Entertaining Agents

for Multi-Player Role Playing Games

Mads Granding <madsg@cs.auc.dk>
Rasmus Toftdahl Olesen <rto@cs.auc.dk>

Supervisor: Peter Bøgh Andersen <pba@imv.au.dk>

July 20, 2004

AALBORG UNIVERSITET

Aalborg University
Department of Computer Science
Frederik Bajers Vej 7, Building E
DK-9220 Aalborg Ø.
Telephone (45) 96 35 80 80
Fax (45) 98 15 98 89
http://www.cs.auc.dk

Title:

Entertaining Agents
for Multi-Player Role Playing Games

Topics:

Agents, Computer-Games and Story-Telling.

Project-term:
DAT-6, spring 2004

Project-group:
d611a

Members:
Mads Granding
Rasmus Toftdal Olesen

Supervisor:
Peter Bøgh Andersen

Copies: 6

Pages: 63

Date: Jul 20. 2004

Synopsis:

Working from a goal to create entertaining and interesting agents for RPGs, we design an agent architecture. The architecture incorporates many elements designed for instilling it with a personality of its own, and for using this personality to make the agent entertaining.

The means for making the agents interesting, and giving them personality, include: Using vector-distances to select the one sentence, from a set of sentences, that best matches the agents personality. Selecting actions and plans based on parts of the agents personality described as a vector. Cognitive dissonance is used by the agent to decide how to interact with players and other agents.

We also touch upon subjects, such as quad-trees, networking in multi-player games, and planning, as they relate to our architecture.

Contents

List of Figures

Chapter 1

Summary

Our goal for this project has been to design entertaining agents for multi-player role playing games. Role playing games for computers are either single-player or massively multi-player. However, we wish to make an architecture that is more akin to a real-life, pencil and paper role playing game. These games usually have a much smaller number of players, than current online role playing games. They also focus more on interesting characters, than on the levelling-treadmill of most computer based games. We implement key parts of both the world, and the agents for such a game.

We consider if "simply" adding more intelligence to the agents will make them more interesting and entertaining. We conclude that intelligence is a means that can help make the agents entertaining, but will not automatically do so. In games, entertainment is the goal, and intelligence that does not entertain is a waste of the precious resources of the computer.

We create a dialogue representation that pairs a plain text sentence with an expression that is easily interpreted by the agents. To each of these sentences, we assign a vector that describes the "mood" of the sentence using attributes that denote if the sentence is polite or rude, happy or sad, and so on.

Given several different sentences that express the same thing, the agent can select the one that fits its personality best. It does this by comparing the vectors on the sentences with a corresponding one, that defines how that agent speaks. This vector is part of what defines the agent's personality.

By using cognitive dissonance, the agent can decide how to interact with different characters, based on whether it likes the character or not.

The agent also has the possibility to keep secrets, and even lie to certain characters.

When performing actions, the agent should give the player hints about what it is doing. Otherwise the agent might seem to be running around aimlessly, when it does in fact have reasons for its behaviour.

Previously games have had a rather simple interface for communication between player and agents. The player could simply select a predefined sentence from list. Using template sentences, with slots that can be filled in by the player, gives the player a greater degree of freedom. For instance, one slot in a sentence could contain locations, and the player would only be able to select a location to fill that slot.

In order to make agents that are more entertaining than those in most games, we would like to give them similar possibilities to those of the player. This also means that the agent should be able to make plans in order to achieve its goals. If more than one plan achieves the goal, the agent can select the plan that best fits

its personality. It uses the attributes obligation, desire, and ability to make this decision. These attributes are part of the agent's personality.

The agents and players connect to a world-server, which contains the actual game world. Through the world-server the different clients can interact with each-other and with other objects in the game. The world-server also ensure that every-one plays by the rules. All the objects in the world is contained in a data-structure called a quad-tree. The quad-tree allows fast access to adjacent objects, which makes calculating which object are visible to a client easy.

Using a possible scenario from a role playing game, we have found that our architecture does add new means for making entertaining and interesting agents. Through scripting, we have all the same possibilities as other role playing games, but our components add additional possibilities.

Chapter 2

Motivation & Goals

This report is the result of the second of two semesters work on agents. The work on this semester has been focused on implementing both agents and a world for these agents to inhabit. We have especially focused on agent/player interaction, and how to make agents that are interesting in a game context.

We focus on role playing games (RPGs). In our opinion, this is the genre of games where the possibility of implementing interesting agents is greatest. This is because role playing games focus more on characters than first person shooters or real-time strategy games.

2.1 Role Playing Games

RPGs have room for interesting agents, because they often have many different characters, with their own distinct personalities and their own goals. In RPGs the objective is not to get the highest score, but rather to experience being part of the plot, and interacting with other characters. Ideally this should be an immersive experience for the player. At least this is the case with a traditional, pencil and paper, RPG.

In a pencil and paper RPG there is a social aspect. A small group of players share the experience, and they all have important parts to play in the plot. Usually, the players will cooperate to advance the story. Computerised RPGs only have one of these two aspects. Either there is a social aspect, or you are important to the plot, but never both.

In single player RPGs there is no social aspect, like there is in a real life RPG. You play as one or more characters following some quest. Often the goal is to gain enough experience points to be able to defeat some powerfull monster in the next dungeon. But your role *is* the most important in the plot. Without it, the story would simply not advance. This works against trying to create a immersive world, the world seems to stand still when you are not interacting, i.e. if your quest is to stop the enemy's army from attacking, by sabotaging their siege-engines, they will not attack even though you ignore this quest.

In a MMORPG (massively multiplayer online RPG), the social aspect does exist. You can meet hundreds or thousands of other players in the game world. However, the main drive of the game is the "levelling treadmill". The goal is to collect enough experience points for the character to gain a level, such that ones character is better than the characters of other players. The MMORPG misses out on one of the most important parts of both single player RPGs and real life RPGs - the feeling of importance. These games have an overall plot, and not just a series of disjoint quests. *Your* character is important to this plot. *You* are the hero. The fate of the

world rests on *your* shoulders. In a game with thousands of players everyone cannot be important. In fact "no-one" is important in an MMORPG. If a player stops playing the game, it will have no effect on the rest of the game.

It is puzzling why no one has made a computer game that combines these two aspects of pencil and paper RPGs. So, we want a game where the player can be an important part of the plot, and with the possibility of a social aspect. In other words, the game should be multiplayer, but not "massively" so, just like a traditional RPG. Our architecture should accommodate these features.

One of the reason this has not been done yet is the need for better AI.

2.2 Entertaining Agents

In a traditional pencil and paper RPG the dungeon master plays the part of the non-player characters, major as well as minor parts, friendly and unfriendly. The dungeon master also has the job of adapting the story according to the players' actions. In a computer RPG we need agents to play the parts that the dungeon master plays.

In current computer RPGs the AI cannot play a part in any way resembling that which players can. The agents are at most equipped with a static schedule, e.g. a shopkeeper opening the shop each morning and closing it again in the evening. However even this amount of agent AI is rare, most agents are either standing somewhere waiting for player interaction or simple critters/monsters designed to be killed for experience points and items. The agents does not act on their own accord. They are purely reactive rather than being active agents. They have neither goals nor beliefs.

If we want agents that can play these parts they need to be able to actively pursue goals on their own. If the player does something that interferes with the agent's plan, it should be able to alter its behaviour accordingly, just like the dungeon master would.

A good dungeon master would plan and act each role differently, and give each character its own personality. Likewise the agents should also exhibit some form of personality.

Above all else, a computer game is a means of entertainment, and therefore our architecture should focus on this particular aspect. To be entertaining it is important that the player does not become frustrated, because he does not understand the reasoning behind the agents behaviour. Therefore, it is important that the agent shows its plans and emotions such that they are easily understandable to the player.

2.3 Goals

From the preceding parts of this chapter, we can establish a set of goals. We need to design an architecture for a multi-player RPG world, and to implement key elements of this architecture. We need networking code to deal with the multi-player aspect, and we need a world-server that supports a sufficiently large number of players and agents. Likewise, we need to design an architecture for the agents inhabiting this world. We will focus on the parts of the agents that pertain to making them entertaining, and allowing it to act as a player in the game. In order to do this the agents should have the ability to have goals and make plans. The interaction between player and agents should reveal that the agents all have different personalities.

Chapter 3

Architecture Overview

Using the goals from the previous chapter as a starting point, we can begin to define the components in the system.

From the goals, we can derive some components that the system should contain:

- World
- Players
- Agents

The world can contain a number of agents and players. The players and agents need to modify, or interact with, the world. This can be described as a diagram as shown in figure 3.1.

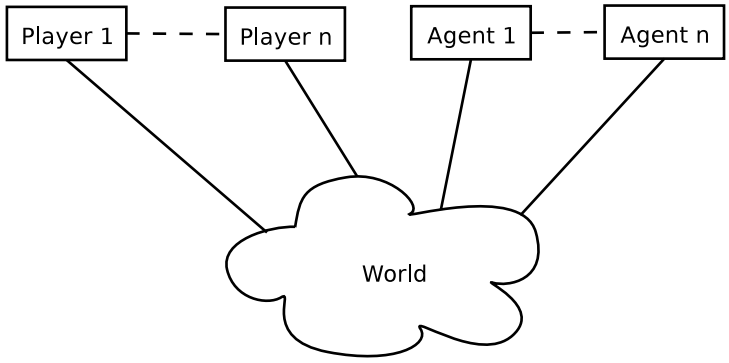


Figure 3.1: Overview of the architecture

After this very basic overview, we will go into details of the requirements for the different components.

3.1 World

The most important responsibility of the world is managing all objects in the world. Objects can be both players, agents and static objects such as a building or a sword lying on the ground. In the most basic sense, the world is a container for all these objects.

When players or agents modify the world, the other players or agents should be notified of these changes. This should also apply to communication between other players or agents.

The world should allow the players or agents to pick-up or modify static objects in the world. It should also make it possible to use the acquired objects in different ways. For instance, use it as a weapon, or give it to another player or agent.

Some rules of the world must also be implemented, to allow for similar possibilities of interaction for all players and agents.

3.2 Player

The player should be presented with a view of the world. This view should include everything her alter-ego (avatar) can observe in the world, including sound and other non-visible things.

The player should be able to control her avatar in the world using the keyboard and mouse. The player's avatar has many things in common with the agents. Things, such as, moving about in, and interacting with, the world. I.e. they pick things up, and use them in exactly the same way.

3.3 Agent

Agents need the same ability to modify the world as the players do, but it needs to act according to a role in the story. We do not want these roles to be scripted, but rather have the agents act dynamically, in much the same way as a player would. In order to do this, we need to represent the beliefs and goals of the agent, as well as a mechanism for achieving these goals. Another important part of the agent is how the agents behaviour should change to achieve their goals. The problem of combining actions to achieve goals is generally known as planning, and is described in chapter 10.

3.4 Multiplayer

One thing mentioned in the previous motivation chapter that we haven't covered yet in this chapter is the multiplayer aspect of the world. Several players and agents should be able to inhabit the world over a network. This leads us to the development of a network protocol for communication between the world (server) and the players and agents (clients) - a standard client/server setup.

Chapter 4

Are Intelligent Agents Interresting Agents?

When making agents for games, you have to consider exactly how intelligent to make them. A computer game has a limit on how many resources it can use, and only a small part of these resources are used on AI. So, when designing a game, you must decide how many resources to spend on the AI. Most games prioritize having pretty graphics over having better AI. We choose not to use bleeding edge graphics for two reasons: The machines we have available to develop it are not top of the line, and we prioritize having reasonably intelligent agents higher.

4.1 Intelligence

So, exactly what do we mean when we say that the agents should be intelligent? This depends on the type of game in question. In a real-time strategy (RTS) game it is about building and using the right units, and attacking in the right places. In a first-person shooter (FPS) it can be things like teamwork and manouvers. In the book "AI Game Programming Wisdom" [8] there are many examples of these types of AI.

However, our agents are for RPGs. In RPGs there are two kinds of agents. There are those agents you could call characters, and those that are simply "cannon fodder", whose purpose it is to be killed by the main characters. Being made for combat, the "cannon fodder" agents could use parts of RTS and FPS AI. The agents we deal with is those that are characters in the game.

As previously mentioned, these character agents cannot be said to be very intelligent in most RPGs. Often they have a piece of information, that they will simply give to the player when prompted. Other times they will send the player on a quest: "Bring me the sword of Azgelblort, and I will give you the shield of the dwarf king". This is not what we understand by intelligence. To be intelligent the agent should be able to take appropriate action to fulfill its goals. Essentially, it should be an agent capable of planning, as described in chapter 10.

In this way an actual "thought process" will be the base of the agent's actions. It will still be able to do the same things, but now the agent will think: "I want the sword of Azgelblort." and "I have the shield of the dwarf king" and "the player wants the shield", and from there decide that sending the player on the quest is a good idea. Similarly, when prompted for information, the agent can decide if it is actually a good idea to answer truthfully.

4.2 Entertainment

One must remember that the point of games, is that they are entertaining. The agents can have all sorts of advanced AI and be very intelligent, but if they are not entertaining their intelligence is irellevant. From a technical point of view, the agents might be interesting, but the game designer must consider if the players will also find them interesting. If not, the extra AI is just a waste of resources.

This is not to say, that intelligent agents cannot add entertainment value to a game. They can, if the intelligence is used in a way that is visible to the player, and makes him go "that was clever" when the agent does something. In some games, like RTS games, it is easy to see when the opponent does something clever. For instance, if he uses special manouvers when attacking or builds his base in a particular way. In a RPG, where the agents are like actors, it is more difficult to immediatly see if they are being "clever". You cannot see the plans and goals inside the "head" of the agent. Thus, it may be difficult for the player to determine if the agents actions are smart. Therefore the agent must somehow show the player what he is doing, but in a non-obvious way.

Chapter 5

Communication

As we have already mentioned, it is our opinion that a key part of the agent is the ability to relate what is going on inside its "head" to the player. We split the things that goes on inside the agent, in to these three core aspects:

Personality A more or less static part of the agent that affects the way it expresses itself.

Emotions The effect that different situations and attitudes towards other characters have on the way the agent expresses itself.

Cognition The actions of the agent, and the reasoning behind these actions. The actual thought-process of the agents, so to speak.

The following three chapters deal with each of these aspects. The way the agent relates these things to the player can be both verbal and visual.

We also need a representation of the lines of dialogue that will be passed between agents and players. The following describes a representation that allows it to be easily understood by other agents, as well as the player.

5.1 Dialogue Representation

A line of dialogue is represented by two strings. One string is the actual line presented to the player. The other string is a representation that is easily interpreted by an agent. For instance, the sentence "I'm a lumberjack" would be represented as JOB(lumberjack, Agent). Here the representation takes the form of a belief from the agents knowledge-base.

The previous example illustrates an agent relating information from its knowledge-base. However, it is also possible to query the knowledge-base, which is how we represent questions. The sentence "Where is the micro-film?" would be represented by LOCATION(?, micro-film). The agent that is being asked will then use this as a query to its knowledge-base, finding any beliefs that match LOCATION(?, micro-film) where the "?" is a wildcard.

Emotions, attitudes and opinions towards other characters can also be represented. "I hate, I hate, I hate Peter Pan!" would be represented to an agent as NEGATIVE(Captain Hook, Peter Pan). Although this takes the form of a belief, it does not actually come from the knowledge-base. These inter-agent attitudes are represented in the dissonance-table described in section 7.

In section 7 it is also described how these attitudes can affect the way an agent responds to queries. It may choose not to answer if it dislikes the character whom is asking.

There can also be other reason not to answer a question. For instance, a game might require agents to keep secrets from each-other, or try to decieve each-other.

5.2 Secrets and Lies

One thing that the players in a traditional RPG are able to, that agents in a computer game has not thus far been able to do, is to deceive one-another. One thing that we can do to give the agents this option, is to modify the way they respond to questions. When asked a question the agent can either reply normally, decide that it is a secret that he will not tell, or try to deceive the asker with false information. Secrets are handled by a mapping from a belief to a set of characters sharing that secret. In other words, if the asker is not in that set of characters, the agent should not answer his question. Lies are handled in a similar way, but instead of only mapping to a set of characters, it also maps to an alternative belief that these characters should be told. For instance if Gandalf is asked who has the ring by a character he does not trust, and he knows this to be Frodo, he will look up HAS(Frodo, The Ring) in his lies mapping and find (Sauron, Saruman, Wormtongue, Gollum, HAS(Aragorn, The Ring)). Thus Gandalf should lie, and say that Aragorn has the ring. If the character who is asking is not one of these, he would look it up in his secrets mapping and find (Sam, Frodo, Pippin, Merry, Gimli, Legolas, Aragorn, Boromir, Galadriel, Elron). These are the characters he trust with the information. If it is anyone else, he would refrain from answering.

Chapter 6

Representing Personality

An important part of all forms of entertainment, both film, games and books, is to have interesting characters. Usually for a character to be "good" he has to have an interesting personality, and most importantly that personality has to come across to the player. The things the character say, the way in which they are said, the way the character moves and acts. All these things should match the characters personality. In order to ensure this, it is important that the characters personality remains consistent throughout the game. To make this easier it is a good idea to have the characters personality defined on paper for easy reference. That a description of the characters personality is done, is more important than which tools or methods are used to created that description. However, we will briefly describe the method we have used to design all our characters.

6.1 Character Diamonds

In designing our characters we use concepts from David Freeman's book "Creating Emotion in Games: The Craft and Art of Emotioneering" [3]. The book presents a number of tools and techniques for making the different parts of a game deep and interesting. He calls these techniques deepening techniques and interesting techniques, respectively. He has techniques for dealing with dialogue, characters, relationships between characters and many other aspects of games.

As mentioned we want to create characters with interesting personalities. Consequently, we will look at what he calls a "NPC Interesting Technique", which simply means that it is a technique for making non-player characters interesting.

A concept he uses when designing characters is the concept of traits. A trait is a major part of a characters personality and can be manifested in both action and dialogue. Only those aspects of a character that define his personality are considered to be traits. If a character becomes angry, when someone cheats in front of him in the line at the supermarket, this is not necessarily a trait but simply an appropriate response in the given situation. Likewise, if a character only eats thai food, or walks with a limp, this is not a trait but a quirk. Quirks add detail and texture to a character, but are not essential to the characters personality.

The list of traits that define a character is called that character's "character diamond". Usually a character has four traits, thus the name "Diamond". However, both three and five traits can be used. If a character has more than five traits his personality can become muddled, and it can be hard for the player to make sense of all the different traits. Minor characters may even have only one or two traits.

An example character diamond for a pirate could be:

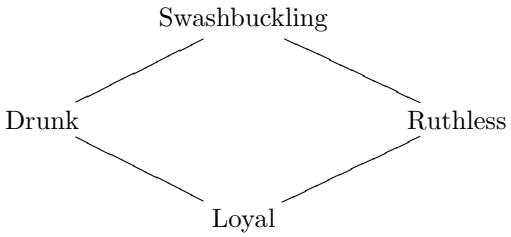
Swashbuckling He never says no to swinging on a rope with a rapier clenched between his teeth.

Drunk Rum is a pirates best friend, is his motto.

Ruthless If you are not his friend, he does not care if you live or die.

Loyalty Once you are his friend, he is loyal forever.

This could also be written as:



This only serves as a more visual representation of the diamond, but the placement of the individual traits has no significance.

Some traits might be a false front or "mask". For instance, if we were to add the trait "remorseful" to the pirate character, his drunken ruthlessness might be a mask to cover this. Using "masks" is what Freeman calls an "NPC deepening technique", and it is used to add depth to a character.

This technique can give us characters with interesting personalities, but how do we convey these personalities? Two ways are with the graphics used to represent the character, and with the things the character says.

6.2 Personality in Graphics

The way a character looks is not necessarily connected to their personality, but it can be. Both quirks and traits can be shown using graphics.

Many quirks can be shown in graphics. For instance, in the LucasArts adventure-game "Indiana Jones and the Fate of Atlantis", the main bad-guy always runs his fingers through his hair. See figure 6.1. This is a quirk that is shown in graphics. Other quirks could be a character that walks with a limp, or always chews gum. These kind of quirks are not essential to the characters personality, but help add extra detail to the character.

Figure 6.1: Klaus Kerner from "Indiana Jones and the Fate of Atlantis".

Some, but not all, traits can be shown using graphics. If it is a trait of a character that he is happy or sad, then this could be shown with the graphics used to

represent the character. However, traits such as being naive or polite could not. They have to be represented through the dialogue.

6.3 Personality in Dialogue

One thing to consider when creating dialogue for NPC's, is that several sentences can convey the same information. These sentences can be very different in style. For instance, it might be a part of a characters personality that he swears a lot. This will, of course, affect the choice of sentence. He would not say "the bucket seems to be leaking" but rather "there is a damn hole in the bloody bucket", which fits his personality better. If it is a player created character this is all right because the choice is the player's, but if it is an agent, he needs to choose the right one. This means that the agent needs some way to compare a sentence to his personality. So how can we accomplish this?

We have a series of attributes for the sentence templates. Examples of attributes could be; how rude, friendly, sad, unfriendly, funny, sarcastic, polite or happy the sentence is. The exact attributes are determined based on which characters are in the game. These attributes create a vector where each attribute is a dimension. We will call this the sentence vector. The sentence vector is set when the template is created. This is a subjective process in which the game-designer will go trough all the attributes, and assign a value to each. For instance, the sentence "there is a damn hole in the bloody bucket" will probably get a high value in the rude attribute, and low in polite.

Each agent will have a similar vector that describes the way he expresses himself. We call this the agent vector. The agent vector has all the same attributes as the sentence vector. Each attribute value describes a facet of the agents personality that manifests itself in the way the agent talks. Now, when the agent has arrived at a group of sentences expressing what he wants to say, he can simply compare these vectors to choose the sentence that fits his personality. This is done by taking the vector distance from the agent vector to each sentence vector. The sentence vector that has the shortest distance is chosen, as it is most like the agents personality. This approach was inspired by our work on a spam filter based on "weighted inverse document frequency" [11], although this topic has nothing to do with games or agents. In this project, comparing vectors was used for classifying texts.

By combining attributes that are opposites, or near opposites, we have arrived at the following axes on the vector: friendly/unfriendly, funny/serious, happy/sad, and polite/rude. In this way we can create a vector for the sentence "there is a damn hole in the bloody bucket". This could be: <friendly=0.0, funny=0.2, happy=-0.4, polite=-0.8> or similar. Different axes might be needed in other games where the characters have other personalities.

We use the euclidean distance $\sqrt{\Sigma(xi - yi)^2}$, between the two vectors x and y. If two vectors are exactly the same, the distance will of course be zero. This corresponds to the sentence matching the personality perfectly. Our attributes have values ranging from 1 to -1. Thus, the maximum distance for vectors with only one attribute would be $\sqrt{(1-(-1)^2}$ or $\sqrt{4}$. For vectors with n attributes the maximum distance is $\sqrt{4 * n}$. However, it is unlikely that the distance will be near the maximum, as this implies complete disagreement on all attributes.

The sarcastic attribute is slightly tricky. It is not an axis like the others. Either the sentence is sarcastic or it is sincere, there is no in between. Thus, it should be a boolean attribute on the sentence if there are any characters that need it. If a character were to say "what a wonderful smell you've discovered", but he is sarcastic, what would another agent think? The literal meaning and the intended meaning

of the sentence are very different. We assume that our characters understand sarcasm, and the agent would receive the intended FOPL expression. However, if the game design requires characters that do not understand sarcasm, this can be solved by sending two FOPL expressions. One containing the literal meaning of the sentence, and one containing the intended meaning. For Instance the literal meaning of "what a wonderful smell you've discovered" is represented by `Good( Smell )`, but the intended meaning is `Bad( Smell )`.

6.3.1 Testing the Vector Comparison Approach

In order to test if comparing vectors is a usable approach, we have written a small test. We make two agents with differing personalities express three beliefs. As the two personalities we have chosen the two cartoon characters Tintin and Captain Haddock, because they have well defined, and very different personalities. We create personality vectors for these characters.

Tintin <friendly=0.7, funny=-0.5, happy=0.5, polite=0.8, sober=0.7>

Haddock <friendly=-0.5, funny=0.7, happy=-0.4, polite=-0.7, sober=-0.8>

Here we have added the sober/drunk axis, because the character Haddock needs it. However, in most games this attribute is probably not needed. The sober/drunk axis is different in that it can be affected by using an object in the game, i.e. drinking a glass of whisky. We have chosen the following three beliefs for the characters to express:

FUN(Games) , that the agent thinks that games are fun.

PLAYS(Self, Games) , that the agent plays games itself.

STUPID(Non-players) , that the agent finds people, that do not play games, stupid.

For each of these beliefs we make three or four different sentences, to express it in different ways. Below are tables for each of the beliefs, showing the different sentences, their vectors, and which character selected which sentence. The vectors all follow the <friendly, funny, happy, polite, sober> pattern. The first table shows the FUN(Games) belief.

Sentence	Vector	Chosen By
Games are fun.	<0, -0.4, 0.3, 0.1, 0>	
Games are thoroughly enjoyable.	<0, -0.1, 0, 0.9, 1>	Tintin
Those games are like, really, really... erm... tip of my tongue, good!	<0, 0.3, 0.4, 0, -0.6>	Haddock
Games are absolutely marvellous.	<0.3, 0.2, 0.6, 0.2, 0>	

Haddock uses the sentence "Those games are like, really, really... erm... tip of my tongue, good!", whereas Tintin chooses to use the sentence "Games are thoroughly enjoyable." Haddocks choice is obviously the one that fits his personality best, but "Games are absolutely marvellous." also seems to fit Tintin's personality. However the distance between agent-vector and sentence-vector shows that "Games are thoroughly enjoyable." is a better fit. It has a distance of 1 from the agent-vector, whereas the other candidate had a distance of 1.22882. With five attributes the distance lies between 0 and $\sqrt{20}$ (4.4721). Considering this, the difference between the sentences is not great. If the agent is required to express the same thing twice (e.g. the player asks him twice), perhaps it should be allowed to choose sentences that have a distance that is a small percentage larger than the closest one. In this way it is possible to avoid the agent repeating itself endlessly.

The sentences for the belief PLAYS(Self, Games) is shown in the following table.

Sentence	Vector	Chosen By
I play games.	<0.3, -0.1, 0, 0.3, 0>	
I like playing games myself.	<0.4, 0, 0.5, 0.2, 0>	Tintin
I play games all the bloody time.	<-0.3, 0.1, -0.2, -0.4, -0.6>	Haddock

Again Tintin chooses a line that fits his mild-mannered personality, whereas Captain Haddock jumps on the chance to swear.

The next table is for the STUPID(Non-players) belief.

Sentence	Vector	Chosen By
People that don't play games must not have the right sort of mind.	<-0.3, -0.3, 0, 0.4, 0>	
I certainly can't see the reason why some people don't play games.	<0, -0.3, 0, 0.8, 0.6>	Tintin
You must be an ignorant imbecile, not to play games!	<-0.7, 0.3, -0.3, -0.6, -0.1>	Haddock

Tintin says "I certainly can't see the reason why some people don't play games.", because he cannot bring himself to call someone stupid, even though that is his belief. Haddock, on the other hand, does not care that he is rude and unfriendly, just as his agent-vector dictates. In the sentence "You must be an ignorant imbecile, not to play games!", the difference between Tintin's and Haddock's personalities is clear. The sentence has a distance of only 0.842615 to Haddock's agent vector, but 2.41661 to Tintin's. Again we can see that a sentence is unlikely to reach the maximum distance. Even though this sentence is one that Tintin would never say, it is still far from the maximum distance.

If we put these sentences one after another, we find that Tintin would say: "Games are thoroughly enjoyable. I like playing games myself. I certainly can't see the reason why some people don't play games."

And Haddock would say: "Those games are like, really, really... erm... tip of my tongue, good! I play games all the bloody time. You must be an ignorant imbecile, not to play games!"

These bits of text *do* express the right beliefs, but there is room for improvement. The text does not seem to flow well. This could be solved using **reference** as described in chapter 18 of [5]. Reference means that we can replace some words in our sentences with other words, such as "he", "it", "they" and so on. For instance if Tintin was asked "what do you think of games?" he could reply: "They are thoroughly enjoyable. I like playing them myself. I certainly can't see the reason why some people don't play them." Of course, reference is more complex than this. For instance, a reference does not last forever, so to speak. If we added just one more sentence, we should probably use "games" instead of "them" or "they". Similarly, if we put a sentence unrelated to games into the text, we would also need to use "games" in the following sentence.

Chapter 7

Representing Emotions

In order for NPC's to be really interesting, they need to be able to display their emotions to the player. The agents do not really need to *have* emotions. We only need to convince the player that they are having them, which should be a much simpler task. Internally the agent's emotions could just be represented by a variable or a state, but towards the player it should use all the same tools as an actor would. This can be facial expressions, body-language, tone of voice, and the things said and done.

In academia however, more research seems to be focused on simulating emotions realistically, than on making them interesting and dramatic to a player. For instance in [6] Kesteren, Akker, Poel and Nijholt use neural networks to teach an agent how its emotional state should change given different stimuli. In a game, not only would this be overkill, it also contains no way to make the player emotionally involved. Similarly, in [2] Dolores Cañamero presents a system that simulates emotions with hormones, bloodpressure and other physical effects. However, these emotions are displayed to the user as a series of numbers. So, while these approaches are technically interesting, they do not provide us with a way to display our characters emotions in an interresting and dramatic way.

If we instead look at books on game design, and books written by game designers, maybe we will find something. Surprisingly, this is not the case. The books we have looked at are either too technical, or too abstract. None touch on the subject of how to convey an agents emotions. For instance, David Freeman's book [3] is too abstract. It tells us how to design our characters, story arcs, backstories, etc. so they contain emotion. But it does not tell us how to convey these emotions. On the other hand many books, such as Erik Bethke's "Game Development and Production" [1], mostly touch on bussiness aspects, while others dwell on technical detail, still others contain post-mortems and anecdotes from games the author has worked on. But none of the books we have examined focus on the important topic of: "How do I get what I have designed across to the player?"

This means that we have to find our own way to do this. If we maintain that our agents need to be able to express their emotions using facial expressions, body-language, tone of voice, and the things it says and does, this gives us a good starting point.

First we will look at the dissonance table, which is at the core of our agent's emotions. The dissonance table allows an agent to have positive or negative attitudes towards other characters and events.

7.1 Dissonance

The dissonance table is a matrix containing $n \times n$ elements. Along each of the axes are n objects or events. It is the same n elements along both axes. Each element in the matrix denotes an object or events attitude towards another object or event. From the know attitudes, other attitudes can be derived using certain rules. Implementation details on the dissonance table can be seen in section 11.4.2.

As an example of the power of the dissonance table, we have devised a small example consisting of three objects in Haddock's table: Tintin, Haddock, Rastapopoulos and one event: Fighting Crime. Tintin promotes fighting crime, while Haddock promotes Tintin. Rastapopoulos obstructs Tintin, a graphical representation of these relations are shown in figure 7.1.

	Fighting Crime	Haddock	Rastapopoulos	Tintin
Fighting Crime				
Haddock				+
Rastapopoulos				-
Tintin	+			

Figure 7.1: Initial dissonance table

After applying the rules to the initial table, we get the new attitudes shown in figure 7.2.

	Fighting Crime	Haddock	Rastapopoulos	Tintin
Fighting Crime				
Haddock	+	+	-	+
Rastapopoulos	-	-	+	-
Tintin	+	+	-	+

Figure 7.2: Dissonance table with derived attitudes.

Applying the rules gives a lot of new information. For instance we can now see that Haddock does not like Rastapopoulos. The table does not only contain Haddock's attitudes, but also the attitudes that Haddock thinks that others have. An example of this is that he thinks that Tintin dislikes Rastapopoulos.

7.2 The Importance of Graphics

As mentioned, facial expressions, body-language and gestures are important in conveying emotions. The way to simulate body-language and facial expressions is, of course, through graphics.

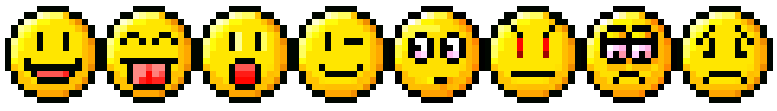


Figure 7.3: Emoticons from the instant messaging software RhymBox.

As the emoticons from RhymBox[1] shown in figure 7.3 illustrate, it is possible to represent recognizable facial expressions using simple graphics. Of course such

[1] http://www.rhymbox.com

simple graphics are rarely usable in games, but the principle is still sound. In a game using 2D bitmap graphics, a different sprite for each emotion would be needed. In a 3D game a different texture for the characters head might be enough, at least for the facial expressions. Similarly, body-language could also be created using different sprites in bitmap graphics. In 3D new animations of the character models are needed. Sadly, these techniques are rarely use to create interesting body-language, and more often for "cool moves", such as spinning your flaming sword above your head, or doing an impressive kung-fu kick.

7.3 Emotions in Dialogue

Emotions can be expressed through dialogue in different ways. There are things that are said because of emotions, and there are things that were going to be said anyway, but were changed by emotions.

There are several different types of lines that are said because of emotions. There are lines which are direct emotional responses to situations or events. These can be lines such as: "Yuckk!", "Oh no!", "Dang it all to heck!" and so on. An agent can also express its feelings on past events: "I cried when Mama died.", "The day you were born was the happiest day in my life." and so forth. Things that are in a characters dissonance table can also be expressed. This could be sentences such as "I hate, I hate, I hate Peter Pan!" or "I love you."

Another way of showing emotions in dialogue, is through emotional tinting of things that the agent were going to say anyway. In order to do this, we can return to our sentence vector.

7.3.1 The Agent Vector Revisited

An advantage of the vector comparison approach is that the agent vector does not have to be static. It can change with experience, or be modified to the current situation.

For instance, a character that is otherwise friendly can be very harsh when talking to another character that he really loathes. This ties together the agent vector with the dissonance table. Both if an agent has a positive attitude towards another agent in its dissonance table, and if it has a negative attitude, this can have an effect on the agent vector used when talking to that agent.

Similarly, when a situation calls for some sort of emotional response, the agent vector can be modified. If an agent drops something heavy on his foot, he is more likely to swear. If one of the other characters, a friend, dies, the agent will be more sad in the way he expresses himself.

If we return to the example of Captain Haddock, he would talk in a different way depending on whether he is talking to his friend Tintin, or to their nemesis Rastapopoulos. A way to implement this is to have three different personality vectors. One that is used when talking to characters for which the agent has a positive attitude, one for those that are neutral or unknown, and one for negative. This will make him choose different sentences based on whether he likes or dislikes the one he is talking to.

Similarly, if Haddock's bottle of whisky is smashed, he will also react. However, this sort of reaction should be scripted in most cases, but it might also be accompanied by a change to his agent vector. This change would subside over time, such that Haddock's rage would disappear and be replaced by his normal mild-mannered personality.

Chapter 8

Representing Cognition

This chapter deals with how to convey to the player, that the agent is percieving and understanding the world, that the agent have a "thought process", and what the reasons behind it's actions are. To have a reasonably precise description of all this, using one word, we call it cognition.

So how do we convey these internal processes to the player? The only real tool we have that can do this, is dialogue.

8.1 Running your Mouth

In order to give insight into what cognition processes are going on inside the agent, we can let him explain them all in great detail. Sentences that deal with percieving and understanding the world could be: "I see a tree.", "I am hungry." and so on. A sentence that deals with the agents current action and motivations for that action could be: "I drink beer because I like beer." The problem with these sentences is that they are what David Freeman [3] calls and "robo-speak". This means that, while the lines explain what they need to explain, they are boring and unrealistic (nobody really speaks like that). The lines are also "on the nose", they are over-obvious and to the point. Saying things in a round about way, or only hinting at them, is much more interresting. A better way of saying "I drink beer because I like beer." would be simply "Mmmmh!"

The agent also needs to be able to explain several actions at once. This can be a simple chain of actions, like "I will gather wood to build a fire.", or more abstract plans, such as going to Mordor to destroy the ring.

8.2 Withholding Information

There are things that the agent might not want to say in the presence of certain other characters. If the agent is in the presence of such a character, he should either refrain from talking, or say that he is doing something else, than what he is, infact, doing.

8.3 How much to say

This section deals with the fine line between being easy to understand and becoming a farce. Or, in other words, how much information is too much information.

The "I see a tree." and "I am hungry." examples from before are examples of saying to much. The agent could say these things if he is really hungry, or if he sees an interesting artifact, an object of great value or importance.

Generally speaking, the more obvious it is visually what is going on, the less need there is for the agent to explain it. The less obvious, the more need to explain. For instance, if an agent is chopping wood with a great big axe, there is no need for him to explain what he is doing. He may explain *why* he is building a fire, if this is interresting. On the other hand, if an agent is just standing around on the middle of a bridge, he may mutter to himself: "I hate guard duty. Just standing here all day!", which gives the player a much better chance of understanding what is going on.

Chapter 9

Communication Interface

Now that the agents have a way of communicating their beliefs, intentions, and so on, to both players and other agents, we need to find a way to let the player communicate to the agents. We need to define an interface for this communication.

We will look at a couple of games from the Adventure and Role Playing genres, and examine their interface for communication. The reason for looking at games from these two genres, is that communication between player and agents typically play a larger role than in other genres. The games we will look at are:

The Secret of Monkey Island by LucasArts, or Lucasfilm Games as they were then known. A classic graphic adventure with a point-and-click interface. The way dialogue works in this game is similar to many other games.

Sam & Max Hit the Road , also by LucasArts. While also a point-and-click adventure, this game has a very different way of doing dialogue.

Icewind Dale, Baldur's Gate II by Black Isle Studios and Bioware respectively. These two role playing games use the same game-engine, and as such their interfaces are very similar.

9.1 The Secret of Monkey Island

While this is an old game, the way in which player to character interaction is done, is still used. When talking to a character the player is presented with a list of, often humorous, possible sentences say. Depending on the choice, new possible sentences may become available, or the NPC will terminate the conversation. See figure 9.1 for an example of this. So basically, as a player you have several choices of what to say, but the individual choices and replies to them are scripted in advance. Also the characters in this game are not really agents, and are in no way autonomous. So, of course, they can not be said to understand what the characters say to them, but because everything is scripted they will always be able to react in a way that fits what is said to them.

9.2 Sam & Max Hit the Road

While this game is in many ways similar to "The Secret of Monkey Island", the way the player interacts with characters is quite different. Instead of sentences the player is presented with a series of icons representing different topics of conversation, i.e. if you can talk about another character, that would be represented by a

Figure 9.1: Monkey Island dialog screenshot.

The available lines are:
"Hello, head.",
"May I please have that necklace?",
"Thank you for leading me to the ghost ship.", and
"Well, it's been nice. I'll just put you back now."

Figure 9.2: Sam & Max Hit the Road dialog screenshot

small picture of that character. See the example in figure 9.2. However, the player has no real control over what is said, just the topic.

9.3 Icewind Dale & Baldur's Gate II

Both games utilise a dialog system similar to the ones found in the classic adventure games, such as the previously mentioned Monkey Island game. For each NPC, there are only a limited number of topics of conversation. Like in Monkey Island the player is presented with a list of sentences to choose from.

Figure 9.3: Icewind Dale dialog screenshot.

The available lines are:
"Yes. I wanted to ask you some things about Easthaven.",
"Of course. I am a visitor to this town, and there are things I would know.", and
"I must take my leave. Farewell."

9.4 Our Interface

The common thing about all the above games is that they present the player with a small selection of things to say, but they do not offer any real freedom. Sometimes the choices are even just different ways of saying the same thing. This can be the right way of handling dialogue, if the game has you playing a certain character with a certain personality, or if the game relies on jokes, like the "Monkey Island" or "Sam & Max" games. However, if you create your own character in the game, as in traditional RPGs, this way of handling dialogue may feel restrictive, as the player is unable to choose what *he* thinks the character should say.

What we would like, is to give the player greater freedom in saying what he wants. Also the agents should be able to "understand" what is being said, and

Figure 9.4: Baldur's Gate II dialog screenshot.

The available lines are:
"I'd like to see what you have available.",
"You have every type of drink available? I hardly believe that.",
"You must hear things…a friend of mine has been taken captive by the Cowled Wizards. Do you know where they keep their prisoners?",
"Perhaps you could tell me about the war between the thieves' guilds I've been hearing about.", and
"I'll be on my way."

react accordingly. This also has the effect of making the player feel more part of the game world, and more important to the story.

The problem lies in how to make this freedom of expression possible in a way that does not feel cumbersome. By letting the player input text, like when chatting with humans on-line, anything can be said. However the problem of having the agents understand this is insurmountable. What we could do, is to use a large number of template sentences with slots for the player to fill out. This is similar to the way it is done in case grammar [7], and it might be possible to make an even more flexible interface by following the case grammar methodology more closely. However, such an interface may easily become too cumbersome to use for the player, with too many choices, and slots to be filled in.

However, even an interface that uses slots still presents the player with too many sentences at the same time. One way to lessen this problem is to group sentences together according to type. One type of sentence could be questions. These could then also be grouped by type. One type of question could be those regarding the location of things. These can be grouped into those regarding the location of characters, those regarding the location of objects and those regarding the location of places. If the player has chosen that he wants to say a question regarding the location of a place, he could then be presented with a list of sentences fitting this category: "Where is <location>?", "How do I get to <location> from here?", "Which way to <location>?" are examples. Here <location> is where the user can select the place he wishes to ask about, for example "Bag End". Some sentences might need to be present in several different groups. "Is there a <profession> in <location>?" deals not only with a location, but also with those agents that are of a certain profession. We can also use these slots when querying the knowledge base. For instance, a query, for asking "Can you give me directions to a tailor?", could be: LOCATION(Tailor, $location:1) & DIRECTIONS($location:1), where $location:1 is a variable. First he would look up the first part and answer: "There is a tailor in Bree." He would then use what he has found to look up DIRECTIONS(Bree).

In this way we can group sentences together until they can be presented in manageable numbers. However, the focus on making an interface that gives the player a large degree of freedom of expression, should not affect the simplicity or ease of use negatively. For instance, a form-filling approach will create a tree of choices. It is better to make this tree wide, rather than deep, as this would mean that the player should go through many levels of the tree, even to say the simplest things. On the other hand, if it becomes too wide, all benefits of the tree structure is lost, as the player still has an overwhelming amount of possibilities presented at once.

Chapter 10

Planning

Our planning architecture resembles that of the STRIPS (Stanford Research Institute Problem Solver) planner, a further discussion of this planning architecture is made in [4] and [9] chapter 11, page 377-380.

10.1 Beliefs

The first part of the architecture is the representation of the agents beliefs about the world, that is, they are not facts of the world, but they are what the agent believes to be true. The agents knowledge about the world is not perfect, but they plan as if it were. This opens for the possibility that the agent will make mistakes. We do not consider this to be a problem, as agents who make mistakes can give a more interesting game-play.

We use first order predicate logic (FOPL) to represent the beliefs of the agents, that is, FOPL without implications or combinatorial symbols such as *and* or *or*. Such a predicate could be:

HAS(Blackbeard, Axe)

These beliefs are stored in a knowledge base.

10.2 Actions

In order to do any planning, the agent must have some actions available to it. Actions change the world from one state to another.

Actions consists of conditions, effects and operations. Conditions and effects are lists of beliefs. Conditions are what should be true before the action can be executed, and effect are the additional beliefs that are true after the action is executed.

Each action contains some operations that should be executed when the action executes. That is, these operations are what actually gets executed, that actually changes the state of the world. For instance, the operations of the action called "Frodo picks up the Ring" could be to remove the Ring from the ground, add the Ring to Frodo's inventory and let the Ring change Frodo's personality.

An agent should have attitudes towards each action, these attitudes describe how the agent "feels" about executing a particular action. These attitudes takes the following form:

Obligation denotes how much the agent feels it *ought* to do the Action.

Desire denotes how much the agent *wants* to do the Action.

Ability denotes how confident the agent is that it will be able to do the Action.

Other attitudes can be added by the game designer, if they are needed. Each agent has an attitude-vector, with corresponding axes. These attitudes are used in the planning to ensure that the actions the agent chooses, is in accordance with the characters personality.

The game-designer should also set a range of values the attitudes can take. We suggest using the range from -1 to 1 and striving to keep the values close to 0, such that -1 and 1 has impact as extreme attitudes.

10.3 Goals

Each agent must have one or more goals that it work towards, the agent must choose one of these goals to pursue. A goal is the final action in a chain of actions forming a plan. The agent has complete freedom to choose whichever of its available goals, it is able to construct a plan for. The game designer can control what order the goals should be pursued in by using the conditions of the goals to require that other goals should be pursued first.

10.4 Perceptions

Two possible means of perception handling are possible: automatic and active.

With automatic perception the changes to the knowledge-base are automatically done by the world/low-level agent. This gives us a rather clean interface between the low-level and high-level of the agent. For instance, the position of the agent and other objects and what is currently observable by the agent. Things like updating the dissonance table with attitudes to agents/events based on perceptions are not, or at least should not be, decided by the low-level agent. It is very much a thing for the high-level agent to decide. And perhaps the dissonance table should not be updated according to the agent's perceptions, as this is not an easy task. Also, it adds complexity that may not have much effect in the way of making the game interesting. However, updating it from conversations with other agents is much easier, and also easier for the player to follow.

Active perception, on the other hand, requires the high-level agent to ask the low-level agent to check the world state. If a condition to an action requires the agent to be inside a given area, one of the conditions would be something like: AT(Agent, *area*), where the AT predicate is linked to a method in the low-level agent returning true or false. This gives a simpler interface between the low-level and high-level agent, the only thing the high-level agent can ask the low-level agent about is boolean expressions.

Both methods of perception has their strong and weak points. Automatic perception gives rich possibilities of the low-level agent, while having the potential of putting too much intelligence into it. The active perception gives the perception authority to the high-level agent, while having a simple interface for interaction between the two levels of the agent.

We have chosen to use a combination of these two techniques, mainly to lower the cost of the perception step to a level acceptable in modern games. There is no clear line between the responsibilities of the low-level and high-level agents other than a qualitative measure of which level would yield the lowest processing overhead.

As an example, the problem of path-finding, finding a path between two points in the world, is divided between the low-level and high-level agent. The low-level agent deals with avoiding to run into trees and other objects in the game, while the high-level agent deals with the bigger picture: getting from one area to another. To draw a parallel to Tolkien's Lord of the Rings Gandalf knows that he can get from the Shire to Rivendell and then over the mountains to get to Mordor, that is his high-level agent talking, while the low-level agent deals with taking one step after another and avoiding obstacles in the way.

10.5 Planner

Planning deals with connecting actions together, finding effects that satisfies the conditions of the previous actions. The planner start with a goal and finds the actions that have effects that satisfies the conditions of the goal. As such we do backwards planning, we start at the goal and work towards an initial action that has all its conditions satisfied by the agents knowledge base.

This design also has the possibility that planning may yield several viable plans, so we need some way to tell which plan to start executing. We would like to use the plan that is most true to the personality of the agent. As mentioned in section 10.2 each agent has its own personal attitude towards each action, if we combine this with an attitude-vector on each agent we can use this to construct an equation that will calculate a value for the level of personality coherence that each plan presents. The equation used to score the actions are shown in equation 10.1.

$$S_a = \sum_a \mathrm{Action_a} \times \mathrm{Agent_a} \tag{10.1}$$

The Action values are the agent attitude towards specific actions, the Agent values are values from the agents attitude-vector. The a variable goes through all the different attitudes: Obligation, Desire, Ability.

A plan can then be reduced to the median of all the actions it consists of, and the plan with the highest value can begin execution.

10.6 Failed plans

One thing we have not considered yet is what will happen if (or more likely when) a plan fails in the middle of execution. There are several possibilities. Planning could start all over. This would yield a more intelligent agent, but the question is whether this is what we really want. It might be more interesting if the agent just tried to devise a plan to either satisfy the failed action, or to replace the failed action. And only if no viable plan could be found, the agent would plan from the start again, maybe also trying to find a new goal. But before this happens we need do decide how a failed plan is detected.

In [9] section 12.5, two different detection methods are described: action monitoring and plan monitoring. Both methods are executed before any action is executed. The difference lies in how much checking is done. Action monitoring only checks the conditions of the action that is about to be executed. One could say that the agent only looks one step ahead. Plan monitoring checks all conditions of the remaining plan, the agent is thus able to make assumptions as to whether observations about the world state changes the requirements of some plan to be executed in the future.

Plan-monitoring would yield more intelligent agents. Using action-monitoring the agent would continue executing its plan, even though an event that would make finishing the plan impossible happened right in front of it. For instance, if an agent planned to transport a cart-load of potatoes across a bridge, and the bridge collapsed right in front of him, he would continue collecting potatoes until he had a cart full. Only then would he discover that his plan was impossible.

Now that we know that a failure has occurred, we need to do something about it. The agent knows what action will fail and now need to react to these new conditions. As mentioned, two approaches could be taken: re-plan from start or repair current plan. Re-plan from start just runs the planning algorithm again, maybe choosing a new goal instead of the current. Repair current plan tries to find a way to go from the current (unexpected) state, to the state the agent was supposed to be in. Repairing the current plan is almost always a better alternative than the more expensive re-planning. The agent already has a partially working in-progress plan, and with some mending we might be able to transform it into a working plan again. When choosing between several plans to repair the current plan, the agent applies the calculations described in section 10.5 above, to find the repairing plan that is more true to the agent's nature.

Chapter 11

Architecture

In this chapter the system required to support the goals described in the first chapter is designed.

Apart from the goals in the first chapter we have focused strongly on dividing the system into smaller components with weak dependencies. This is done to make the system easier to manage. It also allows both ourselves and other developers to use the parts of the system they might need in other projects, without having to adopt the whole system. It is our hope that this will lead to a higher adoption rate of the different components that make up the system.

One of the goals discussed in section 2.3 is the multi-player requirement. We have chosen to achieve this using a standard client-server approach, where players and agents are both clients. This approach gives us a clear cut definition of which world state is the "correct" one, the world server always contain the "correct" world state. The clients thus has to have full trust in the world server, while the world server only trusts the clients marginally. Everything the clients decide to do, is simulated inside the world server to assure correct simulation. This is also done to assure that no clients has more control over the world than others.

As a consequence of the above, the agents that inhabit the simulated world also have the same limitations as the players, regarding how much of the world they can perceive.

We have chosen to implement the management of the world state in a component called Odin, this component also contains the network "glue" code required to connect the clients to the server. Clients use a class called `NetClient` to talk to the world server. This class maintains the clients world state and communicates with the server to update its internal world state, and to communicate client actions to the server.

This system architecture yields a system overview as shown in figure 11.1.

Also visible in this view of the Odin component is an isometric client. This is a component for presenting the player with an isometric view of the world state contained within the low-level client. What an isometric view is exactly and how this relates to the client is described in section 11.3.

By choosing to design and implement loosely coupled components it is simpler to add other clients to the system, we have e.g. implemented a small web application that shows the number of users in a world state in a web browser, an example screenshot of this example can be seen in figure 11.2.

Our agent architecture simply connects to the world server using this `NetClient` class, and thus becomes just another part of the client/server model in the Odin component.

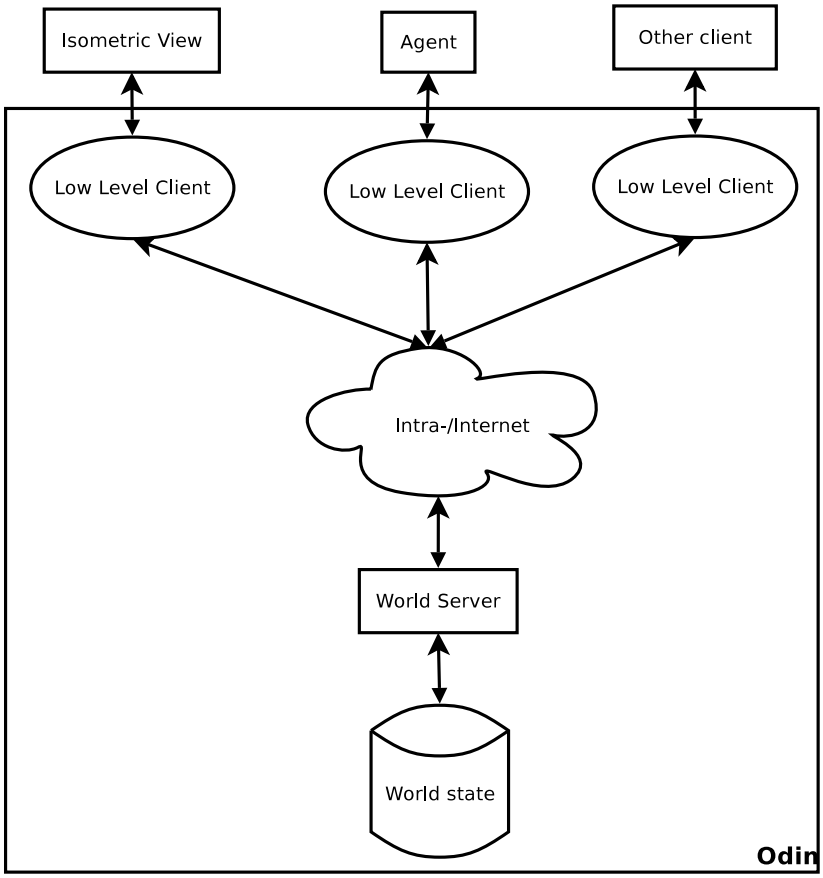


Figure 11.1: Overview of the networking architecture

Server: localhost

Servername:	Mobile Test Server
Hosted by:	Rasmus Toftdahl Olsen
Administrator:	rto@cs.auc.dk
Softwarename:	Odin WorldServer
Softwareversion:	0.0.0pre
Hardware:	Pentium III 1Ghz, 512MB RAM
Maximum number of concurrent users:	10
Number of users:	3
Users online:	1
World size:	1000x1000

Figure 11.2: Example output from the server info web application

11.1 Quad-Tree

The first problem to be tackled is how the objects should be represented in the world state. The representation needs to be both fast to access and efficient, both memory and processing wise.

Our initial thought was to use fixed/discreet positions for objects, that is, each object had an integer positions, and the ojects could only move from one integer position to another. The world was thus represented as a 2-dimensional array, each position in this array could be an object.

This idea is not in itself a bad idea, if the world is small, but one of the ideas was that the code should be able to allow hundreds of clients in a vast virtual world. If the world becomes just a bit larger than a normal single-player game world, the memory requirements for the world became much too large. With an object size of 12 bytes, two integers for the x and y positon and a pointer to an object type. A world of just 1000 times 1000 would take up 11.4 million bytes (mega-bytes) of memory. To enable the world to contain hundreds of clients and millions of objects a much larger world would be required, probably somthing along the lines of 1.000.000 times 1.000.000. This would yield a memory requirement of 10.9 trillion bytes (tera-bytes), too large for any current hardware to handle. Another world-representation clearly needed to be designed.

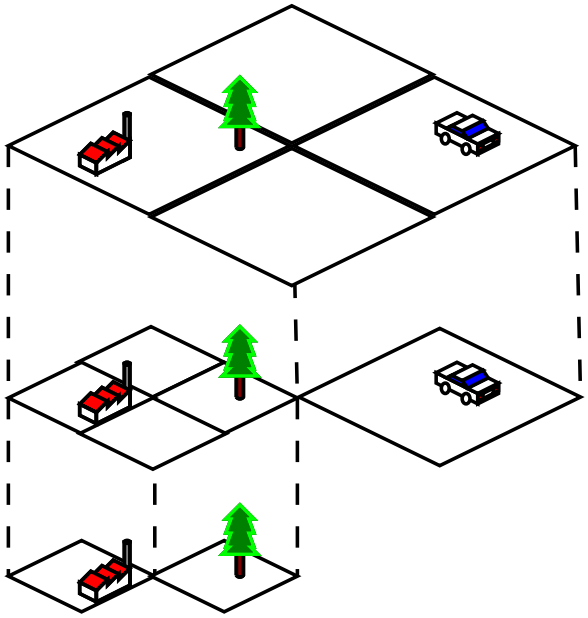


Figure 11.3: The splitting of the quad-tree

A common way to represent a world containg objects is to use a quad tree. A quad-tree is a tree data-structure where objects are placed according to their position in a 2-dimensional plane. The overall idea is to divide the world into four smaller parts: north-west (NW), north-east (NE), south-west (SW) and south-east (SE), these parts are divided further into fours until each object has an individual part, as shown in figure 11.3. Figure 11.4 shows the four quadrants that the world is initially divided into, and 11.5 shows this as a tree.

The whole idea of using a world representation is, besides being able to do collision detection between objects, to be able to find objects adjacent to another object quickly, e.g. to calculate what a client with a given visual radius can perceive.

A quad-tree differentiates between nodes and leaves, the tree itself is just a

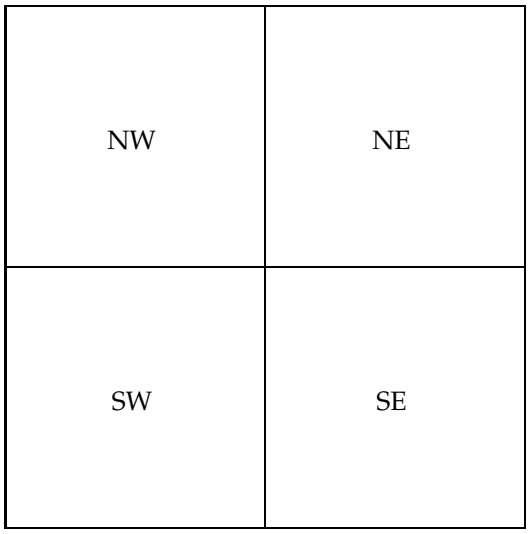


Figure 11.4: The intial quad-tree division of the world

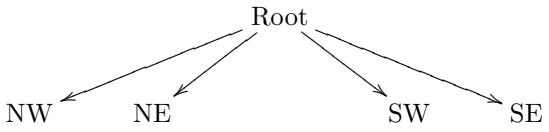


Figure 11.5: The initial quad-tree represented as a tree

node. Nodes contain other nodes and/or leaves, and leaf contains an object attached to the tree. A node contains four pointeres to child-nodes/leaves - one for each direction: NW, NE, SW and SE.

Each node and leaf contains the borders for the area of the world that they cover. This means that the borders of the root node are the borders of the world. In other words the root node covers the entire area of the world.

The algorthim for inserting an object is as follows:

1. Set root node as current node.
2. While current node is non-empty, and current node is not a leaf.
   (a) Find the child-node of the current-node that contains the position of the object to be inserted.
   (b) Set that child-node as the new current-node.
3. If current-node is empty, we insert a new leaf in the current-node's parent at the position in the direction of the object.
4. Otherwise if current-node is a leaf:
   (a) Convert leaf to node.
   (b) Insert leaf's old object into the new node.
   (c) Insert the object to be inserted into the new node.

The conversion of a leaf to a node is best illustrated using figures. If we have an object inserted into a tree in the root's NE child, as shown in figure 11.6 and then insert another object that is also in the root's NE child, the leaf that the first object previously occupied is now converted to a node, and the two objects is inserted into that node as shown in figure 11.7.

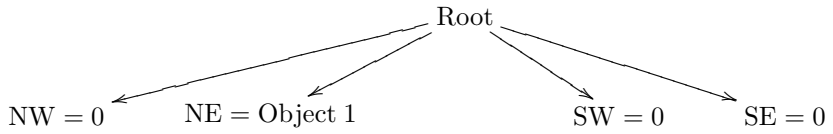


Figure 11.6: A quad-tree containing one object

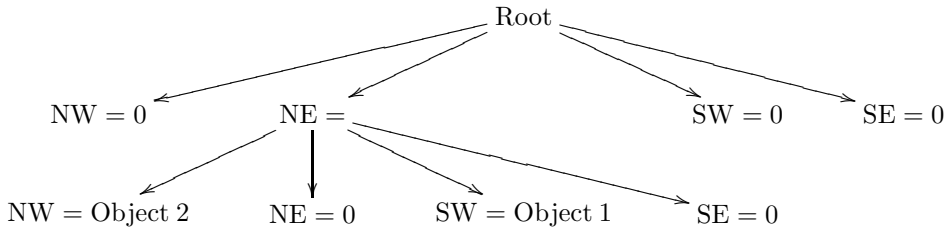


Figure 11.7: A quad-tree containing two objects, both in the root's NE child node

A place where our quad-tree differs from most other quad-trees out there, is that we have constructed a way for objects to be aware of the "level" that they are on, that is e.g.: sea-level, ground-level, in the air, etc. Besides defining the level they are on, objects can define what other levels they block. Objects blocking several levels could be a tall tower that both blocks the ground level and air-level.

Another way to solve this could be to use an oc-tree, a cousin of the quad-tree that is usually used to represent 3-dimensional worlds. The oc-tree splits a cubic space into eight equally sized smaller cubes. It quickly became apparent however, that all our objects would be ground-based, and as such an oc-tree would be overkill.

11.2 Odin

Odin is a library we have designed and implemented, it is the library containing the actual game-logic such as manipulation of the game world, keeping track of objects, object types, scripts, game rules. Another feature of Odin is the network sub-system. Besides containing a World class for representing the world, it also contains a WorldServer class that makes a World network aware, that is, it enables remote players and agents to log into the world and manipulate the world remotely. Odin also contains a NetClient class that makes it simple to write a network client, without requiring any actual network programming.

The networking protocol in Odin is based upon XML (eXtensible Markup Language), all Odin classes are derived from a class called Unknown, this class contains two method stubs to serialize a deriving class to XML or create an object from XML. This simplifies the design greatly as all Odin classes just need to define how

they should be serialized to/from XML, and the networking system will be able to serialize any object before sending it over the network. When an network packet is received it is parsed to an XML tree, and depending on the value of the root node, it is converted to the appropriate object.

For example a XML serialized object, would take the following form:

```
<object area="circle" id="8" x="46.019863" y="21.179991"
        r="0.700000" type="dwarf2" movex="0.000000"
        movey="0.000000" ismoving="0" level="ground">
    <blocks level="ground" />
    <properties>
        <property name="name" type="string"
                  value="Joel Stronginthearm" />
    </properties>
</object>
```

From this example we can also observe another nice feature of using XML, when a class contains another class, it can just serialize that object along with it's own data and it is equally easy when the object should be created from XML again. In this particular example each Object also holds a Properties class, a class for storing general information pertaining to an Object, in this case: the name of the object.

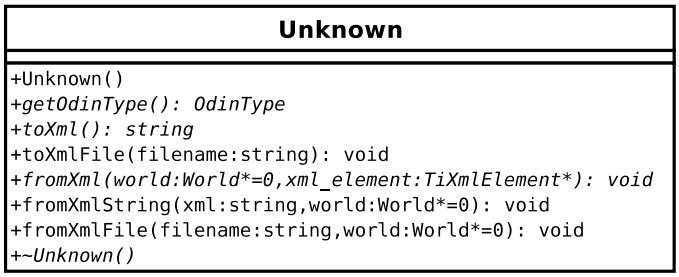


Figure 11.8: The Unknown class

We will not go into great detail about the Odin library, but will examine some of the key classes of the system.

The first class we will deal with is the Unknown class, a class that declares some methods that makes the rest of Odin able to serialise any inheriting object to XML. The Unknown class is defined in figure 11.8.

The three most important methods are getOdinType, toXml and fromXml. toXml serialises the object to XML and fromXml re-serialises the object from Xml. The TiXmlElement parameter of the fromXml is just an XML element from the XML parser we use[1]. A reference to the World is also supplied because an object might need to lookup some objects to fully serialise itself, e.g. the inventory needs to look up the objects it contains. The getOdinType simply returns an enumerated value, such that objects can tell the difference between, for instance, an object and a world.

The next class we will focus on is the Object class, this class represents any object that exists in the world, the uml diagram for the class is shown in figure 11.9. We do not include the getOdinType, toXml and fromXml methods, but they are, of course, defined in the Object class. Other methods that are not presented in the graph are methods for manipulating the inventory and calling scripts.

[1] http://grinninglizard.com/tinyxml/index.html

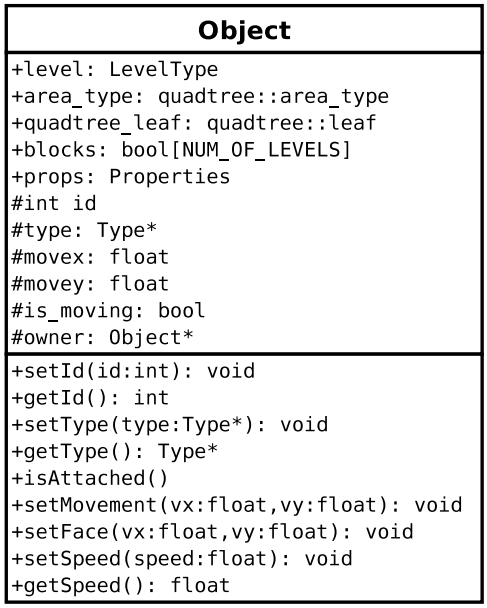


Figure 11.9: The Object class

Each Object has a unique id along with a type, e.g. Dwarf or Tree. The Object class also contains methods for manipulating the movement vector of the object. The movement vector is worth considering in detail, we want to express both movement, which direction the object is facing, and the speed of movement. This can be implemented using a single vector and a boolean stating whether the object is moving or not. If the object is moving the length of the vector is equal to the speed of movement. This arrangement means that the object will always face the same way as it is moving, this is a valid trade-off to having a separate vector for the direction the object is facing. Even this little increase in the memory footprint of an object would yield a substantial increase in the system requirements if millions of objects are supported. Any object can be in one of two states: attached or detached. An attached object is attached in the sense that it is positioned somewhere in the world. If it is detached it is in another object's inventory, and the object's owner reference is set.

The last class we will examine briefly is the World class, containing all objects, and simulating the world step by step. In figure 11.10 the most important methods of the World class are shown.

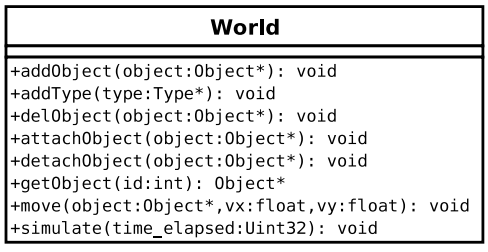


Figure 11.10: The World class

It contains methods for attaching/detaching objects, and moving them about, as well as adding and deleting objects. The simulate method simulates the world, the parameter to this method is the number of milliseconds that should be simu-

lated. For instance it will see to it that an object has moved the correct distance along its movement vector.

11.3 Isometric 3D

Isometric 3D, also known as semi-3D, is a common world view used in role playing games, this is the interface through which the player interacts with the world. While it gives the impression of being 3-dimensional it is in fact just 2-dimensional. One pro of this view is that it gives the player a good overview of the world. The player is commonly positioned in the middle of the view, and the player can observe the part of the world within the avatar's viewable area. This area is commonly circular, thus giving the player the ability to "look behind his back". This is obviously not a realistic view, but it gives the player some advantages over a first-person view. Human beings (and indeed many other animals) has the ability to turn their head without turning their body, this is normally a very quick motion and give humans a way to quickly get an overview of the environment they are in at the moment. This it not possible using a first-person view, for this to work the player should be able to move the view-direction independent of the move-direction using some secondary way of control (secondary to the mouse/joystick used to control the movement). And even if this secondary control could be possible it is still very unlikely to be as intuitive as the normal human way of doing it. Indeed, to make this work perfectly (without using a 6-sided cave environment) a device would have to use the actual head-momvements of the player while placing the world view in front of the player's eyes, that is, some kind of head-mounted goggles.

Our isometric 3D library has the ability to render a world, either stored locally or sent over a network. The player can navigate using the mouse, and make the avatar move about in the world. The view also contains methods for converting from world coordinates to screen coordinates, and vice-versa. These methods allow the objects to be positioned correctly on the screen, and allows the player's mouse clicks to be converted to world coordinates. A screenshot of a running view is shown in figure 11.11.

11.4 Agent architecture

Before defining the actual agent class, we start by describing some classes that the agent class utilises.

Along with these classes the corresponding file-formats for the serialisation of the agent is also described.

11.4.1 Knowledge Base

To contain the beliefs of an agent we use a knowledge base. First we define the Belief class in figure 11.12. Also included in this figure is the Beliefs class, this is simply an automatically growing array of Belief(s) that is implemented using the C++ Standard Template vector [10].

The KnowledgeBase class described in figure 11.13 contains the beliefs of an agent, it also contains methods for querying the knowledge base for the existence of some belief. These methods are called find, and comes in three different variations:

find(string) Return the beliefs whose predicate is equal to the supplied string.

Figure 11.11: A screenshot from our isometric 3D library

find(string, int) As the above but only returns beliefs that has the supplied integer number of terms.

find(string, int, string[ ]) As the above method, but considers the string array as a list of terms. Only returns the beliefs where term_n of the supplied terms is equal to term_n of the belief. The string "?" can be supplied to tell the query method to not consider one or more terms.

A query for HAS(?, The Ring) is written as find ("HAS", 2, ["?", "The Ring"]) and could yield a result like such as HAS(Frodo, The Ring).

The file-format for the knowledge base simply takes the form of one predicate per line in the file, e.g.:

```
has(Frodo, The One Ring)
location(Frodo, The Shire)
```

11.4.2 Dissonance Table

As explained in chapter 7 a good way to represent emotions is to use a dissonance table. In figure 11.14 the interface of our DissonanceTable class is described. Events and objects are added to the table by using the addObject and addEvent methods. Attitudes between two objects/events are defined using the setAttitude method, or the obstructs/promotes methods.

To do one calculation of the table, the calc method is be called. The table can then be examined using the isPositive, isNeutral and isNegative methods (or the getAttitude method).

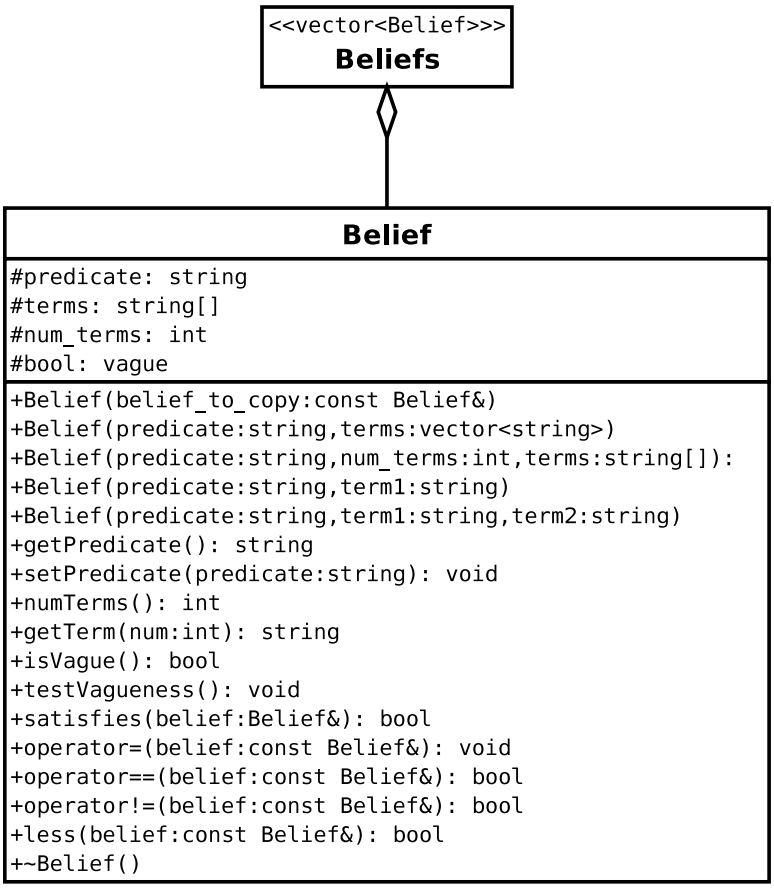


Figure 11.12: The Belief class

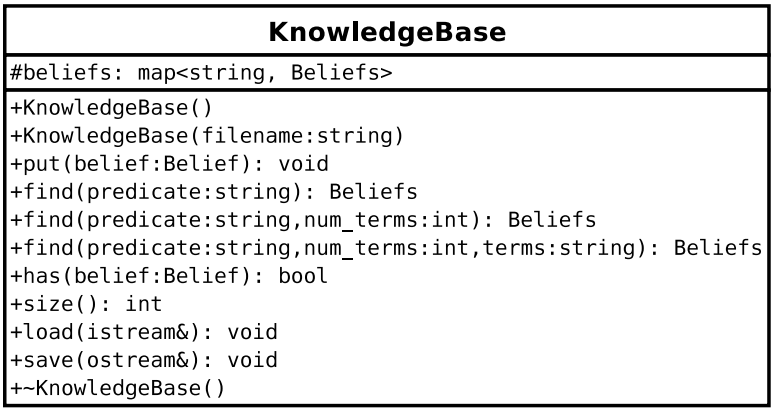


Figure 11.13: The KnowledgeBase class

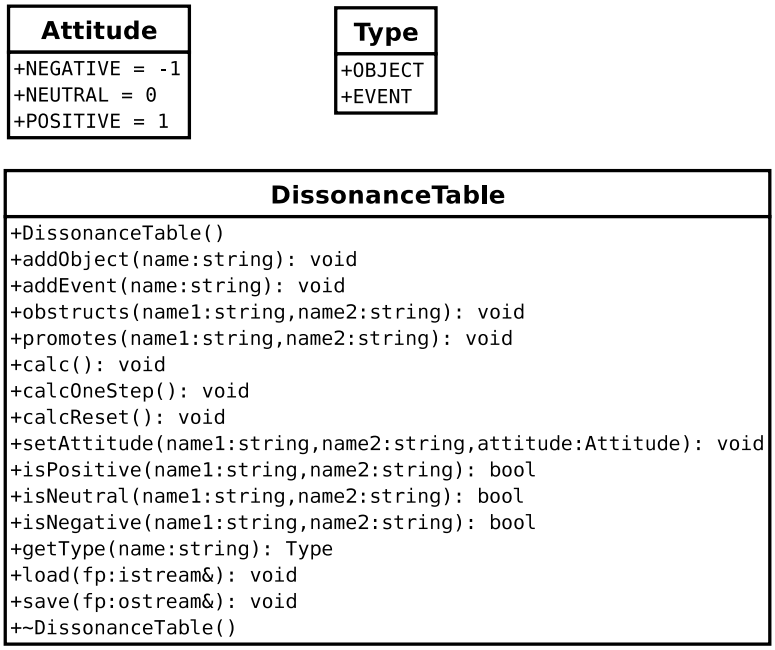


Figure 11.14: The DissonanceTable class

The calc method is actually just a wrapper around the calcOneStepmethod. This method is called until the CalculationDone exception is thrown. The calcOneStep method maintains internal pointers to keep track of where the calculation of the dissonance table is. The calculation can then be resumed whenever it is usable. This allows for a very flexible class that can calculate as fast or as slow as is required. For instance, the entire table could be calculated each iteration or one step per iteration.

As an example for using the DissonanceTable class we have implemented the example discussed in section 7.1:

```
DissonanceTable table;
table.addObject ( ‘‘Haddock’’ );
table.addObject ( ‘‘Tintin’’ );
table.addObject ( ‘‘Rastapopoulos’’ );
table.addEvent ( ‘‘Fighting Crime’’ );
table.promotes ( ‘‘Haddock’’, ‘‘Tintin’’ );
table.promotes ( ‘‘Tintin’’, ‘‘Fighting Crime’’ );
table.obstructs ( ‘‘Rastapopoulos’’, ‘‘Tintin’’ );
table.calc();
```

A query for `isNegative(“Haddock”, “Rastapopoulos”)` would now return true. If we used `getAttitude` it would return `NEGATIVE`.

The file-format of the dissonance table has two parts, first the declaration of objects and events, and then the attitudes between these objects and events. Objects and events are declared by their name, followed by “, O” or “, E”, which is interpreted as either object or event. Attitudes are declared using the names of the two objects or events with a plus or minus in the middle to signify either a positive or a negative attitude, e.g.:

```
Tintin, O
Haddock, O
```

```
Rastapopoulos, O
Fighting Crime, E

Haddock + Tintin

Tintin + Fighting Crime

Rastapopoulos - Tintin
```

11.4.3 Planner

The planner is the most complex class in our agent architecture, it must order actions to execute in order to get from the start state to the goal state.

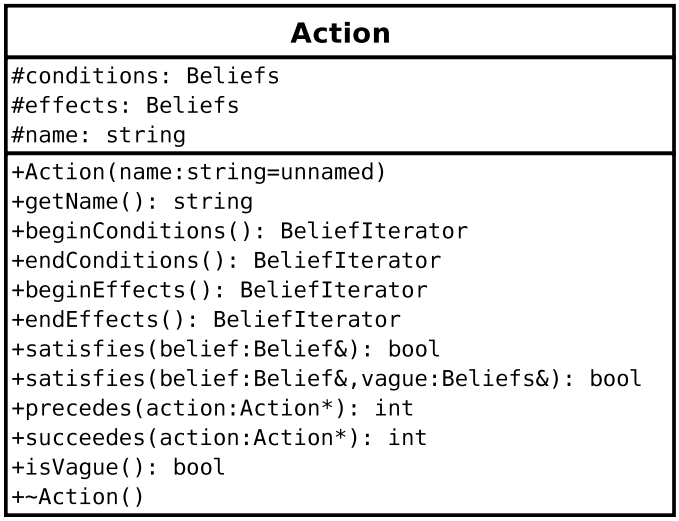


Figure 11.15: The Action class

The first class needed is an Action class, pictured in figure 11.15. This class contains some conditions that needs to be met for the action to be executed, and some effects that become true after the action has been executed. The Action class also contains methods for testing whether an Action can precede or succeed another Action. Precede here meaning that an Action's effects satisfies all the conditions of another Action, and succeed meaning that an Action's conditions are met by another Action's effects. These two methods are crucial for the Planner class, shown in figure 11.16, to establish a working plan.

The Planner class is given a list of goals to choose from, and the Planner selects a goal to pursue. However, we do not deal with goal selection, so for now, the game designer has to set a goal to be pursued.

Inside the Planner class is also a list of actions that the planner uses to form a plan.

When the planner tries to create a plan to reach a goal, it actually constructs a tree. The root of the tree is the goal and the leaves of the tree are the start points of the various available plans. Each node in the tree is represented as a PlanNode. The PlanNode class may seem overly complex, but we have to consider the possibility (or rather likeliness) that one action might need several actions to satisfy all its conditions.

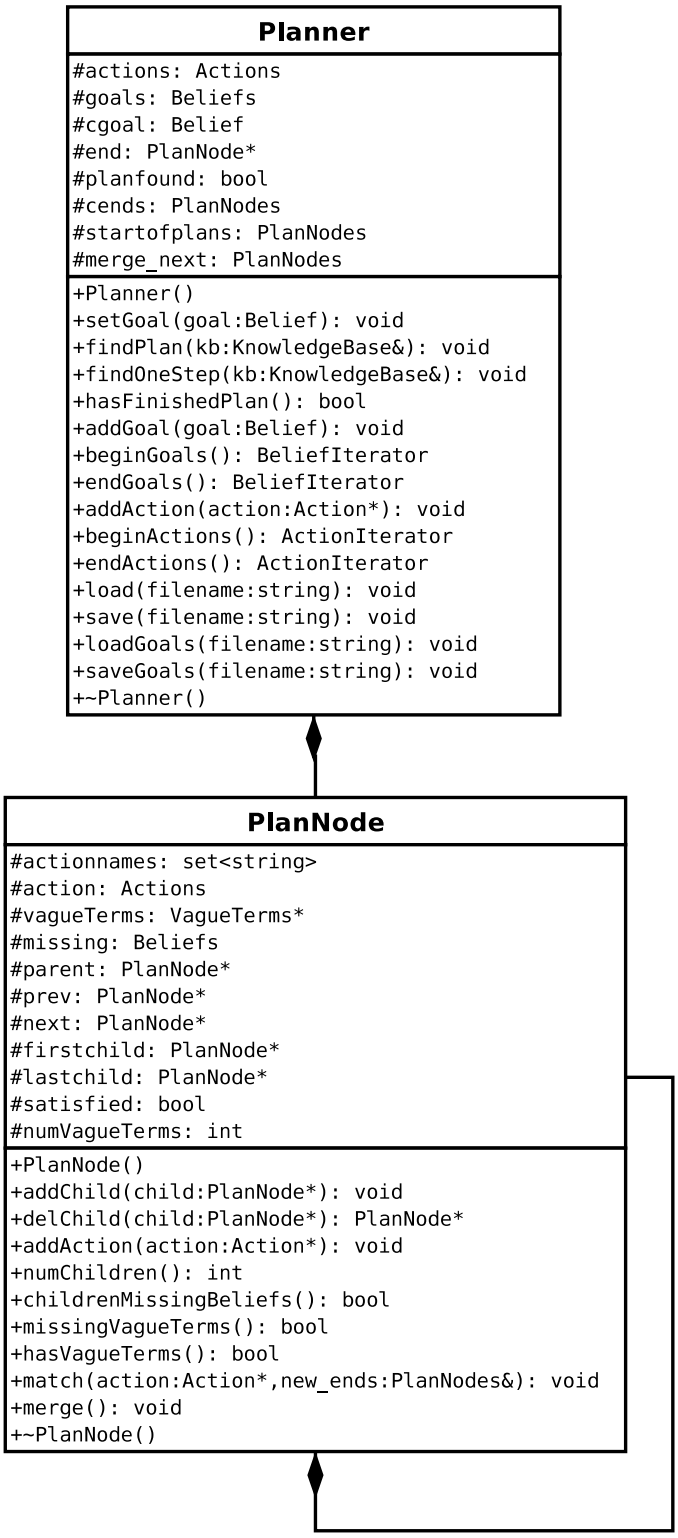


Figure 11.16: The Planner class

The Planner class works in steps. Each time the findOneStep method is called, another level is built upon the current tree, if possible. The nice thing about this arrangement is that the game-designer is able to tune the "speed of thought" of the agent. This should yield very different agents. This should be especially true when combined with the step-by-step calculation of the dissonance table.

The serialisation of the planner is done in two stages. The first stage is the goals, which are simply a list of beliefs, and can as such be serialised as a knowledge base.

The actions of the planner is defined using the following pattern:

```
name: conditions > effects
```

If there are more than one condition or one effect, they can be joined together using the "&" (and) character.

An example list of actions are shown below.

```
Sauron Claims the Ring : has ( Sauron, The Ring) >
                         claim ( Sauron, The Ring )

Find Frodo and catch him : find ( Sauron, Frodo ) >
                           caught ( Sauron, Frodo )

Kill Frodo : caught ( Sauron, Frodo ) > dead ( Frodo )
```

11.4.4 Communication system

Our architecture should also deal with what an agent would say in a given situation, what it would answer to a specific question, and more specifically how it would answer it.

As described in section 6.3 we want to employ a sentence-vector for all the different lines that the agents can say, and then compare that to the agent-vector.

The sentence-vector should therefore be modelled as a class, and since all the vectors share the same axes, they should be stored in a central place, in a SentenceSystem class. The sentence-vector is then reduced to be an n-tuple where n is the number of axes in the SentenceSystem, we can still look up the value of a sentence-vector by it's name rather than just the index of the particular axis in the tuple, this is done simply by looking the name up in the SentenceSystem use the returned index to find the value. This also means that all axes in the SentenceSystem must be defined before the sentence-vectors are created. The Uml diagram for the SentenceSystem and the SentenceVector is shown in figure 11.17.

The Lines class is just a container class containing an agent-vector, a line, and the ability to find a Line given some beliefs that this line should include. The Lines class contains a mapping from a belief to a list of lines that include the mapped fact. The Uml diagram for the Lines and the Line classes are shown in figure 11.18.

The Line class contains a list of beliefs that shows the information it conveys, it also contains the actual string that is to be uttered to the user and finally it contains a sentence-vector for each line. This sentence-vector is defined objectively by the game-designer, such that any line's sentence-vector can be compared to the agent-vector.

The lines can also be saved and loaded to a file with the following format:

```
Text to say <sentence-vector> beliefs expressed in the text
```

If some axes of the sentence-vector are undefined, they are set to 0 (neutral).

An example of some lines that the Sauron agent can say is shown below:

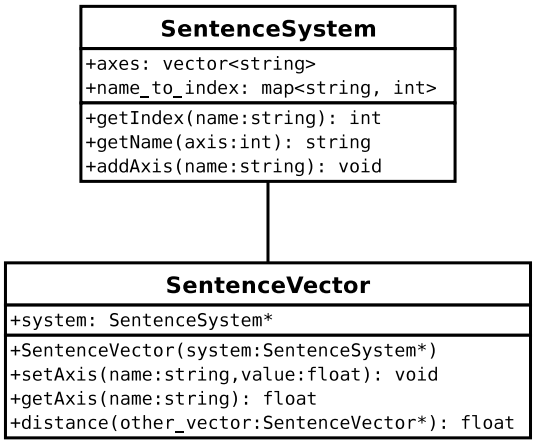


Figure 11.17: Uml diagram for the SentenceSystem

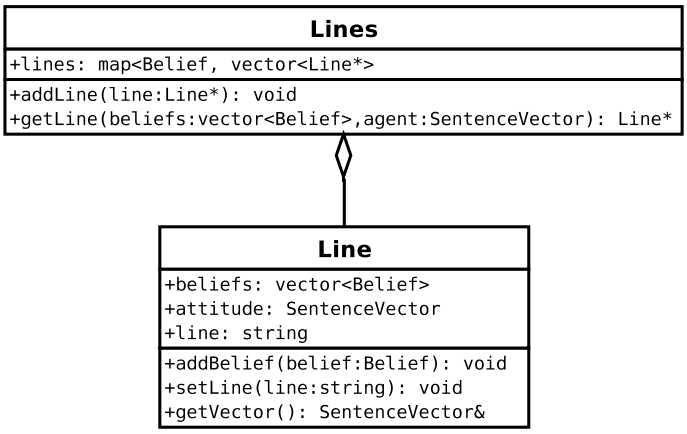


Figure 11.18: Uml diagram for the Line and Lines classes

```
Frodo has the Ring        < polite=0.1, funny=-0.2 >
                          HAS(Frodo,The Ring)
Bloody Frodo has the Ring < polite=-0.6, happy=-0.7 >
                          HAS(Frodo,The Ring)
```

11.4.5 Putting it all together

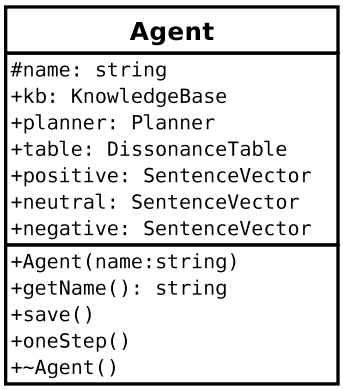


Figure 11.19: The Agent class

We now arrive at the point where we can combine the KnowledgeBase, Planner, DissonanceTable and SentenceVector classes into a combined Agent class. This class is shown in figure 11.19.

The oneStep method utilises the calcOneStep and findOneStep of the DissonanceTable and Planner classes, to find out what the agent should do in one iteration.

The save method of the Agent class simply calls the save methods of the underlying classes. The filenames used for saving is simply the name of the Agent postfixed by an common identifier for each class, e.g. the Sauron agent would consist of the following files:

```
Sauron.agent
Sauron.kb
Sauron.dissonance
Sauron.goals
Sauron.actions
Sauron.lines
```

The `Sauron.agent` file format has not been explained yet. When someone asks the agent a question it needs to know how to answer. Because an agent can have three different attitudes towards the asker, we give the agent three personality vectors that it can use when finding the lines matching the beliefs.

These personalities are defined in the `Sauron.agent` file, an example of such a file are shown below.

```
positive: friendly=1, funny=1, happy=1, polite=1, sober=1
neutral: friendly=0, funny=0, happy=0, polite=0, sober=0
negative: friendly=-1, funny=-1, happy=-1, polite=-1, sober=-1
```

Chapter 12

Multi-player

Using the client/server networking architecture inherent in the Odin library, we can build a multiplayer game, as described in the motivation & goals chapter. We decided on this, primarily to ensure that clients can trust that the other clients cannot cheat. By doing all simulation on the server, the server is the authority on the world's state. Clients also simulate the world, but must change their own version of the world when the server tells them to do so, at least if they want to stay in synchronisation with the world that the rest of the clients share using the server.

12.1 Client honesty

Because we have settled on an open protocol, with open source, the server cannot trust the clients in any way. A client with bugs or with cheating features should not be able to interfere with the server's world state. This means that the server needs to always check the packets sent by the clients. Furthermore the server should only send information relevant to the client, e.g. only transmit the other objects and their movements and positions if these objects are within the visual range of the client's avatar, this also goes for talking/chatting. Only talking within the client's avatar's hearing radius should be proxied to the client.

Differentiating between what objects are visible/hearable is not enough. Objects may contain information that should not be sent to the client. One obvious thing that should not be transfered as part of an object is the password, if it is a player or agent. Other things include inventories, spellbooks and other "personal" things.

12.2 Synchronisation

A problem that always occurs when you deal with network programming (or any programming dealing with several threads of execution) is synchronisation. When you have several clients with their own view of the world, and the server sends packets with updates to the client, the received updates will most likely be older than what the client has simulated on it's own, because of the delay caused by transporting the packets over the network. Consider the following scenario dealing with the movement vector and position of an object:

Description	Server	Client
Initial state	Movement = (1,0) Position = (5,5)	Movement = (1,0) Position = (5,5)
Next iteration	Movement = (1,0) Position = (6,5)	Movement = (1,0) Position = (6,5)
Object changes movement vector, server proxies this information to the client	Movement = (0,1) Position = (6,5)	Movement = (1,0) Position = (6,5)
Next iteration	Movement = (0,1) Position = (6,6)	Movement = (1,0) Position = (7,5)
Movement change information arrives at client	Movement = (0,1) Position = (6,6)	Movement = (0,1) Position = (7,5)

As can be seen from even this simple example, some action must be taken by the client to keep in synchronisation with the server. A way to keep somewhat clean of the problems is to send the position "every now and then" and then the client just positions the object in question at the position given. This is how it is implemented in our system at the moment.

A more clever way to do this is to send the position when the movement was changed and then use that information to calculate how far the object has moved since the client got out of synchronisation. When you know how far the object has moved, you also know how much time this has taken (movement is calculated using the time elapsed and the movement vector), this amount of time can then be used to calculate how far the object has moved since the movement vector was changed. This process is known as interpolation, a mathematical term describing the calculation of missing points using the surrounding points [12].

Chapter 13

Game Scenario

We have made a scenario, which is supposed to resemble a situation from a game. The scenario should have both player and agent controlled characters, and should use the different methods for giving the agents personality and for making them entertaining. The scenario is meant to point out places where our different agent components can be used, but also where they fail, and something else is needed. We do not examine the networking and multi-player aspects of the game scenario.

13.1 The Characters

Our scenario takes place in a small dwarf village. Four dwarfs live in the village. They are:

Ethan Stronginthearm This is the character the player controls.

Joel Stronginthearm This is the player's brother. Together they run the family sheep-farm.

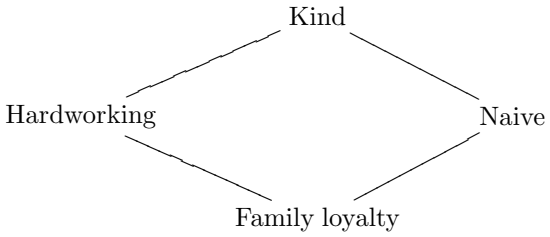
Blackbeard Hoodwink This unsavoury character sits around in front of his house, drinking mead, all day long.

Agi Hammerthief This is the local entrepreneur. He has a large farm where he grows barley from which he makes mead. He would like to have an even bigger farm.

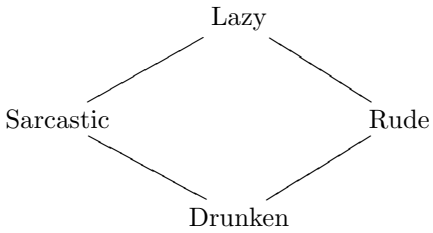
We have used the character diamond technique mentioned in section 6 on our characters to create the following character diamonds:

Ethan Stronginthearm Since he is the player's character, we let the player decide his personality by controlling the actions and dialogue he uses.

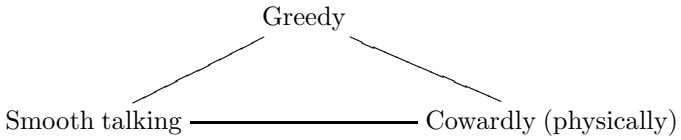
Joel Stronginthearm



Blackbeard Hoodwink



Agi Hammerthief



These traits should then be reflected in the way the characters talk and act.

Now we should also define the characters agent-vectors, for use when selecting dialogue. As before Ethan does not have a vector since he is player controlled. We define three vectors for each character, corresponding to the positive negative and neutral attitudes in their dissonance table.

Joel Stronginthearm

Positive <friendly=0.8, funny=-0.2, happy=0.7, polite=0.8, sober=0.8>
Neutral <friendly=0.5, funny=-0.2, happy=0.5, polite=0.7, sober=0.8>
Negative <friendly=-0.5, funny=-0.2, happy=-0.5, polite=0.3, sober=0.5>

Blackbeard Hoodwink

Positive <friendly=0.1, funny=0.7, happy=0.3, polite=-0.6, sober=-0.9>
Neutral <friendly=-0.5, funny=0.7, happy=-0.4, polite=-0.8, sober=-0.9>
Negative <friendly=-0.9, funny=0.7, happy=-0.8, polite=-0.9, sober=-0.9>

Agi Hammerthief

Positive <friendly=0.9, funny=-0.3, happy=0.6, polite=0.9, sober=0.8>
Neutral <friendly=0.8, funny=-0.3, happy=0.5, polite=0.8, sober=0.7>
Negative <friendly=-0.7, funny=-0.3, happy=-0.5, polite=-0.8, sober=0.5>

As we can see in Agi's character diamond he is both greedy and smooth talking. As a result of this, he should not use his negative vector, even when talking to characters he does not like. As he is interested in buying their farm, the actions he use should override the vector, if there is any chance of a deal.

13.2 Relationships

Here we will briefly describe the relationships between the different agents. Blackbeard and the Stronginthearms are neighbours. Blackbeard is always rude and his garden is a mess, which could be the cause of a bit of dispute between them. Agi Hammerthief also lives near the Stronginthearms. He has offered to buy their farm, but they declined because the farm has always been in the family. Agi has always been very friendly towards them. Agi sells Blackbeard all the mead he drinks. Blackbeard cannot pay for it all, but Agi lets him have a running tab. The tab has grown rather large.

13.3 The Scenario

Due to the nonlinear nature of our game, describing exactly how the game will proceed is not possible, but the following description explains the plot and one possible way it could be revealed. Things that happen internally in the agent is **written in bold** letters.

Joel Stronginthearm comes running up to his brother, the player, and tells him: "Someone let out all the sheep!" **It finds a sentence expressing FREE(Sheep).** If we could use the term "our sheep" instead of "the sheep", this would establish that the two characters know each-other, without the need to explain the entire back-story to the player. However we have not found a way to do this, that still preserves what is gained from using slots.

"Come on back to the farm. Help me round them up" says Joel. **He says a sentence that matches GOTO(Ethan, Farm) and another that matches CATCH(Ethan, Sheep) and CATCH(Joel, Sheep). The reason he asks Ethan this is because his goal is to bring Ethan to the farm.** The walk to the farm leads them past Blackbeard Hoodwink's house, thus letting the player see that he looks like a villain. This is of course because the game designer have laid the map out in this way.

While they are walking Joel talks about how this is just the latest in a string of bad things, amongst these things are: "This is the worst thing since someone poisoned the well". **The things Joel tells Ethan here would most likely be scripted. They would be operations in the action that moves him back the farm. It is unlikely that a sentence fitting this pattern could be used elsewhere in the game, and thus it might as well be scripted.**

Once they reach the farm Joel says that he can manage catching them alone, and asks Ethan, the player, to find out who let the sheep out. **He says !CATCH(Ethan, Sheep) and gives him the goal to catch the perpetrator.** This gives the player a quest. Of course the player can choose not to do this, but our little game has no other quests to complete. Joel begins rounding up the sheep.

Upon examining "the crime-scene", the player will find an axe that seems to have been used to make a hole in the fence. If he examines the axe he will find that it is inscribed with the name Uberbrucker Hoodwink. If he asks Joel about it, he will say that it belongs to Blackbeard. **Joel has the belief OWNER(Blackbeard, Axe).** If the player goes to ask Agi he will say he does not know whose axe it is. **The belief is in Agi's secret mapping.** If he asks Blackbeard, he will say that it was not his axe, and that his axe is in the backyard where he uses it for chopping wood. Upon examining the backyard the player will notice that no axe is there. Confronted with this, Blackbeard still claims that he had nothing to do with it, and that the axe must have been stolen. **This is something we cannot actually do. They are referring to the same object, but they have differing beliefs about it. Blackbeard knows the location of his axe, and thus this axe cannot be his.** If asked if anybody has been to his house, he starts raving about how Ethan should just mind his own: "That's none of your flipping business! Sod off!" **Blackbeard's attitude towards Ethan changes from neutral to negative.**

When Joel is done rounding up the sheep, he will join Ethan. Him and Blackbeard will start talking, if the player is done. Joel: "You let out all our sheep! This is just like that time you burned down our barn." Blackbeard: "That was a bloody accident! I was very, very drunk at the time." Blackbeard says that he just fell asleep with his pipe lit. They start to argue, possible sentences: "My father always told me to never trust a flaming Stronginthearm", "Don't you have anything to do? Your garden looks like a mess!", "Now, can't a bloke smoke his pipe without getting bloody screamed at?", "Well, not if you fall asleep in the middle of it", "You are just lazy!", "That's right! Come closer and we'll see whose got a lazy arm!", "A

Stronginthearm would never hit anyone!", "Ha, more like Weak-in-the-arm, can't even hurt a paralyzed piss-ant!". Blackbeard will be very rude, and Joel will not be able to defend himself in the argument. Joel becomes sad, and starts talking about how him and Ethan are going to loose the farm, Blackbeard breaks in and shows his warm side: "Sobbing ain't gonna help nobody, grab a beer, let's drink our sorrows away." Joel replies: "What sorrows . . . you have sorrows?", Blackbeard: "I've got a bloody running mead tab at Agi's - i think i'm in over my darn head! Now he's talking about taking my blooming house". Joel: "Well, maybe i jumped to conclusions about you. I better go back to the farm and watch the sheep, maybe he'll be back." Joel leaves towards the farm. Blackbeard: "He's ok, just a bit too soft." **This argument would probably be scripted, ending with Blackbeard and Joel changing their attitude to positive.**

Now the player can continue talking to Blackbeard. He has become slightly more cooperative. If he is asked about who has been to his house he replies: "Well, that bastard Agi, god bless him, brought me some mead earlier. But no one else." **Blackbeard is more cooperative now, since his attitude towards Ethan has changed to positive.**

When he talks to Agi, Agi appears happy to see him, and smooth talking as always. **Because Agi has an interest in buying the farm, his negative attitude towards Ethan will be overridden.** He meets Ethan happy and forthcoming, he starts of with a sales-pitch:"Mr. Stronginthearm, did you finally decide to sell your farm, damn good offer, the last one! I want to make you a deal that YOU'RE happy with. Because if YOU'RE not happy, I'M not happy. But I KNOW you're going to leave here happy today. How do I know this? Because I'm willing to pay the same price, even though your sheep escaped. You go home and tell your brother that. I'm sure he will be happy too. Making people happy makes me happy." **This sales pitch is scripted, it is several sentences that follow the same style and form one coherent whole. This is not easily achieved without scripting.** After this it's the player's turn to ask questions. Whatever the player asks, Agi will not say anything that can incriminate himself. He will admit to visiting Blackbeard because he knows that there are witnesses (Blackbeard). If the player asks him if he had been to their farm, he will deny it. The player will have the ability to ask Agi about any of the crimes committed: burning the barn (if Blackbeard hasn't said he did it yet), poisoning the well and letting out the sheep. As the answer to the last accusation, Agi replies: "Look, I've never been caught doing anything illegal my whole life. Listen, nobody saw me do it, so it can't very well be me, right? When you've got no proof you've got no case. And anyway, do you really think I would do something that stupid with somebody watching me? You must be joking! If you said you found my scarf you might have a case, but I wore my raincoat for pete's sake! I mean. . . I didn't! I mean, I would have. You know, it's messy on a sheep farm, not that I did go, but I would have worn it. . . I didn't do it, nobody saw me do it, you can't prove anything" Agi stops. He turns this way and that, and then says: "Are you going to hurt me now? You are, aren't you? Please, I can't stand violence!" He turns and runs away. **This sequence is also scripted, much like at cut-scene in other games.**

Now that we have seen what parts of this game scenario we can deal with using our different components, we are ready to conclude on our goals.

Chapter 14

Conclusion

With the networking abilities of the Odin library, we fulfil the goal of making the game architecture multi-player enabled. We do not know exactly how many players we can handle using this library. However, since it is not supposed to be massively muti-player, we can certainly handle enough concurrent connections.

Using the quad-tree we can support a very large number of objects in an expansive world on a normal workstation pc. Our world can be much larger than single-player RPGs but smaller than massively multi-player worlds, which is ideal for our purpose (i.e. a multi-player RPG).

Our goal of making entertaining agents have been addressed in several separate components. Each of these components makes up a part of the complete agent architecture.

The agent-vector has a number of attributes that gives the agent a value for parts of its personality that has influence on the way it communicates. The comparison of the agent-vector and the sentence-vectors, allows the agent to select sentences in accordance with the personality the game-designer has defined. Thus, the dialogue regarding the same topic, will be different when a player talks to different agents.

The attitude-vector has the attributes: desire, obligation and ability, that define the parts of its personality that deals with the selection of its course of action. For each action it has corresponding attributes. Using the vector, and the attributes on the actions, the agent can select between plans. When there are more than one plan that satisfies a goal, the agent can compare each plan, and select the one that fits its personality.

We have a dialogue representation that gives the agent more freedom of expression than agents in games traditionally have. It allows the agent to express the beliefs in its knowledge-base, and also allows it to have secrets and tell lies.

The main objective of games, are to be entertaining, and as such, the agents in the game should also be entertaining. We believe that we have succeeded in making an agent architecture, that makes it possible to create agents for RPGs that are more entertaining than those found in current games. We say this with confidence, because our architecture allows our agents to do everything that is done in those games by using scripting, but in addition to this we allow for non-scripted behaviour.

Bibliography

[1] Erik Bethke. *Game Development and Production*. Wordware publishing, 2003.

[2] Dolores Cañamero. A hormonal model of emotions for behavior control, 1997.

[3] David Freeman. *Creating Emotion in Games: The Craft and Art of Emotioneering*. New Riders, 2003.

[4] Mads Granding and Rasmus Toftdahl Olesen. Story-telling based agents. Master's thesis, Aalborg Univerity, 2003. Mid-way report.

[5] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, 2000.

[6] Aard-Jan Kesteren, Rieks op den Akker, Mannes Poel, and Anton Nijholt. Simulation of emotions of agents in virtual environments using neural networks. In *Learning to Behave: Internalising Knowledge.*, volume 18 of *Proceedings of the Twente Workshop on Language Technology*.

[7] Van Parunak. Case grammar: A linguistic tool for engineering agent-based systems.

[8] Steve Rabin. *AI Game Programming Wisdom*. Charles River Media, Inc., 2002.

[9] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2. edition, 2002.

[10] Bjarne Stroustrup. *The C++ Programming Language*. Adisson Wesley Longman, third edition, 1997.

[11] Takenobu Tokunaga and Makoto Iwayama. Text categorization based on weighted inverse document frequency. Technical report, Tokyo Institute of Technology, 1994.

[12] Eric W. Weisstein. Interpolation. In *Mathworld*. Wolfram web resource, 2004.