NTNU
Norwegian University of
Science and Technology

Intelligent agents in computer games

Karl Syvert Løland

Master of Science in Computer Science
Submission date: June 2008
Supervisor: Helge Langseth, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This project will look on the possibility for a computer to play a game using the same percepts and actions as a human player. Is this possible to do without using too much of the resources available and stalling the computer game? Is this possible without having to "cheat" and get sensory data from the game-engine. This is something we are going to test using the Quake II game from ID Software and especially look into being able to map the environment and identify enemies using computer vision. We hope to be able to at least navigate around the environment using the learned map and avoiding obstacles using computer vision. The choice of agent architecture are going to be an important part of making this work.

Assignment given: 21. January 2008
Supervisor: Helge Langseth, IDI

NTNU

Det skapende universitet

Intelligent agents in computer games

TDT 4900 Computer Science, masters thesis

Karl Syvert Løland

PREFACE

This report is a result of the project from the course TDT 4900 Computer science master thesis. The project was performed spring 2008, by a master student from the masters degree program in computer science at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The project continues on previous work done by Karl Syvert Løland and Stein Gunnar Grastveit.

The assignment was to check if a intelligent agent can learn to play a game using the same inputs and outputs like a human uses and what agent architecture would be best for this purpose. The programming and designing in this assignment was done as a pair, but we ended up writing separate reports.

I would like to thank my supervisor Associate Professor Helge Langseth for guidance and help trough the project, Stein Gunnar Grastveit for putting up with working with me on this assignment and John Eggett for being the sport he is and for letting me interview him regarding the subject this thesis brings up.

Abstract

In this project we examine whether or not a intelligent agent can learn how to play a computer game using the same inputs and outputs as a human. An agent architecture is chosen, implemented, and tested on a standard first person shooter game to see if it can learn how to play that game and find a goal in that game.

We conclude the report by discussing potential improvements to the current implementation.

Trondheim, June 23, 2008

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Description

This project will look on the possibility for a computer to play a game using the same percepts and actions as a human player. Is this possible to do without using too much of the resources available and stalling the computer game? Is this possible without having to "cheat" and get sensory data from the game-engine. This is something we are going to test using the Quake II game from ID Software and especially look into being able to map the environment and identify enemies using computer vision. We hope to be able to at least navigate around the environment using the learned map and avoiding obstacles using computer vision. The choice of agent architecture are going to be an important part of making this work.

1.2 Introduction

When people are playing a first person shooter game they use strategy and try to find the best solution to a situation. Humans not only have an image on a computer screen, but memories about the past are also used to solve a situation or avoid a dangerous situation. When a human player explores some part of the game environment, he will try to map the area into memory and make a plan and goals for where to go next. When moving around like this and suddenly an enemy jump around a corner the human will react fast to avoid the danger.

The motivation behind this project is to make an intelligent agent that are able to play a first person shooter like a human player. Let the computer interact with the game the same way a human player does, and have the same perceptions.

1.3 The scope of this report

In this report I am going to mainly concentrate on the agent architectural part of this project. I will mention the parts on this project that's explicitly done by Stein Gunnar Grastveit at a design level only, and not go in any depth on parts that I have not participated in. (The computer vision part.)

This report will go in detail on the InteRRaP architecture and why this architecture may be good to use in computer games and test this. An implementation of the InteRRaP architecture to try to test this will be made and explained in detail.

The report will then test the use of the InteRRaP architecture and discuss what can be changed, if anything, to make the InteRRaP architecture better for use in computer games.

Chapter 2

Agent architectures

This chapter will make you an introduction to what an intelligent agent is and then continue to explain two simple agent types. Ending with explanation and introduction to two hybrid architectures between the two agent types.

There are no universally accepted definition of the term agent. Since it would be hard to continue explaining about agent architectures without having a definition I will present the agent definition suggested by Michael Wooldridge in [Woo05]:

> *An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.*

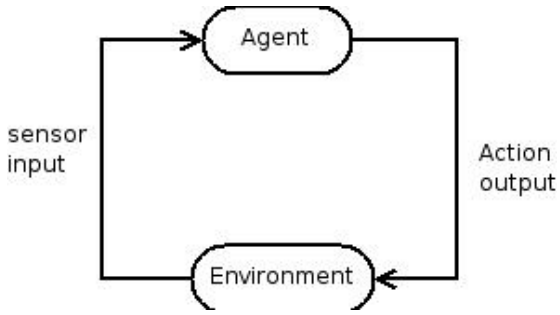

Figure 2.1: An agent in its environment.

Using that definition Wooldrige proposed figure 2.1 illustrates an abstract view of an agent in its environment.

2.1 Purely reactive agents

Purely reactive agents are agents that choose their actions without any reference to prior perception information (the past). This type of agent chooses it action all based on current perceptional information (the present). This type of agent does not do any other than just respond directly to the environment they exists in. So in other words they only iterate trough all perception data and react to it if there is anything to react to. A purely reactive agent can be represented abstractly in the following function:

$$Ag : E \rightarrow Ac$$

A small example of a purely reactive agent would be an agent that controls the water-level in a dam. And lets water out trough the watergate by opening it when the water hits a certain level, and closes the gate when the water hits a certain level.

2.2 Utility-based agents

A utility based agent are an agent that have several available runs. Each run contains a set of actions to perform a certain goal. A utility is a numeric evaluation on how good a particular run is given the current perception data from the environment. A utility based agent contains a function that calculates the utility for a run as a real number based on the data from the perceptions from the environment. This utility function, as it can be called, can be represented abstractly as follows:

$$u : R \rightarrow R$$

The agent performs the run with the highest utility. This ensures that the agent is trying to maximize its own performance.

2.3 Goal based agents

Goal based agents are agents that make a plan of actions to perform to satisfy their goal. This selection of actions are often complex and searching and planning are used. The plans are are made given the goal and percepts. This function then checks if the goal is satisfied in the percepts already. If so then no plan will be generated. The planning involved in selecting actions can involve a history of previous information from the percepts and use pattern in these to recognize what needs to be performed.

2.4 What makes an agent rational

The ability of being rational is the ability to choose its actions based the information from the environment and on knowledge about the environment it exists in. Knowledge about the environment includes information of its past experiences and how the environment evolves or behaves. Wooldridge and Jennings have in [WJ94] come up with a list of what kind of abilities a rational agent is expected to have:

> *Reactivity.* Rational agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives.
>
> *Proactiveness.* Rational agents are able to exhibit goal-directed behavior by taking the initiative in order to satisfy their design objectives.
>
> *Social ability.* Rational agents are capable of interacting with other agents (and possibly humans) in order to satisfy their design objectives.

2.5 BDI agents

BDI agents are an example of rational agents. The BDI architecture are inspired by the Belief-Desire-Intension model of human practical reasoning that was developed by Michael Bratman in [Bra87]. This model that Bratman developed was a way of explaining the human practical reasoning. BDI agents are developed with Beliefs, Desires and Intensions.

Beliefs represent what the agent believe about the environment and itself. New beliefs are generated from perceptional data and old beliefs. A belief that an agent has can be wrong, and can be changed in the future.

Desires represent which motivations the agent have. These desires are more what an agent wants to accomplish then a goal. Using the term goal we add a restriction, since the set of goals must be consistent [Spa03] . By consistent I mean that an agent should not have concurrent goals like a goal to go out and another goal to stay at home.

Intensions represent the desire that the agent has chosen to do and committed to. The intension are chosen by the agent based on the agents beliefs and desires.

2.6 Hybrid architectures

For an agent to be capable of both reactive and proactive behavior, an introduction of a hierarchy of interacting subsystem layers is a well used option to solve this. It

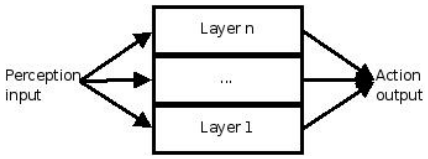

Figure 2.2: Horisontal Layering

is at least two layers in a hybrid architecture, one to deal with reactive behavior and one to deal with proactive behavior. Horizontal layering are a layered architecture where all layers are connected to the perception input and action output. Figure 2.2 shows an abstract overview over horizontal layering. A horizontal architecture are most often in need of a control function that decides which of the layer that has "control" of the agent at the time. This control function can be considered a bottleneck in the decision making of the agent.

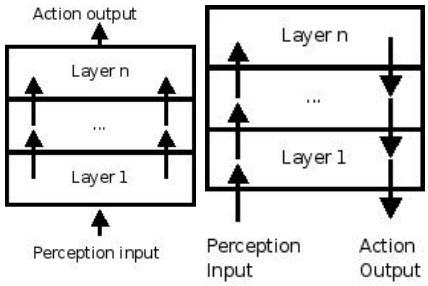

Figure 2.3: Vertical Layering. Left: One pass control. Right: Two pass control.

Vertical layering are a layered architecture where at least one layer deals with the sensor input and action output. There are two types of vertical layered architectures. One is *one-pass* architecture and the other is *two-pass* architecture. One-pass architecture is a vertical layered architecture where the control and information flows sequentially trough each layer, until the final layer generates an action output. An abstract overview of one-pass architecture are illustrated in the left part of figure 2.3. Two-pass architecture is a vertical layered architecture where the information flows up the layers and control flows down. An abstract overview of a two-pass architecture are illustrated in the right part of figure 2.3.

2.6.1 TouringMachines

The TouringMachines architecture consists of three layers that continually produces suggestions of actions for the agent to perform. The TouringMachines architecture are a horizontal layered architecture with a control subsystem which is

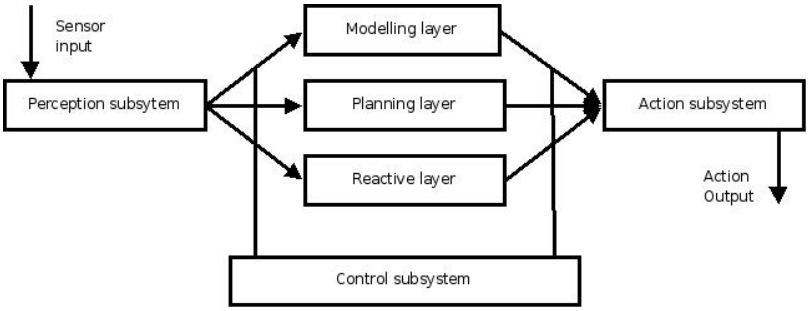

Figure 2.4: TouringMachines

responsible for deciding which of the layers that should have control of the agent at any given time. The control subsystem is implemented as a set of control rules which can either suppress sensor information from reaching the control layers, or censor actions chosen by a control layer. Figure 2.4 gives an abstract overview of the TouringMachines architecture.

The reactive layer in the architecture provides the architecture with the ability to respond immediate to changes or events in the environment. This layer maps the perception input to a desired action, just like a purely reactive agent.

The planning layer in the architecture provides the architecture with a proactive behavior. The planning layer does not create plans from scratch but rather uses a library of "skeleton" plans called *schemas*. So to achieve a goal the planning layer searches trough the library of schemas to find one that matches that particular goal.

The modeling layer in the architecture provides the architecture with representations of the various entities in the environment. The modeling layer generates goals which it passes down to the planning layer to solve. These goals are generated using the representations within the modeling layer so the agent can avoid conflicts with other agents.

2.6.2 InterRRaP

The InteRRaP architecture is a vertically layered two-pass architecture that contains three layers. The bottom layer is the behavior layer which deals with reactive behavior. The middle layer is the local planning layer which is responsible for the proactive behavior and uses planning to achieve the goals. The top layer is the layer responsible for the agent's social interactions.

All of these three layers are connected to a knowledge base that represents the environment in a different lever of abstraction that's appropriate for the connected layer.

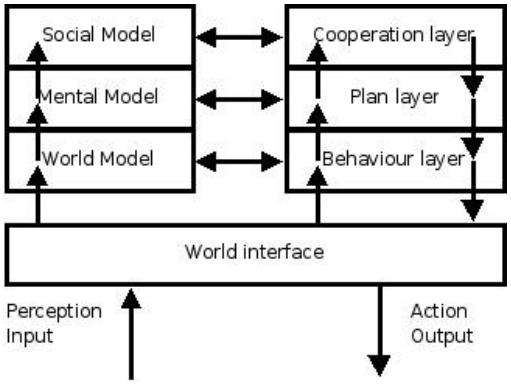

Figure 2.5: InteRRaP

The layers in the InteRRaP architecture interact with each other using bottom-up activation and top-down execution to achieve the same end. Using bottom-up activation the layers passes control to the layer above if itself are not capable to deal with the current situation. Top-down execution is used for a layer to make use of facilities available by the layer beneath to achieve its goal(s).

So if the bottom reactive layer can deal with the current situation it does so, if not it passes the control up to the local planning layer. If the local planning layer can deal with the situation it uses top-down execution to achieve its goal(s), if not then it passes the control up to the cooperative layer which again uses top-down execution to deal with the situation if it can so, if it can't solve the task then an empty top-down execution occurs.

Chapter 3

The environment

The environment is the world where the agent operates in. An environment can have many properties that can be classified. Russel and Norvig suggest the following classifications. [RN03]

> *Accessible versus inaccessible:* An accessible environment is one which the agent can obtain complete, accurate, up-to-date information about the environment's state. Most real-world environments (including, for example, the everyday physical world and the Internet) are not accessible in this sense.
>
> *Deterministic versus non-deterministic:* A deterministic environment is one in which any action has a single guaranteed effect - there is no uncertainty about the state that will result from performing an action.
>
> *Static versus dynamic:* A static environment is one that can be assumed to remain unchanged except by the performance of actions by the agent. In contrast, a dynamic environment is one that has other processes operating on it, and which hence changes in ways beyond the agents control. The physical world is a highly dynamic environment, as is the Internet.
>
> *Discrete versus continuous:* An environment is discrete if there are fixed, finite number of actions and percepts in it.

The more accessible an environment is, the simpler it is to build agents that operate effectively within it. This is something that is very logical, since for an agent to be good and make the right decisions it depends on good and accurate information from the environment. An agent that do get inaccurate information from the environment will make bad decisions more often.

3.1 Quake II

Quake II is a game developed by IDSoftware [IDS97] and released to the public in 1997 . The main person behind this game is John Carmac, one of the worlds best game developers known for his mathematical skills and ability to create lightweight games that looks good. [Wik08b] Carmac "invented" the 3D FPS genera with his game Wolfenstein 3D. It existed first-person games before Wolfenstein 3D but none of them utilized a gun and shooting. Quake II the environment that are used by the agent in this project.

Quake II lets the user to chose between Software or SDL/OpenGL to render the graphics. To be able to get the screenshot from the display-card the game must use SDL/OpenGL rendering. This is because if the game was software rendered there would be no framebuffer and depthbuffer on the graphics-card to get the screenshot and the depth image from. A typical screenshot of the game is shown in figure 3.1.

Quake II was released under a GPL license in 2001. This made the source-code open and made it possible for the public to look at in and change it and build modifications of the game or completely new games based in the engine. The possibility for changing the source-code of the game was the main reason for us to use this game as an environment. Its source-code is written completely in C, and are fairly easily readable.

3.2 Quake II as an environment

Based on the classifications suggested by Russel and Norvig [RN03] we can say that Quake II is an inaccessible, non-deterministic, dynamic and continuos. Quake II is an inaccessible environment since we cannot obtain all information about the state of the environment and there is no guarantee that the information we obtain from the environment is complete or accurate. In Quake II there are no guarantee that our actions have the preferred effect that we want, this makes it a non-deterministic environment as well. In Quake II our agent will also not be alone, so there is other agents (Non Playing Characters or bots as they are called in the game) that also can change the environment. This makes Quake II a dynamic environment. The amount of actions and percepts in Quake II are not a fixed one. These classifications of the environment is something we have to have in the back of our heads when we are choosing an architecture to use within it.

Figure 3.1: Screenshot of the Quake II game

3.3 Agent architecture for use in Quake II

A game uses a lot of the available resources on a computer and requires the player to often react very quickly to get further in the game. Norling discusses in [Nor03] the possibility to use a BDI agent to model human behavior in games, based on how humans answer when asked how they think. Using a hybrid architecture with a reactive layer and a planning layer would satisfy a BDI architecture for planning and a reactive architecture for quickly reacting.

Both InteRRaP and TouringMachines are a hybrid architecture with proactivity and reactivity. TouringMachines is a horizontal layered architecture and requires a control subsystem that can be a bottleneck into the agents decision making, and also require us as developers to make control rules to control the flow of information in the architecture. InteRRaP on the other hand is a two-pass agent architecture and does not require a control subsystem since the architecture itself takes care of the information flow. InteRRaP also have the benefit of having a knowledge base that makes it able to better represent the environment for the different layers. It is also assumed that since there is no control subsystem needed for InteRRaP and all layers in InteRRaP does not create an action at the same time like in TouringMachines, that it do require a little less computational resources when running. InteRRaP seems to be the architecture that would fit best into a computer game.

Chapter 4

The InteRRaP agent architecture

This chapter contains a brief introduction to the InteRRaP agent architecture as explained in [FMP94]. There exists several versions of the InteRRaP architecture and the most recent one is presented in muller-pischel94c [FMP94]. The InteRRaP architecture is an extension of the RATMAN architecture [BM91]. It was developed for interacting robots. An evaluation of InteRRaP is performed in the loading-dock application described in [MP93].

4.1 The overall structure

InteRRaP is a layered architecture where the signals and data go bottoms up and the actions go top down. Figure 4.1 shows a good overview of the architecture. The world interface (section: 4.2) are the lowest layer and is the layer that interacts with the environment. The world interface sends sensory and perception data to the world model (section: 4.4.1)and performs actions.

The behavior based layer (section: 4.3.1) is a reactive layer and reacts to known situations and events. If there are not any known events or situations where the behavior based layer knows how to solve it, then the signals and data are sent upwards to the local planning layer (section: 4.3.2).

The local planning layer is a goal based layer. The local planning layer checks of there are any known situations that it recognize and has a plan ready for. If it has a plan for the current situation it performs the actions of the plans by sending the actions down to the behavior based layer that sends in to the world interface and performs them. If the local planning layers does not recognize then the signals and data are sent upwards to the cooperative planning layer (section: 4.3.3).

The cooperative planning layer is a layer that creates joint plans, i.e., plans that resolve or avoid conflicts and allow several agents to cooperate in joint goals.

To provide a better understanding on the general principle about how the

behavior based layer, local planning layer and cooperate planning layers works together I will provide an everyday example of this:

> *Imagine that you are sitting in your office and you feel like you want a cup of coffee. The planning layer makes a plan for you to go the canteen and you head out of your office heading for the canteen. The plan generated consists of a route down the corridor, down the steps and into the canteen that are located at the ground floor. You are heading down the corridor to the steps when a coworker suddenly slams his door open right in front of your nose. The behavior based layer inside your brain suddenly recognizes the situation and takes over and you avoid the opening door by taking a few steps in the opposite direction of the door. You then continue on the plan to get to the canteen for a coffee. When you arrive at the canteen the plan generated by the planning layer is complete. Now you still want that cup of coffee, but you find out you don't have any money on you. Neither the behavior layer or planning layer recognizes the situation so the control is passed upward to the cooperative layer. The cooperative layer recognizes the situation and generates a plan which involve communicating to the colleague standing behind you and ask to borrow a small amount so you will be able to get yourself that cup of coffee.*

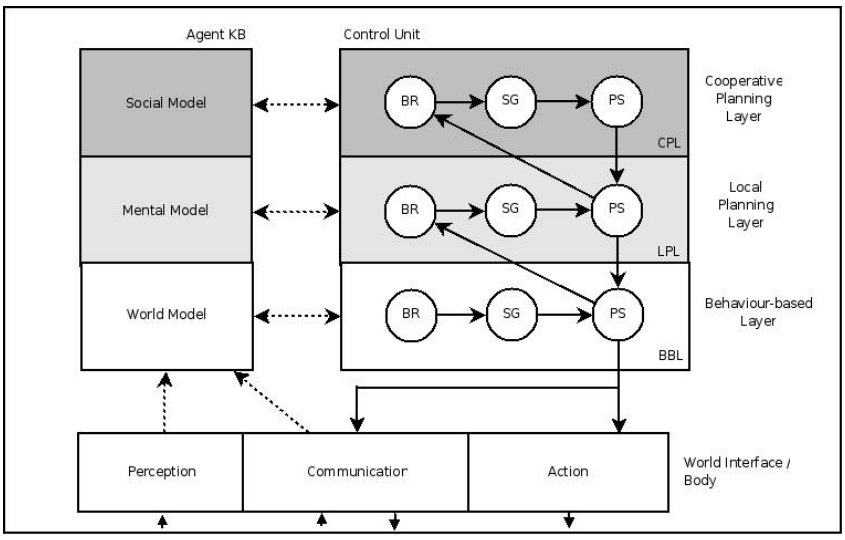

Figure 4.1: Overview of the InteRRaP architecture

4.2 World Interface

The world interface is the agents connection to the environment. The world interface performs the actions selected by the control unit (section: 4.3) on the environment. These actions are stored in the action-module within the world interface and activated by a top-down approach from the behavior based layer. The world interface contains the perceptrons available to the agent and transfers the perception data from the perceptrons to the world model (section: 4.4.1). The world interface is the part of the InteRRaP architecture that connects the agent to the environment and other agents, so it sends and receives messages with other agents and translate their responses into messages usable within our own agent if needed. An illustration of the world interface is shown in figure 4.2.

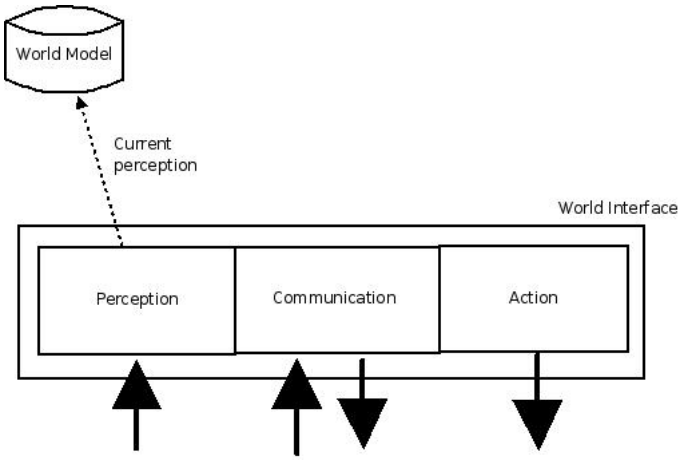

Figure 4.2: Overview of the world interface

4.3 Control Unit

The control unit is controls the flow of information between the behavior based layer, local planning layer and cooperate planning layer. The control unit encapsulates these three layers. Every layer in the control unit has three functions, belief revision (BR), situation recognition (SG) and planning and scheduling (PS). An abstract illustration of a generic layer in the control unit are shown in figure 4.3.

Belief Revision The BR function translates the current perception information and old beliefs into new beliefs.

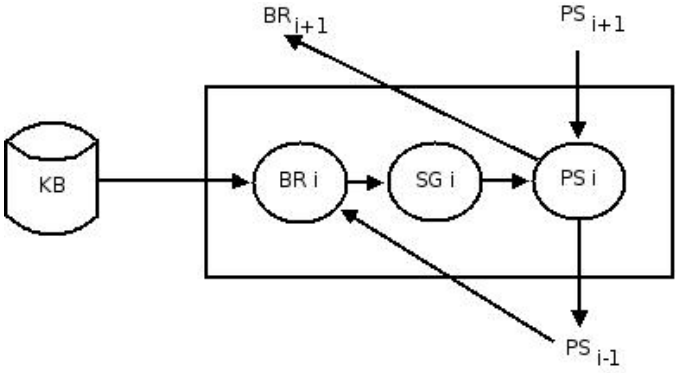

Figure 4.3: A layer in the InteRRaP architecture.

Situation Recognition and Goal Activation The SG function uses the current beliefs and goals to make new goals. When the SG function on a layer is invoked , it uses the beliefs generated by the BR and recognizes situations for this corresponding layer if there exists some. For every known situation there is a desire associated with it, and the desires of the recognized situations are sent to the PS.

Planning and Scheduling The PS maps the current beliefs, desires and intensions into new intensions. The PS chooses which of the desires, if more than one desire sent over from the SG, that will perform best given the situations and pick the plan connected to that desire and send it to the layer beneath it. If no desires are sent from the SG the PS invokes the BR of the layer above.

4.3.1 Behavior Based Layer

The behavior based layer is a reactive layer. This layer is the first layer in the control unit, and therefor have a first priority to check for situations and send actions down to the world interface to do. The behavior based layer mainly takes care of problems and situations that happen sudden and unexpected and requires a reactive action. If the behavior based layers recognizes a situation the above layers are not invoked and saves some computational power.

An example of a situation that the behavior based layer is designed to comprehend may be: A box fall down in front of an agent and the agent have to avoid hitting it.

4.3.2 Local Planning Layer

The local planning layer is a goal based layer. The local planning layer recognizes situations where a plan is needed to achieve the goal. When a situation is recognized there may be several plans that can be generated to achieve the desired goal, but only one of those plans gets chosen. The plan that gets chosen gets broken into actions that the behavior based layer can understand and passed down to the behavior based layer.

An example of a situation that the local planning layer is designed to comprehend may be: The agent has "seen" the goal and needs to find the fastest and safest possible way to it.

4.3.3 Cooperative Planning Layer

The cooperative planning layer uses information from other team-members or their location in the world to create plans to satisfy its goal, either by creating a plan to avoid conflicts with other agents in the environment or create a plan that involves the abilities of the other agents and cooperates in a joint goal.

4.4 Knowledge Base

The knowledge base contains all the knowledge within the agent. The knowledge base is designed so each of the players in the control unit have their own layer of knowledge in the knowledge base. Each layer in the knowledge base has their distinct trades that will be explained in the following sections.

4.4.1 World Model

The world model represent what the agent observe at time t. It is a representation of the reading from the perceptions to the agent. This information is overwritten by new reading from the perceptions and therefor does not store previous knowledge about what the agent have observed at time $t - 1$. The world model is used by the behavior based layer.

4.4.2 Mental Model

The mental model is a representation of the world model where observations at time $t-n$ are stored. This makes it able to remember past situations as well as the current ones. The mental model is used by the local planning layer. The mental model makes the local planning layer able to create good plans based on current situations and previous situations.

4.4.3 Social Model

The social model represents the environment and other agents in it with the abilities of the other agents. This makes the agent able to generate plans that involve cooperation with other agents to solve a common goal. This is the highest level of knowledge in the knowledge base.

Chapter 5

Design choices

In this chapter there will be discussed the design choices and the challenges we had during our design process. The possible alternative solutions will be discussed and a decision will be made on how we want to develop our prototype.

5.1 The InteRRaP library

In the pre-project we did in the autumn semester of 2007 [LG07] we created an InteRRaP library that could be used to easily and fast create an agent that used the InteRRaP architecture. This library is to be used now to ease the development process a little. This library is mainly designed after the design in [LG07] but there are some changes to the design that are explained in more detail in this chapter. A major issue with this design of the library is it's learning capabilities. In [LG07] there are mentioned several places in the architecture to implement learning. The places being the Belief Revision (BR), Planning Scheduler (PS) and the perceptions. We stated then that learning would best be used in the local planning layer to learn witch plans that are best suited for the situations that arise. How this can be done will be explained in 5.1.1. The InteRRaP library supports an easy integration with perceptions so we can fast make a new perception if we need so. The InteRRaP library has support for rules to be made and changed during runtime without the need to recompile the library. The rule-files are simple scripts and are explained more in detail in 5.5.

5.1.1 Learning in the local planning layer

As mentioned earlier there are a couple of places in a layer one can implement learning. Implementing learning in the BR would mean it would learn to better translate the perceptional input into situations. A way of implementing learning

in the PS would be to make it choose the plans to schedule for the world interface better. Learning in these places may not be the best solution. If we instead implement learning in the Situation Recognition and Goal Activation (SG) module we teach the SG to recognize what plan are best suited for a situation that occur.

To learn the connections between one or several situations and a specific plan, a matrix that connects the situations and plans with a weight can be used. This would be a simple solution that would fit right into the SG. An example of what a matrix like this might look like can be seen in figure 5.1. Every rule available in the local planning layer will be represented by a row in the matrix, and every situation known will make up the columns. New situations can be added in runtime as they occur by adding a new column and initialize its weight for every rule with the default weight the rule has. (The default-weight a rule has, is a global-variable named priority in the lua-files.) A new situation will be generated if there is any combination of perceptual inputs that never has occurred before. A situation consists of all the perceptions with a particular combination of inputs. So every situation has a different combination of perceptual input. This means that the agent is only in one situation at a time. So the plans to be chosen from have the weights that the plans have for that particular situation, and these plans with their weights would be sent to the PS for a decision.

The weights can be tuned in positive or negative direction based on the evaluation on how the plan performed when it has been executed. To be able to evaluate a plan like this, the situation that triggered the plan has to be remembered together with the knowledge of the plan that was selected for execution. When the plan is finished executing, we can then evaluate the plan and change its weights to that situation up if it succeeded or down if it failed. The way a plan is represented in the InteRRaP library from [LG07], there is no way of evaluation. The representation of the plan has to be changed in the rules and methods for evaluation have to be made. How we can chance the rule-files and the representation of the plan will be explained more in detail in 5.5.

5.1.2 The cooperative planing layer

The cooperative planning layer is the layer in the InteRRaP architecture that selects a plan that can be executed if there is a need of cooperating between 1 or more other InteRRaP agents as explained in more detail in section 4.3.3. In our case this layer need not have any rules made for it . The agent we shall make will be the only agent in the environment. If we introduced more InteRRaP agents into the environment the need for planning and learning in the layer may arise. Since we are not introducing more InteRRaP agents into the environment or think of doing it we will leave the layer ”empty” and the information-flow will go trough it as a normal layer. In this ”normal” flow the BR will not find any new beliefs, the

	S1	S2	S3	S4
P1	1.0	0.8	3.2	0.1
P2	1.1	1.6	2.1	1.1
P3	1.9	1.0	1.9	0.5

Figure 5.1: A situation/plan matrix with example values

SG will not find any desires and the PS will not chose an intension. The layer will then pass down to the local planning layer that it did not find anything it could do.

5.2 The map

In the pre-project we did in the autumn semester of 2007 we used at XY coordination based map [LG07] as shown in figure 5.2. This map expanded quadratically so the map always have the size NxM. Every tile in this map held information about the cell as well as a memory of what it had contained previously at time $t - n$. The size of this map got very big and used a lot of memory. It ended up containing a lot of information about tiles we never had visited or would end up visiting in the near future. This was not a very memory efficient way of creating a map, and searching after a path in this map would require a test to filter out all the tiles in the map that did not contain any information. During testing we discovered a big flaw with this way of making a map as well. Since the unit used in this map was a step in the environment for the agent, and since we had to wait for 3 images to come trough to see if we were at a wall. The map mapped the steps the agent took (but did not really move, since it was stuck at a wall). This made the map not very accurate since the agent got confused where in the map it was.

5.2.1 Requirements for the new map

After what we did in the pre-project in the autumn semester of 2007, we now figured out that we needed a list of requirements before we started making a map. The minimal requirements we figured out we needed based on the previous

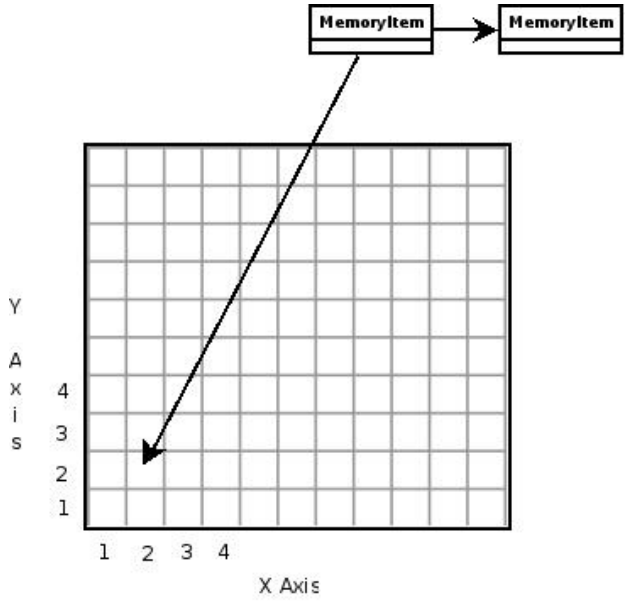

Figure 5.2: Representation of the old map with data in position (2,2)

experience as mentioned above was:

After the experience with the pre-project there no special requirements were set for the map we now wanted to have a short list of requirements for the new map ready before we began developing it. This small list was made over a discussion about what both of us meant was needed. The list ended up to be three main requirements that was the following:

- *Fast to search trough* for an A to B path.
- *Accurate mapping* from the environment to internal map representation.
- *Low memory usage.*

5.2.2 The new map

To make the map take less space it consists of tiles linked together. These tiles are linked together in a way where a tile contains a link to its neighbors in north, east, south and west direction. A tile and its linkings are shown in figure 5.3. If one of its neighbors are not yet discovered the link to it will be empty. This make the map take less space since we now discard the unexplored tiles in the map. A tile contains only the information of what it consists of, and does not have any memory as the previous map mentioned earlier had. A tile has the possibility to consist of several things like floor, wall, goal, etc.

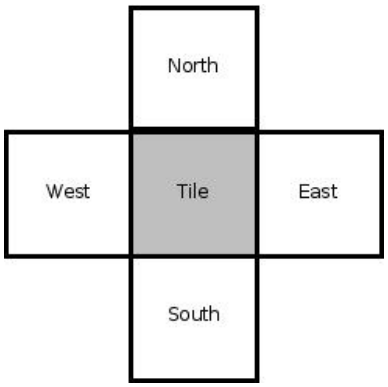

Figure 5.3: A map tile and its neighbors

To create a new tile the map needs the position in the environment it will represent. This position in the environment is used to calculate the position of the tile. A tile can represent a NxN part of the environment. When the position in the map of the tile is calculated the tile finds its neighbors in the map and links itself to them and makes them link themselves to it. This makes the new tile a part of the old map. Using the environment position instead of steps like we used in the pre-project to represent the position of the agent within the map, the accuracy of the map is as good as it can be.

5.3 The game server

The game server needs to be able to collect data and screenshots from the environment. The environment is the Quake II game explained in 3.2. To get the information we need we can ether change the source-code of Quake II or use the LD_PRELOAD instruction to load a library into the game. LD_PRELOAD is an instruction to tell the loader to load additional libraries into a program. This makes us able to load additional libraries into an already compiled program.

To change the source-code of the game would mean that the solution would not be very general for other games, and just Quake II. To make it work on other games we would have to have the source-code available for us to use it as an environment.

The solution of using LD_PRELOAD will make us able to make a shared library that we can use on more then the one game as an environment without much modification to the shared library itself. Whenever swapbuffer is called from the game, the swapbuffer method in our shared library is run instead of the method in the OpenGL library. The override of the swapbuffer method using the shared library are shown in figure 5.4 ,where the normal flow is drawn with a dotted line, and the flow using the LD_PRELOAD instruction are drawn with a solid line.

This will make the shared library only able to be used on OpenGL based games, but a very similar technique can be used on DirectX [Mic06] based games as well. This technique in DirectX is called ”DLL injection” [Wik08a].

The swapbuffer method in our shared library will have to call the actual swapbuffer method in the OpenGL library so that the game can render to screen.In our swapbuffer method we can access all global variables within the game, and call on OpenGL methods. The data from the game is then easy accessible since Quake II is implemented using the programming language C and uses global variables to store data. So this solution is the one that would fit the project best.

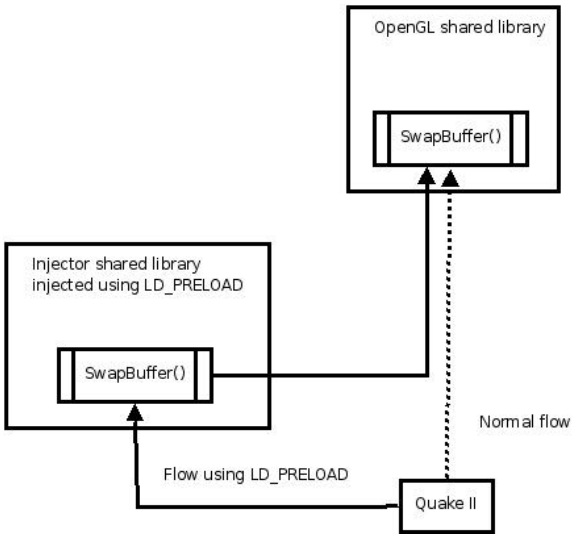

Figure 5.4: A overview of the usage of LD_PRELOAD

When we have the image data and position data needed we have to send it over to the agent. A simple TCP/IP client/server architecture will be ideal here. This will make the environment able to run on a different machine if ever needed, as well as run both the agent and environment on the same machine. To set up this we can use the network library. This network library makes it easy and fast to set up a network connection. This library is explained in more detail at section 6.4. The environment will become the server and the agent will become the client. The server (also referred to as ”the game server”) will then continuously send data from the environment to the agent for it to use. An illustration of this simple TCP/IP setup is shown in figure 5.5.

Figure 5.5: Simple TCP/IP setup

5.4 Computer Vision

To be able to utilize the health information on the image we get sent over from the game server we have to parse it somehow. An idea could be the use of training an artificial neuron network to parse the health of the player. This will involve a lengthy training process [Cal99] together with the time it will take to implement it and that is not very desirable. The health is displayed in the pictures as numbers with a black border around them as shown on the left part of figure 5.6. A quick way to parse them would then be to run them trough a threshold function that would make all thats not the color black to white, and all that is the color black to black. This would make a representation of the numbers as shown in the right part of figure 5.6. The ”cleaned” image of the number was then divided into 9 equal sections as shown in figure 5.7. The number can then be identified by counting the number of black pixels in every section. Then for every number we had a vector of nine integers that would identify it. This solution will not be very resource demanding as one can figure from the explanation of the method above.

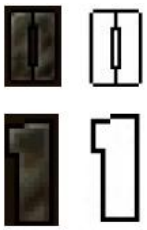

Figure 5.6: Before and after a threshold function

To be able to recognize an enemy advanced computer vision algorithms have to be applied. Instead of developing all these ourselves OpenCV can be used. OpenCV has many capabilities and the demonstration program showed us some very good results. It has AI and machine-learning methods, the possibility of image sampling and transformations, methods for computing 3D information and high-level computer vision methods [Hew07]. Using the facial recognition algorithms in

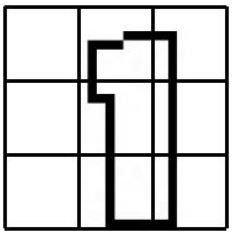

Figure 5.7: The grid used to parse the number

OpenCV we can recognize an enemy in front of us facing us.

5.5 Plans

The rules will have to be able to be evaluated. To do this there are two possible solutions. One solution is to evaluate it using an evaluation method in the InteRRaP library and the other one is to let the file containing the rule have a method to evaluate itself. To use an evaluation method in the InteRRaP library would mean that that the evaluation method need to be general for the current domain the agent exists in. It is not an easy task to make such a general evaluation method, and if one are made it would stand against the reason behind making the InteRRaP library in the first place. Since the reason to make a InteRRaP library was to make a general library that would work on several domains and not only one.

The best solution is then to let the rule-file have an evaluation method to be able to evaluate itself. The rule-file would then have to contain its goals and have them available to the evaluation method. To create variables in runtime when using the lua script language is very easy. The evaluation method then has to query the environment at the time of evaluation to check if the goals have been accomplished.

So the rule-files will now contain a global initial weight variable, a global goal variable, a method for creating a plan and a method for evaluate itself.

Chapter 6

Implementation

This chapter will explain in more detail how the different parts in this project are implemented. The code-base at the end extended to approximately 10,000 lines of a mixture between C and C++ code. The LUA code is not counted. It was developed an InteRRaP library that uses rules written in LUA, an agent that makes use of the InteRRaP library and has a GUI for debugging purposes, a server wrapping around the Quake II environment and a network library.

6.1 The InteRRaP library

The InteRRaP library is based on the InteRRaP library used in the pre-project. As stated in chapter 5 the library needed to be changed to implement learning, and thus some parts of it had to be made from scratch to support this. I will in this section give an overview of the implementation of the library, and go in more detail only where the changes were made.

6.1.1 World interface

The world interface is developed as a public class with public methods to access or invoke the different parts it contain. The world interface also contains two private classes, Action and Perception, that represents the action and perception modules shown in figure 4.2. The world interface class is shown in detail using UML in figure 6.1.

performing a plan or action

The public methods DoLplPlan(char* plan, int sID), DoBblPlan(char* plan) and Continue() are the methods used to make the world interface perform actions from a plan on the environment. The DoLplPlan and DoBblPlan methods are

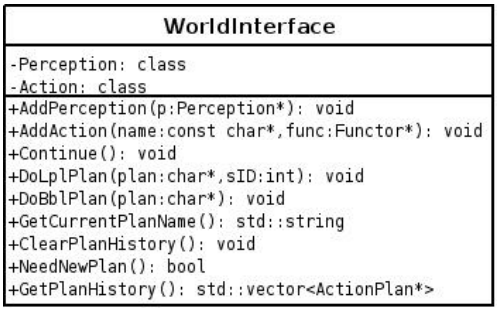

Figure 6.1: UML class diagram of the WorldInterface class

very similar, only differences are that DoLplPlan method only gives the plan to be run access to the mental model and the DoBblPlan gives the actions access to the world model. In figure 6.2 an overview of what the methods do are shown. This figure represents both methods, since the execution in them both are the same. The methods DoBblPlan and DoLplPlan are from now referred to as the *DoPlan method* for easier reading, since they are so similar.

The DoPlan methods iterates trough the lua files in the rules folder for the current layer, ether LPL or BBL, opens the file and check if its name are the same as the string-value passed as an argument. If no file that matches are found, the method vil exit. If a match is found then the name of the rule and the situation responsible for the rule to be chosen are stored as global variables within the world interface. These two global variables will from now on be referred to as plan-name (for the name of the current plan that to be executed) and SID (for the situation responsible for that plan). The SID is only set if the current layer are LPL, if the current layer are BBL the SID are set to a default value of -1. The plan is then parsed and the lua-function named Rule() 6.2.3 are executed and this produces a list containing of actions to be performed are stored to a global variable called list. The method Continue() is then called.

The Continue method performs the next action in the plan currently stored in the plan-name global variable. If this global variable is empty then it just returns doing nothing. When the next action is performed, the method checks if there are more actions left in the plan. If there are none, then this means that the plan are done executing and if it is a plan that origins from the LPL layer, it starts to evaluate it. The evaluation is done by parsing a lua-file again and executing a method in the lua file called Post() this method returns -1,0 or 1. If the return value is 1 the plan was a success, if the return value are 0 the plan returned neutral and if the return value are -1 the plan failed. The neutral return value means that the evaluation was no good and is not usable for further use in adjusting weights later on. (More on that in section 6.2.3.) When the evaluation is done, the plan-

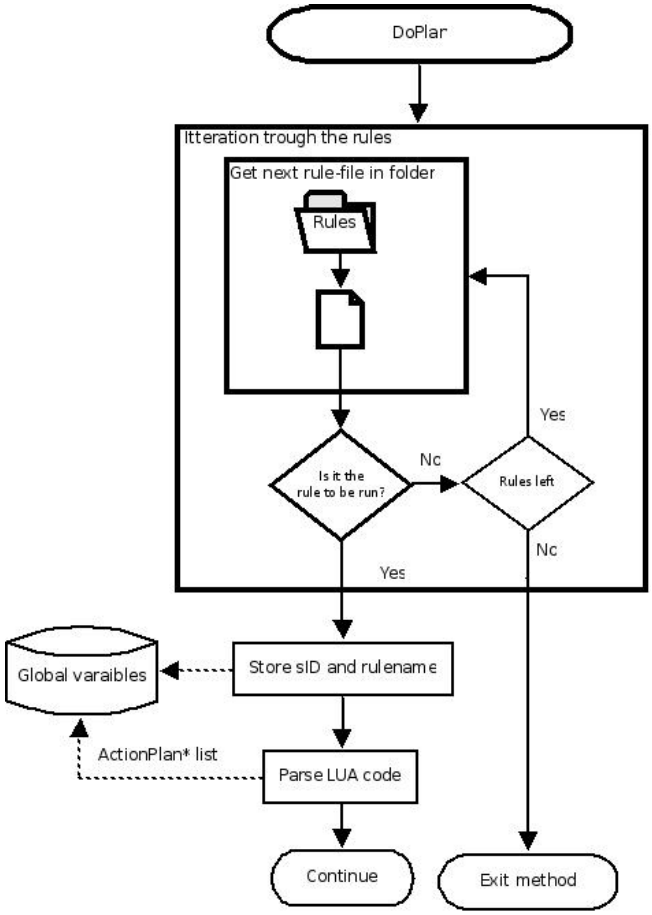

Figure 6.2: Overview of execution and parsing of a new plan or action.

name SID and evaluation value are stored in a log called PlanHistory for later use by the learning process in the LPL explained in section 6.1.2.

Perceptions

The perceptions are added to the world interface trough the method AddPerception where a pointer to a perception-class is passed as a parameter. This perception is stored in a list in the world interface and available to insert data to the knowledge base using the GetPerceptions method that returns a list with pointers to the perceptions. The perception-class contains methods for inserting data to the perception and retrieving it. An abstract overview on how a perception-class works and its data-flow are illustrated in figure 6.4

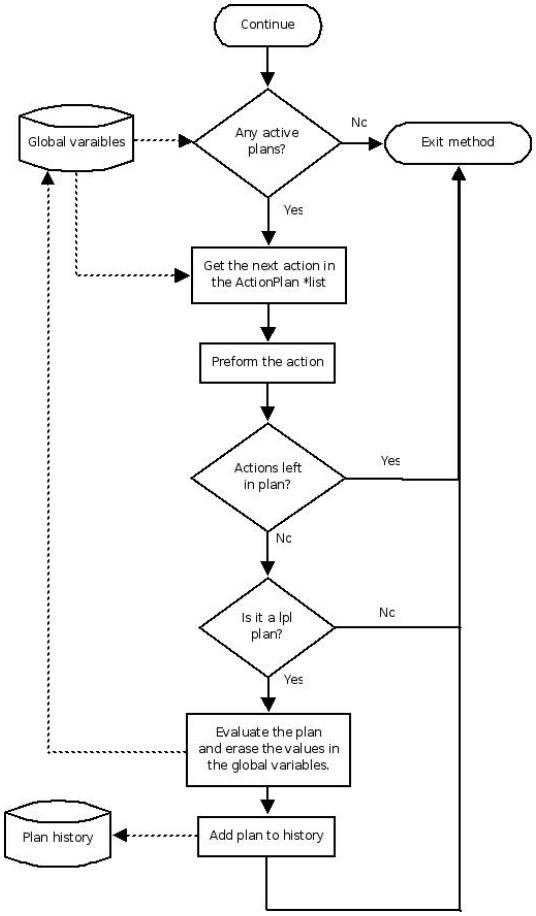

Figure 6.3: Overview of execution of an existing current plan.

6.1.2 Control unit

The control unit is implemented as a simple class that contains 3 members. The members are instances of the BBL, LPL and CPL classes. It contains an invoke methods that starts the information flow within the control unit. This is done by letting the control unit call the invoke method in the BBL class. The implementation of this class / container is unchanged from the pre-project[LG07].

The behavior based layer

The behavior based layer wraps around 3 classes. These 3 classes are the bblBR, bblSG and bblPS that are implementations of these 3 modules, as described in 4.3, specially tailored for the behavior based layer. The implementation of the behavior based layer is almost unchanged from the version used in the pre-project except for major changes in the bblSG module and minor changes in the bblPS

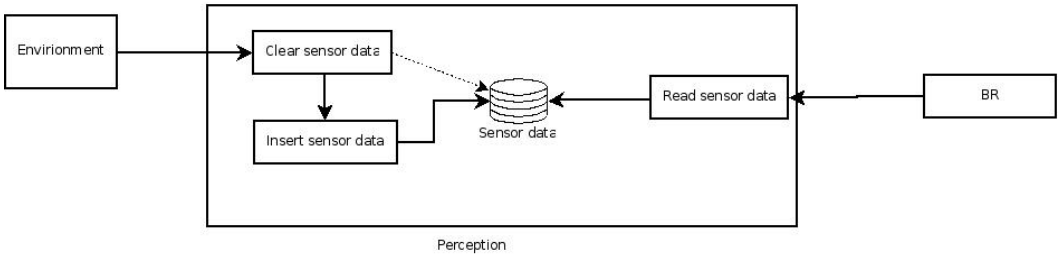

Figure 6.4: Abstract overview of the inner workings of a perception-class

module.

The bblBR module reads the perceptions and translates it to beliefs it stores in the world model. This is implemented by getting the list of Perception pointers from the world interface and traverse trough them en read the data each perceptron contains.

The bblSG module are now implemented to recognize a situation and select the action actions corresponding to it by first parsing trough all the lua files, where each one represent an action to take, and keep a track of the lua-files that wants to do some action on the environment. If a lua-file wants to do some action on the environment means that the action can be corresponded to a current situation. The bblSG puts the names of these actions in a list that it passes on to the bblPS. This list contains the name of the actions and its weight. The weights are implemented so one action may have a higher priority then others if there is any need for it.

The bblPS module goes trough the list from the bblSG and chooses the best reactive action to do given the current situation(s).The best reactive action is chosen by sum all the weights of the actions in the list, divide every weight of a rule with this sum. This generates a fraction that represents how big chance the action has to be chosen. A random variable between 0 and 1, R, are chosen and a new loop starts. In this new loop the fractions are added to a variable V that starts with the value 0.0. Every iteration of the loop the fraction-value for one of the actions are added. If the value of V is greater then the value of R then the last action that had its fraction added wins and becomes the action to perform.

An example would be that a list containing two actions with both weight 1 are sent to the bblPS, and the fraction for both actions becomes 1/2. First action in the list gets represented from 0 to 0.5 and the other action will get represented from 0.51 to 1. A random float number are then chosen between 0 and 1, and if this number is <=0,5 the first of the two actions are chosen, if the number is >0.5 the second action is chosen.

The local planning layer

The local planning layer wraps around 3 classes, lplBR, lplSG and lplPS that are implementations of the BR, SG and PS modules as described in 4.3. The local planning layer have undergone a total rewrite since the pre-project. This has been done to support learning in this layer. The local planning layer have an initialization method that are invoked when the InteRRaP library are started for the first time. This method iterates trough all the lua-files available to the local planning layer and makes a list over every available plan and its initial weights. This initialization process also initializes the BR,SG and PS modules within the layer.

The lplBR is initialized with the list containing the name of the plans and their weights. This list is then stored as a member of the lplBR class for later use. When the lplBR are invoked it gets the list over plans that have been executed if any form the world interface. The weights for those plans are then adjusted for the situation that was responsible for the plan based on their evaluation. The data in the world model are then traversed trough and if a new situation are discovered it adds this as a new column in the matrix and initializes its weights with the plans to the default weights for the plans. This is then our current situation, and no need to check which situation the agent are in. If no new situation are discovered the lblBR find the current situation by checking the data in the world model. The situation is sent to the lplSG and the matrix containing the weights are stored and updated in the mental model.

The lplSG is invoked with the current situation from lblBR as a parameter. The plans and their weights for the situation are sent as a list to the LPL's together with the current situation.

The lplPS then sums the weight with a power of the value K together to a total weight. ($Sum = \sum_{i=0}^{n} W_i^K$, where n is the total number of plans sent to the lplPS.) This is very similar to the summing of weights done in the bblPS, the big difference are using the weight in power of K we get a bigger number to use when a plan are to be chosen. The plan is chosen by getting a random float number from 0.0 to the value of the summed weights, and then start to sum the weights over again. In the loop summing the weights again there is a test that tests if the sum have become larger then the random number. If not then sum next weight. If the sum became bigger, then the plan connected to the weight last summed are the plan to be executed.

The cooperative planning layer

The cooperative planning layer are implemented as an empty layer. If information are sent upwards to the layer, it just sends empty commands downwards.

6.1.3 Knowledge base

The knowledge base is developed as a class containing the three layers world model, mental model and social model. The world model is a class containing a primitive hash-table containing the data it contains. The knowledge base wraps around a method for inserting data into this table, and another method for extracting data from the table. The mental model stores the matrix that connects the situations and the plans with a weight as well as storing all the situations that have occurred. The social model are implemented as an empty class as a place-filler if it ever needs to be implemented.

6.2 The agent

The agent is implemented by using the InteRRaP library. The agent connects the perceptions from the environment and the actions available to perform with the InteRRaP library. It is developed as a tcp/ip client that connects to the environment. It tries to connect to a hardcoded IP-address, and the InteRRaP library are initialized and started when a connection have been established. There are 2 conditions that needs to be true for the agent to consider a connection established. The first one is that besides the normal handshake-packages used by the network-library, a package containing position-data and image-data have to be received. The second condition is that the environment have sent a package containing how long a step with on move action, and how big an angle are rotated with one rotate action. This makes the agent able to better calculate the amount of moves needed to go a certain distance or rotate to a certain angle. When a connection have been established and the InteRRaP library have been initialized and started, the agent waits for new data from the environment and makes this data available for the perceptions to use and invoke the InteRRaP library once every second.

6.2.1 The QT GUI

The agent has a GUI for debugging and interacting purposes. This GUI is developed using the QT [Tro08] library developed by the norwegian company Trolltech. This GUI contains 3 main frames mainly. A text debug area on the bottom part of the frame, an image area to display the image-data received from the environment

and an area on the upper right side to put buttons for interacting with the GUI. The image area can switch between showing the color image received, the depth image received or a visualization the map generated by the agent. This GUI makes it possible for observing the agent in more detail in runtime.

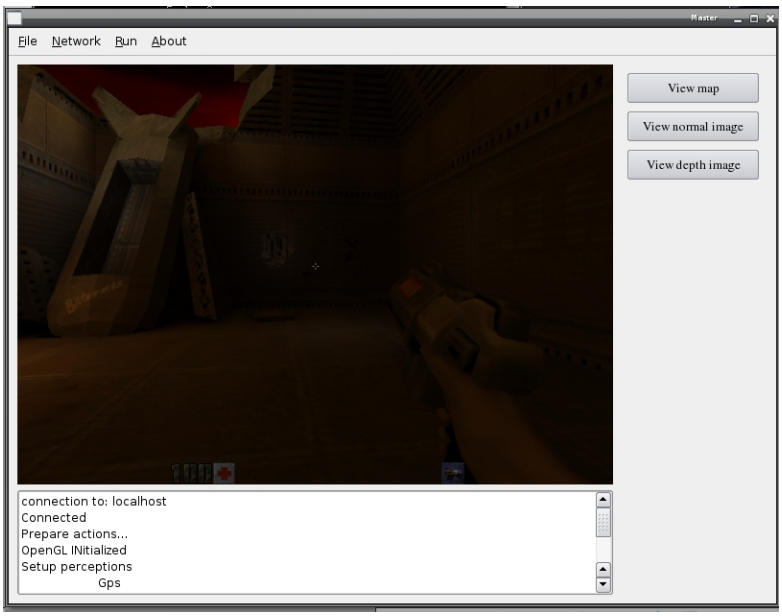

Figure 6.5: Screenshot of the QT GUI

6.2.2 The map

The map is integrated into the agent as a more or less stand-alone module. To be able to interact with the map at a high design level the map had to be implemented into the agent and not the InteRRaP library as it was in the pre-project.

The tiles that the map consists of are stored in an array where the position in the array is the identity of that particular tile at that position in the array. So a tile that has the ID 15 are stored at position 15 in the array, this makes it very efficient retrieving data for a tile we know the ID of since no search trough array is needed. A tile is represented with the MapRegion class. The map contains methods for adding a new tile into the map, load a map from a file, save the map to file and a method for find a route from a to b. When adding a new tile to the map the neighbors for that new tile have to be found. This is done by going trough the entire array of tiles and checking if the new tile represents a part of the environment thats ether north, south, west or east of that old tile. If the new tile is a neighbor then links are created between those two tiles in the direction they are neighbors.

6.2.3 Rules

The rules are written in the lua scripting language. There are no need for a recompile when one wants to add another rule to the agent or a change is made in an existing rule. This is because lua is parsed at runtime.

Rule parsing

The rules are parsed using the lua C api [Ier06]. The lua C api takes care of executing the lua-script, all we need to do is tell it which file to execute and what methods and/or global variables that will be available from the C code. With this api we are able to make a connection between lua and C. There are methods made available from C to the lua-script depending on which layer of the InteRRaP architecture that parses the rule. These methods make the lua-script available to communicate to the compiled program and call methods. When parsing the lua-script only the method Rule() within the script are executed, all other methods are ignored and will only be parsed if they are used from the Rule-method. It is this Rule-method that calls all the methods in C and generates actions for the agent to perform.

To be able to evaluate a rule after all actions it generated have been executed by the InteRRaP agent, an evaluate method had to be implemented in every lua-script that would be executed by the local planning layer. A method called Post() in the lua-script is executed to evaluate itself. To be able to do this a global string-variable in the lua-file called ”goals” contain variables that will be globally available trough out the lua-file. The goals-string contains a set of variables that will be available for Rule() to store the goals in, and for Post() to check the goals. This goal-string is not a static declaration of the goals, but rather a declarations of variables available for that purpose later in Rule() and Post(). This string is parsed in C before Rule() is called and injected into the lua-parser as global variables available. The variables that are in the string are separated by a comma and stored in the mental model until the evaluation are done. So lets say the string ”goals” contain ”gTest1=0,gTest2=10”, then a variable called gTest1 and gTest2 will be available to the lua-script with values 0 and 10. These global variables can be read and written to, so if you write 60 to gTest1 then gTest1 will have this value until its changed or the lua-file are done and evaluated. Using these global variables we can store goal-values or other values we want to use when we evaluate the rule later. So of I wanted to go to a tile with ID 6 in Rule() and made a plan to do so, I’d store the ID in a global variable and then in Post() I can evaluate and see if we are in a tile with ID 6 as we wanted.

Structure of a rule

I will here show an overview of the structure of a rule. The rule shown beneath is written in lua.

```
----------------- Testrule1.lua --------

rulename = "Testrule"   -- name of the plan, used as a identifier
priority = 1.0          -- Initial weight used by the LPL
ruletype = 0            -- 0=plan, 1 = subgoal/reactive, 2 = subsubgoal

goals = "dummy=0"       -- initializes a global variable named dummy
                        -- to be used globally in Post() and Rule()

function Rule()         -- The main function that generates a action or plan
                        -- Available functions from the InteRRaP library:
                        -- DoAction(string query)
                        -- Query(string query)
end

function Post()         -- Evaluation function
                        -- Available functions from the InterRRaP library:
                        -- QueryPast(string query)
                        -- Query(string query)
   return 0             -- returns: -1=failure, 0=neutral, 1=success
end
```

6.3 The server

The server application starts the Quake II game with LD_PRELOAD=injector.so in front of the command that starts the game. The server then starts up as a network server accepting connections on port 6800 and opens the shared memory allocated by the injector. The injector is a small library that overwrites the swap-buffer function and dumps the image from the frame-buffer and depth-buffer into an allocated shared memory so the server can access it later. It also obtains the address of the swapbuffer method in the OpenGL library so it can call that every time itself gets called. This ensures that the frame from the game still gets shown.

The server then enters into a loop where it does two procedures every iteration. The first procedure are a check if there have been any network packages received since last time. And if so parse them. The packages received are in all cases except

the initial handshake commands to be sent to the actuators. This will be described in more detail beneath. The second procedure is a check if the data in the shared memory segment have been changed since last time, and if so send the data to the client. A more detailed explanation ow what types of data this is and how this is sent are written beneath.

6.3.1 Sensors

The sensors and perceptions are implemented in the injector. The sensors implemented are a coordination sensor, a frame- buffer sensor and a depth-buffer sensor. These sensors are updated with new information every time the Quake II game calls the swapbuffer method that the injector are overriding. The information from the sensors are placed in a shared memory segment, so the server can send this to the client. This information is stored in the memory as a struct. This struct has 5 members. The xyz coordinate, the depth-image and the color-image. The server then reads this data if it has changed and packs it in to a package to be sent over the network before sending it.

6.3.2 Actuators

The actuators are the movement-keys available in the Quake II game. The server receives network packages from the client with the keys to be pressed down. The keys get pressed down by the client using the X11 api. But before a key can be pressed, the server have to get the X11 window-id of QuakeII. Since this ID can change every time the game starts, it has to be retrieved every time we start our server. When a key to be pressed have been received we have to translate it to the X11 code for that key. After the key have been translated a XEvent have to be sent. The event message gets generated with all the values it need to contain to send a KeyPressedEvent to X11. This event contains the key to be pressed, what window to press it in and for how long it will be pressed down.

6.4 The network library

The network library supplies us with an easy to use and understand API for networking between our client and server. This library supports peer-2-peer networking and client-server networking, but not both at the same time. You choose which of them you want to use when you initialize the library trough the API.

Sending and receiving of packages runs in separate threads and received packages go into a buffer and waits to be read. Thread safety is ensured. There is no max-limit on how big the packages or data chunks can be. If they are bigger than

16KB the library will split them up in 16KB chunks and send them one chunk at a time. This decreases the strain on the network adapter when big packages are sent.

All in all this self-made library makes the usage of networking an easy task, and it is only a matter of 3-4 API calls to set up a connection and begin sending packages back and forth.

Chapter 7

Testing

This chapter contains the tests done using the prototype implementation of the InteRRaP agent on the Quake II environment.

7.1 Test 1: Recognizing enemies using OpenCV

The test is to test a perception able to recognize enemies in real-time using the OpenCV library.

7.1.1 Why

The reason behind this test is to see if its feasible for the prototype implementation to use a perception that identifies enemies and send data to the agent about if there is an enemy visible or not in front of the agent. A successful result would mean that the agent could identify enemies in the environment and eventually kill them before the enemies kill him (the agent).

7.1.2 The setup

The setup for this test is a game server running the Quake II environment and the prototype implementation as a client connected to it. The prototype uses the OpenCV library with a pre-learned XML-file of training data to try to identify an enemy. The training-data is generated using OpenCV and a data-set consisting of 3000 negative examples (images not containing an enemy) and 7000 positive examples (images containing an enemy in them). The generation of the training data took 3 days in total.

7.1.3 Results

In figure 7.1 8 of the results from this test are pasted together into one big figure. These results give a good overview on how the test went. In 3 of the 8 images shown in figure 7.1 managed OpenCV to identify the enemy. This shows that the OpenCV library with the training data used a low chance of identifying an enemy in an image. The circles in the images show where OpenCV found an enemy. The identification process took approximately between one and two seconds too perform.

7.1.4 Discussion

This test shows us that, using the training data we made, identifying an enemy using OpenCV is not very feasible. Only having a little success-rate on identification makes the result from the identification process very unreliable. A wrong identification would result in the agent trying to shoot something that is not there, and could in worst case scenario make the agent go closer to an enemy thats not there. (If the plan performed by the prototype includes some actions making it move closer to the target for a better aim.)

Using between one and two seconds to identify an enemy is acceptable though. The worst case of two seconds will not lead to unusable data because they are too old. ("The transduction problem" [Woo05].)

7.2 Test 2: Learning in the local planning layer

This is a test to see if the learning we have implemented works as we want it in the local planing layer.

7.2.1 Why

The reason behind this test is to see if the agent is capable of learning what plan to execute for its current situation, based on how well or bad it has done with the plans in the situation before. What we want to see here is that the agent wants to explore in the beginning when most of the area around it is unknown to it. And after it has explored what it can explore it learns that exploring is not that good of an idea anymore and starts doing other plans.

7.2.2 The setup

The setup for this test is the prototype implementation of the agent in a big room with no obstacles. We accomplished this room in the game by letting the agent

go trough walls so the test would not be disturbed by to small corridors. The environment is the Quake II game "encapsulated" by the game server application. The prototype agent has 3 plans available: Explore1, Dummy1 and Dummy2. Explore1 is a plan that makes the agent explore an unknown tile next to the tile it is currently standing, and are successful if that tile gets explored. Dummy1 is a generic plan that does nothing, but have a 85% chance to return neutral, 5% chance to return failure and 10% chance to return success. Dummy 2 are an also a generic rule that does nothing, but this one have a 93% chance to return neutral, 2% chance to return failure and 5% chance to return success.

7.2.3 Results

The results of this test is given in figure 7.2. These results was what we wanted to see in this test. In the beginning the exploration plan have a lot of success and gets rewarded a lot for that, something that make the exploration plan more likely to be run next time as well. After time is approximately 800 the exploration plan starts to get punished. This punishment is because the exploration plan is called, but there are no more to explore, so it returns failure every time its executed. After a little while one of the dummy rules takes over the biggest chance to get selected, since it has been rewarded so much and now have a big weight. The exploration converges to a weight where the plan no longer gets chosen.

7.2.4 Discussion

The results of this test showed that the learning in the local planning layer worked as planned. Since one of the dummy plans have a little more chance of success it managed to follow the exploration plan very closely when gaining weight. Since we used a "big room" without any obstacles in this test we managed to show a proof of concept. But if we used this in a corridor or a room with a lot of crates, the agent would get to a point where he had only explored tiles around him much faster. So to entirely explore a map only one simple exploration rule like the one used in this test is not enough.

7.3 Test 3: Finding the goal

This test is to see if the agent can find the goal. If the agent find the goal, this test will also check if the agent uses the knowledge about the whereabouts of the goal to get to it faster.

7.3.1 Why

To be able to solve the game, the agent have to find the goal. So if the agent cant find the goal it will not be able to win the game. So to be able to show that InteRRaP works on a computer game environment it has to find this goal.

7.3.2 The setup

The setup for this test is the agent in a normal map in the game with small open areas, corridors and crates. The environment is as the previous test the Quake II game ”encapsulated” by the game server application. The agent has 3 plans available for it to use to find the goal. The plans are: Explore1, Walker and GoToGoal. Explore1 is the same plan used in test2 and are a plan that makes the agent explore an unknown tile next to the tile it is currently standing, and are successful if that tile gets explored. Walker is a plan that goes to an explored tile that is next to the tile the agent is currently standing on, and are successful if the agent gets to that tile. GoToGoal is a plan that generates a route to the goal if the position of the goal is known and are successful if the agent gets to the goal position.

7.3.3 Results

The results of this test are given in figure 7.3. This graph shows that the agent found the goal after a little while exploring. The next time the agent tried to find the goal it used almost one sixth of the time it used the previous run. When the goal position is known the agent uses very little time and uses mostly the same time finding the goal after that.

7.3.4 Discussion

This test was a great success. The agent found the goal as hoped and was faster to find it again after it had learned the position of the goal. After knowing the position of the goal, the time to find it varies a little every time. This may be because it is not always the GoToGoal plan gets selected right away and some of the other plans might have had a chance to execute a time or two before GoToGoal was executed. This test also shows that the InteRRaP architecture managed to control a player in the Quake II game and with the use of the perceptions that were implemented it managed to solve the game by finding the goal.

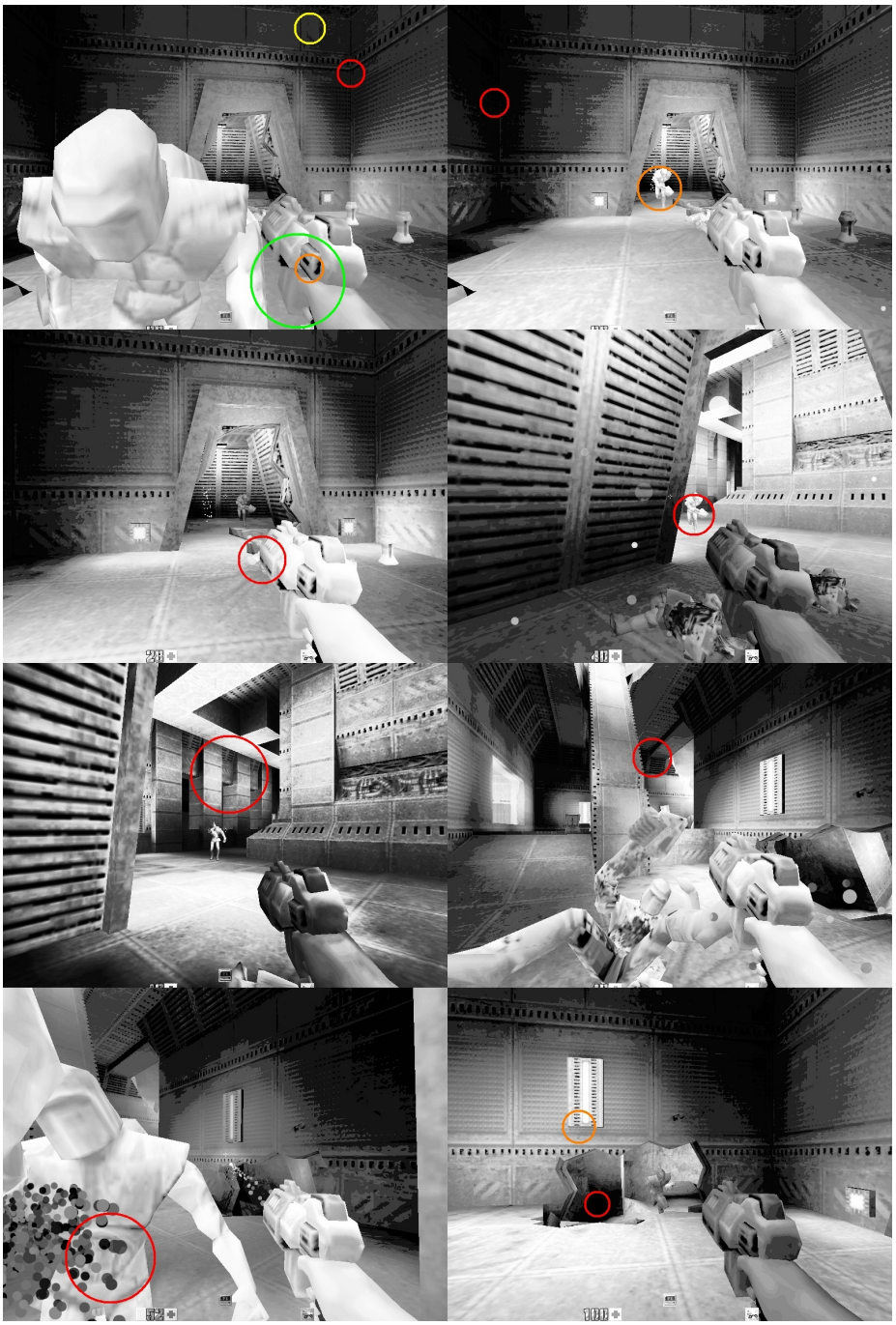

Figure 7.1: The results of Test 1

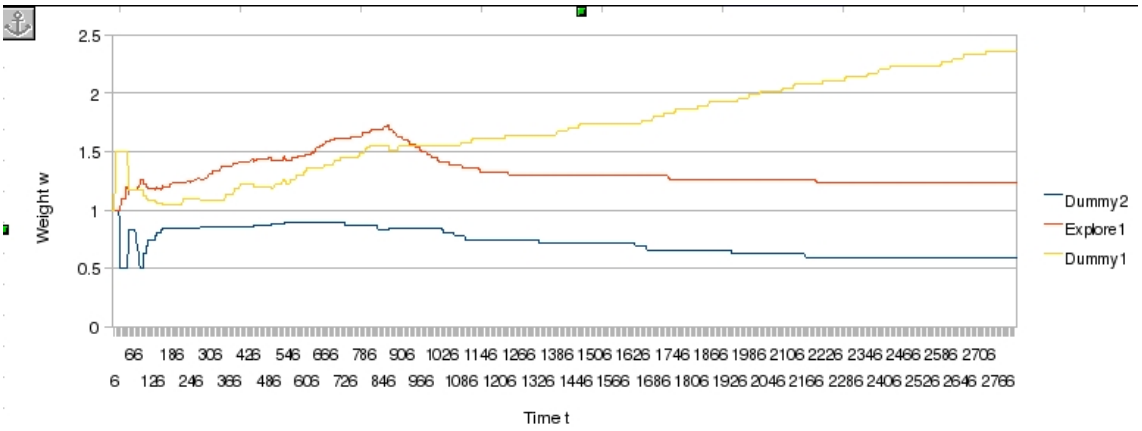

Figure 7.2: The results of Test 2

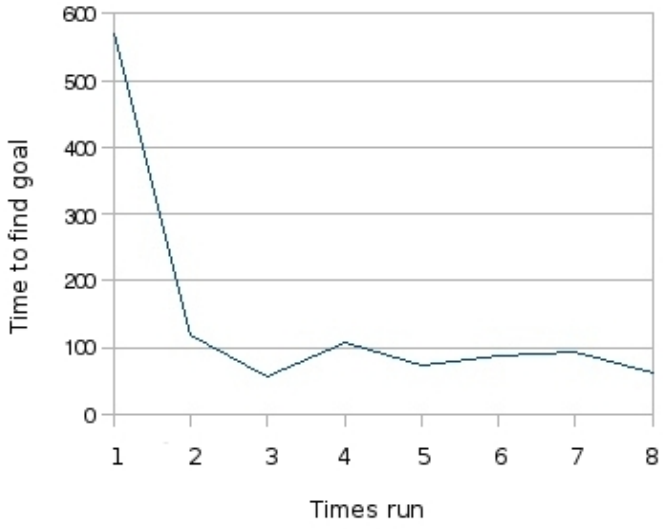

Figure 7.3: The results of Test 3

Chapter 8

Discussion and future work

This chapter will contain a discussion about if intelligent agents, and especially InteRRaP agents, are usable for computer games. Any changes that can be made to make intelligent agents more suitable for the game environment will also be covered.

8.1 Intelligent agents in games today

Using the above question I called up John Eggett a game designer of Team17 in Newcastle and asked him it. I have explained to him the InteRRaP architecture and the basics of intelligent agents on a previous occasion. His opinion was that intelligent agents as they are today might not be very usable in computer games, mostly because of how games are designed today. Todays games are designed with very little A.I. in mind, and what may seem like A.I. to the person playing the game is most often just predefined scripted actions, or scenes as Eggett called them. Between the rendering of frames there is little time to let any A.I. make decisions, and that is mainly an "error" in the game-design making it not very easy to implement any sort of intelligent agents in the game. Eggett stated that most games today cheat to make the player believe there is some sort of decision making going on, while in reality there is none. When asked if he saw a future where this would change he responded that he didn't think so with the normal way of designing a game today.

An example of a game that has succeeded with using agents in computer games is F.E.A.R created by Monolith Productions. Jeff Orkin is the person behind the A.I. in this game and used an agent architecture that resembles MIT Media Lab's C4 architecture [Ork05] [BID+01]. The agent architecture was successfully accomplished by distributing the processing of costly preconditions over many frames and caching results for the planner to inspect on-demand [Ork05]. What

Orkin means by preconditions is the parsing of sensory data and getting sensory data from the environment, since the architecture only calls the planner when there are new perceptual data available. In Orkin's agent architecture an agent has a blackboard, working memory, subsystems and sensors. The subsystems are targeting, navigation animation and weapon systems. Sensors detect changes in the world and put the data (percepts) it collects in the working memory. The sensors places the data it is calculation on in between frames on the blackboard so the sensor can start calculating the data where it stopped when the frame was to be rendered. The planner then uses the percepts in the working memory in its decision making and communicates the instruction to its subsystems using the blackboard. Instead of letting the planner do costly computations on-demand sensors were used to reduce the cost over these computation over many frames and cache the results in working memory instead of only one frame that would have been the case if the planner did the computations.

8.2 The InteRRaP architecture

As we have seen in this report, the InteRRaP architecture is not optimal for computer games, although its computational complexity is rather low for a hybrid architecture. So there is a potential for this architecture to be modified for better use within a game-engine to control NPCs (Non Playing Characters). However, if the costly calculations in the architecture can be distributed similar to Orkin's architecture [Ork05], the InteRRaP architecture can be very usable for games. This would mean that a blackboard has to be introduced in the world interface so the perceptions can be handeled like the sensors in Orkin's architecture. Letting the sensors reduce the cost of the expensive calculations in the local planning layer and distributing it over many frames instead of only calculating it between 2 frames would most likely give as good results as Orkin experienced using his architecture. An abstract illustration of the proposed changes are given in figure 8.1. The knowledge base will get data from the perceptions,communication unit and information about the success of a plan (containing several actions) that were executed using the blackboard in the world interface. The control unit will pass actions down to the blackboard for the action unit in the world interface to execute along with plan information so it can know what plan the actions belong to. The control unit will only be invoked if there are new perceptual information that can lead to a new plan or action, or if there are no plan currently executing. Since all heavy calculations are now distributed between frames and the control unit is not invoked all the time, the architecture will not give big spikes of processor usage between the frames, but rather have a steady and lower flow of processor usage all the time so the rendering can go more smoothly.

Another possible change is to replace the rules/plans by python scripts instead of using lua. Lua has shown us many of its weaknesses during this project. It is easy to integrate in C, but does not have as big standard library available to use in complicated plan generation. Python is a bit harder to implement, but its worth using the time to do it to have the big standard library included in python available to use in plan generation.

Using python the actions available to the library implementation could also be generalized and placed in scripts instead of hardcoded in the implementation as it is now. This would greatly increase the generalization of the library and make it easier for users to create actions available to the agent, since no C knowledge would then be needed.

Most of these changes would make the architecture differ from the original InteRRaP architecture, but make it better to use in computer games. The reason for making these changes is that computer games most often have only a small "time-slot" available to do A.I. calculations. The concept used by the agents in F.E.A.R. is a concept that could be used in the library to make it better for games to do heavy calculations between frames and distribute them over several frames instead of only between the 2 frames that we use now in this project.

8.3 Using computer vision

Using computer vision we made perceptions for the prototype implementation. The main reason behind this was not to "cheat" too much and try to give the same percepts to the agent what is given to a human. This was something that slowed down the agents decision making. Using computer vision as a perception was tested in test 1 (in section 7.1), and this test shows that using OpenCV to recognize an enemy was not a success. Using OpenCV we didn't manage to recognize enemies sufficiently and it took too long to identify an enemy. To make the architecture perform better on games computer vision has to be discarded and replaced by "cheating" techniques. These "cheating" techniques uses information from within the game engine to give to the agent. No or very little computation is needed to get this information, as with computer vision would need extreme computation to get the same data without this "cheating". The cost of not cheating is too high to justify it. Even with the backside of having to generalize the perceptions and plans/rules to a specific game for the agent, instead of having the ability to use same perceptions and plans/rules on several games.

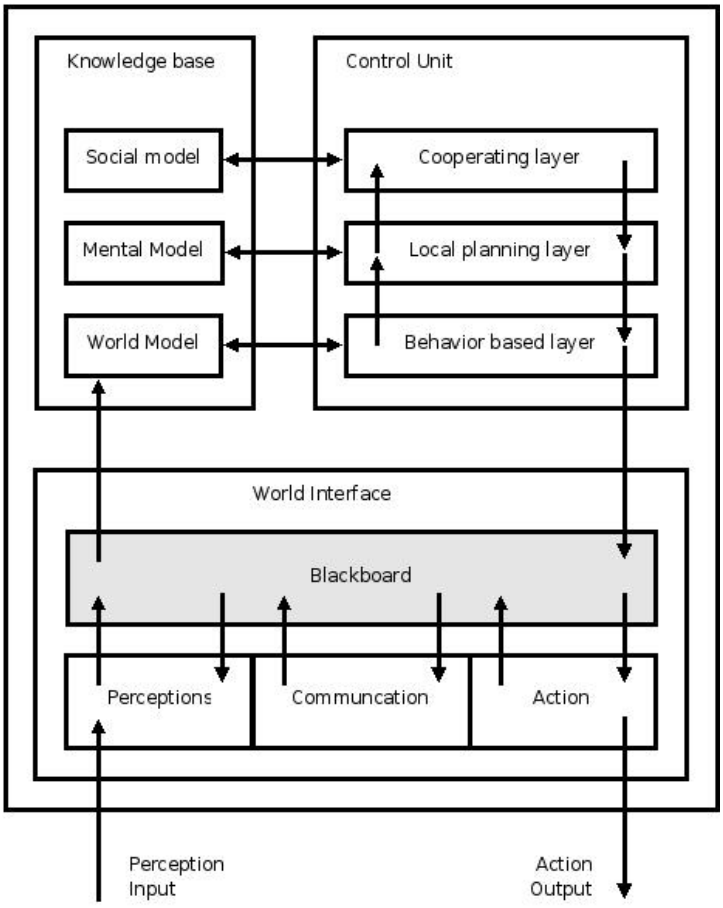

Figure 8.1: Proposed change in the InteRRaP architecture for games.

Chapter 9

Conclusion

To play a computer game with the same percepts and actions as a human player an intelligent agent have to use computer vision. Computer vision proved in test1 to be hard to accomplish with a good success rate and without using a lot of computational power. This can however be solved by introducing a blackboard into the agent architecture and let the perceptions distribute their computations between frames. But even if its solvable with distributing the processing power between frames it is not very feasible since another problem can occur even when distributing the processing. This problem is the ”transduction problem”, and are the problem were the translation from environment to symbolic takes too long time and the data is no longer usable when they are done translating.

Using a little cheating technique with getting the in-game position of the agent we were able to successfully map the environment. The agent learned from his mistakes like he was supposed to, and learned how to get to the goal when the goal was discovered in the environment. The test runs showed that the agent architecture used have a potential for use in computer games, and with some changes have an even better potential.

Appendix A

Explore1.lua

Explore1.lua is the simplest exploring plan available for the agent. This plan checks to see if there are any unknown tiles around the agent. If there are any unkown tiles around the agent, then one of these tiles get randomly chosen to be explored and generates a action-set for doing so.

```
rulename = "Explore1"
priority = 1.0
ruletype = 0
goals = "gYaw=0,gUkjente=0,gTileID=0,gDirection=0"

function Rule()
   AntUnknown = 0
   NextPos = nil

   east = Query("East")
   south = Query("South")
   north = Query("North")
   west = Query("West")
   yaw = Query("Yaw")
   myPos = Query("Position")
   gTileID = myPos

   if(east == "unknown") then
      AntUnknown = AntUnknown +1
   end
   if(south == "unknown") then
      AntUnknown = AntUnknown +1
```

```
end
if(north == "unknown") then
   AntUnknown = AntUnknown +1
end
if(west == "unknown") then
   AntUnknown = AntUnknown +1
end

if(AntUnknown >= 1) then
   math.randomseed(os.time())
   randTall = math.random(AntUnknown)
   gUkjente = AntUnknown

   NumUnknown = 1


   if(east == "unknown") then
      if(NumUnknown == randTall) then
         NextPos = 0
         gDirection = 1
         randTall = 0
      else
         NumUnknown = NumUnknown +1
      end
   end

   if(south == "unknown") then
      if(NumUnknown == randTall) then
         NextPos = -90
         gDirection = 2
         randTall = 0
      else
         NumUnknown = NumUnknown +1
      end
   end
   if(north == "unknown") then
      if(NumUnknown == randTall) then
         NextPos = 90
         gDirection = 0
         randTall = 0
```

```
            else
               NumUnknown = NumUnknown +1
            end
         end
         if(west == "unknown") then
            if(NumUnknown == randTall) then
               NextPos = 180
               gDirection = 3
               randTall = 0
            else
               NumUnknown = NumUnknown +1
            end
         end

         if (AntUnknown > 0) then
            DoRoatations(NextPos,yaw)

            step = Query("StepLength")
            step = tonumber(step)
            steps = 100/step

            for i=0,steps,1 do
                DoAction("moveforward")
            end

            gYaw = NextPos
         end

      else
         gUkjente = 0
      end
   end

   function Post()

      result = -1
      ukjente = tonumber(gUkjente)
      expectedYaw = tonumber(gYaw)
      retning = tonumber(gDirection)
      myPos = Query("Position")
```

```
   curYaw = Query("Yaw")

   if(ukjente ~= 0) then
      if(myPos ~= gTileID) then
         liste = GetNeighbourInfo(gTileID)
         forventet = tonumber(liste[retning])
         if(forventet == 1) then
            result = 1
         else
            result = -1
         end
      else
         result = -1
      end
   end

   return result
end

function DoRoatations(goal, yaw)

   goal = tonumber(goal)
   yaw = tonumber(yaw)

   goal180 = goal+180

   divider = Query("YawStep")
   divider = tonumber(divider)

   if(yaw > 0 ) then
      goal180 = goal180 - yaw
   else
      goal180 = goal180 + (yaw * -1)
   end

   if(goal180 > 360) then
      goal180 = goal180 - 360
   end

   if(goal180 < 0) then
```

```
      goal180 = goal180 + 360
   end

    goal180 = goal180-180

   if(goal180 < 0) then
      diff = goal180 * -1
      rotation = diff/divider
      for i=0, rotation, 1 do
         DoAction("rotateright");
      end

   else
      rotation = goal180/divider
      for i=0, rotation ,1 do
         DoAction("rotateleft");
      end

   end
end


function Length(x0, y0, x1, y1)
   if(x0 > x1) then
      dx = x0 -x1
   else
      dx = x1 - x0
   end

   if(y0 > y1) then
      dy = y0 - y1
   else
      dy = y1 - y0
   end

   l = math.sqrt( (dx * dx) + (dy * dy) )
   return l

end
```

Bibliography

[BID+01] R. Burke, D. Isla, M. Downie, Y. Ivanov, and B. Blumberg. Creaturesmarts: The art and architecture of a virtual brain, 2001.

[BM91] H.J Börckert and Jörg P. Müllerl. Ratman: Rational agents testbed for multi agent network. Technical report, 1991.

[Bra87] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[Cal99] Robert Callan. *The essence of Neural Networks*. Practice Hall Europe, 1999.

[FMP94] Klaus Fischer, Jörg P. Müller, and Markus Pischel. Unifying control in a layered agent architecture. Technical Report TM-94-05, 1994.

[Hew07] Robin Hewitt. Seeing with opencv. 2007.

[IDS97] IDSoftware. Quake ii, May 1997. `http://www.idsoftware.com/games/quake/quake2/`.

[Ier06] Roberto Ierusalmischy. *Programming in Lua*. Lua.org, March 2006.

[LG07] Karl Syvert Løland and Stein Gunnar Grastveit. Intelligent agents in computer games, 2007. Prosjektoppgave.

[Mic06] Microsoft. Directx, 2006. `http://msdn.microsoft.com/en-us/directx/default.aspx`.

[MP93] Jörg P. Müller and Markus Pischel. The agent architecture inteRRaP: Concept and application. Technical report, German Research Center for Artificial Intelligence, 1993. RR 93-26.

[Nor03] Emma Norling. Capturing the quake player: using a bdi agent to model human behaviour. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 1080–1081, New York, NY, USA, 2003. ACM.

[Ork05] Jeff Orkin. Agent architecture considerations for real-time planning in games. In R. Michael Young and John E. Laird, editors, *AIIDE*, pages 105–110. AAAI Press, 2005.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[Spa03] Luca Spalazzi. M. j. wooldridge, reasoning about rational agents, intelligent robots and autonomous agents series, cambridge, ma: The mit press, 2000, isbn 0-262-23213-8. *Minds Mach.*, 13(3):429–435, 2003.

[Tro08] Trolltech. Qt cross platform application framework, 2008. `http://trolltech.com/products/qt/`.

[Wik08a] Wikipedia. Dll injection, 2008. `http://en.wikipedia.org/wiki/DLL_injection`.

[Wik08b] Wikipedia. John d. carmac, 2008. `http://en.wikipedia.org/wiki/John_D._Carmack`.

[WJ94] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice, 1994.

[Woo05] Michael Wooldridge. *An introduction to MultiAgent Systems*. Wiley, 2005.