

1 Project Report

Goal Based Agents in a Competitive Environment

A 3rd year project by:

Luca F. Beltrami

Supervised by:

Nathan Griffiths

1.1 Abstract

The project presents an implementation of the Spreading Activation model of AI by Pattie Maes. The project includes a world framework where the agents can live, an implementation of the model that attempts to overcome the limitations of the original paper and an agent that demonstrates the capabilities of the implementation.

Keywords

Artificial Intelligence, Agent, Spreading Activation, Simulation, OpenGL

1.2 TOC

1 Project Report.....	1
1.1 Abstract.....	1
1.2 TOC.....	2
1.3 Introduction.....	3
1.3.1 Objectives.....	3
1.3.2 Academic Aims.....	3
1.4 AI theory.....	4
1.4.1 Spreading Activation Model.....	4
1.4.2 Model Limitations.....	5
1.4.3 Extensions to the model.....	7
1.4.4 Other approaches.....	8
1.4.5 Spreading Activation model vs other AI models.....	10
1.5 Project organisation.....	11
1.5.1 Time tabling.....	11
1.5.2 Why C++.....	11
1.5.3 Development methodology and tools.....	12
1.6 World framework.....	13
1.6.1 General design considerations.....	13
1.6.2 World.....	16
1.6.3 Entities.....	17
1.6.4 AI interface.....	18
1.6.5 Combat.....	19
1.6.6 Output.....	20
1.7 AI implementation.....	21
1.7.1 General design consideration.....	21
1.7.2 Graph vs set.....	22
1.7.3 The StateSymbol class.....	25
1.7.4 The GoalSymbol class.....	25
1.7.5 The CompetenceTemplate class.....	26
1.7.6 The selector class.....	27
1.7.7 The Agent class.....	27
1.8 The Actor class.....	30
1.8.1 Expected behaviour.....	30
1.8.2 Spreading Activation network.....	30
1.9 Evaluation.....	31
1.9.1 Correctness.....	31
1.9.2 Performance.....	31
1.9.3 "Smartness".....	31
1.9.4 Ease of use.....	32
1.10 Further Work.....	33
1.10.1 World Framework.....	33
1.10.2 AI.....	34
1.10.3 Actor class.....	34
1.11 Conclusion.....	35
1.11.1 Usefulness of the model in practice.....	35
1.11.2 Project success evaluation.....	35
1.12 Acknowledgements and thanks.....	35
2 Appendices.....	36
2.1 Appendix A: The model file format.....	36
2.2 Appendix B: File list.....	37
2.3 Appendix C: Running the demos and building the project.....	38
3 Bibliography.....	39

1.3 Introduction

1.3.1 Objectives

The objective of this project is to provide an implementation of Pattie Maes Spreading Activation Networks as detailed in “How to do the right thing” [MAES89].

The project will provide:

- A number of classes to that implement the basic functionality of the agents, such as building the agent network and spreading energy though it. The classes will be extendible to allow simple implementation of agents in a particular domain.
- A world framework where agents can exist and act and a output system to show the state of the world in real time. The world framework should be able to support a few tens of agents at a time.
- Some sample agents that demonstrate the capabilities of the agent implementation.

For flavour reasons the world framework is taken to be a “fantasy” world where agents are heroes fighting monsters and hunting treasures for profit and glory. This “setting” has very little bearing on the actual project apart from justifying/requiring a combat system.

1.3.2 Academic Aims

The basic description of the Spreading Activation Model [MAES89] makes assumptions about the environment the agents will act in such as agent actions being atomic and the agent already knowing most of the world (for further discussion see the section on model limitations). This project tries to adapt the model to an environment where most of these assumptions cannot be made.

Another challenging aspect of the project is the design of the world framework so that the agent design is not tied to it and both are easily extendible. Since the framework will make no use of concurrent threads or processes it will need to provide apparent concurrency for the agents.

1.4 AI theory

1.4.1 Spreading Activation Model

The Spreading Activation model was first presented by Pattie Maes in 1989 in her paper “How to do the right thing”. The model is an attempt at bridging the gap between traditional planning and reactive agents by taking a novel approach to the whole problem.

Maes' agents are constructed as “a society of interacting, mindless agents” [MAES89](p3), in other words they are built out of simple modular blocks each of which is capable of carrying out one “action”. The appeal of this design comes from the distributedness, modularity, emergent global functionality and robustness of the system.

The novelty is not in the way the agent is structured but rather in the way the agent is controlled. More specifically Maes rejects the idea that bureaucratic modules whose only use is to decide which other modules should be used are necessary to such a system.

In Maes' approach the “action” modules themselves “decide” which other modules should be activated and which should be inhibited, using a simple but powerful algorithm that features some global parameters used to tweak its behaviour.

In Maes' own words “the algorithm completely integrates characteristics of both [symbolic and connectionist] approaches by using a connectionist computational model on a symbolic, structured representation” [MAES89](p5).

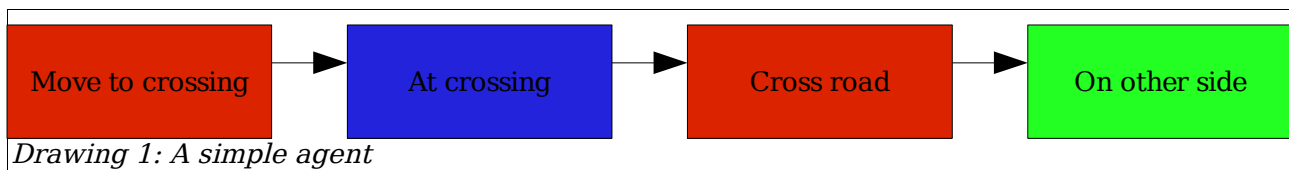
The abstraction the model uses to decide which module to activate is to give each module a certain amount of “activation” energy and the module with the highest amount of energy is picked.

The modules (henceforth known as competences) are defined by a quadruple (c, a, d, α) where c (the prerequisite list) is the set of propositions that have to be true for the module to be executable, a (the add list) is the set propositions that will be true after the module has finished executing and d (the delete list) is the set of propositions that will be false after the module has finished executing and α is the current energy level of the competence. Of course this is the definition of a competence from the Spreading Activation algorithm point of view, every competence must also define how it actually carry out its job upon activation but this is irrelevant as far the algorithm is concerned.

Goals are just propositions. All the propositions that are used by the system (as pre or post conditions or goals) will henceforth be knowns as symbols.

The system can be seen as a network where competences and symbols are nodes. The add, delete and prerequisite lists define the edges.

Energy is injected in the network by state and goal symbols. True state symbols inject energy in all the competences that list them as prerequisites, goal symbols inject energy in all the competences that have them in the add list and remove energy from all the modules that have them in the delete list.



Energy is then spread between competences. A competence that is not executable (not all its prerequisite list is true) will spread energy to the competences that might make their prerequisites true and executable competences will spread energy to competences that depend on their add list to become active since its likely they will be active in the near future. Competences will also remove energy from other competences that might remove symbols in their prerequisite list.

Formal definitions of how energy is to be spread are given in the paper [MAES89](section 3)

1.4.2 Model Limitations

1.4.2.1 Atomic competences

The model does not processing while a competence is active, in fact it considers activating a competence an atomic step. This means that changes to the environment that occur while the competence is executing which might not be atomic and take considerable time, such as a competence that requires extensive movement, are ignored until the whole action has been carried out. This might just lead to suboptimal behaviour in some situations (such as ignoring a precious treasure that is revealed while moving past it) and to downright irrational behaviour in others (such as continuing a combat that has become unwinnable), unless competences are more than “mindless”.

The approach taken in this project does not fully solve this problem, but the environment is constantly evaluated, even while a competence is executing so that nothing will be missed, even if it not instantly acted upon and some intelligence is embedded in the competences to ensure that irrational courses of action are abandoned (see section 1.4.3.1)

1.4.2.2 Single exit competences

There is an implicit assumption in the model that competences will either succeed and their add and delete lists will be applied to the current state or it will fail and the state will remain the same, although what the exact semantics of the failure are is unclear since the state of the network would not be the same (the failed competence has 0 energy).

Still there is no provision for competences that might result in different outcomes depending on events beyond the agent's control.

There are 2 possible approaches to solving this problem:

- The competence provides add and delete lists only for one outcome. The symbols that represent the other outcomes are not connected to the competence. This is not a satisfactory solution because the energy spread semantics are broken.
- The agent provides all the possible outcome symbols in its add and delete lists, but doesn't guarantee activating any. While this solution still breaks the energy spread semantics slightly it does mesh better with the rest of the design.

1.4.2.3 Static network

As mentioned in section 6 of the paper [MAES89] the entire network must be instantiated before the agent can start using it, otherwise, if nodes were added at runtime, energy levels in the network would be uneven and would lead to suboptimal decisions. Moreover no variables can be used in the definition of competences or symbols. To avoid these limitations symbols and competences use indexical-functional aspects to define symbols, competences and goals. This way every competence and symbol has to do some very simple searches (possibly greedy algorithms are enough for these) on the environment (see section 1.4.3.3 for how it was implemented in the project).

1.4.2.4 Network structure affects decisions

Given the way energy is spread the way the network is structured heavily influences the decisions that are taken, for example a module with more prerequisite symbols will usually gain more energy than one with less. Moreover a competence that leads to multiple goals will gather more energy than one that leads to only one way of achieving the same goal.

This might be considered a feature rather than a limitation but it does mean that network design requires careful consideration.

This project makes no attempt at solving this limitation because it is intrinsic in the energy spread algorithm and changing that would probably lead to different semantics for the entire system.

1.4.2.5 Limited goal semantics

One of the limitations that was felt the most during the initial design phase was the limited goal semantics: there is no way for an agent to prefer a certain goal to another and theoretically all the goals are achievable. With this it is meant that there is a sequence of competences that leads to achieving the goal, and once the goal is achieved it can be ignored (apart from making sure it isn't undone). What is not considered are unachievable goals, goals that the agent strives to but never completes, such as “collect as much money as possible”.

Turns out that with the non deterministic add and delete list semantics one such goal can be defined as a symbol that is never true. This way the agent will carry out all the competences that lead to satisfying the goal without ever succeeding and thus repeating the same actions (which, being defined by indexing-functional aspects can in fact be quite different actions).

Solving the priority issue required a slight modification of the energy spread semantics. The energy spread from goals is multiplied by a priority factor so that goals with a higher priority will spread more energy.

1.4.2.6 Lack of memory

The final major limitation of the algorithm presented in the paper is that the agents lack memory of their earlier actions and failures. This can lead to an agent repeatedly attempting the same course of actions over and over without making progress. No attempt was made at solving this problem in itself because it was felt that careful network design could reduce the risk of such an eventuality to negligible levels.

1.4.3 Extensions to the model

To solve some of the problems mentioned earlier some extensions had to be made to the model.

1.4.3.1 Intelligence in competences

In the paper competences are defined as “mindless”. On the other hand while a competence is executing the network is not being updated. This leads to problems when defining some competences, for example a competence that makes the agent explore the environment would be hard to implement as mindless. If the state of the exploration was to be saved as symbols in the

network it would be at best impractical and at worst useless. Moreover the competence might have to do a full exploration of the environment before completing which would definitely be suboptimal. On the other hand if the state of the exploration is saved within the competence itself the exploration could be interrupted at any point without problems.

A similar argument can be made for the implementation of a combat competence, the competence itself should decide when to fight and when to break off from a combat that has become too dangerous.

1.4.3.2 The selector pattern

To implement competences and symbols that are defined using indexing-functional aspects of the world a limited amount of search must be implemented in each to identify the feature of the world that best suits that aspect. Often this search is repeated in many different symbols and competences, for example an agent that crosses a road might have the following competences:

- Go to nearest crossing
 - PRECONDITIONS: none
 - POSTCONDITIONS: at nearest crossing
- Cross Road
 - PRECONDITIONS: at nearest crossing
 - POSTCONDITIONS: on other side of road

Here “Go to nearest crossing” and “at nearest crossing” both need a way to identify a specific crossing out of the many crossings in the world, furthermore they have to agree.

This is what sparked the idea of Selectors. A Selector is a parameter that is given to symbols and competences and acts as an index into the entities of the world. The project uses 3 selectors: one selects the most precious entity known to the agent and is used by the competences that achieve the goal of collecting money, another selects the entity that provides the most glory and is used by the competences that achieve the goal of making a profit, the last selects an area to explore.

Since the selector is shared between all the symbols and competences that require that particular index there is no possibility of disagreement about which entity suits that index best, moreover the search is encapsulated thus reducing the effort of implementing the system.

1.4.4 Other approaches

Many other approaches have been taken in the development of agent architectures. One of the simplest is to use a state machine. Under this approach states usually represent actions taken by the

agent and transitions are external or internal events to which the agent responds. The appeal of this approach lies in the simplicity of implementation and the intuitiveness of the AI behaviour. It is trivial, by simply examining the state machine, to say how the agent will act, furthermore it is hard to beat the performance of these agents since they perform. On the other hand this approach is rather limited in the fact that it is incapable of reacting to events that are not programmed into it and will always react the same way to each of those events. In a sense a state machine based agent is the quintessential reactive agent, incapable of “consciously” working toward a goal, all the planning has to be done by the implementer, the agent can only enact one of the pre built plans that was chosen, by the implementer, during the agent implementation. Despite all these limitations surprisingly good results can be obtained in fairly controlled environments [PIRANJAN98]

Another approach to the design of agents is to describe the world as a number of possible states with the actions available to the agent being the edges connecting them. The agent's view of the future then takes on the shape of a tree (or a graph), from each node the possible actions lead to different possible future states of the world. Events beyond the control of the agent can also be represented by introducing “chance nodes”, nodes in the tree whose outgoing edges represent the possible outcome of the external events (for example the result of a die throw). Chance nodes can also represent the actions of an opponent in a competitive environment. Agents that represent the world in this way usually find a complete solution to the problem before beginning execution, using algorithms such as A* and minmax (depending whether the search involves search nodes or not). The main downside of these algorithms is the inability to deal with incomplete information about the world and, in the case of A* external changes to the world state. Minmax can deal with changes to the world state but does so in the rather brute force way of recomputing the optimal solution after every action-chance event pair. Still many optimisations have been developed over the years and variations on minmax are at the base of many of the most sophisticated agents such as the chess master Deep Blue. Another issue with these algorithms is that they require a heuristic to judge how “good” each possible future state is compared to the others. Producing a good heuristic is definitely a non trivial problem. Finally the range of events that the agent can “understand” is limited by the design of the nodes (both chance and choice). [RUSSEL03](chapter 5, 6)

Variations on this model use, instead of a single tree of possible states, a set of trees each representing a possible world given the current limited information available to the agent. Selection algorithms used on these agents often use “plans”, prebuilt sequences of actions that are known to, given certain preconditions, to lead to certain results. The advantage of using plans is two fold,

firstly a plan is a higher level construct than the actions the agent can take allowing the implementers to ignore some of the details when examining or designing the agent behaviour, secondly being a sequence of actions with known postconditions a plan can drastically reduce the amount of search the agent requires to achieve its goals thus improving performance. One example of this architecture is discussed by [RAO91] and [RAO95]

A totally different approach is taken by planning agents. A planning agent uses logical propositions to represent the state of the world. Actions the agent can take and its goals are described by more logical propositions. The agent will then use inference rules to derive a sequence of actions that will make the goals true given the current state. In some variations pre built sequences of actions might be used to speed up the search. The attractiveness of this type of agents lies in the reusability of the inference engine. All the application specific parts of the agent (state description, goals and action) are in the inference engine language and can be changed without modifications to it. Another very appealing feature of planning agents is the possibility for completely unexpected behaviours to “emerge” from the inference engine. On the other hand the state of the agents and its future evolution are very hard for a human to understand by simply examining the agents knowledge base. [RUSSEL03](section 4)

This brief description does not cover all the types of agents developed many of which take some aspects of each of the above and combine them in one or more layer of decision making.

1.4.5 Spreading Activation model vs other AI models

The Spreading Activation model as mentioned before sits between planning and reactive agents and between classical (in 1989) programmed AI and connectionist approaches. The model takes some of the advantages of each type without necessarily suffering from the same weaknesses.

From planning agents it gets the ability to decide on a course of actions and carry it through to achieve a goal, without requiring all the processing to be done at once and losing the ability to react to a changing environment. On the other hand unlike planning agents there are situations where a Spreading activation agent gets stuck in a loop of ever failing actions.

From reactive agents it gets the ability to react to a changing environment but with very little risk of losing sight of its final goals. Like in a reactive agent selection of the next course of action is quick and efficient.

From classical AI it gets competences which are understandable by humans, represent a specific

capability and whose code (and by implication knowledge) can be shared among multiple agents with ease, at the same time the Spreading Activation models keeps the capability to produce emergent behaviour like a connectivist agent.

On the other hand the Spreading Activation model can't provide some things other models can.

A planning agent might be able to guarantee the ideal solution given a static environment, the Spreading Activation agent will only guarantee a “good enough” solution. And while it can provide some emergent behaviour it won't be on the scale of a fully connectivist agent nor it will be as predictable as a classical one.

The Spreading Activation model could be considered an implementation of the BDI (belief, desire, intent) model. State symbols in the network represent the agents beliefs about the world, the goals represents the agent desires and the energy levels in the competences represent the agent intent. The intent is not represented by the energy levels in a single competence but rather in overall spread of energy over the entire network. A chain of competences with above average energy levels can be thought of as the agent intending to eventually execute those actions.

1.5 Project organisation

1.5.1 Time tabling

Project development proceeded according to the following time line:

- Term 1 – Week 1 – 3: Initial research and specifications
- Term 1 – Week 4 – 5: World framework design
- Term 1 – Week 6 – 10: World framework implementation
- Christmas Break: World framework debug and AI research
- Term 2 – Week 1 – 8: AI design and implementation
- Term 2 – Week 9: Presentation
- Easter Break: Report

During the early weeks of Term 2 several problems with the main development machine (requiring a full reinstall of the system) caused significant delays from the original time table presented in the specifications in Term 1. These delays required the original plans to be changed slightly.

The time left wouldn't allow for a proper implementation of 2 types of agents as planned originally, so the Spreading Activation model was picked as the one to fully implement and evaluate. The reason for the choice was that the Spreading Activation Model was known to work even if many implementation details were unclear. On the other hand the partial planning design was going to be built from scratch using a mix of techniques presented in various papers and there was a major risk of failing to produce a working agent.

1.5.2 Why C++

The choice of using C++ to implement this project is the result of many factors. First and foremost I wanted to demonstrate my ability to use the language. While Java is used in many companies C and C++ are still the workhorses of the industry.

Secondly many of the features of the language are appealing from a software design point of view: overloaded operators and templates for example are two features that are not used much in the project but in the few places where they are used they are invaluable. Another feature of C++ that was greatly appreciated was the lack of a garbage collector. It might sound strange a garbage

collector removes deterministic destruction of objects which is a key point of many idioms used in the project. Plus using smart pointers ensures that objects with complex lifetimes and ownership are destroyed correctly.

A third reason for choosing C++ was the abundance of libraries that could be used. Java has an excellent library but it's harder to find 3rd party libraries to cover what the official library doesn't cover.

1.5.3 Development methodology and tools

Development didn't follow any specific methodology. About half of the design work was done before begging implementation loosely following the waterfall model, in this phase most of the high level interfaces were set. Once implementation begun the original design was heavily refractored to deal with unforeseen circumstances and often with details that should have been considered but were missed due to inexperience.

Tight implementation debugging cycles were run to check all sub modules before checking a full modules. This much reduced the number of “wild bug hunts” across most the code base but sadly didn't remove them all. Again many of these were due to inexperience in detecting hints of where a bug actually lay and concentrating the search on other sections of the code.

Most of the coding was done using various combinations of Kate and Kdevelop, whose excellent syntax highlighting was very much appreciated even though code completion was mostly useless.

gcc was used to compile the code and apart from one strange bug with const iterators was perfectly suited to the task.

Finally gdb and Valgrind were used for debugging. While gdb is an excellent debugger Vlagrind was extremely useful in hunting segmentation faults that could occur at any point of the main loop.

Cvs was used to provide access to the code from any machine and also as a backup mechanisms.

Late in the project svn was used in place of cvs due to a better handling of binary files so that the report could be shared efficiently too.

1.6 World framework

1.6.1 General design considerations

1.6.1.1 World framework competences

The world framework is responsible for running the virtual world the agents live in. Where the world ends and where the agent begins is not something that is obvious. For example the agent might include the knowledge of how to move itself in the world and how to perceive other entities or it might just be limited to making decisions and giving orders to an entity that's part of the world.

I decided to follow the latter route because it allows for a cleaner interface between world and AI.

Thus the world framework needs to represent the entities that exist in the world, know where they are, be able to say what they perceive (if they have senses) and enforce the rules of physics.

Everything that exists in the world is represented by an entity. To allow more complex entities (such as those representing an agent) to maintain the necessary state and implement the needed operations without forcing simpler entities to do so as well (and avoid the performance costs) an “extension” mechanism was defined. In short a basic entity would only store its position and orientation in the world. A more complex entity would also possess extensions that allowed it to carry out other functions such as being rendered, participating in combat, interface with an agent, etc.

Entities are stored in a tree structure sorted by position, so the world framework can quickly retrieve entities based on their location. Entities are also uniquely identified by an ID string and can be retrieved using that too.

Perceptions are dealt with by one or more Perception Managers, each encapsulating a sense.

Perception Managers are not responsible for generating percepts, each entity must generate the percepts about itself (after all only the entity itself knows how it looks and sounds), their only task is to deliver the percepts from the generating entity to all the entities that are perceiving it.

The final part of the framework is the path finding logic. The reasons why path finding is included in the world framework and not in the AI are fairly simple: firstly there's already a very good and well known path finding algorithm, A*, and the way it works does not lend itself to a tighter integration with Maes' Spreading Activation agents; secondly moving path finding into the AI would require either much data duplication or exposing the inner workings of the world framework

to the agents.

1.6.1.2 Non quantized world

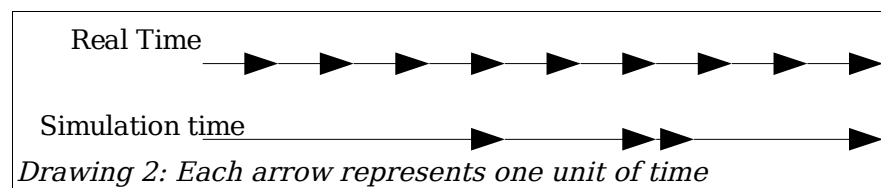
One of the objectives of the project was to make both space and time as continuous as possible within the simulation. Thus all positions are stored as floating point numbers and can assume any value (within the limitations of the hardware). The only part of the world framework that is quantised is the path finding logic since A* works best (must work?) on a quantized world.

This decision proved to be a double edged blade: on the one hand removing quantisation causes all manner of problems, especially on the timing side (see the next section); on the other hand it does also remove a lot of annoying artefacts caused by quantisation, such as things moving faster when moving diagonally, or being unable to do so.

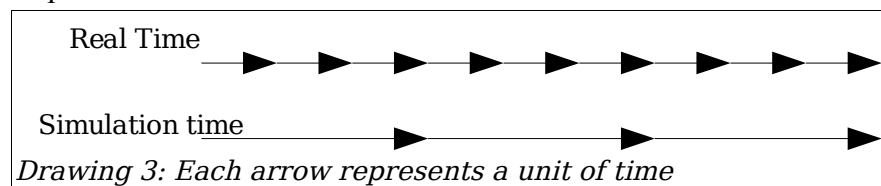
1.6.1.3 Timing

There are 2 main ways to make time flow in a simulation:

- 1) simulation time flows independently of real time, sometimes faster, sometimes slower.



- 2) Simulation time flows in sync with real time, maybe faster or slower but always with the same ratio between speeds.



The choice of which method to use was heavily influenced by the desire to play the output of the project in real time and as it was being computed “on the fly”.

The first method is the more straight forward to implement (compute current state, increment simulation time, compute current state, ...) but, to achieve the desired result, requires the simulation to guarantee a certain rate of state computation. At this stage of the project I had no idea how long the state computations would take and how much the time taken would vary.

The latter method is much more better suited to producing real time output on the fly but has some interesting implementation issues.

In the first method each iteration of the simulation loop knows exactly how long it will last in simulation time, in a way it can look a bit in the future of the simulation and say “I will end at this point in time”. By contrast the second method cannot do this. Since the time take to compute the new state is unknown, the ending time of the iteration in real time is unknown (this is true for both methods) and simulation time is anchored to real time so the ending time of the iteration is unknown, we've lost the ability to “peek” into the future.

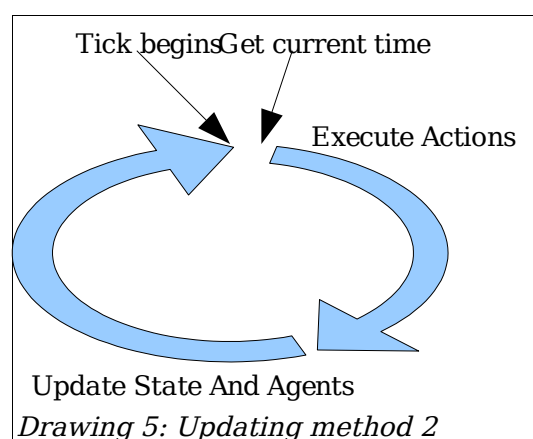
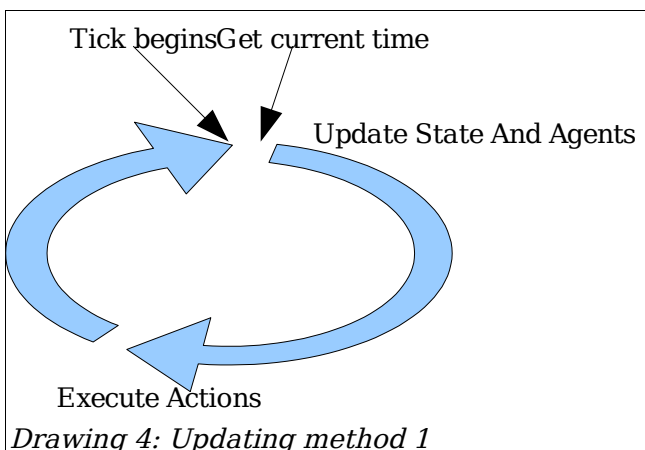
The final result is that under the first method each iteration can compute the new state in the straight forward manner:

- 1) get current simulation time // easy, a fixed increment on the previous time
- 2) deliver perceptions to each entity
- 3) let the agents decide what to do
- 4) execute actions // we know exactly how much time entities have to execute their actions before the next iterations so it is easy

On the other hand the second method needs to do something like this:

- 1) get current simulation time //need to get new system time
- 2) execute actions //note that this is the step 4 from the previous iteration under the first method. It has to be done now otherwise it is unknown how long entities had to execute their actions
- 3) deliver perceptions to each entity
- 4) let the agents decide what to do

This sequence is counter intuitive, at first sight it looks like entities act before deciding what to do, in reality the 2 methods are exactly equivalent with just a swap between the reading of the current time and executing of actions.



The other important issue to consider was how to handle race conditions between entities trying to interact with other entities. For example if 2 entities were trying to pick up the same entity at the

same time which one should succeed?

Since there is no true concurrency in the world framework there is no risk of 2 entities modifying another entity at the same time, the only issue is to guarantee a fair environment to the agents, without some entities always getting priority over other entities.

The simplest way to approach the problem is not to do anything counting on the non quantised world to provide a “random” ordering of the entities attempting the same action. This, coupled with very short iterations, would make race conditions a rare occurrence and a negligible problem.

A slightly more complex solution is to randomise the order in which entities update their state . This solution can cope fairly with frequent race conditions.

The most complex and most flexible solution is to implement a scheme where all actions for a given iteration are first “declared” by the entities, then potential conflicts are resolved (possibly using different algorithms for each kind of conflict) and finally all actions are carried out. This system is not only fair but allows to customise the way each conflict is resolved.

In the end the first method of “not really doing anything about it” was chosen. The reason was that at that point there was no evidence the system would need a more complex solution, which could always be implemented later if necessary, and it would free time to move on the more important parts of the project.

1.6.2 World

1.6.2.1 Entity storage

Since different parts of the world framework would need to access entities in different ways based on different information so different data structures are used to store entities.

Much of the world framework functionality needs to access entities based on their position in space. The data structure most suited to for this task was found to be a variation on the octree theme. An octree is a tree structure where the world (generally a cube) is recursively partitioned in 8 parts each becoming the root of a sub tree. The main downside to a basic octree is that it has 8^n nodes at each level of subdivision. One solution to this problem is to, instead of subdividing to a fixed depth, subdivide until there is only a certain number of entities in each node. This solution works well and is only moderately complex but for the world framework there was a better solution (in my opinion).

Generally octree are used to store spaces that are roughly the same size along all 3 axis while the worlds represented in this project where going to be mostly 2 dimensional; and equal number of subdivisions on each axis would lead to nodes that were extremely thin in one dimension while still too large in the other 2.

The tree implemented in the project can subdivide each node in 1, 2 or 3 dimensions depending on the shape of the node. The implementation is fairly trivial and gives less nodes than a standard octree.

Using a quadtree (a structure similar to an octree that subdivides a plane in 4) could have been another solution but would have affected performance in case more 3 dimensional worlds were used, without much simplification of the code.

Some other parts of the system might require to identify a specific entity, without knowing its location. To provide this data efficiently all entities are inserted in map that associates a string (the entity's id) each entity allowing fast retrieval of entities by name.

1.6.2.2 Path finding

For path finding the straightforward solution of using the A* algorithm was chosen. A separate data structure is used for the pathing computations since A* requires a quantised world to work in. The structure is a simple 3 dimensional array of "pathing cells". Each cell stores 2 bits of information: the coordinates of its centre, used for all the cost calculations and as the point entities move through when going through that cell, and whether entities can move through the cell or not. To reduce the task of keeping the pathing map up to date only some entities are considered to block pathing depending on a value attached to every entity, whenever one of the path blocking entities is move the pathing map is rebuilt from scratch. In the final implementation only entities representing walls block pathing. Another reason why most entities do not block pathing is to completely avoid collision issues between moving entities.

1.6.2.3 Perceptions

Perceptions are handled principally by Perception Managers. Perception Managers are modelled after the Observer pattern: entities subscribe to a Perception Manager for each sense they have (only sight is implemented currently) and the manager provides them with all the Percepts they need for that sense. The Perception Manager does not generate the Percepts on its own, every entity provides it with the percepts associated with it. The reasoning behind this design is as follows:

Perception should be a passive activity for entities, at any one time entities should have the most updated percepts available (in fact this not true in the current implementation but that is intentional to hide the fact that entities are updated sequentially).

By encapsulating the mechanisms of perceiving in Perception Managers it is very easy to modify what senses an entity has access to at runtime. Moreover the manager could carry out optimisations on the way it picks up and delivers perceptions over multiple entities. Neither of these points is used in the current implementation.

Finally a perception manager has no way of knowing exactly how to build percepts about entities so each entity provides it with all the percept it should generate.

Percepts are discussed later in the AI interface section.

1.6.3 Entities

An entity is anything that exists in the world, whether it is movable or static, visible or invisible. All entities have a name, a position and an orientation. The name of the entity is unique and is used to reference the entity directly.

Some entities need more functionality, which is provided by extensions. The extensions implemented in the project are:

- Combat: the entity is capable of participating in combat
- Item: the entity is an item that can be picked up, carried around and dropped
- Handler: the entity is capable of manipulating items
- Movable: the entity can move or be moved around
- Renderable: the entity can be rendered
- Sensitive: the entity has senses and can perceive things
- Smart: the entity supports an agent
- Tangible: the entity can be perceived (and thus has to provide percepts to PerceptionManagers)

Every entity has a number of methods of the form `getExtX()` where `X` is the name of the extension. These methods return an appropriate extension object that implements the interface of the extension, for example the `Renderable` extension implements the `render()` method used to display an

entity. If an entity does not support a given extension the corresponding get method will return null. This property is used extensively to test if a given entity support certain extensions.

The extension classes in general only provide basic implementations of their interface (sometimes they only declare their interface) and keep only the state that is strictly part of the extensions. This is because they are not meant to be used directly but as parent classes of the actual extension objects used in the system. There are 2 ways they could be extended:

The first is to make “runtime” extensions. From each extension base class a full implementation of the extension is derived. Every entity contains all the extension objects it needs and puts them in contact when they need to (for example the Smart extension needs the percepts from the Sensitive extension).

The second option, “compile time” extensions, is that every entity inherits from both the base entity and the extensions it wishes to support and provides a full implementation of all of them. This way is less elegant than the first but is simpler since extensions can just share the variable that are needed by more than one. The project uses this method since only a small number of entity types where needed.

1.6.4 AI interface

1.6.4.1 Percepts

Percepts are object that describe a single item of sensory input. In the current implementation every entity provides 1 percept per iteration (theoretically it would be 1 per sense but only sight is implemented). Every percept is identified by an id and every percept from the same entity for the same sense uses the same id making it easy for the perceivers to determine if they are sensing the same thing.

Percepts also carry information about what sense they represent and at what time they begun and at what time they ended (the time information is not well implemented in the project but it wasn't of much use).

Percepts in theory should store all the information about the generating entity that can be accessed by using a given sense. In the current implementation they only store a reference to the parent entity and the perceivers use that to access further information. This was done mostly to simplify this part of the program even though it does break the Agent / World framework interface

1.6.4.2 Orders

Orders are the other half of the interface between AI and world framework. Orders encapsulate a high level action such as walking or picking up something, hiding the implementation details of the framework and allowing the agents to work with meaningful actions.

The currently implemented orders are:

- **Attack:** makes the entity attack another nearby entity. Both entities must implement the Combat extension.
- **Drop:** makes the entity drop an item it is carrying. The executing entity must implement the Handler extension and the target entity must implement the Item extension. While implemented it is never used and no entity actually keeps track of what it is carrying.
- **Pick Up:** makes the entity pick up a nearby item. The executing entity must implement the Handler extension and the target entity must implement the Item extension.
- **Walk:** makes the entity move in a straight line to a target location. The executing entity must implement the Movable extension.

Some of the orders could have been extended to be even more “high level” than they are: instead of the Attack order there could be a Fight order that handles the entire fight and the Walk order could handle path finding. On the other hand it was felt that doing so would move too many of the decision that should be handled by the AI into the world framework. The current orders are meaningful but atomic actions.

Unlike the percept interface the order interface is never broken. Using the percept interface as first envisioned turned out to be too cumbersome under the current design while the orders worked very well for their intended purpose.

1.6.5 Combat

While the combat system was just a minor part of the project a few words on how it works are necessary to fully understand the workings of the project.

Entities that engage in combat can perform only one action, Attack, implemented in the order Attack. How skilled an entity is at attacking is represented by the attack score. Whenever an entity is attacked it automatically defends itself, how well it can do so is represented by the defence score.

When an entity attacks it generates a random integer between 1 and its attack score, the defending

entity generates a random integer between 1 and its defence score. The ration of the 2 random numbers is the amount of damage the defending entity takes.

The amount of damage an entity can take is represented by its health score. If the health score drops to 0 the entity dies. Every entity regenerates health over time.

1.6.6 Output

The output module of the world framework is a very simple 2D OpenGL based renderer. The design of the interface between output module and the rest of the framework theoretically allows replacing this renderer with another more or less sophisticated one with no change to the code outside the output module itself.

The module works using models each of which is can render itself to the screen. A renderable entity might have one or more models associated with it and decides which to render at any point. The same model can be shared across multiple entities and receives entity dependent information, such as position and orientation, from each entity at the time of rendering.

Entities request models by name from a globally known model factory that creates the model object and returns it to the entity. How the model factory constructs the model object is not the entity concern. In the current implementation there is a very rudimentary file format for describing models.

1.7 AI implementation

1.7.1 General design consideration

The interface between world framework and agent consists in the world framework requesting the agent to “update” itself (take into account the changes to the environment and decide on the future course of actions) every tick. The agent gather information about the world by calling the `getAllPercepts()` method (part of the `ExtensionSmart` interface). This method returns a list of all the percepts the entity supporting the agent experienced during the last tick. After making a decision the agent will “communicate” it to the supporting entity by calling the `setNextOrder()` method (also part of the `ExtensionSmart` interface).

This interface design means that the agent has a number of crucial tasks to perform:

Firstly it must (obviously) be able to decide on a proper course of actions.

Secondly it must be able to remember past percepts so that things not in the immediate sensory range of the entity are not ignored.

Finally it must be able to translate percepts into symbols that are meaningful to the Spreading Activation Network.

All these tasks can only be carried out when the agent is asked to update itself, the agent is “inactive” the rest of the time.

Given the way the Spreading Activation model works the agent may carry out all or only a few of the above tasks on each update. On every update the agent must process the new percepts and update its memory to avoid missing out anything of importance. Making decisions, in this case by running the enough iterations of the energy spreading algorithm for the network to pick a competence, only occurs when no competence is running. As a consequence it is pointless to update symbols and selectors except when the network is about to be run.

The agent must also consider situation when the last order given to an entity has been completed but the competence is not completed yet, in this case the agent should only pass a new order to the entity without running the decision making algorithm again.

The agent memory can be implemented as a set of percepts using the percept id as the key. This gives a fairly efficient check for duplicates. Using a set indexed by a hash table might have been better but it is not a standard part of the STL (C++ standard container library) and would have

required implementing a hash function.

1.7.2 Graph vs set

1.7.2.1 Set pros and cons

In the paper, when presenting the expressions that describe how energy spreads between competences and symbols according to the Spreading Activation model, a number of sets and set operators are used. For example the expression that defines the energy input in a competence due to the active prerequisite state symbols is:

$$inputFromState(x, t) = \sum_j PHI \frac{1}{|M(j)|} \frac{1}{|c_x|} \quad where \quad j \in S(t) \cap c_x$$

Where PHI is the amount of energy injected by the state per true state symbol, $M(j)$ is the set of competences for which j is a prerequisite, c_x is the set of prerequisite symbols of x and $S(t)$ is the set of active symbols at time t .

One approach to implementing the Spreading Activation model would be to directly translate these set expressions into code.

The advantage of doing so is that it should be quite simple to determine whether the implementation is correct or not. Also there should be very little redundancy in the stored data: a set of symbols, a set of competences, each competence holds some references (pointers) to some symbols, all the other information is generated on the fly, so to say. On the other hand many derived sets need to be allocated and deallocated during each processing iteration such as $M(j)$ in the example above, not only that they need to be created from very scattered information.

A quick performance analysis is needed before a comparison with other ways of implementing the model can be made.

The standard c++ library makes these guarantees on the performance of set operations:

- insertion is $O(\log n)$
- deletion is $O(1)$
- find is $O(\log n)$
- intersection is $O(n)$

One very important simplifying assumption is made for this analysis: the size of add, delete and prerequisite lists is fairly similar for all competences and grows linearly with the size of the network. Also it shall be assumed that the number of competences in the network and the number of symbols are in a fairly constant ratio (both are in $O(n)$ to the size of the network).

i will indicate the number of competences in the system, j will indicate the number of symbols in the system. Given the above assumptions anything in $O(i)$ or $O(j)$ is in $O(n)$.

For every update cycle several expressions need to be computed:

`input_from_state(x,t)`

`input_from_goals(x,t)`

`taken_away_by_protected_goals(x,t)`

need to be evaluated for every competence. Each expression will need to allocate a set in $O(j)$ and evaluate a number of sub expressions (the number of sub expression is in $O(j)$) each of which requires the allocation of another set in $O(i)$. Overall the evaluation of these 3 expressions is in $O(n^3)$.

`spread_fw(x,y,t)`

`spread_bw(x,y,t)`

`takes_away(x,y,t)`

need to be computed for every competence pair. `spread_fw(x,y,t)` and `spread_bw(x,y,t)` taken together allocate at least 1 set for every x in $O(j)$. `takes_away(x,y,t)` also allocates at least 2 sets for every evaluation (in $O(n)$ time).

Overall the `spread_*` expressions will have to evaluate a number of sub expressions in $O(i^2*j)$, with each sub expression being of similar complexity to the ones from the `input_*` expressions. The overall complexity is thus $O(n^4)$.

`takes_away(x,y,t)` is again similar, the overall complexity is (n^4)

Thus the complexity of the entire state update is in $O(n^4)$ but with quite large hidden constants due to the many set allocations and repetitions of similar operations. It is also quite likely that the size of the add, delete and prerequisite lists will grow faster than $O(n)$.

1.7.2.2 Graph pros and cons

Another approach to implementing the Spreading Activation model is to use a graph structure. After all the model is supposed to represent a network of competences and symbols.

The graph approach requires some modifications to the way all the expressions are evaluated. For example instead of evaluating `input_from_state(x,t)` for every competence, each active symbol will inject a certain amount of energy in each of the child competences.

Overall this approach probably requires more memory to be stored (depending on how the graph is implemented) but many of the sets that had to be computed on the fly for the set approach exist explicitly in the representation. For example `M(j)` is just the outgoing edges from symbol `j`.

Another advantage to the graph approach is that each of the 6 expressions shown in the paper is broken up in multiple parts each residing in its own method making it easier to follow each one.

Finally the graph method feels more object oriented than the set method.

Now to analyse the complexity of the graph approach

Assuming the graph is built as a directed network of competences and symbols where competences are children of symbols in their prerequisite list and parent of the symbols in their add and delete lists.

Assuming as with the set approach that size of the add, delete and prerequisite lists is proportional to the size of the network.

Each node in the graph has a `spreadEnergy()` function.

For state symbol nodes this function spreads the correct amount of energy to every child node in $O(n)$.

For goal symbols it combines parts of `input_from_goals(x,t)` and `taken_away_by_protected_goals(x,t)` adding energy to competences that might activate the symbol and removing from those that might disable it. Again it is in $O(n)$.

For competences the function is more complex, as it integrates the `spread_*` expressions and `take_away(x,y,z)` which are already the most complex of the expressions and is in $O(n^2)$ complexity.

`spreadEnergy()` is called once per every node in the graph giving an overall complexity of $O(n^3)$.

The hidden multipliers are fairly large, like for the set approach but since there are no memory

allocations involved they should be smaller.

It should also be pointed out that the set based approach could be optimised quite a bit if the expressions in the paper were not followed literally but broken up like they have been in this approach.

Ultimately the graph based approach was not chosen for its better performance but because it was found much easier to code and debug.

1.7.2.3 Graph implementation

The graph objects implemented to support the graph implementation of the model take a fairly naïve approach to the problem. A graph is a list of nodes. A node is a list of edges and holds a reference to a “content” object.

Since the lists are not sorted adding a node to the graph and adding an edge to a node are both in $O(1)$. Adding an edge to the graph is in $O(n)$ where n is the number of nodes. Removing an edge is also in $O(n + m)$ and so is removing a node (m is the average number of edges in a node).

Edges store 3 pieces of data: a pointer to the parent node, a pointer to the child node and a boolean label that indicates whether they are enabling or disabling nodes.

Each node stores both incoming and outgoing edges and a pointer to a “content” object. The “content” is either a Symbol object or a Competence object and provides the `spreadEnergy()` method. The choice of splitting nodes in node and content was to allow debugging of the graph before implementing the Symbol and Competence classes. It could have been done by using inheritance but composition seemed a cleaner solution.

1.7.3 The StateSymbol class

In addition to implementing the `spreadEnergy()` function the StateSymbol class must know when it is enabled or disabled depending on the memory of the agent and its current percepts.

The subclasses of StateSymbol, most of which are parametrised by a Selector, are:

StateSymbolAt: true if the agent is at a specific location.

StateSymbolAtSelect: true if the agent is at the location determined by the selector

StateSymbolDying: true if the agent is below half health

StateSymbolNotSafe: true if the agent can perceive entities that can fight

StateSymbolKnowSelect: true if the agent knows an entity that can be selected by the Selector parameter

StateSymbolDontKnowSelect: the complement of StateSymbolKnowSelect

StateSymbolAtItemSelect: true if the agent is near the entity chosen by the Selector parameter and the entity is an item

StateSymbolAtMonsterSelect: true if the agent is near the entity chosen by the Selector parameter and the entity is capable of combat.

Each of these subclasses implements a custom update() function that is used to determine the truth value of the symbol.

1.7.4 The GoalSymbol class

The GoalSymbol class is very similar to the StateSymbol class. The main difference is the spreadEnergy() method. Like StateSymbol objects each GoalSymbol object must be able to determine its truth value.

The subclasses of GoalSymbol are:

GoalSymbolAny: a generic unattainable goal

GoalSymbolSelect: parametrised by a Selector, this goal is a permanent goal that cannot be achieved. It is also parametrised by a priority value which indicates how important the goal is to the agent. The energy input into the system by the goal is equal to the normal energy input for a goal times the goal priority times the value of the currently selected goal entity.

GoalSymbolAlive: probably badly named, since this goal like GoalSymbolSelect cannot be achieved and yet agents are “alive”. It probably should have been named GoalSymbolHealthy as it comes into play when StateSymbolDying is true (see the Agent class section for a full description of the agent)

All the goals in the project are permanent goals that can never be attained by the agent (they never evaluate true) so that the agent will forever try to accomplish them.

1.7.5 The CompetenceTemplate class

This class is what is usually referred to as the competence class. It is called CompetenceTemplate because instances of the class are the actual competences. In addition to implementing its own

version of `spreadEnergy()` `CompetenceTemplate` must be able to determine whether it is executable or not (by examining its prerequisite set) and be able to actually execute.

Most competences when executing will pass an order to the entity supporting the agent, wait for it to finish executing, pass another order and so on until they have passed all the orders needed to carry out the task the competence represents.

Once all the orders are carried out all the symbols in the competence add list should be active and all the symbols in the competence delete list should be inactive but, since the add list and delete list only indicate the possibility of those symbols activating or deactivating, it might not be so. For example `CompetenceFightSelect` has `StateSymbolDying` in its add list but it won't be necessarily activated after every fight.

Moreover every competence might abort during execution if completing the competence is no longer possible or the best course of action. Again taking `CompetenceFightSelect` as an example the competence might abort if the agent is losing the fight.

The competences implemented in the project are:

`CompetenceExplore`: moves the agent to a random location in the world.

`CompetenceFightSelect`: parametrised by a selector, fights the target picked by it. Will abort the fight if the agent is loosing.

`CompetenceMoveSelect`: parametrised by a selector, moves to the location picked by it.

`CompetencePickUpSelect`: parametrised by a selector, picks up the item selected by it.

1.7.5.1 Competences vs orders

There is no 1 to 1 correspondence between orders and competences, an order is too small for the agent to use directly in the network and since competences actually host parts of the agent they cannot be made into orders.

For example if the Attack order was a competence most of the time there would be no change to the state of the network after it was executed, except that the attack competence would have no more energy making it an unlikely candidate for being picked again. The agent would most likely wander off to do something else. The end result would look like agents where severely suffering from attention deficit disorder. On the other hand making a fight order would move too many of the choices that should be handled by the agent to the world framework.

As it is with the current design each competence can host parts of the agent that would be difficult to describe in the spreading activation network without having to relinquish control to an external party.

1.7.6 The selector class

Selectors implement most of the “second layer of intelligence” in the system. Some of it is part of the competences but selectors implement the most important parts. The a Selector object as it;s name implies selects the best entity or position to fulfil a certain goal. In the current implementation 3 Selectors are implemented: SelectBestGlory, SelectBestGreed and SelectExplore.

SelectExplore ideally would keep track of the areas the agent already explored and make it explore new areas, in fact is selects random locations in the world.

SelectBestGreed and SelectBestGlory select the best entity to fight or pick up for glory or greed. Each considers a various factors when making the choice including the value of the entities and how dangerous they are to fight and produces a weighted value for the selected entity to be used by the goal objects.

1.7.7 The Agent class

1.7.7.1 Responsibilities and operations

The Agent class is technically still part of the world framework and defines the interface between it and agents. Actual agents are implemented in a subclass of Agent. The only agent type implemented in the project SpreadingAgent implements all the remaining functionality needed for a full implementation of the spreading Activation model.

Every agent receives a pointer to the entity that supports it upon creation and implements an update() function that is used to update the state of the agent each iteration.

SpreadingAgent's update() method carries out a large number of task.

Firstly it processes the current percepts to determine if the agent discovered anything new and if that is the case stores it into the agent's memory. Next it also check if anything the agent knows about is no longer there. This second operation is extremely important without it an agent that decided to pick up something was the best course of action would forever try to pick it up after it was gone (the Spreading Activation Network has no memory to determine if the action it is attempting now was attempted earlier and whether it failed or not).

Processing percepts to see if something new has been discovered is trivial, for each percept of interest the agent stores it in its memory. If another percept with the same id was already there it is overwritten.

Updating memory to “forget” things that are no longer there is more complex. For every percept in memory the agent has to check if it is within its sensory range. If it is the agent checks if it also appears in the current percepts, if it doesn't the entity associated with it is no longer there and thus the percept is removed from memory. While not the most sophisticated approach to the problem, this solution does solve the problem quite simply and effectively.

After processing the current percepts the agent checks if an order is in mid execution, if so `update()` ends as the agent waits for the entity to be done executing it. If there are no orders on the entity but a competence is executing, the competence is told to deliver the next order to the entity and `update()` ends.

If there are no orders and no competences executing then the agent needs to pick the next action. First all the selectors are updated so they all select the best target from the agent's memory.

Next all the symbols (state and goal) in the network are updated to they reflect the current state of the agent and the world.

Finally energy is spread around the network until one competence has enough energy to be picked. When one is chosen it is activated and the first order from it is passed to the entity.

The second job of the `SpreadingAgent` class is to provide a nice interface to for creating the network. The class provides a number of methods to insert and remove nodes and create edges:

`insert()` takes either a competence smart pointer, a symbol smart pointer or one of each and a boolean. The first 2 versions of the function insert the competence or symbol into the graph (after creating a node to host it), the latter creates an edge between 2 previously inserted nodes.

`remove()` takes either a competence or a symbol smart pointer and removes it from the graph together with the corresponding node and associated edges.

Note that all these functions work with smart pointers not regular pointers to simplify memory management. Smart pointers are used in many other areas of the project but this is one of the few where they are part of the interface.

The final job of `SpreadingAgent` or of a subclass is to create the network when an object is instantiated. The `SpreadingAgent` constructor creates the network used by the agents in the project, a

subclass of `SpreadingAgent` could implement a completely different network.

The network is created simply by creating all the required nodes and linking them using the above functions, no digging in the internals of the class is needed.

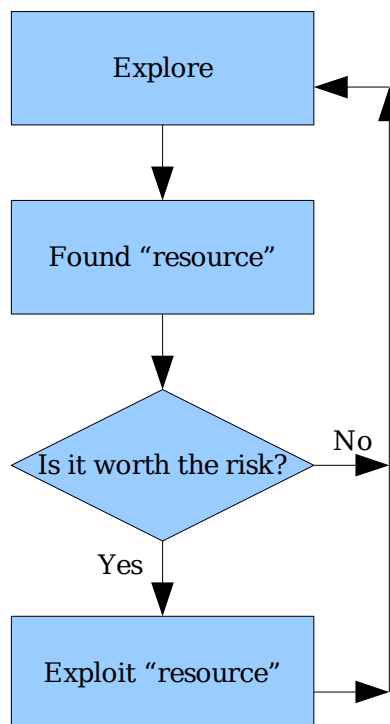
1.8 The Actor class

The actor class is not part of the world framework or of the AI system, rather it uses both to implement a rational agent and the entity supporting it. The actor class is a world framework entity implementing all the extensions with the exception of ExtensionItem. Thus an Actor object can move, engage in combat, interact with other items, perceive and be perceived and support an agent.

The agents that are attached to Actor objects are of the class AgentActor, a subclass of SpreadingAgent that upon construction initialises itself with the Spreading Activation network it uses.

1.8.1 Expected behaviour

The desired behaviour of these agents was to explore the world until they encounter something of interest (a treasure or a monster worth glory or money). At that point the agent should evaluate the risk of attempting to claim the reward from the encounter. In the case of items there is no risk in doing so but in the case of monsters the task might be quite dangerous or outright impossible, of course the more tempting the reward the more risk the agent should be prepared to take. If the reward is deemed worthy of the danger it should be claimed otherwise the agent should resume exploring in the hope of encountering better prey.

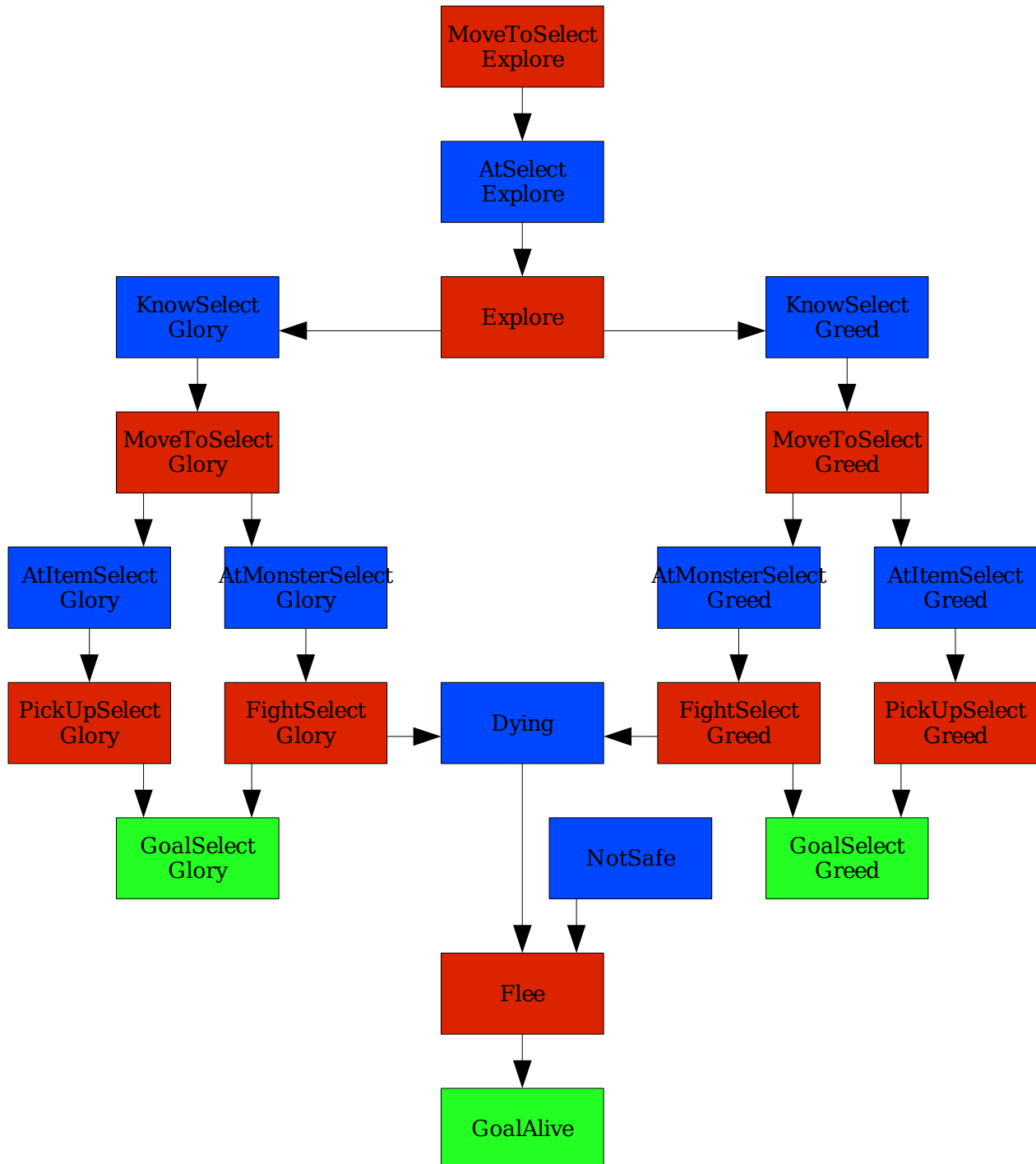


In addition the agent should be parametrise by its lust for gold and glory. Some AgentActor objects

will only desire gold, other only glory and every shade in between should be possible.

1.8.2 Spreading Activation network

The final network design for AgentActor is the following:



The network is split into 2 paths one that achieves the goal of glory and one that achieves the goal of greed. Both paths have exactly the same structure and use the same competences parametrised by a different selector.

1.9 Evaluation

1.9.1 Correctness

To test whether the model was implemented correctly the robot example from Maes' original paper was implemented. For simplicity the agent was implemented with a map containing all the symbols in the world and their value (an integer representing their number of occurrences in $S(t)$). The StateSymbol and GoalSymbol objects were implemented as proxies to the values in the map. Competences upon activation would simply modify the map according to their add and delete lists.

The code for these classes is in the AgentTests.h and EntityTest.h files.

The only problem in implementing the tests was determining the values of the parameters of the network since they are not given. θ , Φ , γ and π were determined by inspection of the output reproduced in the paper. δ was not used in the example and was guessed.

When run the agent executed exactly as on the paper, following the same sequence of actions and taking the same number of iterations to complete its task (see demo_6_robotPainter).

1.9.2 Performance

1.9.2.1 AI

From the performance point of view the project has been a success. Demo 2 (demo_2_massive) is designed to show this. The demo creates 50 agents in the same world as demo 1 and there is no noticeable slowdown (on the development machine: P4 3.2 GHz, 2 GB ram). Probably the main reason for this is that due to the way the agents are designed “thinking” time is quite rare. Most of the time agents will be mindlessly executing the next order which might take several hundred ticks, thus the chance of more than a handful of agents “thinking” at the same time is quite low.

1.9.2.2 World framework

The performance of the world framework on the other hand is quite abysmal. Specifically the path finding algorithm can take a couple of seconds or more to plot a course. This probably due to the large number of allocations that are carried out when computing the path. On the other hand replacing the path finding routines would take very little work, since they are encapsulated in their own object.

1.9.3 “Smartness”

The objective of the actor class was to produce a rational agent that would search the world for glory and or gold and try to collect as much as possible of one, the other or both depending on its priorities. The first demo included with the project (demo_1_general, see Appendix C for details) shows 3 such agents in a world. In the demo the multicoloured squares represent the agents, the red diamonds represent monsters and the yellow diamonds represent piles of treasure. All monsters and treasures are worth both gold and glory and the agents are interested in both.

When running the demo some minor glitches in the AI behaviour can be noticed: sometimes the agent will walk up to a treasure and walk off without first picking it up. This is likely to be due to interference between the glory and greed paths in the network, both might have accumulated significant amounts of energy for different target entities and the agent will oscillate between which to pick. Solving the problem would be simple if a symbol other than a goal could inhibit a competence but under the current semantics and network design there is no obvious solution apart from further tweaks to the AI parameters.

After a while the agents will start moving around randomly without collecting the last few treasures and monsters, this is due to the fact that those entities are not within accessible pathing cells and are therefore ignored.

Demo 3 (demo_3_selector_issues) shows what a bad AI parameters can cause. The 2 agents for ever oscillate between 2 treasures, one with a high gold value and one with a high glory value. In this situation the energy input from the goals is too low compared with the energy input from the state.

Demo 4 (demo_4_priorities) demonstrates (not too clearly) how the agents will prioritise goals. There are 2 agents in the world and eight treasures in two groups of four, the top row of treasures has a high glory value and the bottom row has a high gold value. The agent on the left prefers glory to gold and the agent on the right prefers gold to glory. Each agent will first collect the treasures it prefers before collecting the other ones.

Finally demo 5 (demo_5_combat) shows the rationality of agents when approaching combat. The demo once again features two agents and three monsters. One of the monsters (the one on the left) is worth a fair bit of gold and glory and is not too dangerous, another (the one on the right) is just as dangerous but is worth very little, the third (the one at the top) is worth colossal amounts of glory and gold but is also far too dangerous for an agent to fight. The agents will target the monster on the

left and ignore the monster on the right, there is no reason to take risks for such little reward. After a while one of the agents might attempt attacking the monster at the top because given the massive reward it is worth the risk in combat (in the network term enough energy from the goal will eventually accumulate on the competences leading to a fight with this monster). The agent will then engage in combat, quickly realise the combat is a losing proposition and break off.

Overall the agents can be said to act fairly rationally pursuing their priorities and avoiding dangers that do not lead to sufficient rewards. They also display a human like tendency to, every now and then, take immense risks that could lead to immense rewards.

1.9.4 Ease of use

One other aspect of the agent classes that was evaluated during the project was how easy it is to write an agent behaving in a specific manner. The end result is not the most user friendly but, assuming symbol, competence and selector classes are given, it is quite easy to build a network. For example the road crossing agent discussed in earlier sections could be implemented with the following code:

```
SelectorCrossing selector; //create a selector
SymbolPtr atX = SymbolPtr(new StateSymbolAtSelect(&selector));
SymbolPtr crossed = SymbolPtr(new GoalSymbolOnOtherSide());
CompetencePtr goToX = CompetencePtr(new CompetenceMoveToSelect(entity,
    &selector));

// entity is the entity supporting the agent
CompetencePtr cross = CompetencePtr(new CompetenceCross(entity));
insert(atX);
insert(crossed);
insert(goToX);
insert(cross);
insert(goToX, atX, true);
insert(atX, cross, true);
insert(cross, crossed, true);
insert(cross, atX, false);
```

To follow the creation is initialisation idiom this code should be part of the constructor of a subclass of AgentSpreading, but this is not necessary since all the insert() methods are public, on the other hand the selector must be a member variable of the agent class. To ensure that selectors are updated at the correct time the agent must override the AgentSpreading customUpdate() method to update the selector.

Implementing custom Symbols and Competences should also be fairly easy. Symbols will require the overriding of the `update()` method inherited from `StateSymbol` or `GoalSymbol`. Competences will require a bit more work overriding the `activate()`, `nextOrder()` and `isDone()` methods of the `CompetenceTemplate` class.

Overall this should be fairly easy, much easier than designing the Spreading Activation network correctly. At first sight the network design seems easy enough once the competences available to the agent have been decided but deciding on the appropriate symbols to use as prerequisites and to link competences can be tricky and trickiest of all is finding the right values for the parameters controlling the network behaviour.

1.10 Further Work

1.10.1 World Framework

The current implementation of the world framework should provide a good platform for much further development. Extensions that require a truly 3 dimensional world will only require a new output module, the rest of the world framework makes no assumptions of 2 dimensionality.

A very simple input system is provided (as a very thin layer over SDL) and can be extended to provide all the controls that might be needed without changes to the core system.

The main extensions the framework itself needs are not to the core but to those parts that are built to be extensible.

First it would be nice to add more Sensor Managers particularly hearing and to modify the Sight Sensor Manager so that it takes into account entities that block line of sight.

Secondly more types of entities and orders should be provided to allow more variety in what the agents can do.

On the other hand if this project was to be truly used for research or other applications (as the basic platform for a video game for example) I would strongly suggest a full rewrite of the core parts to simplify the implementation of some more exotic functionality.

During the implementation of the project several flaws and inelegant constructs became clear.

The first major change would be to move entities away from using inheritance to implement extensions to composition. Using composition has many advantages:

First it encourages to provide the full logic of each extension within the extension class itself instead of having it mixed in the entity, this should simplify debugging extensions.

Secondly it is possible for entities to add or remove extensions at runtime and it's much easier to integrate new extensions in an older entity.

Given the way the extensions interface is designed this move should be fairly simple but is hindered by one of the most inelegant constructs of in the World Framework: the update() method on entities. In the current design every entity implements an update() methods that knows how to update its state. Some entities do not require to be updated (for example static walls), others require updating their state only under very special circumstances yet, at the moment, every tick the system must call

update() on them. Moreover most of the state update logic is not in the update() method but in the Order classes, which were designed for this purpose. Annoyingly though the update() method still contains a few key parts of the state updating logic which are closely tied to the entity and the extensions it supports.

If entities are to support extensions through composition this is a fatal flaw. The solution is to remove the update() making entities completely static on their own. Other objects would implement an “Updater” interface that provides an execute() method. Updater objects would be observers to the central time control object of the framework. Each tick the time controller would call execute() on all the currently registered updaters. Extensions whose state is a function of time (such as the current ExtensionCombat) could register themselves as updaters too.

This change would also ease the integration of a physics engine by making it easier to ensure that entities do not misbehave.

Another change that could be very interesting is to remove position from the state of the entity. An entity's position in the world is not really part of the internal state of the entity but more part of the state of the world. Thus entities in the world could be contained within a “slot” object that represents the position within the world. The advantage of this design is apparent when considering items. When an item is lying on the ground it is in a certain position in the world. When the item is carried by another entity what is its position? Should it be relative to the carrier? By having the entity know the “slot” it exists in all these problems can be encapsulated within the “slot” itself. An entity laying on the ground will be in a “world slot” which just stores the entity position, an entity carried by another will be in a “inventory slot” which might reference the position of the carrier.

1.10.2 AI

The AgentSpreading class requires little further work. As of now it implements correctly all the functionality mentioned in the paper [MAES89] and seems relatively bug free. On the other hand there are many improvements that could be done to ease the development of specific agents that use it as a base. The first and most crucial one is how the AI parameters θ , Φ , γ and δ are passed to Symbols and Competences. At the moment these parameters are stored as static public constants in the AgentSpreading class, this means that every agent will use the same parameters and it is impossible to modify them in a subclass. The Symbol and Competence interfaces should be rewritten to take these as parameters passed either on creation or in the parameters list of the spreadEnergy() method.

Another useful change would be to integrate the graph node class and the symbol and competence classes more tightly. This change would simplify the `spreadEnergy()` code and probably improve performance.

Finally providing a bigger library of predefined symbols competences and selectors (including a properly implemented `SelectExplore` class) would make it much easier to implement new agents.

1.10.3 Actor class

While the actors developed in the project are capable of behaving somewhat rationally they are far from the capabilities of agents developed in similar environments (referring to games such as [MAJESTY] or [SIMS]). The actor class should be extended to handle more complex task such as being able to hunt non static monsters, understand the concept of a “base”, communicate with other agents and maybe collaborate with them.

The latter is probably the most complex extension, many papers have been written on algorithms describing how agents can form groups independently on developer control such as [GRIFFITHS03].

One simple solution might be to develop a pair of competences “LeadGroup” and “FollowLeader”, that allow the agent to lead a group and follow a leader respectively. FollowLeader in its simplest form could just be to obey every order from the group leader, but LeadGroup is far more complex: the competence must complete for the agent to make more decisions while still leaving the agent in “group leader mode”. This could be done by having the competence set a symbol `GroupLeader` and having all the other competences produce additional orders for the subordinates if the symbol is set. This solution is extremely primitive and doesn't explain how groups are formed or disbnaded.

Another approach to the follower side of the problem could be to treat the orders received from the leader as percepts and have symbols in the network that respond to them, also the leader should be able to override the selectors in the followers to maintain coordination. At this point the leader could have a separate “Group Control” network that is activated when leading a group. This would then take care of coordinating the group generating orders for every follower network in the group (the leader's included). This second approach doesn't break the Spreading Activation network design as much and allows followers a much higher degree of autonomy, in this system a follower could decide it would rather break off from the group instead of following the leader if the orders are against it goals. The former approach could do so but it would have to implement all of this in

the FollowLeader competence instead of having it neatly integrated in the network. Moreover in this latter approach orders can be much higher level instead of havign to be the low level order to the entity.

1.11 Conclusion

1.11.1 Usefulness of the model in practice

Despite its limitations the Spreading Activation model can certainly be used as the basis for agents in real life applications thanks to its nice mix of qualities: a Spreading Activation agent is easy for developers to use, it requires (relatively) little processing power and is capable of responding dynamic environments without losing sight of its objectives.

The ease of implementation follows from the nature of the network components. Competences represent concrete understandable high level actions, symbols represent high level features of the environment, the low level details are encapsulated within these two concepts. The implementation of the energy spread algorithm can be reused without worry, like the inference engine of planning agents. Competences and symbols are more application domain specific but it is quite likely that they could be reused across different agents used in the same environment. Finally once the implementation of competences and symbols is defined the network can be described in a fairly simple language making changes to its structure fairly easy to carry out.

The agent requires little processing power thanks to its ability to carry out “incremental” searches. A tree based algorithm will only search a limited number of paths to the goals, often only one complete path, by contrast the Spreading Activation model “searches” all the possible paths and records their “score” in the energy level of each competence. If a specific path suddenly becomes impossible the agent already has done most of the search for alternatives. Also the search space is tiny compared to that used by both tree/graph based algorithms and planning agents due to the massive encapsulation of low level concerns done by the competences and symbols.

Finally the agents demonstrates a nice combination of goal orientedness and reactivity to the environment and “ratio” of these two characteristics can be tweaked by changing the network parameters.

The extensions to the model made in this project are somewhat natural extensions of the original ideas. Making the add and delete list of competences non deterministic results in possibility of describing competences with multiple non deterministic outcomes and allows the agent to plan around all of them. The extension is quite natural when one remember that even in the original paper a competence could fail and execute without changing the state of the world.

Selectors are a way of implementing indexing features of the environment that makes competences

and symbols even more reusable. A selector is an index in the features of the world that retrieves the object or location that most closely fulfil a certain characteristic. Again there is encapsulation of low level details so that the implementation and understanding of the system may be easier.

There are a number of papers that have been written about implementing or extending the Spreading Activation model, but sadly it was impossible to gain access to them. Some of them are:

- Maes, P. 1991. A Bottom-Up Mechanism for Behaviour Selection in an Artificial Creature. In From Animals to Animats, First International Conference on Simulation of Adaptive Behaviour. MIT Press, Cambridge, Ma.

Details the use of spreading activation network in a more complex environment

- Tyrrell, T. 1994. An Evaluation of Maes' Bottom-Up Mechanism for Behaviour Selection. In Journal of Adaptive Behaviour 2(4): 307--348.

A critique of the approach

- B. Rhodes. PHISH-nets: Planning heuristically in situated hybrid networks. Technical report, MIT Media Lab, 1996

An improved version of Spreading Activation Networks

1.11.2 Project success evaluation

Most of the basic goals of the project as presented in the original specifications have been achieved. The biggest change from the original objectives was dropping the comparison between Spreading Activation network and partial planning based agents for reasons already explained in section 1.5.1. The biggest difficulty encountered during project development was time management. The first point where time management was a problem was when writing the first time table, as I had no idea of how long each section would take. As the project progressed it was possible to keep up with the original time table until the early weeks of Term 2 where software problems cause considerable delays. The remainder of Term 2 was spend trying to catch up with the original time table while other academic commitments took up much more time than expected.

Now with the project completed I have a better idea of how to estimate the time taken to develop a given piece of software and how to interleave multiple commitments.

The major design related difficulty encountered when developing the software was how to object life management. Without a garbage collector the life of every object has to be tracked by hand and

the object has to be destroyed when it is no longer needed. For most object the easiest way to do so is to identify another object that “owns” it and have the owner take care of destroying the object. In other cases the life time of the object is more complex as ownership is shared across many others. During the design phase very little effort was put in determining the exact owners of each object and how to best manage its life time, this caused sever problems in the later stages of development when many unforeseen non trivial life times emerged requiring extensive modifications of the code base.

Overall the project might have benefited from the use of more formal design tools to more clearly define the competences of entities and many other classes. During development some parts of code were shuffled back and forth between different classes before settling down.

1.12 Acknowledgements and thanks

Several external libraries where used in this project to simplify and speed up development:

Most of the mathematical and geometry classes used in the project are from the Ogre 3d engine (all the files used are in the OgreImports directory)

Input and output are done through the Simple Direct Media Library (SDL) which greatly simplifies integrating OpenGL with the X protocol.

Finally a few libraries from the boost project were used extensively: boost::any is at the core of the variable system for entities, boost::lexical_cast was used for debug output and boost::smart_pointers was fundamental. Other boost libraries such as boost::graph were used as inspiration for the implementation of some of the code.

I'd also like to thank Dr Nathan Griffiths for his invaluable help and counsel throughout the development of the project.

2 Appendices

2.1 Appendix A: The model file format

The files that store model description are plain text files in the following format:

- All the lines are terminated by a new line (`\n`)
- The first line contains the number of vertices in the model;
- The following lines contain vertex definitions, one per line
- Vertices are defined as `x,y,z;r,g,b` where `x`, `y` and `z` are the coordinates of the vertex from the model origin and `r`, `g`, `b` are the colour components of the vertex (0 – 255)
- The first vertex to be defined has index 0, the next index 1, the last `#vertices – 1`
- The line after all the vertices are defined contains the number of triangles making up the model
- The following lines contain face definitions, one per line
- Faces are defined as `i,j,k` where `i`, `j` and `k` are the indices of the vertices that make up the triangle in counter clockwise order

Any error in the file syntax will cause the program to fail loading the file and exit.

2.2 Appendix B: File list

On the CD you will find the following directories: demos, docs, external_libs, references and src.

demos contains five demos of the project

docs contains a copy of this document, the original specifications and the presentation in pdf format

external_libs contains the source code of all the libraries used in the project (except Ogre). Refer to the individual libraries for installation instructions

references contains a copy of all the paper referenced in the project in pdf format

src contains the full source code of the project. For building information refer to the next section.

2.3 Appendix C: Running the demos and building the project

To run the demos simply cd to the demo folder on the CD and run them. Running them from another directory will result in failure to find the model files.

In all the demos in addition the agents there are 5 blue, green and red triangular objects spinning near the centre of the world and another one randomly moving around, these are there just to show that the program is running even when the agents might appear to be stuck.

To build the project copy the src directory somewhere. Ensure that all the required libraries are installed (boost and sdl). Install scons anywhere you like. Then cd to the project directory and run scons. The compiled program will be called project.

You can pass a few options to scons to tailor the building process.

debug=1 will build with debug symbols.

optimize=1 will build with optimisations.

mac=1 will change the settings to build under OS X.

3 Bibliography

3.1 Sources referenced in the report

- [SIMS] - EA Games; The Sims; 2000; <http://thesims.ea.com/>
- [GRIFFITHS03] - Griffiths, Luck; Coalition Formation Through Motivation and Trust; 2003
- [MAJESTY] - Microprose; Majesty; 2000; <http://www.majestyquest.com/>
- [MAES89] - Pattie Maes; How to do the right thing; 1989
- [PIRANJAN98] - Piranjan, Madsen, Granum; Bouncy: an interactive life-like pet; 1998
- [RAO91] - Rao, Georgeff; Modeling rational agetns with a bdi architecture; 1991
- [RAO95] - Rao, Georgeff; BDI agents: from theory to practice; 1995
- [RUSSEL03] - Russel, Norvig; Artificial intelligence, a modern approach; Prentice Hall; 2003

3.2 Other Sources

3.2.1 Papers

- [BOUTILIER92] - Boutilier; Toward a logic for qualitative decision theory; 1992
- [CAICEDO01] - Caicedo, Monzani; Toward life-like agents: integrating tasks, verbal communication and behavioural engines; 2001
- [DECKER95] - Decker, Lesser; Designing a family of coordination algorithms; 1995
- [PAULS01] - Pauls; Pigs and people; 2001
- [POLLACK90] - Pollack; Plans as complex mental attitudes; 1990
- [RAO96] - Rao; AgentSpeak(L): BDI Agents speak out in a logical computable language; 1996

3.2.2 Books

- Deitel & Deitel; How to program C++; Prentice Hall; 2003
- Bruegge, Dutoit; Object-oriented software engineering; Prentice Hall; 2004
- Somerville; Software engineering; Addison Wesley; 2004
- Gamma, Helm, Johnson, Vlissides; Design patterns; Addison Wesley; 1995
- OpenGL ARB; OpenGL programming guide; Addison Wesley; 2004

3.2.3 Websites

All the following website were valid as of 26/04/2006

Dinkumware STL reference guide: <http://www.dinkuware.com>

SGI STL reference: <http://www.sgi.com/tech/stl/>

Ogre: <http://www.ogre3d.org>

SDL: <http://www.libsdl.org>

boost: <http://www.boost.org>

Scons: <http://www.scons.org>