EASIMEN - A sandbox for artificial creatures in simulated environment

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc.
Study program: Computer Science, Programming

Prague 2011
Acknowledgments

I would like to take this chance and thank my whole family for supporting my studies, for their trust and patience toward me, and for contributing to me becoming the person I am today.

I would like to dedicate this work to all of them, especially to my grandfather, Tánczos Péter, who could not live to see me finish it.

I would also like to thank my friends for the support and help they gave me during the time of development, to my supervisor, Mgr. Roman Neruda, CSc., for accepting this project, for his guidance and for being cool 😊, and to Mgr. Lukáš Chrápa, PhD for his patience and leniency at the beginning.

I am also in debt to Hajdics Balázs, who helped me to look into the “dark depths” of his DotSceneFormat Ogre-Addon project, to Steve ‘Sinbad’ Streeting for his wonderful creation, the Ogre3D open-source graphics rendering engine – and while at it, the whole Ogre community –, to Russell L. Smith, the original author of the ODE physics engine, and the guys who help to maintain it, to Paul Cheyrou-Lagrèze (a.k.a. ‘Tuan Kuranes’) and ‘sfmonster’, the previous and current maintainers (as of 2011) of the OgreODE wrapper Ogre-Addon project, to ‘Crazy Eddie’, the creator of the CEGUI GUI system, and the free open-source software community in general for providing such invaluable and grand software projects, and contributing selflessly to the world without any charge, for the enrichment of others.
I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Article 60 paragraph 1 of the Copyright Act.

I agree with lending the thesis.

Prague, 27th May 2011

Attila Hencz
Název práce: EASIMEN - Sandbox pro umělé bytosti v simulovaném prostředí
Autor: Attila Hencz
Katedra / Ústav: Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: Mgr. Roman Neruda, CSc., Ústav informatiky AV ČR, v.v.i.
e-mail adresa vedoucího: Roman.Neruda@mff.cuni.cz

Abstrakt: Předložená práce se zabývá vytvořením přizpůsobitelného skoro-univerzálního simulačního prostředí pro ztelesněné umělé inteligence (umělé bytosti), kde simulace probíhají i v reálném čase. Dále je také zkoumána problematika běhu takového systému a metody použité při těchto simulacích. Stávající implementace simulačního prostředí (EASIMEN) je zjednodušené a složí k demonstraci návrhu. Přitom je k dispozici pár jednoduchých ukázkových modulu specifických implementací pre potužitou architekturu umělé inteligence navržené autorem (BIAR), které jenom znázorňují jednotlivé aspekty této problematiky a slouží jako šablony pro vytvoření sofistikovanějších modulu v budoucnu.

Klíčová slova: umělá inteligence, real-time simulace, 3D grafika, fyzicky simulované virtuální prostředí, umělá bytost, kombinování přístupů k problematice umělé inteligence, návrh umělé inteligence založené na biologii, EASIMEN, BIAR

1AV ČR = Akademie věd České republiky
2v.v.i. = veřejná výzkumná instituce
Title: EASIMEN - A sandbox for artificial creatures in simulated environment
Author: Attila Hencz
Department / Institute: Department of Theoretical Computer Science and Mathematical Logic
Supervisor of the bachelor thesis: Mgr. Roman Neruda, CSc., Institute of Computer Science of the ASCR\(^3\) v.v.i.\(^4\)
Supervisor’s e-mail address: Roman.Neruda@mff.cuni.cz

Abstract: The present work investigates the creation of a flexible near-universal simulation environment for artificial intelligence with embodiment and real-time simulation in mind. Also, a look is taken at the problems associated with the functioning of such a systems and the methods that can be used in a simulation of this kind. The current implementation of the simulation environment (EASIMEN) is rather simplified and its purpose is the demonstration of the proposed design and architecture. Additionally, there are a couple of simplistic module implementations available for the underlying artificial intelligence architecture proposed by the author (BIAR). These modules are only mere showcases of certain aspects of the issues at hand, and serve as templates for the implementation of more sophisticated modules in the future.

Keywords: artificial intelligence, real-time simulation, 3D graphics, physically simulated virtual environment, embodied agents, combining AI techniques, biologically inspired AI design, EASIMEN, BIAR

\(^3\)ASCR = Academy of Sciences of the Czech Republic (in Czech: AV ČR = Akademie věd České republiky)
\(^4\)v.v.i. = public research institution (in Czech: veřejná výzkumná instituce)
Contents

1 Introduction 9
   1.1 What others have done ........................................ 11
      1.1.1 The main inspiration ...................................... 12
      1.1.2 Other projects to consider ................................ 13
      1.1.3 Projects on artificial intelligence systems integration .... 15
   1.2 What was I trying to accomplish? ................................ 16
   1.3 How is this thesis further organized? ............................. 17

2 Overview 19
   2.1 Artificial Intelligence .......................................... 19
      2.1.1 Relevant basic concepts of artificial intelligence and embodiment in that context .... 19
      2.1.2 BIAR .......................................................... 20
   2.2 Simulation environment ........................................... 24
      2.2.1 The structure of a full-scale 3D simulation environment .... 24
      2.2.2 EASIMEN ...................................................... 27

3 Implementation 33
   3.1 About the implementation in general ............................ 33
      3.1.1 Programming language ...................................... 33
      3.1.2 External libraries .......................................... 34
   3.2 BIAR ............................................................... 36
   3.3 EASIMEN ........................................................... 43
   3.4 Problems during the development ................................. 45

4 Experimentation 48
   4.1 Modules ............................................................ 48
   4.2 BIAR-testOgre ...................................................... 49
   4.3 Summary ............................................................. 50

5 Conclusions 51
   5.1 Discussion .......................................................... 51
   5.2 Summary ............................................................. 53
   5.3 Limitations and further development .............................. 53

Bibliography 55
List of Figures
Foreword

As I was sitting before the text editor thinking about what to write, I realized, that during all my previous studies (academic and before), I was preparing for *something*... Something specific. Not just for ‘life’, not just for the career of a programmer, but for *something* I always wanted to do: *to understand* and *to create*.

At that time, of course, I didn’t know how this would manifest itself exactly. Still, as my ideas and thoughts were distilling and my knowledge in the field of computer science was growing, they were unmistakably leading me towards the creation of the *something* which eventually became *EASIMEN* and *BIAR*.

At the time of writing this thesis, my main areas of interest regarding computer science were *computer graphics* and *artificial intelligence (AI)*.

However the main focus of this thesis is the second one, artificial intelligence, both of these two areas were considerably influencing me during the whole process of development and writing.

As a consequence, this work is primarily intended for those, who have the aforementioned areas close to their hearts – may they be students, programmers or hobbyists interested in AI.

I hope, that with this thesis and the pieces of code accompanying it, I can aid the willing reader in achieving what it was meant to assist me in: *understanding* and *creation*. 
Chapter 1

Introduction

Nowadays more and more aspects of everyday life are overtaken by computers and automated tools and devices of all kinds. This is generally a beneficial process, but it is also problematic and sometimes even dangerous. Therefore I think it is important that we focus our attention more closely on how these machines work, how they are capable of replacing sophisticated human control or activity in certain areas and why they cannot (yet) do it in others. Also, it is in our best interest to know, how we can make them safer, more intelligent and adaptive, to uphold their usefulness under changing conditions or in environments they were not specifically designed for.

Computers, automated tools and virtual beings

Until recently, a certain level of automation was achieved by using simple, procedural methods. These methods usually take into consideration only what needs to be done to complete a single task (e.g. industrial robots) with little or no respect to the parts of their surroundings not involved in their operation. This could (and it most certainly did in several cases) result in human injuries.

Advances in space exploration has lead to the creation of unmanned probes which can (to some extent) work in hostile, even unexplored environments. However, they are still not fully autonomous – there is usually some kind of human interaction involved –, and it is not unprecedented that they break down, because the technicians preparing them to the mission forgot about some simple but not entirely straightforward possibilities.

There are also UAVs (Unmanned Aerial Vehicles) which fly autonomously based on pre-programmed flight plans using more complex dynamic automation systems. Still, in military applications they have a reputation of being prone to collateral damage and/or erroneous targeting. Although they were developed for the military, they are also used in a small but growing number of civil applications, such as firefighting or surveillance of pipelines.

Another example would be entertainment. In computer gaming, the more and more popular massively multiplayer online games (MMOs), alongside with some of the singleplayer games developed to the finest detail, maintain vast virtual worlds needing to be filled with immersive, believable artificial characters. In film production, movies would become much cheaper to make and of a higher quality, if scenes full of characters or creatures (especially fictional ones) could be produced not by using dummies, supernumerary actors or preset post-produced
non-interactive CGI elements, but virtual entities able to sense and react to the actions of the actors and each other. However, during the creation of the movie ‘The Lord of the Rings: The Return of the King’ there were already battle scenes of huge armies fighting with most of the soldiers driven by artificial intelligence. Although these scenes were post-processed CGI segments and therefore the actors themselves could not ‘participate’ in them, it was already a step made on a long and winding road toward a whole new level of human-AI-interaction.

**Autonomy**

As you can surely see, the issues of autonomy technology are related to military and industrial development as well as to entertainment and other areas of life. These are problems like combining information from different sensors, path planning, trajectory generation and regulation (maneuvering), handling communication and coordination between multiple agents in the presence of incomplete and imperfect information, optimal distribution of tasks amongst a group of agents with time and equipment constraints, and similar single-agent or cooperative scenarios.

The common definition of autonomy is the ability to make decisions without human intervention. In other words, the goal is to teach machines to be ‘smart’ and act more like humans. The reader may correctly associate this with the development in the field of artificial intelligence (AI).

**AI research**

Research in the field of artificial intelligence is quite broad and diversified. Researchers take several substantially different approaches to generate intelligent behavior. Some try to mimic the actions and behavior of living organisms, others go with a low-level approach and try to build up intelligent systems from elements functioning similarly like common biological cells, neurons or even more basic building blocks of life. Still others take a completely different perspective and are designing systems which seem to be nowhere near to ‘life’ in a common sense, yet they behave and act in ways which can be described as intelligent.

**Understand to create**

To be able to create intelligent behavior or to mimic biological organisms, first one has to analyze and possibly understand such behavior, and ask questions like “What is intelligence?” , “How does it work?” , “How does biology work?” , and so on. Answers to these questions may not be straightforward or easy. The multitude of fields of science called the ‘life sciences’ examining these matters can serve as a proof of this. Being familiar with these fields of science can greatly improve ones ability to achieve believable levels of similarity of ‘artificial’ to ‘natural’.  

\[\text{1}\text{a definition of an agent in this context is provided in the section Artificial Intelligence of chapter Overview of this work}\]
Create to understand

On the other hand, the persistent attempts to imitate nature also lead to a better understanding of its processes. Through creating simplified solutions for complicated natural structures we can understand the nature of their workings much more easily—quite the same way as we use simple Newtonian laws to describe the events of our ‘everyday lives’ in a world driven by quantum mechanics, fueled by nuclear power and influenced by effects only the most powerful minds are able to guess at.

Therefore, creating (or attempting to create) better, more efficient artificial intelligence and virtual beings can also result in ... well ... besides better and more efficient artificial intelligence and virtual beings $\sim$, a better and more efficient world to live in. In addition, it can also make us come to know our environment and others better, but if we are perceptive enough, through self-revelation, it can even lead to a better and more efficient self.

How to create?

In order to create AI, the most fundamental way is to get into programming. However, implementing (reasonable) artificial intelligence in any programming language could be a quite hard and time consuming task. There are a couple of development libraries and packages which could be used, but these usually are specific implementations of certain specific AI approaches and designs, and therefore do not allow for much experimentation.

The key to learning and understanding by creation is through experimentation. Hence, it would be much appreciated to have tools enabling us to create different types of AI easily, preferably without much effort, yet still allowing for experimentation – combination, alteration or even completely new ideas.

I have been interested in artificial intelligence from childhood. However, for long, I was only dreaming to become involved in AI research. All because I did not know how to begin. If I had a tool of this kind then, I would probably have learned and done much more—maybe even contributed to the current state of AI research in some way.

1.1 What others have done

At the time of writing, there are finally some projects offering more or less flexible software solutions allowing some experimentation in the field of artificial intelligence.

However, because the exact implementation of these projects did not influence the development of BIAR (or EASIMEN for that matter), I will not discuss them in detail. Still, they are listed here as they are all connected to the topics discussed in this work in some way or another.

I used my own ideas writing this work and developing the attached software, so any similarities of BIAR or EASIMEN to the implementation of the following projects could result only from common base design ideas shared by the authors.
1.1.1 The main inspiration

FEAR

FEAR (Flexible Embodied Agent ‘Rchitecture) is a development library created by Alex J. Champandard as a modular testbed for learning AI. It is described in his book (Champandard[1], 2004, p. 47) for which it has been created. Its goal is to help beginners and software engineers to create advanced artificial intelligence in synthetic characters with simplicity and efficiency. The project includes reusable AI components, portable framework and interfaces to real-time 3D games.

The basic idea of FEAR and the aforementioned book was one of the main inspirations for creating my own architecture BIAR, and EASIMEN, a simulator capable of using it to replace ‘external’ 3D games.

This project could be considered as an alternative to BIAR.

state: version 0.3 (as of 2009)

website: http://fear.sourceforge.net – At the time of writing, it generated a database error, so the project page http://sourceforge.net/projects/fear could be used instead. Also, another site, http://aigamedev.com contains information on FEAR and other AI related subjects.

license: free (GNU GPL)

Pros:

• modularity and extendability,

• allowing for the combination of pre-existing AI components (to some extent) without compiling source code,

• a whole book to back it up with explanation and tutorials,

• portable—however, the platform (i.e. a 3D game) which it is going to be used with, still presents a limitation.

Cons:

• it is ‘only’ an AI module/library and thus it is dependent on an external platform to use it with (e.g. a computer game supporting external AI modules—an interface implementation exists for Quake 2 by Id Software),

• it focuses on the creation of AI for computer games, notably for the FPS (first-person shooter) genre,

• to my knowledge (as of 2009), it is distributed in a source-only format through subversion from the Internet (there is no CD/DVD release even for the book itself) – however, this can also be thought of as an advantage, as no one is forced to stick with some outdated version (as long as the Internet source is available and maintained), still, some entirely working binary releases or SDKs would be nice –,
as of 2009 (version 0.3), there were certain inconsistencies in the code: when I tried to build it, the AI modules and the implementation of the architecture of the same revision were incompatible.

1.1.2 Other projects to consider

Ymir: A Mind Model for Communicative Creatures and Humanoids

Its author states, that Ymir is a broad, generative model of psychosocial dialog skills that bridges between multimodal perception, decision and multimodal action in a coherent framework. It represents a distributed, modular approach that can be used to create autonomous characters capable of full-duplex (i.e. the interaction is open-loop – the exchange of information is not step-lock).

website: [http://xenia.media.mit.edu/~kris/ymir](http://xenia.media.mit.edu/~kris/ymir)

Pogamut

It is stated on its website, that “[m]ultiple game engines provide open scripting languages, which allow users to create scripted AI. External control mechanisms can be connected to many of these games using the scripting languages, thereby providing a tool to control the in-game characters from the outside. Pogamut utilizes UnrealScript (UT2004 scripting language) in this way and also NetBeans Java platform to provide an out-of-the box development environment for AI of virtual characters inhabiting UT2004, Unreal Development Kit (UDK) and DEFCON worlds.

The main objective was to simplify the ‘physical’ part of agent creation. Most actions in the environment (even the complicated ones, like path-finding and gathering information in agent’s memory) can be performed by one or two commands. This enables [the] user to concentrate his efforts on the interesting parts.”


Robocode

On this project we can learn, that Robocode is a programming game, where the goal is to develop a robot battle tank to battle against other tanks in Java or .NET. According to its website, the robot battles are running in real-time and on-screen.

website: [http://robocode.sourceforge.net](http://robocode.sourceforge.net)

ORTS (Open Real-Time Strategy)

ORTS is, as it is described on its website, a programming environment for studying real-time AI problems such as path-finding, dealing with imperfect information, scheduling, and planning in the domain of RTS games.

website: [http://skatgame.net/mburo/orts](http://skatgame.net/mburo/orts)
Soar

Soar is a general cognitive architecture for developing systems – as their authors state – that exhibit intelligent behavior. Researchers all over the world, both from the fields of artificial intelligence and cognitive science, are using Soar for a variety of tasks. It has been in use since 1983, evolving through many different versions to where it is now: Soar, Version 9.

website: http://sitemaker.umich.edu/soar

Alive!

Alive! is a sandbox also developed by Alex J. Champandard (the author of FEAR – see above) for learning AI and game programming. It’s a simple IDE inspired by Self and Squeak but written in Python. According to its website, this project is being developed full time, as part of Game AI Programming tutorials.

website: http://alive.sourceforge.net

Game::AI++

Game::AI++ is another SourceForge-project having A. J. Champandard as a maintainer. According to its description, it is an extensible decision making and control system designed for simulations.

website: http://game-ai.sourceforge.net – Automatically generated (as of the time of writing), the project page http://sourceforge.net/projects/game-ai could also be used.

BOTWORKX: The Ultimate AI Sandbox

Not much could be learned about this project (at least by me), as it has been found as a design concept on a ‘blog’-/‘web-log’-style website. According to that design concept, it should be a “C++ framework and applications for experimenting with artificial intelligence, 3D graphics and simulated physics. It’s primary focus is to provide a testbed for simulated robotics using Ogre 3D and the Open Dynamics Engine.” However, I could not find any indication of an official homepage or project website. Posts containing source code kept appearing on the blog, so chances are that it is an existing but early-stage project.

website: (we)blog: http://www.botworx.org/ – Project design has been outlined in this entry: http://www.botworx.org/2008/03/ultimate-ai-sandbox.html.

Simulator BOB

This project has been described by its author as being “an open-source 3D simulation environment designed for – but not limited to – the simulation of mobile robots. It uses ODE to simulate the [rigid] body dynamics and OpenSceneGraph
for the 3D visualization. The GUI is realized by using MFC and, thus, the software is limited to Win32 platforms. It is written entirely in C++ and is compiled with MS Visual Studio 7.1. (Version 6.0 would probably work as well but [the author] don’t maintain the project files for it.)

The key feature of this application is meant to be it’s simplicity in getting started. You can easily install it and run some existing simulations. When creating your own simulation you only have to care about a very small and simple interface (look at the Documentation page for a Tutorial of a simple simulation).

The author plans to simulate a system of autonomous mobile robots he is currently building up as part of his PhD thesis. Although, as he states on the projects website (as of the time of writing), “there is still a lot to do until this is possible and sufficiently realistic to make sense, it is already functional and surely useful for some applications”.

website: http://simbob.sourceforge.net

1.1.3 Projects on artificial intelligence systems integration

The OpenAIR Protocol (by MINDMAKERS.ORG)

Project description: OpenAIR was created to allow software components that serve their own purpose to communicate with each other in order to produce large scale, overall behavior of an intelligent systems. A simple example would be to have a speech recognition system, and a speech synthesizer communicate with an expert system through OpenAIR messages, to create a system that can hear and answer various questions through spoken dialog. CORBA (see below) is an older but similar architecture that can be used for comparison, but OpenAIR was specifically created for A.I. research, while CORBA is a more general standard.

website: http://www.mindmakers.org

Psyclone AIOS

Project description: Psyclone is a software platform, or an AI operating system (AIOS), developed by Communicative Machines Laboratories for use in creating large, multimodal A.I. systems. The system is an implementation of a blackboard system that supports the OpenAIR message protocol. Psyclone is available for free for non-commercial purposes and has therefore often been used by research institutes on low budgets and novice A.I. developers.

website: http://www.cmlabs.com/psyclone

Elvin

Project description: Elvin is a content-based router with a central routing station, similar to the Psyclone AIOS (see above).

website: http://elvin.org/
OAA

**Project description:** The OAA is a hybrid architecture that relies on a special inter-agent communication language (ICL) – a logic-based declarative language which is good for expressing high-level, complex tasks and natural language expressions.

**website:** Project homepage: [http://www.openagent.com](http://www.openagent.com); Homepage at the SRI Artificial Intelligence Lab: [http://www.ai.sri.com/~oaa](http://www.ai.sri.com/~oaa)

CORBA

**Project description:** The Common Object Request Broker Architecture (CORBA) is a standard that enables software components written in multiple computer languages and running on multiple computers to interoperate. CORBA is defined by the Object Management Group (OMG). CORBA follows similar principles as the OpenAIR protocol (see above), and can be used for A.I. systems integration.


MOSID

**Project description:** The Messaging Open Service Interface Definition (OSID) is an O.K.I. (Open Knowledge Initiative) specification which provides a means of sending, subscribing and receiving messages. OSIDs are programmatic interfaces which comprise a Service Oriented Architecture for designing and building reusable and interoperable software.

**webpage:** Wikipedia page: [http://en.wikipedia.org/wiki/Messaging_Open_Service_Interface_Definition](http://en.wikipedia.org/wiki/Messaging_Open_Service_Interface_Definition)

Unfortunately, while some of these projects are using other platforms (e.g. existing computer games with an AI interface) which is limiting their possibilities and usage, others lack the ability of customizability or extendability, they are not freely available or they focus on a specific type of AI design, or do not satisfy some other goal or intention of EASIMEN and BIAR (see the following section).

### 1.2 What was I trying to accomplish?

The title-name of this work – **EASIMEN** – is an abbreviation for **Embodyed Agent SIMulation ENvironment**, but this name also reflects the intentions to make a flexible easy-to-use (EASI-) AI sandbox supporting not only single but multi-agent simulations of embodied agents (-MEN).
The AI architecture is a sub-project with its own name: **BIAR**. It stands for *Biologically Inspired Agent Architecture*, as its high-level design resembles the structure of real living creatures. Also, BIAR is the main focus of the theoretical part of this thesis.

**Goals**

One of the aims of this work (and the code provided with it) was to create a freely available easy-to-use (user-friendly) and flexible cross-platform software environment which simulates artificial intelligence (AI) in virtual environments, capable of simulating real-time physics with 3D graphical representation.

The other was to design and implement a near-universal AI architecture which enables the combined use and simulation of different (and usually incompatible) AI techniques in almost any kind of environment.

In other words (and in contrast with similar works), the goal of this project was to create a *platform for artificial intelligence and not artificial intelligence for a platform* (as most of the other more complete solutions do).

In addition, BIAR was intended to be independent from EASIMEN, so it could be used in other unrelated projects and solutions, too.

**Who is it intended for?**

This kind of software solution could be suitable for educational purposes, for AI-hobbyists, or even as a base for computer games or computer generated interactive ‘virtual worlds’ with an emphasis on AI.

### 1.3 How is this thesis further organized?

**Overview**  
First, in the next chapter, *Overview*, I take a look at the concept of AI in general, then I introduce the design of BIAR, the AI architecture created and proposed by me.

After that, the description of the concepts related to the creation of a simulator-type application like EASIMEN is discussed, with the design-description of EASIMEN itself following.

**Implementation**  
In the following chapter, *Implementation*, I am going to talk about the methods and means chosen for the implementation of the more important concepts mentioned in the *Overview*, and the issues faced during the process of development.

**Experimentation**  
The chapter *Experimentation* afterwards presents a showcase on the usage of BIAR in the form of a simple test-applications with some commentary.

**Conclusions**  
The final chapter, *Conclusions*, contains discussion on the results and success of this work, draws conclusions and outlines the path for further development.
User Documentation (External)

Information on the compilation of the provided source code and the usage of the built applications provided as an attachment can be found in the User Documentation, which is a separate document under the name User_Documentation_for_EASIMEN_and_BIAR.pdf.

It also contains instructions on how to construct custom scenes to simulate (without any programming necessary).
Chapter 2

Overview

2.1 Artificial Intelligence

2.1.1 Relevant basic concepts of artificial intelligence and embodiment in that context

**Agent** To introduce some of the basic concepts, I use a note by Alex J. Champandard (2004[1], p. 7) on the subject:

[In the field of artificial intelligence], a smart entity is known as an *agent*.

Such computer driven agents are usually interpreted as abstract entities, ‘smart’ programs, which are purely virtual in nature. They are only limited in their working with the data they were presented, by the capabilities of the hardware or the operating environment of the system they work in.

This kind of representation of the controller unit is not that well suited for my goal, the simulation of virtual creatures. As these abstract agents can access data in the system without restraint, their usage in an environment such as what I intend to create could easily lead to the loss of believability of the creatures they are representing.

**Embodied Agent**  *Embodiment* is a concept for representing agents in a different way where the simulated agent is endowed with the characteristics of real living creatures in that it has a ‘physical’ body which is limited by the rules of its environment. Here by physical I mean that it is subject to its environment by whatever rules that environment may be defined.

As Champandard (2004[1], p. 21) defines:

An embodied agent is a living creature subject to the constraints of its environment.

**Note:** As the cautious reader may notice, this definition also applies to real, living, biological creatures.

**Multi-Agent Simulation** In the Introduction, I stated that I would like my solution to support the simulation of single AI entities as well as of ‘multi-agent’ sys-
tems. Let’s define this term by quoting another of Champandards notes (2004[1], p. 7):

A system that handles more than one [such] unit in coordination is known as a multi-agent system.

Now, as I said before, I wanted the simulation environment I was assembling to be capable of handling such multi-agent simulations. But as he continues (Cham-

pandard, 2004[1], p. 7),

Developing multiple agents involves scaling down the AI enough so that it’s feasible to scale up the number of [these computer driven entities]. In essence, it’s about using simpler AI using less memory and processing power—although this is a challenge in its own right!

As he states, there are limitations on the use of multi-agent systems, which are primarily presented by the available hardware to run the simulation on. However, as the computer technology progresses, it is more and more conductive to the simulation of more complex and thus more believable agent-systems.

2.1.2 BIAR

![BIAR design schematic](image)

Figure 2.1: BIAR design schematic

Basic design

The Biologically Inspired Agent 'R'chitecture, is the architecture proposed by me for implementing a flexible and intuitive AI simulation framework.

As the concept of embodied agents itself is so much related to real living biological creatures, I decided to create a high-level design which resembles the structure of such creatures.

The following paragraphs introduce the main components of BIARs design:

**Director**  This is the component which drives the simulation of a virtual environment from the ‘AI-perspective’.

It is responsible for the management of all the Creatures in a single simulation.
Creature  I call my agents Creatures, in alignment with the aforementioned resemblance to biological beings.

One such Creature consists of a Brain, a Body, and the Connections (CNS) connecting their sub-components.

Brain  This component is an analogy to, well, biological brains and the like – who would have thought?...

The Brain is responsible for controlling the Creature it resides in. This is the most important part of a Creature, as this is the component representing the artificial intelligence behind every action and ‘thought’ of a Creature.

It contains Subsystems.

Subsystem  A Subsystem in my concept symbolizes a region in the brain of real creatures with a specific function. This is the most basic computational element the Creature (agent) uses to process information and to interact with its environment.

Body  Again, the name it quite self-explanatory—it is the representation of the body for my virtual beings.

The Body – by its subcomponents – is responsible for the interaction of a Creature with its environment, and thereby limiting it to the actions permitted by the environment through its Body.

Its direct purpose is to be a container for Members.

In addition, it can also serve as an entry point for the Brain, or more specifically its Subsystems, if they want to access the Members of the Body in a ‘structured manner’ (see Body in the following section More details).

Member  A body-Member is the analogy for real-life body-parts.

It can have a Representation, Actuators and Sensors.

It can also have sub-Members (see Body in the following section More details).

World  The World in this context is just an abstraction to provide Representations, Actuators and Sensors for Members. It serves as an interface to a specified type of simulated environment.

Representation  The Representation is an abstraction for simulated entities (e.g. graphical, physical, etc.) in a virtual environment.

Once a Member is connected to such an entity through a Representation, it becomes being represented by that entity in that environment.

Actuator  It is a component to alter the virtual environment or even the internals of the Brain and the Body.

It serves as an interface element to cause changes in a specific World in a specific way for the AI (i.e. the Brain, by its Subsystems).
Sensor  It is a component to gather information on the virtual environment (or the Creature itself).

Serves as an interface element to query the state of a specific World or its changes in a specific way for the AI (i.e. the Brain, specifically its Subsystems).

Connections  Some of the component-types – namely Subsystems, Actuators and Sensors – can be connected to provide information for each other. This allows for the collaboration of the named components.

Connections are ‘made’ using Channels.

Channel  It is an abstraction for real-life nerves. It represents broadcast-type, one-to-many Connections originating from a single Emitter toward multiple Receivers (see Channel in the following section More details).

More details

Director  For every simulation, there has to be exactly one Director to control it. In consequence, to simulate more environments at once, one Director is needed for each and every one of them.

Such simulated environments cannot share components—every Director, with its own Creatures, is completely independent from others.

Creature  In every Creature, there has to be at most one Brain and Body. However, as it could be suspected from the previous statement, neither of them have to exist. The reader could rightfully question the usefulness of this allowance, so I will return to it in the Conclusions chapter.

Brain  As stated before, the Brain manages Subsystems. As a part of this management, aside from creating and destroying its Subsystems, when the Brain itself is required to ‘work’, it should ‘run’ its Subsystems in a meaningful way. (To what I mean by ‘meaningful’, I will return to in the section BIAR of the chapter Implementation.)

Subsystem  Subsystems can act alone, separately from other parts of the Brain, or in co-operation, through their consecutive Connections, managed by the Brain.

The Subsystem is an abstract concept which can be implemented by various AI techniques. There is basically no restriction on how a Subsystem can be implemented, however, an implementation could be inefficient if it involves longer execution time and cannot be divided into iterative steps.

Body  As indicated before, Members in the Body can be connected on other Members, but the graph of the Members has to compose a tree with the Body at the root node—every Member should only have one parent Member, or it has to be connected directly to the Body).

Member  A Member does NOT need to have a Representation. It can just contain/connect its sub-Members. Without any kind of Representation – and therefore without constraints presented by any World –, it does not need to keep
any shape, it could stretch and shrink as it pleases—such a Member is *purely virtual*, or in other words *abstract*.

Of course, an existing ‘elastic’ Representation can also allow for this, but in that case that has to be supported by the World the Representation ‘exists’ in.

If a Member does not have a Representation in the specified World, then it still can retrieve general information and cause global changes in the World through appropriate *Sensors* and *Actuators* (which do not need a Representation to function).

However, if it has a Representation, it can have only one at most. Also, the Member ‘is’ in that World its Representation is from, and it can only have Actuators and Sensors implemented for that specific World.

**World**  This component is usually just referencing an external object entwined with some other part of an application (e.g. with the graphics and/or physics engine).

Various World implementation modules could exist, but all of the Actuators, the Sensors and the Representation can originate from only one of these for every Member. In other words: a Member can be ‘represented in’ only one World (virtual environment), however, a Body can have multiple Members from multiple Worlds—I will return to the implications of this in the *Conclusions* chapter.

Also the components ‘from different Worlds’ cannot interact with each other.

The World-reference (the external object) should be accessible as long as any Member has a Representation in it, but if it is not, that should affect only Members having components from that World – Actuators and Sensors should no longer provide their functions –, while other Members still have to be able to be simulated.

**Representation**  The Representation is not a referenced object itself. It should ‘provide a way’ to get to the simulated entity (as by some identifier) through the World-reference object. This way if a World is not accessible anymore, the Representations in it would also not be available.

**Actuator**  It is implemented for a specific World module, and can be used only by Members ‘in’ that specific World.

It usually manipulates the Representation of the Member (if it has one defined).

**Sensor**  Similarly as Actuators, a Sensor is also implemented for a specific World module, and can be used only by Members ‘in’ that specific World.

Therefore it usually manipulates with the Representation of the Member using it (if it has one defined), too.

**Connections**  Subsystems and Sensors can have *Options* and *Results* of pre-defined primitive types, whereas Actuators can only have *Options*. This is because by definition they should have ‘results’ only in the World their Members Representation is in.

When making a connection the Option of one component is connected on the Result of another.
Channel  The purpose of a Channel is to ‘propagate’ the Value of a Result to its connected Options. It must have strictly one Emitter, which could be any single Result, and ‘lead’ to one or more Receivers, which are the connected Options.

It must have strictly one type specified – the type of the Emitter –, and all the connected Receivers must have that same type.

Also, for one specific Result there may be only one Channel—new connected Options should be automatically added to that Channel.

Note:  Options and Results are discussed in section BIAR of chapter Implementation.

2.2 Simulation environment

2.2.1 The structure of a full-scale 3D simulation environment

The creation of a software application, like EASIMEN has been designed to become, is very similar to the creation of computer games:

- it has to be fast to ensure real-time interaction with the content,
- with preferably pleasing and easily understandable graphical representation,
- simulation of physics on a believable level (emphasis on performance, not real-world precision),
- and a usable AI concept which keeps up with the previous requirements.

Note:  As I have already talked about AI, it will not be discussed in this section again.

The engine

An engine of a software application refers to the core of a computer program. Software engines drive the functionality of a program, and are distinct from peripheral aspects, such as look and feel.

An engine of a simulation framework, like in the case of EASIMEN, comprises a graphics rendering system, a component to calculate the physics of the virtual environment, and another to deal with the simulation of the AI.

In a computer game, it would not be needed to simulate the behavior of physical objects and their appearance as precisely as possible, and our case is no different in that manner. The simulation only needs to be accurate enough to seem realistic. Compromises are allowed, as long as the results are satisfying from the perspective of believability.

Graphics/rendering engine  The graphics engine or renderer is a sub-system of the core of the application. This component is responsible for the rendering of the visual content, such as the GUI overlay and the visual representation of the simulated virtual environment.
The representation of such an environment can be very simple consisting of just points or low-polygon wireframe models, or very complex as the simulation of an army of soldiers for a prestigious (usually high budget) movie, where every entity in the scene is simulated in high detail.

A visual sequence of the later kind takes a lot of time to render—sometimes weeks or even months. It would ensure a high level of believability, but at the cost of being very slow—rendering in such detail is highly computation-intensive. Usually, clusters of computers are required to render these types of scenes for much longer than the length of the animation produced.

Rendering of scenes of the first kind are usually much faster, but it is also much harder for a human being to comprehend and follow their output.

As a result, neither of them are ideal for an observer to interact with the ongoing simulation.

In our case, it is preferable for the observer to be able to easily perceive the changes in the virtual environment in order to be able to react to them. The most suitable solution for this kind of interactivity is provided by real-time rendering engines, which are mostly used in computer games and interactive simulations.

Real-time rendering engine Rendering for interactive media, such as games and simulations, is rendered (calculated) and displayed in real time, at rates of approximately 20 to 120 frames per second. In real-time rendering, the goal is to show as much information as possible in about a $\frac{1}{30}$ of a second—the eye can perceive a motion picture of this rate and faster as continuous because of its physiology. The goal here is primarily speed and not photo-realism. In fact, most of the time exploitations are made in the way the eye ‘sees’ the world, and as a result the final image presented is not necessarily that of the real-world, but one close enough for the human eye to tolerate.

As mentioned above, the significant difference between real-time and non-real-time graphics is the interactivity desired in real-time graphics.

In cases of non-real-time graphics like films, the director has the complete control and determinism of what has to be drawn on each frame, typically involving weeks or even years of decision-making involving a number of people.

However, in the case of real-time interactive computer graphics, usually a user is in control of what is about to be drawn on the display screen; the user typically uses an input device to provide feedback to the system – for example, wanting to move a character on the screen – and the system decides the next frame based on this particular instance of action.

Physics/dynamics engine Another important factor controlling real-time computer graphics is the combination of physics and animation. Such techniques largely dictate where to draw certain objects (deciding their position) on the screen. These techniques imitate the behavior of displayed objects as they are seen in the real-world to a degree that is far more realistic than the degree of realism achieved by computer graphics alone.

The physics engine is responsible for the calculation of the movement of simulated objects in the virtual environment. Such physics simulations are also fairly computation-intensive, as there are usually a multitude of objects which interact with each other in a believable simulation.
For all these objects the engine has to determine their position in a certain future point in time (usually that of the next frame to be drawn) from the actual position, and in the process (in the case of just rigid-body simulations) it has to ensure the modification of the objects path if collisions occur.

Fast checking for collisions is one of the biggest challenges in the world of physical engines, as it is not easy to determine which objects actually interact without testing all of them against all the others. There are optimizations, which reduce the complexity of this testing by, for example, using collision spaces which are basically structured bounding volume hierarchies around groups of objects.

Bounding volumes are minimal covers in the shape of certain computationally cheap objects as, for example, spheres, cylinders or boxes. Testing collisions between spheres is simple: if the sum of their radii is greater than their distance, then there could have been a collision between the objects they encompass in the recent past (usually between two steps of the physics simulation). Therefore, the ‘smaller’ bounding volumes (if there are any) or straightaway the objects included in them should also be tested.

In certain situations however, the shape of such spherical spaces can be ineffective. This is the case, for example, when there is much empty space left inside the volume. To counter this, the shape of the collision space could be chosen according to the shape of the objects, or the shapes of the clusters formed by them with the highest probability, but generally most of the implementations use AABBs (Axis Aligned minimal Bounding Boxes). These are also simple to test, and in general, they serve as a better minimal cover than spheres.

The other important aspect in a physics simulation is the computational model used for determining the behavior of the simulated objects under certain circumstances.

To model the behavior of objects in a virtual world we first must decide which properties of the objects will be kept track of and which physical laws they will obey—it is fair to expect objects to behave according to the basic laws of physics. Newtonian laws describe the basics of the behavior of rigid bodies, and most of the implementations today use primarily these laws—they do not deal with relativity and other advanced concepts.

The GUI

The Graphical User Interface is often neglected in simulator-like programs, however, it is an important part of an interactive application. It can make the use of a certain software easier if the GUI is well implemented and composed, else it can easily cause confusion and disappointment.

Another important aspect of the user interface is its flexibility to support the implementation of various types of interactivity required by different applications.

Ease of use in the development process is another thing to consider. If it cannot be efficiently and comfortably used by the developer, it might be a sufficient reason for the developer to choose another alternative, even if the GUI system in question has the broadest repertoire of features.
The content

It might not matter if a software program is well implemented and highly optimized if the content it provides does not raise the interest of the users the developer is targeting, or it has no useful application. Therefore it is important to contribute enough time for the creation of competitive and well designed content for the code-base of the application.

2.2.2 EASIMEN

![Figure 2.2: EASIMEN design schematic](image)

Basic design

As the Embodied Agent SIMulation ENvironment is a simulator-type application, most of the things discussed in the previous section apply to it.

It allows for running multiple Simulations, each with physics, 3D graphics and AI, separately at the same time.

It can also create multiple windows to show different Simulations, or even different parts of the same Simulation simultaneously, allowing for their much easier overview.

Its basic design is not as structured and complicated as the one of BIAR, but the few elements it consists of are likewise important.

The core components of EASIMEN are

- the Engine,
- the Logger,
- the OgreFramework
- and the Simulation.

**Engine** This is the ‘heart’ of the application. The Engine is started when the application starts, and it manages all the other core components.
CHAPTER 2. OVERVIEW

Logger  It is responsible for handling the huge amount of messages EASIMEN produces while it runs, and presenting them on the standard output and/or writing them in a specified log-file.

OgreFramework  All graphics related functionality is managed by this component, let it be the rendering of scenes, showing and operating the GUI or handling the windows of the application.

By the underlying graphics engine, it also provides some basic filesystem operations to acquire the needed files necessary for rendering.

Simulation  It defines a single simulation run by EASIMEN. More specifically, it contains all the other (non-core) components necessary for a simulation discussed in the previous section. These components take care of

• the physics,
• the graphics,
• and the AI

involved in a single simulation.

DotSceneOgreOdeInterface  This component is listed separately, as it is not a core component, but one implemented as an external software library. However, it is also mentioned here as I was involved in its development as a part of EASIMENs creation.

This is a module allowing for the simulated virtual environment to be defined in and loaded from an XML document. Therefore using it makes possible to run Simulations in different environments without compiling any code.

More details

Now lets see EASIMEN’s design is more detail:

As speed is essential in this type of application, the core components of EASIMEN have undergone certain optimizations as a result of multiple iterations in the development of the core architecture. However, these optimizations will not be discussed here. Instead, I will talk about their final state straightaway.

Engine  A stated before, this component starts up the entire application.

Right on start, it initializes the Logger and OgreFramework, then it parses EASIMEN’s main configuration file, or creates it, if it does not exist, filling it up with all the default values for all the available config-file-options and comments on their significance and usage.

According to the config file it sets up key-bindings and the parsed parameters to be the defaults on creating Simulations.

After that, it creates a default scene, and finally, starts the renderer creating the main window, and the main cycle of the application stepping the Simulations
The amount the aspects of the Simulations are stepped is handled separately for each. This allows for a flexible distribution of the available system resources (CPU, GPU). This is possible through defining another couple of parameters in the config file used as limits for the stepping times of the physics, the graphics, the AI and input scanning.

**Cycle-modes** The Engine supports two modes of operation for the main cycle:

**real-time mode** In this mode of operation, the Engine is trying to maintain the ‘real-time nature’ of the Simulations it runs. This means that all aspects of the Simulations (graphics, physics and AI) are ‘stepped’ with the amount of time elapsed since the last step.

**constant-step mode** In this mode, the Engine is stepping the physics of the Simulations with a constant value, which is one of the config-file parameters. Stepping the physics with a constant amount has its uses in cases for example:

- If a long-term simulation is needed to be finished in a shorter amount of time, it can be achieved by decreasing the resources granted to rendering and input scanning, while disabling the cap on the physics and AI stepping, while using a bigger-constant-stepped simulation.

- If a more precisely simulated result is necessary, using a very small constant step will improve precision but decrease the simulation speed (granting more resources will always help).

Other than this, the Engine is responsible for managing Simulations.

**Logger** The implementation of EASIMEN includes lots of logging to make easier the debugging of the application, and also the generated logs make possible to analyze the run of the Simulations at a later date.

By default, only one common logfile is created. However, this component allows for any number of logs, so if the need arises, each and every component or Simulation can use its own log.

For now, the component itself is just a wrapper for the logging mechanism provided by the graphics engine.

There are two important terms using the logs:

- the *LoggingLevel* – the property of the log specifying what kind of messages will get logged,

- the *LogMessageLevel* – the property of a message to be logged reflecting its importance.

There are 3 levels of both.

For LoggingLevel, these are Low, Normal and BoreMe, with BoreMe being the most verbose, allowing for every message to be logged.

For LogMessageLevel, they are Trivial, Normal and Critical, with Critical being the most important and therefore always getting logged.
OgreFramework  This component is basically a wrapper around the graphics engine. It provides methods to make easy the usage of the renderer and the GUI. Among its functions are

- the management of cameras used to view different parts of Simulations,
- controlling and cycling the cameras in the Simulations as a way of interactivity,
- the management of SceneManagers which account for the rendering of Simulations (one SceneManager per Simulation),
- cycling SceneManagers as a way to observe different Simulations,
- the management of the main window of the application,
- the generation and display of rendering statistics, some online debug information and the simulation-time of every Simulation,
- controlling the operation of GUI pages through a subcomponent.

The GUI system used for EASIMEN is very flexible and customizable. Almost every aspect of it can be defined in XML definitions. This involves the size and type of fonts used, placement and size of elements, which elements are used, labeling of the elements, putting images, videos and non-dynamically-generated text on the elements, the look-and-feel of elements (how they will look like), and so on. For EASIMEN to still be able to use the GUI elements for which it has functionality implemented like buttons, only the name of those elements should be present. The elements themselves can be changed to other types of elements. For example, if there is a button EASIMEN is using, it can be changed to an image being at a completely different place. Simply by having the name of that previous button the function of that button will be executed if the image is clicked.

This flexibility allows for the users to have GUI layouts of their liking. If, say, a user thinks that the placement of a GUI element is unintuitive or problematic, (s)he can move it to a more adequate location. Because these definitions are in files separate from EASIMEN, users can even share them and use a layout created by someone else.

Also, internationalization of EASIMEN can be achieved by simply re-labeling the GUI components in the XML definitions.

Simulation  This component holds all the information and data specific to a single simulation. It contains the graphical and physical structure of the simulated virtual environment, together with a reference to the AI involved in the Simulation in question.

It also holds the parameters necessary for stepping (see Engine). These parameters are stored separately for each Simulation, thus allowing for customization and the distribution of system resources among Simulations.

Aside for these parameters, it also stores (and creates on construction) 3 default cameras for different purposes for every loaded Simulation, to allow the easy navigation of scenes not defining cameras themselves.
It also makes possible to load an XML-defined scene when the Simulation is created, and to be paused or resumed afterward. By default, a Simulation gets loaded in a paused state.

**DotSceneOgreOdeInterface**  As mentioned above, DotSceneOgreOdeInterface enables EASIMEN to load scenes from XML. Just with the alteration of a text-file, the environment could be rearranged, objects can be added and removed, furthermore basically any of the renderer settings can be changed. This allows the user of the application to create or modify scenes to simulate in a relatively easy and flexible way.

There is a DTD (Document-Type Definition) showing the available XML elements and their attributes available to create custom scenes. A tutorial on how to create custom scenes is available in a separate document, the *User Documentation*.

Thanks to my involvement in the project, besides graphical elements and settings there can be also a definition of physical objects in the XML scene-files with the possibility of connecting the physical structures to graphical entities right away in the XML definition.

In addition, there are scene-exporter plugins for 3D content creation software like *3D Studio Max* ([http://usa.autodesk.com/3ds-max/](http://usa.autodesk.com/3ds-max/)) or *Blender* ([http://www.blender.org/](http://www.blender.org/)). These exporters produce XML scene definitions which can be loaded in EASIMEN after some minimal changes, thanks to the DotSceneOgreOdeInterface.

This way, users not comfortable with editing text files can use sophisticated 3D graphical editors to construct their scenes. Also, it could be advantageous to be able to see the scene while assembling it.
Figure 2.3: DotSceneOgreOdeInterface – physics XML element structure
Chapter 3

Implementation

3.1 About the implementation in general

The software solution provided for this thesis consists of more separate projects which were meant to be connected in the application EASIMEN. This project holds the implementations of the engine of the software solution optimized for speed, which enables the simulation of multiple virtual environments simultaneously, combining real-time physics and 3D graphics with a flexible graphical user interface (GUI) showing a demonstrational GUI composition to control the simulation. Another important project, a dynamic library named BIAR, is the implementation of the proposed AI architecture (also called BIAR and created by me for this thesis). All the other projects are supplementary to the above mentioned two:

- modules BIARSSubsystemModule_AreaBoundSimpleReactive, BIARSSubsystem-Module_SimpleAvoider and BIARWorldModule_SimpleOgre implementing primitive sample Subsystems and a World with a similarly simplistic Actuator and two Sensors;
- the DotSceneOgreOdeInterface project for the dynamic creation of simulated scenes in EASIMEN from XML definition files;
- the Tinyxml and XmlNodeProcessor external projects used by DotSceneOgreOdeInterface;
- and finally, the ODE and OgreOde Core external projects used by EASIMEN to simulate the physics of the virtual environments.

3.1.1 Programming language

Considering the topics discussed above, I have decided to use C++ as the programming language for the implementation of the solution. It is fast, it supports the object-oriented concept which is a big help when implementing complex software systems and applications can be developed for multiple platforms as there are compilers for all major operating systems, which is also important from my perspective.

It is said to be hard to work with, as it has no garbage-collection or other similar automatic safeguards against run-time errors caused by the programmer’s
oversights and inadvertence. On the other hand, this is one of the very reasons why it produces faster applications. Furthermore, it allows low level design and much more flexibility. Also, those errors which occur because of a lack of safeguards and automatic resource management can be avoided by using appropriate programming techniques and a little care taken by the programmer.

### 3.1.2 External libraries

As it would have been too complex and time consuming to implement EASIMEN’s every aspect myself, I used some external software libraries. Implementing everything also would have been unnecessary and redundant, particularly because given the limited time, I couldn’t have got even near to their quality and range of functionality they provide.

The external libraries which were used for the implementation of EASIMEN are as follows:

**Ogre3D**[^3] OGRE, the Object-Oriented Graphics Rendering Engine, is a scene-oriented, flexible 3D rendering engine written in C++, designed to make it easier and intuitive for developers to produce applications utilizing hardware-accelerated 3D graphics.

It is a software library which abstracts the details of using the underlying system libraries like Direct3D and OpenGL, providing an interface based on world objects and other high level concepts.

To its advantage, OGRE has a very active community, and new versions are released frequently.

It was SourceForge.net’s project of the month in March 2005[^4], and it has been used in some commercial games such as Ankh and Torchlight, as well as in freely available open-source projects (some even listed among the similar projects to my solution in chapter *Introduction*).

Released under the terms of the MIT License and previously under a modified GNU Lesser General Public License, the engine is free software[^5]. The modification to this license allows users to statically link the library under the same terms as dynamic linking.

As its name states, OGRE is ‘just’ a rendering engine. As such, its main purpose is to provide a general solution for graphics rendering. It also comes with other facilities (vector and matrix classes, memory handling, basic filesystem handling, etc.), but they are considered supplemental. It could not be considered as an all-in-one solution in terms of game development or simulation as it, for example, doesn’t provide audio or physics support.

Generally this is thought of as the main drawback of OGRE, but it could also be seen as a feature of the engine. The choice of OGRE as a graphics engine allows developers the freedom to use whatever physics, input, audio and other libraries they want and allows the OGRE development team to focus on graphics rather than distribute their efforts amongst several systems.
OGRE explicitly supports the OIS, SDL and CEGUI libraries, and includes the Cg toolkit.

In the implementation of EASIMEN, I used the Ogre branch versioned 1.6 (codename: *Shoggoth*). This is also the version used in the official repositories of current Linux distributions (as of the time of writing).

**Ogre-Addons** Ogre-Addons are projects implementing modules for Ogre, enriching it by certain type of supplemental functionality.

The following Ogre-Addons were used for the development of EASIMEN:

**OgreOde** A library implementing bindings to the physics engine ODE/OpenDE. While ODE uses a C-style interface and is completely independent from Ogre, OgreOde uses object-oriented wrapper-classes to provide the same functionality as the underlying ODE does, and is much easier to use. Also, OgreOde uses Ogre-specific types, and arranges for the automatic connection and management of ODE structures on Ogre Entities and SceneNodes.

**DotSceneFormat** Definition of an extensible scene format, called `.scene` which is XML based and can be exported by multiple 3D editor plugins. It is capable of loading graphical scenes into an Ogre-powered software environment by dynamically creating elements of the scene or setting parameters of the renderer itself, described in the XML definition.

The *DotSceneFormat* Ogre-Addon project has been forked by me (with the consent of its author) to add support for loading some of the OgreOde structures from the same XML definition the graphical scene is defined in. Also, some changes have been applied to the original source code so the library would compile using GCC or MinGW, as it has been originally developed using MS/Visual Studio.

**OIS** OIS, the Object-Oriented Input System, is a code library for constructing a human-computer interface with input devices such as a keyboard, mouse or game controller. OIS is designed so that software developers can easily use input from these devices with a computer application.

Also, the use of OIS is fully supported with Ogre to the extent that tutorials and example applications using OIS are provided within the packaging of Ogre releases.

**CEGUI** Crazy Eddie’s GUI system is a graphical user interface library written in C++. It is designed particularly for the needs of videogames, but the library is usable for non-game tasks, too. It is designed for user flexibility in look-and-feel (through defining certain aspects of the GUI elements in XML), as well as being adaptable to the user’s choice in tools and operating systems.

The strength of CEGUIs design is that it is highly configurable. The CEGUI system itself does not directly load files, render windows, directly display
text, or even fetch input from the system. CEGUI interfaces with these through user-defined code.

This freedom allows the user to use CEGUI in any kind of resource management system or operating environment. Input is expected to be gathered by the user's code, possibly filtered as the user sees fit, and then delivered to the CEGUI for window processing.

Rendering is accomplished by a back-end module—in my case by Ogre.

File loading and resource management are handled through APIs into user code. The user is able to define in CEGUI how files are loaded, how memory is allocated, and various other basic tasks. This allows the library to be used in virtually any coding environment.

ODE/OpenDE[7] The Open Dynamics Engine (ODE or lately OpenDE) is a physics engine. Its two main components are a rigid body dynamics simulation engine and a collision detection engine.

It is used for simulating the dynamic interactions between physical bodies in space. It is not tied to any particular graphics package. It supports several geometries: Box, sphere, capsule (cylinder capped with hemispheres), Trimesh, cylinder and heightmap.

It is used as EASIMENs physics engine through the aforementioned object-oriented wrapper, OgreOde.

3.2 BIAR

The working of each BIAR component is based on iterative execution. This means, that every component should provide its functionality in a way, that can be completed in parts or it consists of relatively short, almost 'atomic' actions—it is based on ‘multi-step processing’, so that tasks could be done in pseudo-parallel.

As a consequence, each component has a way to ‘step’ itself. Running the AI system can be abstracted to the cycling of the following two steps:

1. ‘step’ every component
2. propagate all Results to their connected Options

Hereafter, only the implementation of the more interesting BIAR components will be discussed.

Brain

It implements all the decision-making and controlling capabilities of the agent (Creature) in question.

The stepping of the Brain in a meaningful way, as referred to in the Overview, has several stages. By meaningful I mean that it not just iterates though all its Subsystems and makes tries to ‘run’ them even if they cannot (e.g. waiting for a Value of an Essential Option to be propagated to it). This behavior would waste a lot of precious time given a multitude of Subsystems, and therefore make the whole simulation unnecessarily slow.
The principle behind the working of the Brain can be described as follows:

Initial state: finished and not initialized on start

1. Initialize/Reorganize: make a Subsystem ‘ordering’ according to dependence: first add those, which have all Essential Options connected from the ‘outside’ (a non-Subsystem) or have no Essential Options; then add those connected to them; then their followers, etc.—as if in step (2) we were assuming, that all Options connected to from outside the Brain are New

2. create the ‘stepping list’ according to the following rules, by iterating through all the items from the ‘ordering’ created in step (1):
   (a) add all Subsystems (in the order as they occur in the ‘ordering’ made in step (1)), which have all Essential Options New (in reality), to the list
   (b) add those, who would have all Essential Options New having the Results of the ones already in the list (and the input from ‘outside’)
   (c) if iterated through the ‘ordering’, then end this phase – the ‘stepping list’ is finished –, else continue with step (b)

3. Stepping the ‘thought-cycle’: iterate through all the Subsystems in the ‘stepping list’: if a Subsystem is finished or has Usable Result(s), propagate the Values of their (Usable) Results in case any of them is connected to a non-Subsystem; remove those who are finished

4. if any of the Options connected to from outside the Brain becomes New, recreate the ‘stepping list’ (the ‘ordering’ remains) using step (2)

5. if there are no Subsystems in the ‘stepping list’, the Brain is finished, else continue with step (3)

NOTE: If at any point new Connections are made (‘inner’, to Subsystems, or ‘outer’, to non-Subsystems), restart from step (1). This shouldn’t cause a significant drawback, as making Connections should not occur frequently compared to iteration steps.

NOTE: All Options considered above are Essential CONNECTED Options!

Subsystem

The implementation of a Subsystem is a class which is a common ancestor to all the specific Subsystem implementations.

If a Subsystem needs much computation to complete, but the underlying process can be divided into iteration steps (e.g. if it has a cycle), one of these iteration steps should be implemented in the Iterate() method, and the pre-iteration and post-iteration steps should be put into the InitBeforeNewIteration() and FinishAfterIterationCompleted() methods respectively.

If it is a one-step computation, then either Iterate() returning always true or FinishAfterIterationCompleted() can be used to implement it. For further information on the iteration process see the description of methods initialize() and think().

NOTE: This class should not be instantiated!

initialize() Tries to initialize the class before a new iteration. Returning true means that the Subsystem is initialized. Also, a state flag is set in the Subsystem, which can be queried by the isInitialized() method.

Remarks:
• If an iteration has not finished yet, it fails automatically (an ongoing iteration can not be initialized!).

• If the Subsystem is finished and has already been initialized, then it succeeds immediately (nothing to do).

• If the Subsystem is finished but not initialized, then it tries to initialize by calling InitBeforeNewIteration() method.

NOTE: If the initialization succeeded, the iterations-overflow flag is unset, as the previous state should not concern a new iteration. The flag is reset just after the initialization succeeded, so its state could be checked through the initialization process if needed.

NOTE: This method should only be used by the entity which makes the Subsystem ‘think’, therefore that entity should be made a friend.

think()  It tries to iterate (step the Subsystem).

If true is returned, it means that the iteration has finished, else the same iteration needs more iterating. Also, a state flag is set in the Subsystem, which can be queried by the isFinished() method.

Remarks:

• If the Subsystem is initialized and finished, it starts a new iteration with Iterate(1).

• If it is not initialized and not finished, then there is an ongoing iteration: the iterations counter is incremented by 1, then

  – if it is greater then 0, then the Iterate() method is called with the new counter to step the iteration further.

  – if it is 0, that means that the iterations counter overflowed (being unsigned), which is effectively a safeguard upper cap to eliminate infinitely processing a Subsystem. In this situation the HandleIterationsOverflow() method is called to determine what to do next. By default, it returns true, making the iteration finish and setting the iterations-overflow flag to indicate that the iteration finished not because it has reached its goal.

  If iteration is still required after this cap has been reached, descendants can re-implement the HandleIterationsOverflow() method and e.g. increasing one of their own attributes on overflows to keep track of the ‘real’ number of iterations (real iterations = IterationsMax * some overflow counter + iterations) and returning false to continue the iteration, or finally ending it on an overflow (the above equation for real iterations is still correct, since the iterations counter has overflown to 0).

• If it is initialized and not finished, then somehow the Subsystem got into an invalid state and an Exception of type ERR_INVALID_STATE is thrown.

• If it is not initialized and finished, it means that the last iteration (if there was one) has finished, so true is returned immediately.
NOTE: This method should only be used by the entity which makes the Subsystem ‘think’, therefore that entity should be made a friend.

Algorithm The whole principle behind the working of a Subsystem can be described by the following algorithm:

Initial state: \texttt{finished} and \texttt{not initialized} on start

1. if \texttt{finished} and an Option becomes \texttt{New}, checks whether all \texttt{Essential} Options are \texttt{New}...
   (a) if so, calls \texttt{initialize()}
   (b) else, stays \texttt{finished}

2. \texttt{initialize()}: 
   (a) if \texttt{not finished}, cannot initialize (stay \texttt{ uninitialized} and \texttt{ unfinished} - return false)
   (b) if \texttt{finished} and \texttt{initialized}, then nothing to do, stays \texttt{ initialized} (return true)
   (c) if \texttt{finished} but \texttt{not initialized}, then call the virtual \texttt{InitBeforeNewIteration()} method (which is empty by default and returns true if not overloaded in the descendant – a specific Subsystem implementation)...
      i. if that returns true, unset the \texttt{New} state of all the Options (\texttt{Essentials} and not alike) and prepare them for usage, then set \texttt{isInitialized} to true, so the iteration could begin
      ii. else return false (stay \texttt{ uninitialized})

3. \texttt{think()}: 
   (a) if \texttt{initialized} and \texttt{finished}, then start a new iteration: unset the \texttt{initialized} flag, set the iterations counter to 1 and call the virtual \texttt{Iterate()} method with the reset counter (which is empty by default and returns true if not overloaded)...
      i. if \texttt{Iterate()} returns true, then the iteration has \texttt{finished} with the first step, so call \texttt{FinishAfterIterationCompleted()} (which is empty by default if not overloaded), set the \texttt{finished} flag to true and return true (become \texttt{ finished})
      ii. else set the \texttt{finished} flag to false and return false (the iteration is \texttt{ not yet finished})
   (b) if \texttt{initialized} and \texttt{not finished}, then somehow the subsystem got into an invalid state, throw an exception
   (c) if \texttt{not initialized} and \texttt{finished}, then there were no iterations yet or the previous iteration ended – the algorithm is in the initial state –, nothing to do (stay \texttt{ finished})
   (d) if \texttt{not initialized} and \texttt{not finished}, then there is an ongoing iteration—step the iteration: increment the iterations counter and reset it to 0 if it is greater then the \texttt{MaxIteration} variable (which is UInt(-1) - the maximal possible value - by default, leading to a natural overflow to 0 )...
i. if it is non-zero after incrementing, then call \textit{Iterate()} with the incremented counter...

A. if it returns true, then the iteration process \textit{finished}, so call \textit{FinishAfterIterationCompleted()}, set the \textit{finished} flag to true and return true (become \textit{finished})

B. else the iteration has \textit{not} yet \textit{finished}, leave the \textit{finished} flag on false and return false (stay \textit{unfinished})

ii. if the iterations counter is 0 after incrementing, then the counter \textit{overflowed} – reaching a default upper cap (stepping over \textit{MaxIterations}) to prevent infinite iterations –, so handle the situation by calling the virtual \textit{HandleIterationsOverflow()} method (which is empty by default and returns true if not overloaded)...

A. if it returns true, then the iteration had to be stopped, so set the \textit{finished} flag to true, and also set the \textit{overflowed} flag to true, to indicate than the iteration has been forced to finish by reaching the upper iterations cap

B. else the overflow-situation has been handled, and the iteration should continue, so set the \textit{finished} flag to false and also set the \textit{overflow} flag to false (as the overflow-situation has been handled), finally return false (staying \textit{not finished}).

\textbf{NOTE:} If the Subsystem continues the iteration, the counter is incremented before checking zero-state, thus automatically restarting from 1.

4. if becomes \textit{finished}, checks whether all \textbf{Essential} Options are \textit{New}...

   (a) if so, calls \textit{initialize()}

   (b) else, stays \textit{finished}

\textbf{Option}

This component represents a parameter of a Subsystem or a Sensor, however, it is not merely a structure to hold a value. It is a ‘smart’ parameter, which also defines how the Value of the parameter should be handled when ‘used’ or updated.

The behavior of an Option is defined by the following attributes:

\textbf{Value} Holds the value which was propagated to the Option the last time, or a \textit{default Value} (which is described later in this section). It is of a specific type (as referred to above).

\textbf{New?} If the Value of an Option is New, that means that the Option has not yet been ‘used’ (acquired). The Option becomes New after a successful propagation of a Result Value to it. If the Value gets ‘used’, it loses its New state immediately.
Essential? If the Option is Essential, it can be ‘used’ only if it is New. Otherwise, whatever older Value it has can be ‘used’.

Timeout A counter is set with the value of the Timeout on every successful propagation. If it has been set to 0, the Option will never ‘expire’. If it has been set to a value greater than 1, the counter is decremented in every iteration cycle of the simulation. When the counter reaches 1, the Option Value ‘expires’.

Update-preference Refers to the rules that apply to setting a New Value – by overwriting the previous one – for an Option. This usually happens by means of propagating the Value of a Result to the Option. It can either be weak or strong.

weak The Value of the Option can always be overwritten by a New one— even if it is still New and its Value has not yet ‘expired’.

strong If the Option has been ‘used’ (consequently its Value is not New anymore) or it has ‘expired’, it can be overwritten. However, until the Value is still New and it has not ‘expired’, it cannot be overwritten by another New Value.

default Value An Option can also have a default Value. Having a default value can be described by the following rules considering the states of its attributes:

- If it has a default Value, it doesn’t have to be connected on a Result, so it’s not Essential.
- This Value has not been propagated to this Option, so it’s not New. Also, this ensures that the default Value can be overwritten on propagation.
- In order to be able to use the default Value at any time, it will not expire.

As the reader could have noticed, the Update-preference does not play a role in an Option having a default Value. As a result, it can have any Update-preference it pleases.

As a consequence of the Option being designed this way, after setting/propagating a New Value to the Option, the default Value will be overwritten to this New Value, and becomes the default afterward.

Also, an Option can be ‘set up’ to change its Update-preference, Timeout or Essential-state.

When a component having an Option tries to ‘use’ (acquire) its Value, the following restrictions apply:

- If the Option is Essential and its Value has already been ‘used’ (not New), then the Value cannot be ‘used’ (acquired).
- If the Value of the Option ‘expired’, then it cannot be ‘used’ (acquired).
Result

Results are much more simple than Options: their purpose is to store a Value resulting from the parent component’s operation.

Results have the following attributes:

**Value** Holds the Value of the Result. It is of a specific type (as referred to above). This is the Value corresponding to that which is set in Options on propagation.

**Usable?** If the parent component is, say, still iterating, but the iterated Value is ‘near enough’, it can be set to enable the usage of the Result before the end of the iteration. In this case, the Value of the Result will be propagated to the connected Options as New after every subsequent iteration step until the iteration is finished.

**Warning for ‘module-implementers’**: In order for this to work as intended, the programmer implementing the component having a Result should mind these guidelines.

‘Primitive’ types

As referred to before, Options, Results, and therefore Channels, can be of ‘primitive’ types.

For the current implementation of BIAR, these basic types were made available:

- **Bool** for logical values and flags or switches,
- **Int** for computations using discreet, integer values,
- **UInt** for cases needing just positive values,
- **Real** for more continuous representations,
- **String** for sending text or other types of messages.

Also, **vector** variants of all these types are available to send batches of values at once in case of producing multiple values of the same ‘meaning’ in scope of a single iteration (e.g. a Result responsible for providing the X co-ordinate of a detected object in a Sensor could use a **RealVector** instead of a **Real** type to be able to send more values in the case of detecting more objects at once).

Additionally, every class and function in BIAR uses these types, so there should not occur the problem of converting types improperly.

ModuleHandler

This component is not directly part of the AI architecture, as it has nothing to do with AI. On the other hand, this component allows for the modularity of BIAR.

It dynamically (at runtime) loads DLLs (dynamically linked libraries) in a MS/Windows environment or SOs (shared objects) when used under a Unix-type operating system.
It calls the global function `registerImpls()` when loading a module or `unregisterImpls()` when unloading. These two functions should register or unregister component-factory implementations in the factory-manager objects respectively. Then, when a certain type of ‘implementable’ component – namely a Subsystem, Actuator, Sensor or World – is created, the factory-managers call the appropriate functions of the registered factory implemented in the dynamic module.

3.3 EASIMEN

Here I will explain some more details on EASIMEN’s implementation. However, it will not be discussed in such detail as BIAR as its most important aspects have already been outlined in chapter Overview.

Engine

A property of the Engine probably worth mentioning from an implementation perspective is the optimized main cycle.

As mentioned before, the Engine supports running its main cycle in 2 modes: real-time and constant-step. The current implementation makes it possible to switch the cycle-mode at any time during the runtime of the application, however, this mode is common to all simulations—the current Engine architecture runs every Simulation either in one or the other mode. This has been decided to reduce the branching in the main cycle for increased speed of execution.

Also, thanks to the stepping limits mentioned before, EASIMEN will not use up all system resources, just the amount needed to keep the stepping inside of those limits. If all the stepping is done and less time has elapsed than it is needed to comply with those limits, EASIMEN will simply ‘sleep’ until the time comes for the next stepping of the Simulations to take place. This allows for other applications to run alongside EASIMEN without it wasting resources like rendering with unreasonably high framerate.
CHAPTER 3. IMPLEMENTATION

Logger

In EASIMEN, usually 3 types of log-messages are used in accordance with the LogMessageLevel property: informational, warning and error messages.

- **Information-type** messages are logged in the beginning and the end of every called function inside EASIMEN, and during almost every significant operation. These usually tell of the success of operations and stating the current values of involved variables.

- **Warning-type** messages tell of the failure of non-critical operations, or draw attention to non-standard situations (in case something did not happen as expected), but the application could continue without significant changes or problems.

- **Error-type** messages are sent, if something went very wrong, and it causes significant changes to the way the application runs. They are also logged in most cases of fatal errors causing the application to terminate, but in these cases usually exceptions are also used.

DotSceneOgreOdeInterface

This is partially an external component, as for one, it is implemented as a software library separately from EASIMEN, and as for two, I was not the only one developing it.

I forked this project (with the permission of the original author) from the Ogre-Addon project DotSceneInterface which is an add-on module for the renderer to load a graphical scene definition from an XML document. It has been developed on MS/Windows operating system in MS/Visual Studio, and the version I used officially did not yet support other operating systems or programming environments.

My involvement consists of

- converting it to a Code::Blocks project (as I used this free open-source cross-platform IDE for the development of this work),

- making the library compile in the GCC or MinGW compilers,

- making it (almost fully) cross-platform (as of now, the loading of resources from the XML scene definition crashes under Linux, but this is not a critical issue, because those resources can be loaded using the resources.cfg configuration file for all of EASIMEN not just that particular Simulation, and EASIMEN comes buffed with resources anyway),

- some bug-fixes,

- and integrating it (to some extent) with another Ogre add-on, the OgreOde physics wrapper module, allowing for the loading of physical structures (with them being automatically connected to graphical objects) from the definitions in the same XML document the graphical scene is defined in.
3.4 Problems during the development

External libraries

I decided to use the external libraries listed above at the beginning of this chapter to save time not having to implement the functionality those libraries provide, and to have that functionality implemented on a high level. Most of these libraries have been developed for years, and have been successfully used by other projects. However, the first difficulty faced was to assemble the used external libraries to one software application. For that, I had to learn to use and to understand all the libraries separately, then make them work together. This process took a fair amount of time in the beginning of the development. In fact, much more than I hoped it would. During this period, no code has been committed to the project(s), as the testing and learning has been done in small separate test applications.

First iteration

In the first development cycle, the project now tagged as EASIMEN_old has been produced. At first, when the separate libraries have been finally assembled in a cooperative way, I found out that the OgreOde Prefab and Loader modules will not satisfy my needs in defining physics and loading those definitions. This was the time when seeing the flexibility of the DotSceneFormat project, I decided to use its framework to define and load the physics needed for the simulated scenes, too. For this task a much deeper understanding of the library was needed. After reading through the entire source code of the project several times, the creator of the library has been contacted to clarify some details. Finally the project has been successfully forked to support the definition and loading of OgreOde objects from a `.scene` scene-definition file. This is how the DotSceneOgreOdeInterface project came to be. I did not implement the loading of all OgreOde functionality, as the base framework has been already provided: if a new physical object was required, just the parsing of that object would be needed to be added to the existing implementation, which at this time wasn't that hard with the base already provided, but still, a little bit time consuming. So I rather turned my attention to another direction.

At that time I practically had the base components working together and the capability to load the physics to a scene. However, to present the GUI to control the simulation, a GUI layout composition had to be made, which was also time consuming—as perfecting the look-and-feel of an application always is. As a result, I decided to settle with only an example GUI composition to show the capabilities, but not to waste too much development time on making something pretty which doesn't work all that well yet...

Second iteration

I realized, that the core architecture of the EASIMEN_old project was not efficient and suitable enough for the application I intended to make. For example, at the end of the first iteration, most of the classes were made a singleton to be able to communicate without too much complication. It also had the CPU running at maximum. Because this was telling of the flawed design of the core, I
decided to re-implement the engine of the application. Some of the code could be refactored from the first iteration, but not much. The entire architecture of the engine has been rewritten from scratch considering the issues identified after the first iteration and some changes allowing for new possibilities. As a result of this, the engine became faster, and could now simulate multiple virtual environments at once, with economical resource management, and a way to distribute the used resources amongst the graphics rendering, physics simulation, input scanning and AI stepping. This project became the base of EASIMEN in its current state in the end.

Third iteration

At this time, the engine was ready to use, but the GUI module has not yet been added to the new implementation. It was not a priority, as there was still no AI implementation. Therefore the third iteration concentrated on the AI. First, the design of the proposed AI architecture had to reach its final form, as the initial high level proposition was impractical from the perspective of implementation.

The implementation of the AI architecture has been started as a separate library, which is not dependent on the engine or on other parts of EASIMEN so it could be used for other projects, too. Because of the lack of time, only part of the architecture has been implemented, and even that in a simpler form than proposed, to be functional without the missing parts. At the end of this iteration, the abstractions which have been implemented were the Director, the Creature (without connecting the Brain to the Body – as there was no Body yet), the Brain (without connecting Subsystems) and the Subsystem with only one example Subsystem implementation (SubsystemModule_AreaBoundSimpleReactive). This partial implementation already supported the dynamic loading of ‘modules’. At this time, only the example Subsystem implementation module existed, but the framework was already there to load any other kind of implementation (Sensors, Actuators, etc.) in a future release.

At first, the implementation used static functions to instantiate objects, and the object-pool pattern has been chosen for making new instances of objects implemented in modules. However, as it has been found to be highly inefficient for EASIMEN’s purpose, a singleton root object has been created, similarly as presented in Ogre. In the end, for the creation and ‘cloning’ of objects, the factory pattern has been chosen after all.

As the Body concept was still not implemented, no graphical or physical representation was available for the AI. Only a test application was provided to show how to create, clone, delete and step the existing parts of the implementation. Almost all functions have a log output (similarly as shown for EASIMEN before), so it was possible to observe the results of the non-graphical ‘presentation’ in an off-line fashion.

Fourth iteration

In this iteration, EASIMEN has finally been integrated with the EASIMEN_old project successfully, and was only waiting for the AI architecture to be finished.

As with regards to BIAR, the AI framework can already be used as pre-
sented in a graphical test application in the project BIAR (separate from the test framework), but the management of Subsystems (calling their `initialize()` and `think()` methods) and the propagation of Results to Options (a simple test and calling code like `option.setValue(result.getValue());`) have to be done manually by the application using BIAR (instead of this being fully automated).

The main application, EASIMEN, is ready for (demonstrational) use, but as BIAR has not been fully automatized, the code that would step the AI have been commented out (but present in the source code)—this way EASIMEN is functioning as a physical and graphical simulator in the mean time waiting for an AI module to be connected; still, the functionality regarding graphics and physics can be used even without AI.

Also, there has been an intention to create a fully modular ‘test’-framework for BIAR making it possible to demonstrate the full power of the AI architecture and allowing for a cheap way to make simple applications utilizing the functionality provided by BIAR. The test-framework is implemented, however, there are no modules utilizing it yet, as this functionality was not planned for in the initial specification of the work and the small amount of time remaining to the deadline did not allow me to implement such modules.
Chapter 4

Experimentation

As implementing EASIMEN and BIAR was an enormous task, there was not much time left for experimentation. In spite of this, I still was able to implement some modules to try out what BIAR can do.

These modules are highly simplified, and serve only demonstrational purposes, but still, they show that BIAR is working and can be used in the way it was intended.

4.1 Modules

The available modules to try are:

SubsystemModule_AreaBoundSimpleReactive  This module provides a single Subsystem implementation: Subsystem_AreaBoundSimpleReactive. This was the first module to be created and therefore it was designed to be able to express some kind of reasonable behavior even without the presence of a World (a virtual environment) to test it in. The first tests were run in a commandline environment stepping just this Subsystem and analyzing the log output it produced.

The driving algorithm behind this Subsystem is very simplistic: it has defined four ‘imaginary’ boundaries it can not cross and a threshold to evade those boundaries. If it gets nearer to a boundary than its threshold, it would steer randomly until it gets away the boundary. If not ‘near’ to a boundary, it would maintain its direction and go forward.

However, this Subsystem is quite ‘stupid’ indeed, it is already quite configurable. It has multiple Options defining its speed, the angle in which it could turn randomly, a starting direction, the radius of its body (for simplification, it has been considered as a spherical agent), its velocity, the aforementioned threshold, and the panic-switch. As comically as it sounds, the purpose of the panic-switch is the following: In the first implementation, when the agent passed its threshold, it would turn in random directions even if it was leaving the boundary. This struck me as a way of ‘panicking’ in the proximity of a boundary. Therefore I implemented a panic-switch, which would only make the agent steer if it gets even nearer to the boundary but maintaining its direction if it gets farther (but still nearer then the threshold).
WorldModule\_SimpleOgre This module provides very basic interfacing with the Ogre graphics engine but already enabling the testing of agents in a graphical environment (World).

It provides the implementations

World\_SimpleOgre which stores a reference to an Ogre::SceneManager object representing the graphical environment and using string identifiers for Ogre::Entity-s as a Representation,

Actuator\_Position3D which makes possible to alter the position of the Representation (an Ogre::Entity),

Sensor\_Position3D which gets the 3D position of the Representation, and finally

Sensor\_Radar3D which has been added much lately, detecting the ‘marked’ graphical objects inside a spherical space defined by its radar-range Option.

SubsystemModule\_SimpleAvoider is the last implemented module and it uses the Radar3D Sensor to ‘keep its distance’ from the marked objects similarly as if it had a static charge pushing it away from those objects. Therefore it has an Option named repulsion which specifies ‘how much it feels the need to get away’.

This Subsystem can be used to grant any Creature basic means of maneuverability.

**NOTE:** The Radar3D Sensor detects the Axis-Aligned Bounding Boxes (AABBs) around the graphical objects, as this is a way faster method than doing pixel level queries and as we can see in the test application provided with BIAR, this is enough for most cases.

4.2 BIAR-testOgre

This is the the name of the executable produced when building the Debug- or ReleaseTestOgre build targets in the BIAR (or EASIMEN+BIAR) workspace.

It shows a (simple) example scenario using BIAR.

There is a ‘simulation area’ in it bounded by wood-like walls, and in this area 3 Creatures are created:

- ‘Creature1’ using only an AreaBoundSimpleReactive Subsystem,
- ‘Creature2’ having just a SimpleAvoider Subsystem connected, and
- ... [drumbeat] ... ‘Creature3’ which have both types of Subsystems connected in succession.

The first has a simple linear architecture: a Position3D Sensor is connected on the Subsystem and that Subsystem is then connected on a Position3D Actuator.

The second already has multiple input: a Position3D Sensor is connected in parallel to a Radar3D Sensor on the Subsystem and that Subsystem is then connected on a Position3D Actuator as before.
As with Creature3, it has a more complicated but still simple architecture: first a Position3D Sensor is connected on the AreaBoundSR Subsystem, then the output of this Subsystem is used to provide the position for the SimpleAvoider Subsystem (instead the Position3D Sensor when considering Creature2).

When the test application is run, all these Creatures start to act and the user can observe any one of them switching targets for the camera to follow.

**Running the BIAR-testOgre application** As it can be seen in the test application, Creature1 is moving a lot but from time to time it bounces into the boundary (with the default Options defined) and doesn’t care much about anything else in the environment. Creature2 is mostly standing still as it has only the SimpleAvoider Subsystem but if any of the other agents get near it starts to move away. In the meantime, Creature3 uses both Subsystems and therefore it can evade impacts with both the boundaries and the other Creatures alike.

**NOTE:** For more information on running the test application and EASIMEN itself see the *User Documentation* provided in the attachment.

### 4.3 Summary

However, this test application has to be rebuilt and source code has to be modified to experiment with BIAR, it is still easy to do and although the simulated agents are simple in both Body and ‘mind’, it shows that BIAR is working and with more sophisticated modules implemented it could be a useful tool for learning and experimentation.

Also, there are templates for modules next to the current implementations so future modules with new functionality can be easily implemented.
Chapter 5

Conclusions

5.1 Discussion

As it has been promised, I am revisiting some of the concepts of the proposed AI architecture, BIAR, for further discussion:

**Creature**  While a Creature lacking both Brain and Body would be no good for any use, if say, we don’t create a Brain, we still can connect Sensors directly to Actuators. As a result, we could be able to construct purely ‘analog’, ‘brainless’ Creatures, which are similar in principle to the simple white-line-following robots consisting of only two photo-resistors (one on each side), two motors (one on each side), a power supply unit, and having each photo-resistor connected to the motor on the other side. The mentioned setup does not need any processing unit, yet it makes the robot follow a white (or highly reflective) path. According to the same principle, we can produce functional Creatures lacking a Brain completely. It is an other question, how beneficial this would be from a AI-learning perspective, but it shows that the architecture allows for even non-AI-driven entities in the simulations. These kinds of Creatures, as they are computationally less expensive, could be used for instance to populate a virtual World and serve as a target for observation or interaction for more complex and intelligent Creatures.

The imaginative reader could also surely propose some other application which I can not think of at the moment, so in my opinion, there is no reason to make an unjustified limitation possibly preventing future ideas to be realized.

**Subsystem**  As the definition of the Subsystem provides a generalized interface, the modules implementing it could be *internal* – specifically designed to work with this architecture –, or *external* – using some external, already implemented library or algorithm and just creating a Subsystem-wrapper around it.

Possible Subsystem implementations could include modules implementing artificial neural networks, evolutionary algorithms, finite-state-machines, rule-based-systems, script-executing modules, modules implementing formal logic, hard-wired ‘procedural’ Subsystems, etc.

My proposition for implementation should be general enough to accommodate any kind of algorithm, however, the use of a Subsystem implementation could be inefficient, if it could not be divided into ‘iterative’ steps.
CHAPTER 5. CONCLUSIONS

Body  The tree-like connection of Members could be useful for producing situations like the following:
If a Creature ‘looses’ a body-Member, the Member with all its sub-Members gets disconnected from the tree of Members. Therefore, those connections which were directed ‘through’ its parent Member will no longer be valid, but those connected directly to it (lets say, when mimicking reflex-/muscle-memory-like behavior) could still be functional. This way, behavior like in the case of a cut-off uncontrollable but still twitching body-member, could be achieved automatically given a corresponding architecture of ‘internals’ used in the Creature.

The idea of the Body having multiple Members from multiple Worlds, can lead to applications like a game, where there is a ‘material’ and a ‘spiritual’ World, and the Members in the material World account for the body of the game character, while members in the spiritual World represent its spirit or soul.

Member  A Member not having a Representation would allow for a single logical Creature to behave as a more conventional collaborative multi-agent system. In such a scenario, communication between agents (in our case sub-Members of the same Creature) is achieved by simple inter-Member Connections, therefore not being subject to the constraints of the environment. A reader more familiar with movies or computer games could imagine this as a way to create a horde of monsters controlled by one single ‘hive-mind’ communicating through a telepathic connection between them.

Representation  An example of a 3D graphical Representation could be a point, a single sphere, or a complex articulated structure.

An example of a more abstract ‘file-World’ could be the current position in a given file on the hard disk.

Actuator  An example of a ‘physical’ Actuator could be a component exerting force in a given direction with a given magnitude on the Representation of the Member it is included in (being in a physically simulated World).

An example of a textual or a binary ‘file-Actuator’ could be a component to output a single character or byte in an abstract text- or binary-file-based World implementation.

The type of Actuator altering the Brain or the Body can be implemented in an abstract World module dealing with Subsystems, Members, Actuators, Sensors, etc., changing the used components and altering their Connections.

Sensor  An example of a graphical or physical Sensor could be a component to detect the position of the Representation of the Member it is used in.

An example of a binary or text-based ‘file-Sensor’ could be a component to read a byte or character from the specified position in an abstract binary- or textfile-based World implementation.

The type of Sensor ‘sensing’ the Brain or the Body can be implemented in an abstract World module dealing with Subsystems, Members, Actuators, Sensors, etc., detecting the used components and how they are connected.
5.2 Summary

This thesis made an attempt to describe the capabilities of BIAR and EASIMEN. At this point the reader might already have some idea of the vast amount of possibilities they could offer by using and combining various AI techniques implemented as Subsystem modules interfacing with multiple Worlds (virtual environments) even at the same time.

First of all, the high configurability of both BIAR and EASIMEN allows for extensive experimentation. As said in the Introduction this experimentation could enable a better understanding of both ‘artificial’ and ‘natural’ intelligence. This property could allow it to be utilized as an educational software and as a consequence to help willing students in becoming top researchers or leading scientists.

Also, its intuitive design makes it a potential starting tool for beginners in the field of AI helping those, who would happily contribute to the ‘life sciences’, at their first steps on the road. On the other hand, even those could be able to use it who are not familiar with programming (as they for instance operate in other areas), but they are interested in AI or they could use it in their chosen profession.

I can also imagine it as a testbed for AI developers or researchers if they want to try out a new idea before implementing it specifically for their corporate software solution. This could save them time by first perfecting their design in an easy-to-use solution like the combination of BIAR and EASIMEN would be. Without the hazards of experimenting with their official work projects and with the comfort of not having to implement a framework to create and test their new AI design in, they would only have to concentrate on the true AI and let the rest be taken care of by EASIMEN and BIAR.

5.3 Limitations and further development

Because the development of such a software solution is rather time-consuming and complex in nature, more external libraries were used. Also, the main application was implemented in a simplified nature, and the AI architecture is having just a couple of demonstrational AI modules to show certain aspects of the implementation to be able to create more reasonable and useful modules and features in the future.

Because of the magnitude of the task of creating such a complex software solution, it was not possible to implement all the planned functionality and set of features during the available time to finish this thesis. However, I am convinced that the task is very doable, and using the proposed AI architecture, it could be useful not just for beginners and hobbyists in the field of artificial intelligence and artificial life, but it would also have its potential to be a useful and productive tool in the hands of more advanced users (e.g. in the field of game development or AI research) given the necessary time for it to reach a more complete state.

At the time of working on this thesis I was targeting only a single machine to simulate all the virtual environments, but it is taken into consideration to extend the capabilities of the software design by applying a distributed architecture in a future release. The possibilities presented by substituting ‘much more powerful hardware’ for the problem of multi-agent simulations being demanding with, for
example, using distributed computer systems for the simulation of such agent-
systems is promising, as with the proper separation of AI components representing
a single agent, not only distinct agents, but even such basic components of an
agent could be distributed among multiple hardware units.
Bibliography


List of Figures

2.1 BIAR design schematic .............................................. 20
2.2 EASIMEN design schematic ...................................... 27
2.3 DotSceneOgreOdeInterface – physics XML element structure ... 32

3.1 Planned EASIMEN component structure ....................... 43