ACM Press, New York, NY, USA, pp. 263–264.

TXML. `http://sourceforge.net/projects/tinyxml/`

UO. `http://www.uoherald.com/news/`

UsageStats. `http://www.alexaholic.com/tiscover.at+smartertravel.com+hotwire.com+expedia.com`

UT. `http://www.unrealtournament.com/`

Werthner, H. and Ricci, F. (2004). E-commerce and tourism, *Communications of the ACM* **47**(12): 101–105.

Woolridge, M. J. (2001). *An Introduction to Multiagent Systems*, John Wiley & Sons, Inc., New York, NY, USA.

WS3D. `http://www.3drealms.com/wolf3d/`

WTTC. `http://www.wttc.org`

XML. `http://www.w3.org/xml/`

XMLSchema. `http://www.w3.org/xml/schema`