Incorporating Inner States for Agent Systems in Strategy Games

Undergraduate Thesis

Lenka Pitonakova
Supervised by Gordon Clapworthy
Computer Games Development
University of Bedfordshire
May 2009

Abstract

The theoretical part of the project focuses on currently used techniques for obstacle avoidance, task-oriented behaviour and fuzzy logic in the field of games AI. A number of approaches are discussed and evaluated.
The artefact of the project is a strategy game in which units have an inner state which reflects on their actions. This addresses a problem in most of today's strategies - units behave in the same way and display no reactions to how user plays a game. Agents in the game use fuzzy logic to evaluate their current state and adjust their behaviour accordingly. The individualised behaviour was evaluated as an interesting feature by a number of players who tested the game.
Other aspects of games intelligence have also been experimented with - including obstacle avoidance in a continuous environment, finite-state machines, task representation and task-oriented behaviour.

Acknowledgements

I would like to thank my supervisor Gordon Clapworthy who supported me throughout the whole project and helped me keep on track.
I am also grateful to my friends who kindly spent some time testing the project artefact: N. Antolova, F. Grüenke, M. Mačo, P. Pitoňák and R. Vaňuš.

Keywords

Strategy game, life simulation, games artificial intelligence, obstacle avoidance, emotions, inner state, agents, fuzzy logic

Table of Contents

1. Introduction

1.1. Background

This project focuses on strategy games development and games artificial intelligence. The term 'artificial intelligence' in games refers to 'techniques used in computer and video games to produce the illusion of intelligence in the behaviour of non-player characters' (NPCs) [Wikipedia Games AI]. Based on this definition, we can assume that all games which have NPC characters implement artificial intelligence. This is especially true for strategy games where both player's and opponent's (computer's) units can be understood as agents programmed to do various tasks like getting from one place to another while avoiding obstacles, gathering resources or engaging enemies in combat. However, most of today's games use units which don't change their behaviour, nor they directly react to what happens to them or in different cases the units do only that. On one hand there are brilliant strategy games like Command & Conquer or Age of Mythology which most of all implement intelligent enemies, while on the other hand we have life-simulations like the Sims where characters develop as the game progresses. However, there is nothing in between and this is where the gap in the market is.

To address this problem, this project focuses on means of implementing 'inner state' into units controlled by the player. The reason behind this is that even though strategy games are nowadays very advanced, player doesn't directly connect with the units he controls thanks to the fact that all of them are the same and do not mean anything as individuals. Units are all good for exploring and fighting, collecting resources or building structures but the player doesn't need to pick up the ones who would be best for these types of tasks. The units do not complain - they simply do what they are told and are happy to do it for hours of game play.

1.2. Introduction to the Project Artefact

The project artefact is a simple 2D asymmetric strategy game called Alien Farm where player takes care of a colony of alien beings. The aliens can gather resources and breed. Their inner state consists of happiness and activity - two variables which are adjusted for each unit based on what it did previously and whether it liked it or not. Aliens are finite-state machines which have a memory where their actions can be recorded. The units care about the well-being of their colony as well - this is characterised by the amount of food in store and available space in houses.

Units are able to adjust their behaviour based on their happiness. If they are frustrated they will gather and breed slowly, but doing their preferred job makes them very efficient. Also, they get tired when they travel or get old which results in slower movement. Fuzzy logic was used both for evaluation of the current inner state as well as to adjust the behaviour.

The game has been tested by a number of players and their reactions on this type of behaviour are discussed and taken into account in the latest implementation version.

1.3. Aims and Objectives

Aims

1. To create an effective path finding and obstacle avoidance algorithm for an agent system by merging and altering current widely-used approaches
2. To create a game where agents are able to complete various tasks and their behaviour is based on their current state affected by the environment and agent's recent performance

Objectives

- To identify and evaluate existing approaches to path finding and obstacle avoidance
- To identify and evaluate existing approaches to task-oriented behaviour
- To develop a game which implements most effective approaches to the above
- To design and develop processes for agents in the game so that their behaviour is individual and changes by their past experience
- To make the consequences of player's actions easily understandable when it comes to the effect on behaviour of the agents
- To create an easily understandable interface for the game

1.4. Methodology

In the beginning of the project, a Gantt chart was created with a schedule for all desired functionality. The project was meant to follow the Waterfall method with fixed deadlines for a number of stages. However, this became unfeasible soon, especially after the project character has changed from creating a simulation of an agent system able to learn to developing a game where agents have the inner state.

In January new objectives have been set up which required a new Gantt chart. Much more estimated time was added to each of the stages so that there was enough time to implement new ideas during the development. The Iterative methodology was used where the newly implemented features were tested immediately after small time periods. This helped to cope with the time management in a better way - if there was something which needed fixing, the memories about how it was developed were fresh and it was quick and easy to adjust the functionality. Also, some ideas from the agile programming methodology were taken into account, especially adding features during development if there was extra time in the end of a stage.

The final project plan was split into the following stages:

Stage	Due date	Tasks involved
1. Movement	2nd February 2009	agent selection, tasks queue, target following, obstacle avoidance

2. Creating buildings	6th February 2009	creating the graphics, placing buildings on the terrain using the Construction tab
3. Gathering	20th February 2009	setting up the resources (food, grain) and gathering them, breeding, initial speech representation
4. Implementing inner states	18th March 2009	implementation of fuzzy logic, setting up the rules, state evaluation, state graphical representation, balancing the game (resources vs. costs, impact of happiness and activity on the colony's ability to survive)
5. Creating a game	8th April 2009	random map generation, saving/loading, win/loose state, rewards to the player, tutorial
6. Game evaluation and adjustments	20th April	distributing the game to up to 10 people, creating a questionnaire, collecting the views, final adjustments

1.5. Structure of the Report

Chapter 2 provides the information about various AI techniques based on the research done in the beginning of the project. The chapter focuses on path finding and obstacle avoidance, task-oriented behaviour, task processing and task representation. It also provides an insight into fuzzy logic - a technique used for classification of data as well as producing output based on reasoning similar to human.

Further sections discuss building of the artefact. From basic game features like creating the world and making units (chapter 3), through creating terrain objects and buildings (chapter 4), implementing fuzzy logic and inner state (chapter 5), to adding strategy game features like rewards to the player and win and lose states (chapter 6). Chapter 6 also reports on user evaluation and lists improvements which have been done before releasing the game. Chapters 3-6 first talk about how the individual features have been implemented. Testing and evaluation is a part of each chapter and refers to the implemented features. Each chapter ends with a section which lists minor adjustments which needed to be done and features which were added after considering the testing results.
Chapter 7 concludes the findings of this project and provides suggestions for future work which could be based on the artefact. The chapter is followed by the list of references and appendices. Appendix W shows the project poster.

2. The Contextual Review

There are many types of games on the market which are in continuous development as players requirements for a good-quality game get higher and higher. This chapter will discuss the State of the Art in the games industry and present what people expect from different games and what features are most likely to attract the audience.

The section will also provide the reader with an insight into the current techniques used for obstacle avoidance in both tile-based and continuous environments. A significant amount of work has been done on this topic already since the first games were developed. A number of approaches will be taken into account and their advantages and disadvantages will be provided.

Further sections will discuss how a task-oriented behaviour in artificial entities can be created. Task representation and task processing will be described. Also, fuzzy logic will be discussed since it played a major role in creating the project artefact. Fuzzification can be used to calculate entity’s inner state - mood, happiness and awareness of the world around. Fuzzy rules and defuzzification can than help to determine what action to take or calculate an output value.

2.1. State of the Art

Currently the world of AI developers is split into two: one part uses 'deterministic' AI which is used to simulate intelligent behaviour. Such AI is less computationally expensive and does what it is programmed to do – e.g. follow a player or run away from a certain place. Such AI is rather easy to program, control and test. On the other hand, 'nondeterministic' AI is more difficult to predict. Behaviour emerges from a set of low-level rules, i.e. it is not directly coded. The examples of such AI include flocking behaviour or programs able to adapt to the user. Here a bottom-up approach originally introduced by Langton is used – small rules coded into an artificial entity combine in order to produce its behaviour [S. Levy, 1993]. Such AI is mostly used for simulations and research because it needs much more memory and CPU power.

It is necessary for computer games to have some kind of AI implemented. Today's AI is much more sophisticated than in the previous century but still lacks a lot to be realistic. The newly discussed topic is 'Adaptive AI', in other words AI than can learn from its experience and change its behaviour [AI Depot and Planet Crap].

AI is mostly used in the following types of games:

a) **Strategies** (Age of Mythology, Black and White)
b) **RPG game**s (Diablo)
c) **MMO** (Massively Multiplayer Online) **games** (World of Warcraft)
d) **First-person shooters** (Half-life, Halo)

Appendix A lists innovative games which widened our understanding of AI.

In general, the game players' community is interested in the following factors of AI:

a) **Unpredictability** of NPCs, good **interaction** with player [GameAI page A], **believable** behaviour [Slashdot forum and GameAI page C]
b) **A-life** systems which are able to evolve throughout a game and display some kind of autonomy, a 'living world' [GameAI page A]
c) Rules and scripts **adjustable offline** [Game Dev and GameAI page B]
d) **Dynamic generation** of story [GameAI page A]

NPCs and A-life systems should behave and evolve in a similar manner than humans in the real life. Believability means that a player is able to develop some kind of connection with an NPC which makes a game much more enjoyable and interesting.

2.2. Path Finding and Obstacle Avoidance

2.2.1. The Environment

Information about the following section was mostly taken from D.M. Bourg's and G. Seeman's AI for Game Developers (2004). I will reference different sources in the text where needed.

Under 'environment' we understand the virtual world where software agents are placed, move, learn and evolve. Each type of a virtual environment has its structure represented by different types of terrain. Terrain usually affects movement of the entities put into the environment. A map of the terrain is usually stored as a 2D grid or a 3D model.
In terms of how its description is stored and how a programme uses this data, an environment can be understood as *tile based* or *continuous*. Please refer to Appendix B where the two are explained in detail.

There are two ways in which AI can perceive the environment:

a) **offline** - via so called 'waypoints' - locations on a map which represent free space and are connected so that a connection represents a clear path
b) **online** - during movement. This approach is similar to how humans perceive the world around but more difficult to implement as more CPU-expensive as an AI entity needs to gather information about the environment around

2.2.2. Model of an Agent and its Movement

A tile-based environment consists of equal sections therefore there is no real physical model of an object needed. We simply put an image of an object on a tile and move the image where appropriate.

Continuous environment is much more complicated and requires an agent to move according to rules used in geometry. Appendix C describes such model and the movement based on it. AI entities should tend to find the shortest path to get from point A to point B. To do so, they need to know their current location as well as a location of the target point.

The movement should ideally appear [D. M. Bourg and G. Seeman, 2004]:

a) **realistic** - similar to movement of biological entities
b) **efficient** – use as little CPU as possible
c) **reliable** - movement algorithms should be tested properly and work in all scenarios
d) **purposeful** – the algorithms should always fulfil their task

Agents called *animats* were introduced [D. M. Bourg and G. Seeman, 2004] to achieve this type of behaviour. Animats are AI entities able to perceive the environment through sensors like infra-red or sound. They are able to get content of a point in front of them (for example a type of material object consists of like steel, rock, water, etc). Also, they are able to find free space between obstacles by querying the environment about a type of material in front of them.

It is important to adjust sensitivity of the sensors so that an AI entity knows about an obstacle in front of it in a reasonable time before it reaches it. Also, some kind of path planning is needed to make the movement realistic (i.e. AI entities should tend to avoid obstacles following a curved path, not using straight lines and turning at one point).

In 2003, O. Ringdahl has introduced an obstacle avoidance system for autonomous forest machines. The paper refers to trucks which can be sent to a forest and are able to avoid obstacles based on their ability to perceive the environment around them. A truck consists of two parts joined in the middle. This model can be simplified just to one part and the introduced algorithms can be used for any AI entity including a robot, a spaceship or an evil warrior.
To solve the problem of a vehicle getting from point A to point B, Ringdahl has introduced a Pure Pursuit method. Please refer to Appendix D to read more about Ringdahl's findings.
The situation gets more complicated if there are obstacles in the vehicle's way. To deal with obstacle avoidance, various methods have been introduced, including waypoints and A* algorithm [D. M. Bourg and G. Seeman, 2004], vector-field histogram [O. Ringdahl, 2003], potential function and tracing algorithm [D. M. Bourg and G. Seeman, 2004]. Please refer to Appendix E to find out more about these methods and compare their advantages and disadvantages.

2.3. Task-oriented Behaviour and Adaptive AI

2.3.1. Task Representation

The main purpose of each agent system is to complete given tasks. Agents must therefore be aware of their tasks as well as of a way of how to reach a goal. To achieve this, rules for completing tasks need to be created. They can be understood as objects with attributes which can change dynamically.

PAR (Parameterized Action Representation) objects [J. R. Lee and A. B. Williams, 2004] are able to store information about how to accomplish tasks. The logic was used for 'smart avatars' able to receive orders from a player and respond according to their current state and state of their environment.
A PAR object describes:

a) **Target object(s) of an action** - object(s) that can be manipulated or marked for use
b) **Agent** - performer of an action
c) **Applicability conditions** - can include e.g. agent's capabilities or object's configuration
d) **Start/ result** - start and result time, state of the environment at this time
e) **Sub actions** - any smaller actions needed to complete (e.g. 'get to a location', 'turn on flashlights', etc.)

f) **Core Semantics** - components of meaning of an action (e.g. force, path, post-conditions, etc)
g) **Purpose** - conditions to achieve or actions to generate/enable
h) **Conditions of termination** - states of the agent/environment when the action is stopped
i) **Agent Manner** - constraints on manner in which action is performed

The PARs can be extended by additional parameters for personality and mood.

The PARs can have two forms:

a) **Uninstantiated (UPARs)** - carry generic information (parts C-F). These can be understood as 'classes' in C++.
b) **Instantiated (IPARs)** - override UPARs with more specific information (performer, object, manner of performance). These are understood as 'instances' of a class which are created when an instruction is given to an agent.

There are several types of instructions which can be received either from a user or from a system. The most distinctive ones include:

a) **Immediate** - e.g. 'Explore location XYZ'
b) **Conditional** - e.g. 'When there is a steep terrain, go around'
c) **Negative** - e.g. 'Do not cross a swamp'

Lee and Williams (2004) have also described how to construct architecture to allow PARs to work. Please refer to Appendix F to see their model.

2.3.2. Task Processing and Specialisation

We can imagine an agent as a finite-state machine. Finite-state machines have a number of states they can be in and conditions about when a state changes to another [D. M. Bourg and G. Seeman, 2004]. An intelligent agent must know how to decide which state to be in. If all agents start with an 'idle' state, there must be a mechanism which makes them start working on tasks.

V. A. Cicirello introduced so-called 'wasp-like agents' in 2001. These entities remember a response threshold for all existing tasks and tend to pick up one with low response threshold and high task stimulus (linear to the time the task hasn't been assigned). There is a certain probability P for each available job to be picked up by an agent:

$$P(\Theta_{w,j}, S_j) = \frac{S_j^2}{S_j^2 + \Theta_{w,j}^2}$$

Figure 1-1 – calculating probability of picking up a task according to V.A. Cicirello (2001)

Where S_j is a task's stimulus and $\Theta_{w,j}$ is a response threshold of the task type.
Wasp-like agents can roam in the environment until they sense that there is a task around them needed to complete. Alternatively, all available tasks could be stored in a global array or in a global object to which all agents have access. Idle agents are able choose the next most suitable tasks for themselves.

We assume that each task requires a certain tool or characteristics of an agent in order to be completed. For example, exploration jobs should be assigned to agents with more powerful thrusters while gathering resources should be done by agents who have a carriage with them. If an agent is to switch from one specialisation to another, it has to return to the base and swap a current tool for a new one. This is where the response threshold is used. After completing a job A, the response threshold for this type of job is decreased in order to make it more attractive for the agent next time. Moreover, response thresholds for all other types of tasks are increased. If an agent is idle than all response thresholds decrease slightly with time.

Updating the response thresholds this way makes agents specialise in a type of task. A colony should end up being split into 'explorers' and 'gatherers'. However, if for example there are no gathering tasks available but there are idle gatherers within the colony, response thresholds for all jobs (including exploration) decrease incrementally. Also, if the colony doesn't have enough explorers and there are only exploration jobs available, their stimuli increase with time and they become more attractive for gatherers. This makes gatherers start specialising in exploration.

2.4. Fuzzy Logic

2.4.1. Basics of Fuzzy Logic

Term ‘fuzzy logic’ stands for “means of presenting problems to computers in a way akin to the way humans solve them.” [D. M. Bourg and G. Seeman, 2004, p. 188] While computers deal with numbers, people tend to classify their perceptions and act upon this classification. The logic originally introduced by L. Zadech in 1965 provides a way to categorize input values in a way that humans do, i.e. by giving each value a degree of membership to a certain group (e.g. if a person is 1.9m tall, we can say he is 20% medium height and 80% very tall).

Fuzzy logic has a number of uses in computer games [D. M. Bourg and G. Seeman, 2004, p. 190], namely:

a) **Control** – the environment and objects around can be evaluated and steering forces determined
b) **Threat assessment** – a number of factors can be looked at, e.g. if enemy is far or close, if it seems dangerous or friendly, etc. Fuzzy rules (please read 2.4.3. below) can be used to determine an output action.
c) **Classification** – Similarly to threat assessment, we can ‘fuzzify’ (classify using fuzzy terms) various characteristics of objects and let the AI decide upon them. We can either trigger actions or get numeral output (please read 2.4.3. below)

2.4.2. Fuzzification

Fuzzification means converting a numeral input (e.g. height of an object) into fuzzy sets (e.g. ‘short’, ‘medium height’, ‘tall’, etc). A membership function can be used to convert a number into degrees of membership for each of the possible fuzzy sets.

Figure 2-2 shows an example of membership functions for converting a numeral value of height. Note that each of the functions for ‘short’, ‘medium height’ and ‘tall’ have a different shape. There are no pre-defined rules about the shape of the functions, although a few types have been established [D. M. Bourg and G. Seeman, 2004, p. 194-199].

If the input is for example 1.2m, we run this value through all three functions. This will give us membership grades: 100% short, 0% medium height, 0% tall. If the input was 1.9m, the grades would be: 0% short, 50% medium height and 50% tall. This means we don’t only classify objects as ‘tall’ or ‘short’, we can also determine how much tall or short something is and act upon that. In a different case, we could have a rule which says ‘if enemy is 40% *healthy*, start attacking it, otherwise run away).

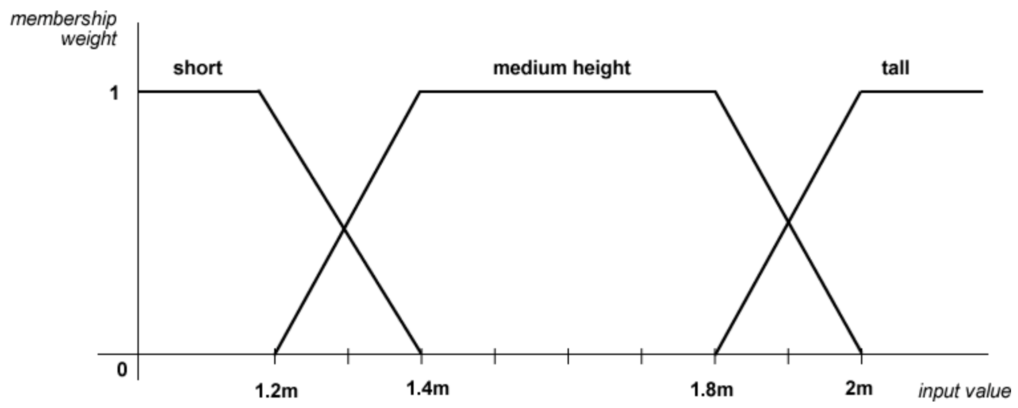


Figure 2-2 – grade, trapezoid and reverse grade membership functions

We could extend add as many fuzzy sets as necessary, depending on what the current situation demands.

2.4.3. Fuzzy Rules and Defuzzification

After the input has been fuzzified, it can be used when determining output. This can be done in two ways: we either set up rules which will fire an action or we produce crisp output numbers.

Fuzzy rules

Fuzzy rules can be applied when we are deciding upon an action to take based on a certain input. After the degrees of membership of a person being short, medium size and tall have been determined, we might want to say e.g. 'if small buy high-heeled shoes'. Moreover, if there is a number of fuzzified sets which classify not only height but also mood, colour of hair, etc, more complex rules can be established, e.g. 'if tall and blond hair buy a black T-shirt'.
Similarly to the Boolean logic, axioms like AND, OR and NOT can be used with fuzzy terms. However, in fuzzy logic functions which will calculate results of using such statements need to be created. Example 2-1 shows how these can be calculated.

```
A or B = MAX (A,B)
A and B = MIN (A,B)
not A = 1-A
```

Example 2-1 – fuzzy axioms [D. M. Bourg and G. Seeman, 2004, p. 200]

As an example, if the hair colour is 50% blond and height is 100% tall, the result of using AND would be only 50% truth. On the other hand, the OR operation would return 100% and trigger an action.

Defuzzification

In Boolean logic, rules like 'if obstacle in front than steer left' or 'if no obstacle go ahead' can be set up. This means that actions can only be taken after objects are completely in front or if the space in front is completely free. On the other hand, having degrees of membership of where the obstacle is provides a more efficient way to determine to what degree actions are done – if obstacle is 70% in front (e.g. it is slightly to the right), the steering can be adjusted accordingly. The same result could be achieved by evaluating the obstacle's relative position as a floating point number – however, we would lose the control of classifying the position which could cause problems if the test was used throughout the code.

To calculate a crisp output (e.g. the steering force), we can use a geometric centroid of the area under the output fuzzy set or, in a much more simple case, a predefuzzified output function [D. M. Bourg and G. Seeman, 2004, p. 204]. To calculate the predefuzzified output, we first assign values to each of the fuzzy sets, e.g. -10 for object to the left, 1 for object in front and 10 for object to the right. Than the degrees of membership are calculated, e.g. object is 0% to the left, 70% to the front and 40% to the right (note that the degrees don't have to sum up as 100%). To calculate the output , the following equation can be used:

```
Output = (0*(-10) + 0.7*1 + 0.4*10) / (0+0.7+0.4) = 4.27
```

Example 2-2 – calculating defuzzified output

If we know that output -10 (object is completely to the left) should result in full steering to the right, etc, we can use the output value to adjust the steering force to the degree it is needed to.

2.5. Conclusion

The chapter gave us an insight into the current state of the AI development and trends. Today's players find it important that the artificial NPCs behave in a sophisticated way and have some kind of unpredictability and autonomy incorporated in their behaviour.
We also discussed how movement in a virtual world can be developed and a number of approaches to obstacle avoidance including waypoints, vector-field histogram, potential function and tracing algorithm. Furthermore, we had a look at task representation and processing, how AI can formalize tasks and how they can be understood as objects which can be manipulated and stored.

In the end fuzzy logic was introduced. Fuzzifying input not only gives developers a chance to produce behaviour based on more human-like classification and rules but also keeps ways of evaluating the input separate from ways of producing the output. This is very important if the input is used throughout the code and changing means of evaluating the current situation would be time-consuming if we were working with Boolean or floating point numbers.
In the next chapter, we will look at creating a 2D world with primitive inhabitants and development of basics of their intelligent behaviour.

3. Creating an Entity in a 2D World

In the previous chapter we discussed a number of theories for obstacle avoidance, path finding and various aspects of artificial intelligent behaviour. We will use the accumulated knowledge throughout the following sections and show how some intelligence can be implemented. Creation of Alien Farm, a simple strategy game where units display emotions and react on tasks given to them by user will be discussed. In this chapter the basics of their behaviour will be described.

There are various aspects which need to be considered when creating a 2D world and entities which live in it. How to represent world objects? How to calculate movement between points A and B and more importantly, how to make sure that an entity reaches its destination while avoiding all obstacles on its way?

The section will describe creation of obstacle avoidance algorithm between circular objects. To solve the problem, vectors in entities local coordinate system can be used to determine relative position and distance from obstacles. Combined with the information about obstacle's proportions, the steering force can be calculated in order to turn an entity away from obstacles. Even more realistic movement can be achieved by adjusting the speed - an entity should slow down when it is close to an obstacle in order to avoid it more carefully.

3.1. The System Structure, World Coordinates and Mapping

Please refer to Appendix G to see a simplified UML diagram of the artefact where all important class members mentioned throughout the following chapters are listed.
The programme is structured in a usual way: there is a game states which represent the main menu and the play state. The play state holds the actual game world where the terrain, buildings and units are placed.

In the beginning of creation of any game, there is a challenging task to accomplish: establish a coordinate system in a newly created world. It is always difficult to navigate in an environment where boundaries are not clear or where you cannot specifically identify where exactly you are. Therefore, first an anchor point should be declared, i.e. a point which can be represented by coordinates [0;0] (or [0;0;0] in a 3D world). Also, movement direction has to be taken into account, in other words, how coordinates will chance if user moves right, left, up or down.

Alien farm has the anchor point in the middle of the world. The game starts with view centred, i.e. point [0;0] is in the middle of the screen. This point was chosen because centre of the screen is the only point common to all screen resolutions – it is sure that if the screen width and height are halved than the rendered view will always be centred.

The scale of the world is 4000 x 3000 units (pixels in terms of rendering), so an object can have coordinates X = <-2000; 2000> and Y = <-1500; 1500>. Appendix H shows the world axes.

Mini map

The mini map is based on an idea of the virtual world shrunken so that it fits into the 285x214px area of the map. To achieve this, the coordinates and the size of all objects need to be converted before rendering the map objects. First the magnifying factor is calculated by comparing the real and the map size. It is then used to convert all three attributes, X, Y and diameter of objects. Example 3-1 shows how to convert the X coordinate of an object:

```
float magnifyFactor = float(iWorldXScale*2/iMapXScale);
float origX = anObject->GetWorldX();
float mapX = centreOfTheMapX+ origX/magnifyFactor;
```

Example 3-1 – calculation of the map X coordinate

Appendix H shows the working mini map.

3.2. Model of an Agent

The following is a description of the classes which an agent is composed of. Please refer to Appendix G to see how the classes link to each other.

The Agent class

The agent class is responsible for general agent activities like calculating the life time, dying, eating and handling the mouse. Furthermore, the following classes have been used to deal with various aspects of the behaviour:

Behaviour engine

The class handles the three agent states: *idle*, *moving towards* and *performing*. Figure 3-1 shows relationships between the agent states. The engine also handles speech, i.e. decides what kind of sentence to say depending on the current happiness and action.

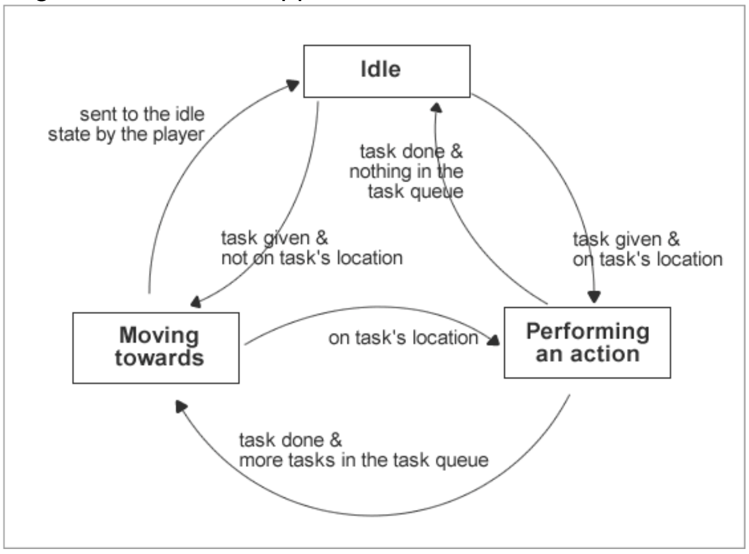

Idle
sent to the idle state by the player
task done & nothing in the task queue
task given & not on task's location
task given & on task's location
Moving towards
on task's location
Performing an action
task done & more tasks in the task queue

Figure 3-1 – agent’s state diagram

Motion engine

The class handles all the movement, target following (i.e. navigation to a given point on a map) and obstacle avoidance. The individual algorithms for calculating movement are described in sections 3.4. - 3.6.

Sensors engine

The engine is used by the Motion and Behaviour engines to get information about the outside world. Among other features, it is able to find the nearest building of a certain type, determine which side an object is on relative to the agent (e.g. north, west, south west, etc.) and find out if an object is in a certain radius around the agent.

Inner sensors engine

The class deals with the inner state variables. The engine uses fuzzy logic to determine degrees of happiness values (happy, normal, sad) and activity values (lazy, moderate, active). It can provide the other classes with:

- individual degrees of these values
- general happiness or activity scale: values are in range <0; 1> and are used to determine percentage of one of the state variables. The scale is used for example to display the state bars (further details will be discussed in section 5.4.)
- general happiness or activity weight: values are in range <-1; 1> and are used to adjust an attribute of agent. If 0 is normal than -1 (i.e. 100% sad or lazy) decreases an attribute value and vice versa (further details will be discussed in section 5.4.).

Memory engine

Memory engine handles the task queue. Tasks (e.g. 'gather grain') can be added to the queue by other classes (mostly the behaviour engine). New tasks are added to the end of the queue and the current task is in the 0th position. When a task has been accomplished (e.g. an agent has its cart full of grain), the whole queue is pushed and the next task is set to current (e.g. 'return to silo').

A task is represented by a struct variable which has attributes X and Y coordinate of the target, type of a task (gather crystals, gather grain, unload cart, no action, go to location, breed) and the array index of the target (e.g. of farm or silo), similarly to the PAR objects [J. R. Lee and A. B. Williams, 2004] (read more in section 2.3.1.).
The type of a task tells the Behaviour engine what to do. Coordinates and array index of the target are used to find the task's location. For the gathering jobs, the target is a farm or a crystal. Unloading takes place in a nearest silo. Target of the breeding task is the agent itself which means it stays in its current location. Finally, target of the 'go to location' task is a place on a free terrain

3.3. Game Interface

The main class which holds the user interface is called CPlayInterface. It consists of a texture put on top of the world with transparent middle, a number of buttons which control the interface and four tabs which are a part of the bottom control panel (Appendix H shows a screenshot of the interface).

3.3.1. Bottom Panel Tabs

The tabs are a 2D array of the CInterfaceObject class. They hold all the tabs objects (i.e. individual building buttons for the construction tab, the god powers buttons, the 'colony information' object and the 'objectives' object) in individual slots along the array's x-axis (1st dimension). The Render function of the Play Interface class renders only objects from a currently selected tab. The Update function works in a similar manner.

The individual objects in the tabs are of separate classes which are subclasses of the CInterfaceObject (Appendix G). This ensures that they can be stored in one array and accessed via for loops but have different functionality.

3.3.2. Interface Dialogues

There is a number of dialogues in the game including: game menu, save dialogue, win/lose dialogues and messages displayed when player wins a god power. Separate classes could have been created for each, but a quicker approach was taken in this case. The CInterfaceWindow class has an enumeration type variable which identifies a type of the dialogue. IF statements are used in the constructor, Render and Update functions to trigger the right actions based on the dialogue type. A common feature of the dialogues is having a back texture and a number of buttons and click areas (click areas are the same as buttons but they don't have any texture). An array of buttons and a number of click areas are members of the CInterfaceWindow class and each type of a dialogue creates different instances of them and handles them individually.

This approach gives the developer better overview of the dialogues behaviour since all code is written in one class. However, if there were too many types of dialogues, they would probably have to be separated into classes because the code would get too complex. In the case of Alien Farm the one-class approach was more effective.

3.3.3. Number of Agents of a Certain Profession

A number of approaches have been taken in order to display a correct number of farmers, miners and breeders on the screen. The problem was in determining in which part of the agent update function should the values change. While it first seemed logical to decrease a number of professionals when a new job has been give to an agent, a problem occurred when agent went to the idle state. On the other hand, a handler for the idle state which would decrease a profession number was useful, but wasn't suitable when an agent went to the idle state while being in a job, e.g. when there were no silos currently available.
There seemed to be a lot of combinations when it came to determining whether to increase or decrease a number of professionals. None of the methods seemed therefore effective enough to cover all the cases.
The final working solution was to set a number of agents in each profession to 0 at the beginning of each loop. Each agent then increases a number in its profession by one so the total always matches with the total population.

3.3.4. Mouse Events Detection

In the earlier implementation, global Boolean variables for left and right mouse down were defined and changed with the state of the mouse. This appeared to be not a sufficient solution for the tutorial since the screen would jump by a couple of slides when a button for the next slide was clicked.
The problem was in the main loop – the mouse was handled on every frame. Therefore when user pressed a button and there were actions associated with it in a current game state, they happened a number of times (loops) while a human would notice a single click. The mouse handling was moved to the programme messages associated with mouse events instead. In the final implementation it behaves as a trigger rather than constantly repeating loop which tests against the mouse state.

3.4. Movement and its Graphical Representation

The implementation of an entity in a continuous environment was based on Bourg's and Seeman's (2004) model (please refer to section 2.2.2 and Appendix C for further details). The thruster and steering forces are represented as one movement vector – its X attribute represents the steering (its positive value means steering left) and the Y attribute stands for the back thruster force (positive value means an agent is moving forwards). The calculation of the movement from the movement vector is as following:

```
angle = m_vMotionForce.m_fx;

//--CHANGE THE ROTATION VARIABLE m_fRotation USED IN RENDERING:
m_pAgent->SetRotation(m_pAgent->GetRotation()+ angle);

//------ CALCULATE THE VELOCITY IN X AND Y DIRECTIONS:
m_fvx = cos(m_pAgent->GetRotation()*PI/180)*g_iTimeSpeed* m_fSpeed;
m_fvy = sin(m_pAgent->GetRotation()*PI/180)*g_iTimeSpeed* m_fSpeed;

//----APPLY THE VELOCITY AND THE THRUSTER FORCE TO WORLD COORDINATES:
m_pAgent->SetWorldX(m_pAgent->GetWorldX() + m_fvx*
(m_ vMotionForce.m_fy*2));
m_pAgent->SetWorldY(m_pAgent->GetWorldY() + m_fvy*
(m_ vMotionForce.m_fy*2));

//------- ADJUST THE SCREEN COORDINATES:
m_pAgent->SetX(m_pAgent->GetWorldX()+m_pAgent->
GetTerrain()->GetGX());
m_pAgent->SetY(m_pAgent->GetWorldY()+m_pAgent->
GetTerrain()->GetGY());
```

Example 3-2 – calculation of the movement using a force vector

Example 3-2 shows that after the world coordinates (i.e. where X = <-2000; 2000>, etc) have been calculated, the screen coordinates are adjusted accordingly. This needs to be done because a user is allowed to move the map and the world coordinates [0;0] are not always in the centre of the screen. We adjust the screen coordinates by adding the difference in which the view moved (`Terrain->GetGX(), Terrain->GetGY()`) to the world coordinates. The X and Y attributes of the movement vector are set to 0 in the beginning of each loop frame and their values are increased or decreased in the target following and obstacle algorithms discussed in the further sections.

To display an alien in different positions (rotations), the alien image could be rotated when rendering is performed. However, this would produce rather awkward images since the game uses asymmetric view. The alien model has been created in 3D Max Studio (a 3D creation tool) which allowed for creation of four views taken from each side and rotated appropriately. When the rotation of an agent changes, the texture image changes as well and is than rotated only by <-45; 45> degrees. Please refer to Appendix I for a detailed description of this approach.

3.5. Navigation Towards a Target

An agent is able to determine its relative rotation towards a certain target location represented by X and Y coordinates. This was done using the X value of a vector pointing from an agent towards a target in agent's local coordinate system [D. M. Bourg and G.

Seeman, 2004, p. 17-19]. If X is greater than zero, the target is on the left side of an agent and it is on the right side if the X value is negative. If X equals zero the target in front so there is no steering necessary.
We can use the X value of the local pointing vector not only to determine target's relative rotation but also to adjust the X component of the thruster force using the following equation:

```
m_vThrustForce.m_fx += vectorToTarget.m_fx*A_CONSTANT;
```

Example 3-3 – calculating the steering force based on target's relative location

Value of the used constant was determined by trial-and-error and set to 10.5.
Similarly, agent's speed can be calculated based on the distance from the target so that the unit slows down when it approaches a desired location. To do this a vector pointing to the target in the global coordinate system is created and its magnitude is taken into account:

```
if (v.Magnitude() < 50){
    m_vForceSteering.m_fy = v.Magnitude()/50;
} else {
    m_vForceSteering.m_fy = 1;
}
```

Example 3-4 – calculating the thruster force based on target's absolute location

Calculations of both local and global vectors are based on Bourg's and Seeman's methods for a vector class [D. M. Bourg and G. Seeman, 2004, p. 349-358]

3.6. Obstacle Avoidance

When there are obstacles around an agent and its sensors detect them, the X and Y components of the thrust force are adjusted based on an angle and a distance from all obstacles around which have a certain distance from the agent [D. M. Bourg and G. Seeman, 2004, p. 75]. The algorithm traverses through all obstacles and gets their positions and diameters from the CTerrain class. For each obstacle, a vector which represents agent's direction and has a certain magnitude (which represents radius of sensors) and a relative vector towards the obstacle are drawn. The obstacle is in the agent's way if

1. Magnitude of the look at vector V is bigger than magnitude of the vector A pointing towards the obstacle
2. Magnitude of vector B (which is a difference between vector A and its projection onto vector V) is smaller than obstacle's radius

Here we can see the benefit of representing the steering force as a real number rather as a Boolean variable (e.g. bool leftThrusterOn, bool rightThrusterOn). Different angles towards obstacles help to determine the final weight of how much the left and right thruster forces need to used. Similarly, different distances and radians of objects help to determine the speed of an agent.
There were some issues with this approach identified in section 3.7. below.

3.7. Testing and Evaluation

The following is a record of testing obstacle avoidance. Each test was done 5 - 10 times and the images below show one situation for each type of test. To track movement of an agent, a member array was implemented which keeps record of previous locations and displays them as a series of 'steps' on the ground. The steps get closer together when an agent is moving more slowly.

Avoiding single obstacles

During this test an agent started in front of an obstacle and was supposed to move forwards/ backwards with the obstacle in its way.

Small obstacle

Agent started from point A, moving forwards towards B, C and D. The path is rather triangular, with maximum distance from obstacle when its centre is in line with the X-axis of agent's local coordinate system. After this point the angle changes rapidly to follow the agent's target.

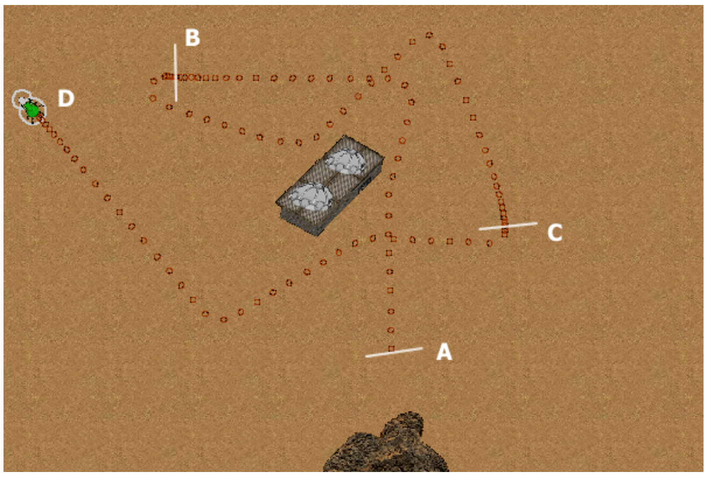


Figure 3-2 – avoiding a small obstacle

Large obstacle

The agent managed to keep a distance from the obstacle while travelling around it. The difference between angles got smaller and the shape of the path looks more like a circle.

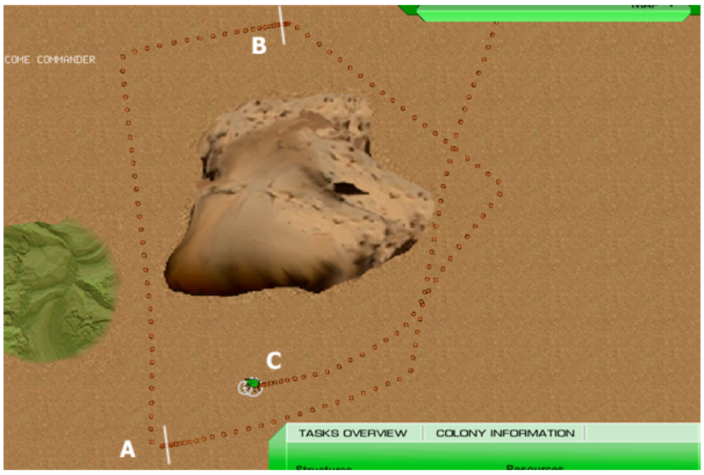


Figure 3-3 – avoiding a large obstacle

A problem occurred when agent was supposed to go too close to an obstacle. It needed to turn couple of times in order to reach the destination.

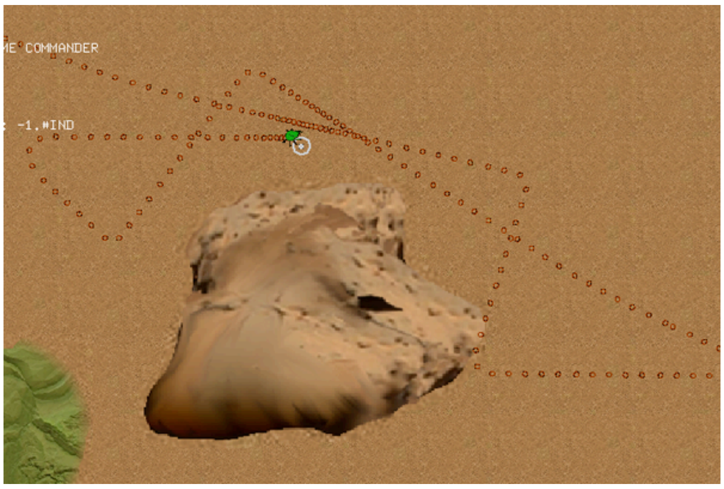


Figure 3-4 – going close to a large obstacle

Avoiding a wall of obstacles

During this test, a wall of rocks was created by placing several rock objects next to each other. Agent was instructed to get from one side of the wall to another.

Testing proves that the method used for calculating the steering angle is not quite suitable for a wall of obstacles. The agent chose to go across the left obstacle since its repulsive effect was neutralised by the right obstacles.

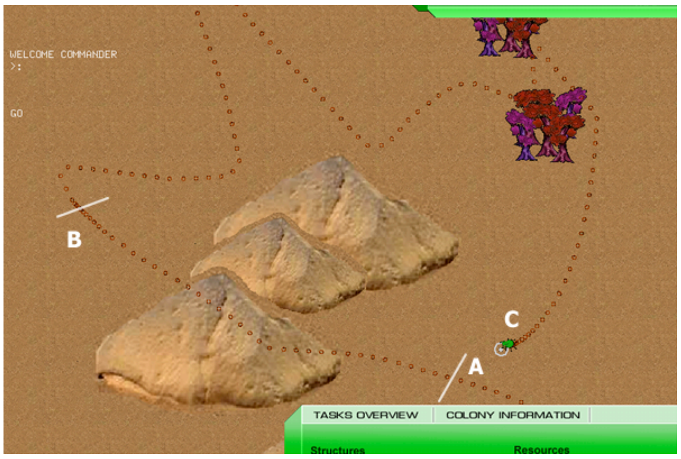


Figure 3-4 – avoiding a wall of obstacles, case 1

In the second case, when the alien was supposed to turn back from B towards C, a forest was in its way. It continued finding a way around by trying different angles. This resulted in taking a route around the bottom part of the obstacle wall.

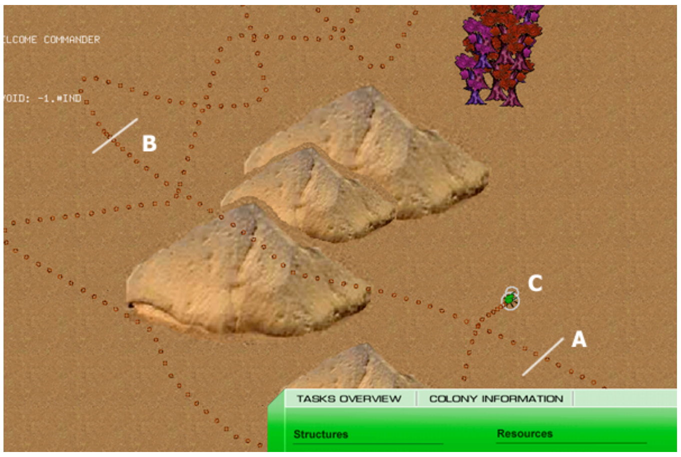


Figure 3-5 – avoiding a wall of obstacles, case 2

Moving between a group of obstacles

The movement between a number obstacles was tested as well. An agent was supposed to find its way across a field of trees and rocks. Figure 3-6 shows that the tests were quite successful apart from the part where the agent crossed a forest while moving towards points D. This was caused by the problem already mentioned during avoiding a wall of obstacles test.

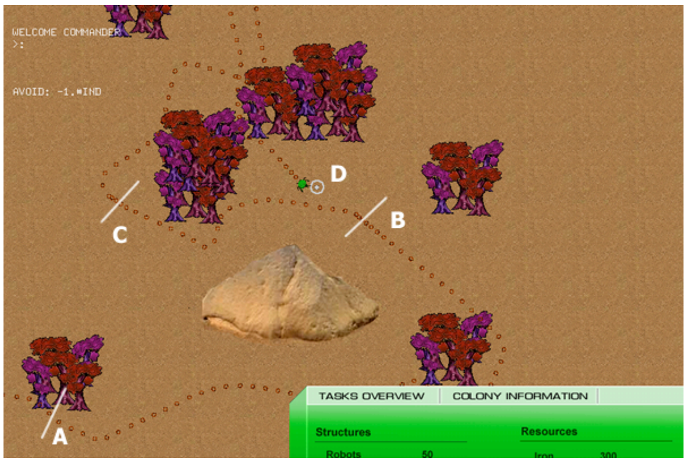


Figure 3-6 – moving between a group of obstacles, case 1

The agent was successful in finding its way in the second case as well. This time there was no wall of obstacles created. The agent 'touched' the forest close to the point B but managed to find its way in all other cases.

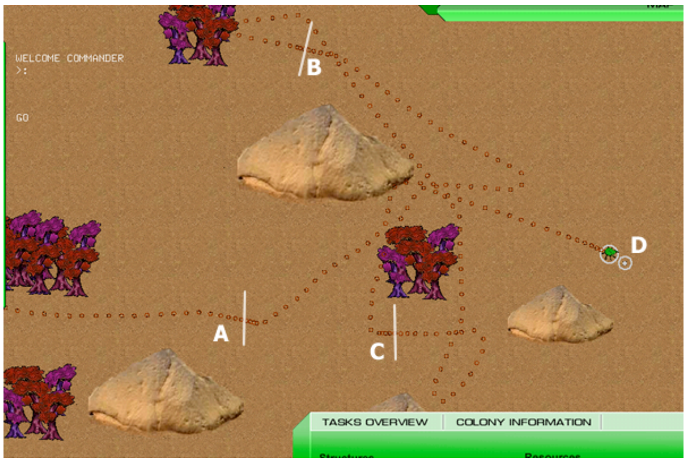


Figure 3-7 – moving between a group of obstacles, case 2

Moving multiple units

During this test, two agents were moved from points A1 and A2 towards a point behind the forest. Figure 3-8 shows that first they ended opposite to each other, facing their target (points B1 and B2). However, a problem occurred when multiple units were moved to one location in a certain angle from their starting location. All ended in one place and covered each other (points C1 and C2).

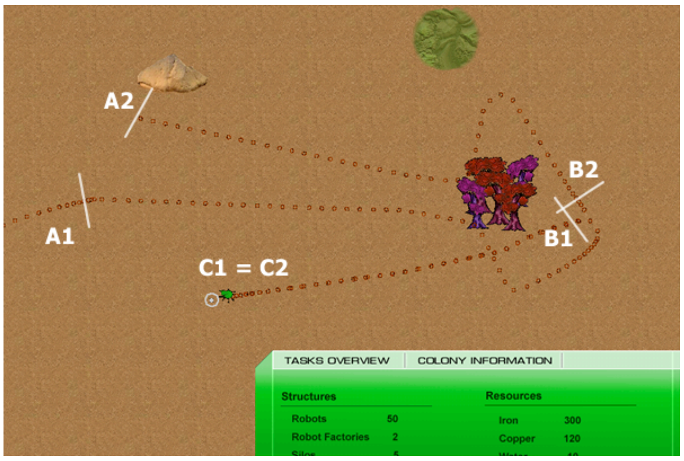


Figure 3-8 – moving multiple units

3.8. Minor Additions and Improvements

3.8.1. Steering Towards a Target When It Is Right at the Back of an Agent

In section 3.5. a target following algorithm was introduced. The technique is based on calculation of relative position of a target and then steering towards it based on a vector pointing to its location.
A problem occurred when a target was right at the back of an agent, i.e. the local vector's X attribute was 0. This resulted in no steering at all and an agent would go straight away from the target in attempt to find it.
The implemented solution tests whether a target is at the back when the X attribute is 0 by comparing the target's position with the position of an agent while taking agent's rotation into account. Please refer to Appendix J where you can find a detailed description of this algorithm.

3.8.2. Obstacle Avoidance

The testing presented in section 3.7. shows that the obstacle avoidance method used by Bourg and Seeman (2004) is suitable for single radial objects or a group of such objects which have a sufficient distance from each other. Agents managed to avoid the objects in each case. However, when it comes to differently shaped obstacles or a wall of objects, the repulsion forces are likely to get neutralised which results in agents taking a way across one of the obstacles.
Since there was not enough time available to implement a different obstacle avoidance solution, rules about creating the world have been established:

1. All obstacles are of a radial shape. This shape has to be apparent from all objects' textures.
2. Obstacles cannot be placed close to each other; certain space must be kept free around each. This needs to be true both when a terrain is created and when buildings are built by the player. Please refer to sections 4.1. and 6.1 where it is described how the system makes sure this is true.

3.8.3. Problem with Agents Covering Each Other

It was shown in section 3.7. that when two or more agents are moved to one location, their ending positions are the same if their paths cross at some point. This results in two agents covering each other which makes one of them not visible.
To overcome this problem, a simple algorithm was added to the agent's behaviour: when it reaches the idle state (i.e. ends its movement) it check against all agents which have a higher array position (a position in the CWorld's array of agents). If this is the case, an agent chooses a random location close to its original position. Appendix K shows implementation of this algorithm.

3.8.4. Agent Circles Around an Object If It Is Sent to Go Inside It

When a *GoTo* task was given to an agent and the target location was inside of a building or a terrain object, the obstacle avoidance algorithm prevented agent to reach its destination and it would circle around the object until it was given another task. The problem was solved by implementing an algorithm in the Motion engine which tests whether a currently avoided obstacle had target of a current action inside it. If this is the case, agent stops when it touches walls of the obstacle. This has proven to be an effective solution for the *GoTo* tasks as well as unloading (target is a silo) and gathering crystals (target is a crystal) tasks. It is not necessary to run this algorithm for the gathering grain task since agents are allowed to go through a farm. Appendix L shows the implementation of this algorithm.

3.9. Conclusion

In this chapter it was discussed how a 2D world can be created and how important it is for any further development to establish its coordinate system. It was shown how an artificial agent can be created and moved in this world, what problems came up during the implementation and how they were solved by adding extra algorithms to the system.

We chose an approach to obstacle avoidance based on extracting information about the world objects around. The used algorithm counts with radial objects and to optimise the performance, rules such as that objects cannot be close to each other were established. This was due to the downsides of the performed vector operations. If the calculations were improved to deal with any kind of shape of objects and a group of objects, the approach would be much more effective. Especially because there doesn’t have to be an abstract layer such as a mesh of waypoints stored in the system. An agent is able to respond to an ‘unknown’ environment simply by querying about surrounding objects’ information. The querying algorithm could also be improved by filtering the array of objects so that an agent doesn’t have to go through the whole array of obstacles but is able to select only the applicable ones.

4. Creating Strategy Game Features

This chapter will introduce the main strategy game features and how their development was done. In each strategy game there are certain resources and buildings which player can build using collected raw materials. Also, some kind of nourishment is required for units to live - Alien Farm uses grain and farms for units to gather it. Furthermore, a colony needs to grow its population. This is usually done either by ‘building’ units or letting units to breed. The project artefact uses the second approach since it is more realistic - each unit can multiply into a number of new ones. This takes both time and willingness to breed.

Saving and loading a game is also important, especially if the game play should last for more than 30 minutes. Players often need to go away and come back and without the ability to save their progress, a game can be rather frustrating. The important question here is how to represent the game world in a way which can be written into an external file and read back when it is needed. XML provides a good structure for representing any kinds of data and relationships between them. Nodes on trees can represent both concrete structures as well as logical groups. A number of XML parsers exist for C++ and the project artefact uses an extended version of open-source parser called TinyXML [L. Thomason, 2006].

4.1. World Objects and Buildings

This section will show how the terrain objects (hills, trees, swamps) and buildings were created in Alien Farm. There are two arrays held by the CTerrain class, one for the terrain objects and one for buildings. Both are subclasses of the CObject class but have different characteristics (please refer to Appendix G to see a UML diagram of the artefact). While the terrain objects are basically obstacles and have no special functionality, all buildings apart from a house can be interacted with. This includes a farm, a crystal field and a silo. The first two are used for gathering resources and the last one for unloading agent's cart (returning resources to the 'base').

There are three types of buildings user can build: house, farm and a silo. The construction panel shows all the buildings types and their costs in crystals (screen shot in Appendix H). Player can click on a building type which attaches the building to the mouse and then place the building on a map.

The icons of buildings are of the CInterfaceBuilding class (Appendix G). They have a Boolean m_bIsDragged member variable which allows for attaching a building to the mouse by being set to true when user clicks on one of the icons. When a building is placed on the terrain, user attempts to do so but doesn't have enough resources or cancels building by clicking back on its icon, the flag is set to false and the cursor is free again.

When user attempts to place an attached building on the terrain, arrays of buildings, terrain objects and agents are iterated through to ensure a building is put on a free space. To do this, the equation of a circle is used:

$$(x-a)^2+(y-b)^2 = r^2$$

Example 4-1: Equation of a circle [MMU Partnership server]

Where r is the diameter of an existing object, X and Y are the mouse coordinates (since the position of a new building would equal to mouse position if it was pressed) and A and B are existing object's coordinates. An extra value is added to the right side of the equation in order to keep space between objects (reasons for this are mentioned in section 3.7.2.)

4.2. Gathering Resources

The Behaviour engine is the main class which handles gathering resources (as well as all other tasks). When a resource (farm or a crystal) is clicked on, a new task is added to the beginning of the task queue stored in agent's memory engine. The Memory engine than deletes all other tasks from the actions queue and sets the gathering task as a current one.
Afterwards, the behaviour engine handles the current task. While gathering, the amount of a resource in agent's cart is increased each frame until it reaches a certain value. The speed of gathering is synchronized with the overall game speed. When the cart is full, the Sensors engine finds the closest silo and sets unloading the cart at this location as a new task. Also, the coordinates of the resource are remembered so that agent can come back to it once its cart is empty again.
When an agent reaches a silo, the unload cart task is executed. This involves decreasing the amount of a resource in the cart until it is empty. The amount of a resource is than added to

the colony's resources, i.e. member variables stored by the CWorld class. Gathering task is added to the beginning of the actions queue with the remembered location of a resource and the whole process repeats.

If a user chooses a different task for an agent, this is added to the beginning of the task queue and any existing gathering tasks are deleted.
Appendix M shows a graphical representation of the gathering algorithm.

There is a limit of one agent working on a farm and three agents working on one crystals field. This forces the player to extend the colony's territory and find new crystals. Each farm or crystal store an array of agents indexes working there. When an agent attempts to start gathering, it first queries the target building whether it is free. If not, it refuses to take the action and switches to the Idle state. If there is a free space, agent registers itself with the building and starts gathering.

4.3. Breeding

Each agent is able to breed, i.e. create up to five clones of itself. The number of clones depends on agent's current happiness. The agents are set to be hermaphrodites so only one alien is needed for multiplication. Breeding is very similar to collecting resources. The cart now acts as a container for counting how many days are left for an agent to multiply. Its initial value moves between 100 and 300 and depends on agent's current happiness. The days left to multiply decrease each frame so that 7 days equals 1 week on the time display at the bottom of the user interface.

When the amount of days is 0, agent multiplies - creates copies of itself by calling a CWorld method (Figure 23). To avoid having the new agents at the same place, i.e. covering each other they are immediately sent to the Idle state. The same algorithm which was used to prevent units ending their movement at one place is used (Appendix K). Before this algorithm was created, a different one was used to send new agents to a place further from their parent. Section 4.6. explains the differences.

Originally, an agent continued breeding until it had a certain amount of children. After observing various players testing the game, the feature has been modified so that the population is easily controllable. In the latest implementation version, agent goes to the Idle state immediately after producing the first population of children:

```
m_pAgent->GetWorld()->AddAgent(thisX,thisY,0);
m_pAgent->GetWorld()->GetAgent(m_pAgent->GetWorld()->GetNumAgents()-1)-
>GetMemoryEngine()->AddAction(0,0,eActionNone,-1,true);
```

Example 4-2 – adding a new agent to the world from the agent class and sending it to the Idle state

4.4. Using XML to Save and Load a Game

Simple txt files or tables can be used to store a small amount of data. However, working with 1D or 2D sets of data starts being difficult when a detailed structure of data (such as information about various aspects of the world) is required. Therefore, a different mean of grouping the information is needed.
XML (Extensible Markup Language) is a general-purpose specification for creating custom markup languages. It is extensible which means that user can define his own markup elements [Wikipedia XML]. In comparison with HTML which is a markup language for browsers, user can define any kind of document markup [TinyXML documentation] and define a structure suitable for the current situation. XML can be used for data transport (as it provides a common structure which can be interpreted by any application), creation of new Internet languages (e.g. XHTML, RSS, WAP, etc.) [W3schools] or by various other applications (like C++, Flash or Java applications) to store data in external files or provide means of scripting the behaviour of such applications (an example would be a Flash photo gallery which reads information about its images from an external XML file).

In Alien Farm, XML was used to save data about the current state of the world and objects inside it to allow user to save a game and load it later on. Five game slots were provided where each of them has an associated xml file stored in the game's installation directory. A XML parser had to be used in order to manipulate the markup using C++. Open source parser TinyXML [TinyXML main] is able to parse and XML document and build a Document Object Model which is a C++ object which can be manipulated and saved. [TinyXML documentation]. To use the parser, additional methods were required, including `GetNodeAt(parentNode, nodeIndex)`, `GetNumberOfChildren()` and `GetFileIsEmpty(fileName)` and methods for saving and loading the game data. Therefore, additional class CXmlHandler was added to the project. The XmlHandler provides an interface between TinyXML and other classes.

4.4.1. Objects Representation in a XML Tree

When a game is being saved, the XmlHandler class queries the CTerrain and CWorld classes about the game state, terrain objects and state of aliens and save this information in a tree structure. Individual branches of the tree represent object groups: terrain objects, buildings and units. Leaves on these branches represent the objects themselves and have certain attributes associated with these objects.
The tree begins with a gameInfo branch which stores all necessary data about the game state, including amount of gathered resources, current time and information about state of the god powers (Appendix O). Terrain objects and buildings form the following two branches of the tree. The leaves store information about their world coordinates. Terrain objects data also includes objects diameters and shape seeds (these two attributes are not necessary for the building objects as their values do not vary). Agents are grouped in the last branch of the tree. The individual sub branches are named according to agents and have attributes such as the world coordinates, age, an indicator whether an agent is dead and various attributes associated with agents' tasks. An agent branch has two leaves which represent the current and the last action. The attributes are a blue print of the Action object attributes (actions representation is discussed in section 3.2.).

The Load screen which can be called from the Main Menu gives a list of the available slots. All slot names are provided, however only the ones which have been used are clickable. The load screen stores an array of five button slots. In the constructor, each save slot is checked whether its corresponding file exists by using XmlHandler's `GetFileIsEmpty(fileName)` method. At the end of the FOR loop, the array holds Boolean variables which tell the system which slots have been used. The Render function than displays buttons on appropriate slots.

When a game is being loaded, the tree is iterated through by the XmlHandler class. First, the game information is read and the current game state is adjusted. Then the terrain objects and buildings are created using their leaves attributes. Finally the agents are placed on the terrain and their state is restored using the two action leaves. The last action is remembered by the Memory Engine and the current action is added to the task queue. The game state is changed to the Play State with a flag which specifies that the terrain has already been created and there is no need for creating a random map or additional agents. When a game starts, all objects are at the right place and agents restore their work where they left it.

4.5. Dying

Implementation of dying gave an extra realism to the game. All agents definitely die when they are 5 years old. However, they might die younger, based on current food store. The death age starts dropping down if the amount of food per alien is smaller than 60. If food per alien is 0, they all die.
When an agent dies it frees its current resource target (a farm or a crystal field, further details in section 4.2.) and is given a different texture coordinates in order to display a grave on its position (Appendix N). Also, its Update function doesn't include any of the algorithms needed by a live agent such as target following, obstacle avoidance and tasks handling. For a certain amount of time the Update function still runs, calculating how long an agent has been dead. When a given time is expired, agent is deleted from the CWorld's agents array and all agents stored after the agent's array index are moved towards the beginning of the array by one slot. The index position they remember is decreased by one. Also, the array of selected agents is adjusted in the same way and the system is notified about having less agents.

4.6. Testing and Evaluation

At this stage of development the game was played at least 10 times which provided an opportunity to try many various objects positions and terrain scenarios. Attributes such as gathering speed, amount of resources agents can carry, amount of clones they produce and time needed to multiply were adjusted to optimise the game play. The following screen shots display a chosen particular test situation each.

Gathering a resource

A selected agent was given a gathering task by right clicking on a farm/ crystal (point A1). It than returned the resource to the silo (B) and repeated the task (A2, B2, A3). The results were satisfactory since the agent continued gathering until it was given another task.

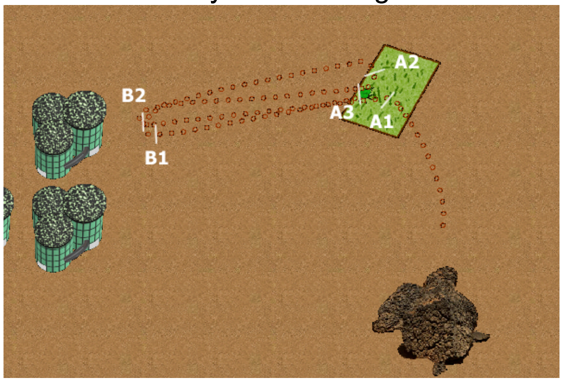


Figure 4-1 – agent given a farming task

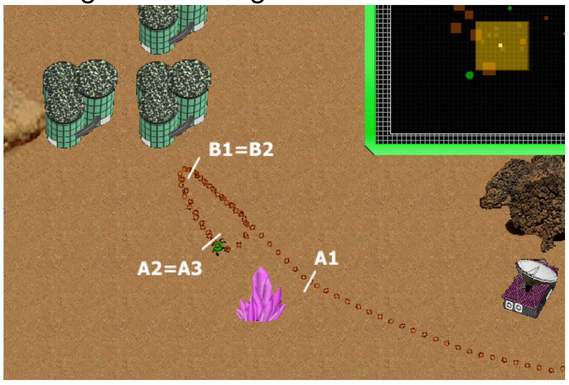


Figure 4-2 – agent given a mining task

Gathering a resource – multiple agents

When multiple agents approached a crystal (silo), each of them stopped in front of it based on where its movement started (a behaviour based on the algorithm described in section 3.7.4.). This prevented agents to group in front of a crystal (silo) at one place, but only when their initial positions were different. Figure 4-3 shows the behaviours of multiple agents (1-3): they all started at different positions (A1-A3) and moved towards the crystal (B1-B3). Then they returned the resources to the nearby silo (C1-C2) and came back to the crystal, now ending closer together (D1-D3). Figure 4-4 shows that after a number of iterations two orange units started gathering from the same location (circled) because of minor changes in their movement trajectories during the iterations. This behaviour prevents user to select the bottom agent while both are gathering.

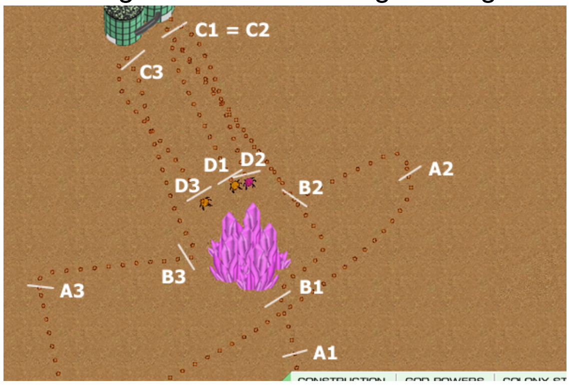


Figure 4-3 – multiple agents given a mining task

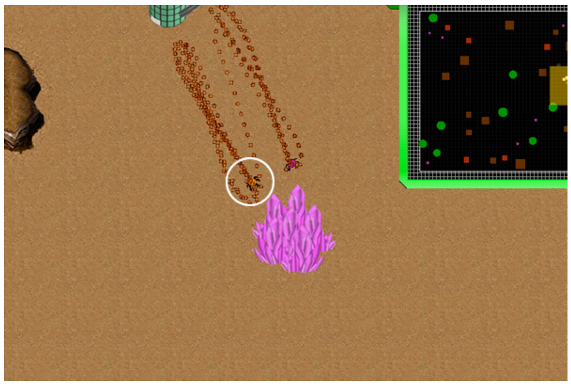


Figure 4-4 – multiple agent mining, after 5 iterations

A way around this problem would involve giving one of the agents a new task to go a bit further to one side and then resume gathering. This would make it gather on a slightly different place than is the position of the other agent. The solution hasn't been implemented in the final version of the game due to time constraints.

Finding the closest silo

During this test there were three silos on the map as well as three resource locations. Each agent was ordered to gather one of the resources (points A1-A3) and bring it back to a silo (points B1-B3). Figure 4-5 shows that all three agents found a silo closest to their location.

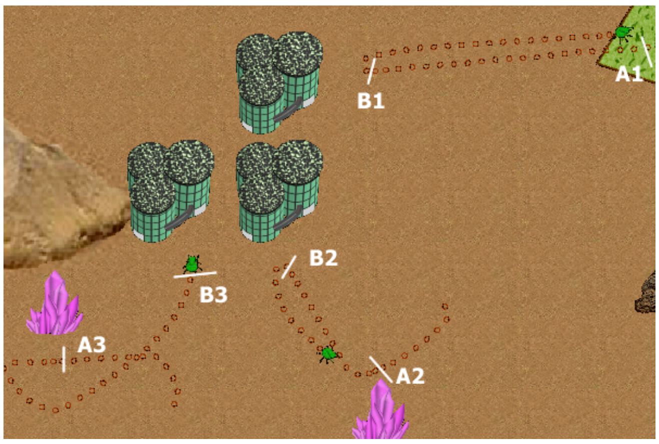


Figure 4-5 – finding the closes silo

Breeding

An agent was ordered to multiply, creating three clones of itself each time. Figure 4-6 shows placement of new clones in the earlier implementation version - the younger the generation, the further from the parent its members went when they were created. Each new agent was given a new position to go to according to the function

```
X = parentX+(1-m_iNumChildrenHad%3)*(30+10*m_iNumChildrenHad);
Y = parentY+(1-(m_iNumChildrenHad+1)%3)*(20+10*m_iNumChildrenHad);
```

Example 4-3 – old algorithm used to calculate a child's position after birth

where the `m_iNumChildrenHad` variable represents a child's number.

Figure 3-6 shows that no two children ended at the same place. This however produced three lines of new agents which was not an aesthetically sufficient solution.

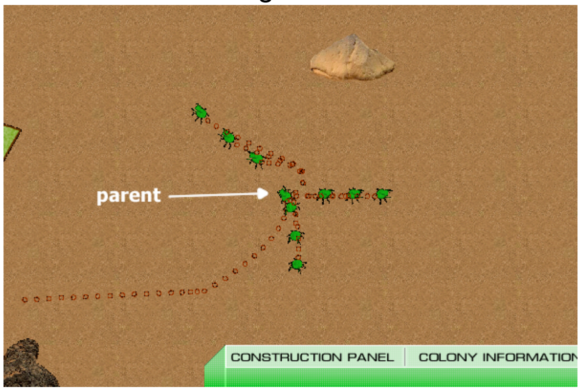


Figure 4-6 – initial implementation of breeding

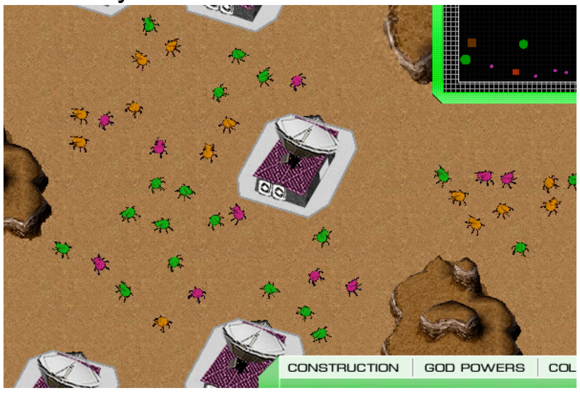


Figure 4-7- improved implementation of breeding

The second implementation was based on a more realistic behaviour discussed in Appendix K: when an agent is sent to the idle state, it chooses a random position to go to. This position is in a square with certain proportions around the original position but excludes the area straight under an agent (so that it doesn't end up at the same or a very similar place after using this algorithm).

Figure 4-7 demonstrates the great improvement in comparison with the first implementation – the positions of newly created units look much more natural. The image was produced by letting first three agents breed. Their children were cloned as well, without moving them. Although it took some time until each agent found a free position, all of them managed to avoid the others while keeping away from obstacles.

Dying

Dying times were tested by speeding up agents' own time speed by 50, i.e. every week counted as a year. Various initial food in store amounts were used to test against the dying conditions: starvation and age.

Saving and loading

Saving and loading has been tested at least 10 times by saving a game in various states to various save slots, checking the XML documents the programme produced, loading these games and playing them further.
Although initial results were good and the game was saved and loaded properly, additional branches and leaves had to be added as the development proceeded (e.g. information about the god powers or agent's previous task). Please refer to section 6.5.2. to read about further improvements which had to be done to provide a stable solution.

4.7. Minor Additions and Improvements

4.7.1. Recognizing Agents in Order to Keep a Record of Them

Each agent is given a random name from an array of possible names created in Agent's constructor and destroyed immediately after a name has been picked up. This helps the player recognize the agents.
Secondly, while the first implementation had only green aliens, further version of the game introduced additional two colours: orange and purple for a faster recognition. The agent texture was doubled in size to allow additional slots for the new colours. Please refer to Appendix N to read about how the texturing algorithm was changed.

4.8. Conclusion

In this chapter it was shown how the main body of Alien Farm has been created. Objects including hills, trees, buildings and resources were added to the World. Resources store information about the workers using them which prevents more than one alien to work on a farm and keeps the amount of miners close to one crystal under 4.
The task processing algorithm of agents has been explained – an agent is a finite state machine and uses the Memory engine to keep track of the current and the last task. The Behaviour engine utilises the cart and uses it when gathering resources as well as to calculate how many days there are left until it multiplies. Finally, each agent keeps record of its age and dies when it exceeds 4 years. However, agents can die earlier if there is not enough food in store.
We also had a look at how an XML tree structure can be used to represent data, particularly information about the world and units in it. XML has proven to be an ideal structure as it can group information to branches to provide a developer with tool easy to test. In the next chapter the aliens will be given a more human-like behaviour which will be based on their happiness and activity.

5. Implementing the Inner State

Most of the current strategy games have units which obey player's commands blindly and always in the same way. Therefore, emotions can be an interesting addition and help the player connect with the virtual more deeply.

Alien Farm uses happiness and activity to represent inner state of each entity. The chapter will discuss how fuzzy logic was used to evaluate the current state and how to use the inner state when adjusting unit's attributes and behaviour. Although fuzzy logic is quite a simple and straightforward concept, it takes a lot of play time and effort to optimise the way it is used by individual entities. There is no general or right way of doing this - each game system requires a unique setting. Important decisions have to be made by a developer in terms of fuzzy membership functions and weight of individual fuzzy rules.

Moreover, some kind of representation of what is happening and why units take their decisions in the way they do is needed. Many games with great AI fail to entertain because the player cannot figure out what exactly affects the units and the game becomes rather frustrating (Black and White 2 is a good example of this). We will discuss both how the current state of an entity is visualised and where the borderline between hints for the player and revealing the whole game logic is.

5.1. Types of Inner State Variables

The implementation version of Alien Farm after chapter 4 had a 2D world with some objects in it and agents which were able to do various tasks like gathering and breeding. Each agent took the same time to gather a resource, produced the same amount of clones, had the same movement speed etc. The reason for implementing the inner state was to make the way units performed given actions different and individual to each agent in order to make the game more interesting and more difficult to play. The CAgentInnerSensorEngine class was created to deal with fuzzy logic and calculate happiness and activity each loop. Two basic inner state variables were introduced: happiness and activity. They both reflect on what an alien has been doing over the last year (in terms of the game time).

Happiness

Happiness level changes based on what task an agent is given to do compared to what job it prefers. It also depends on how much room there is in colony's houses and how much food there is in store.
The emotion affects gathering speed, amount of clones an alien is willing to make and amount of days it will need to do so. This means that player needs to choose right units for a certain type of job – otherwise their work will not be as effective as it could be.
Also, the amount of food agent needs is affected.
Finally, if an agent is in the age when it should die, being above a certain happiness level can prolong its life until the happiness drops.

Activity

Activity represents how tired or active an alien is. The activity level drops when an agent is working or moving from one place to another. Movement speed is affected – a unit moves slower when it is tired. This means that if player doesn't make the aliens rest during their work, it will take much more time for them to drop resources to the silo. Activity recharges when a unit is doing nothing.
Activity also affects the amount of food an alien needs, by a higher amount than happiness does. This brings us back to the concept of making your units rest – player will need much less food in store if he doesn't let the activity of units drop down too much.
Please refer to Appendix P to see the overview tables of how the inner state is calculated and how it affects an agent.

5.2. Evaluating the Current Inner State

As it was mentioned above, fuzzy logic was used to calculate the happiness and activity levels. The important point of fuzzifying is that all fuzzy terms (e.g. sad, normal, and happy for happiness) have values which are calculated at the same time. Their degrees of membership can have values between 0 and 1 but the sum of their values can be greater than 1. The overall inner state level consists of degrees of membership to the fuzzy sets.

5.2.1. Happiness Fuzzy Sets

As Appendix O shows, happiness level is calculated based on agent's current job, number of houses and amount of stored food. Happiness has three fuzzy sets: sad, normal and happy. They depend on three factors; therefore membership to the fuzzy sets needs to be calculated three times. These are the steps taken to calculate membership to each of the 'happy' fuzzy set:

1. Calculate the individual degrees of membership based on current job, houses and food:

```
double jobWeight = FuzzyTrapezoid(jobXval,0.6,0.8,1,1.1);
double housingWeight;
if (aliensPerHouse == 1){
    housingWeight = 1;
} else {
housingWeight = 0;
}
double foodStoreWeight = FuzzyGrade(foodPerAlien,40,50);
```

Example 5-1 – calculation of individual degrees of membership for a degree of happiness

The jobXval value is calculated in a different function (Appendix Q) and is based on agent's current task, preferred job and time spent doing a job. The value is greater than 0.3 if the two match and increases with time. It is smaller than 0.3 if they don't and decreases when agent continues its work (Figure 5-1). This makes agents normal or happy if they do their preferred job. The time doing a job can be a value between 0 and 50 weeks. The value constraints have been established so that the happiness calculation is controllable. If the amount of weeks could be any number, there would be no way of establishing scale for happiness.

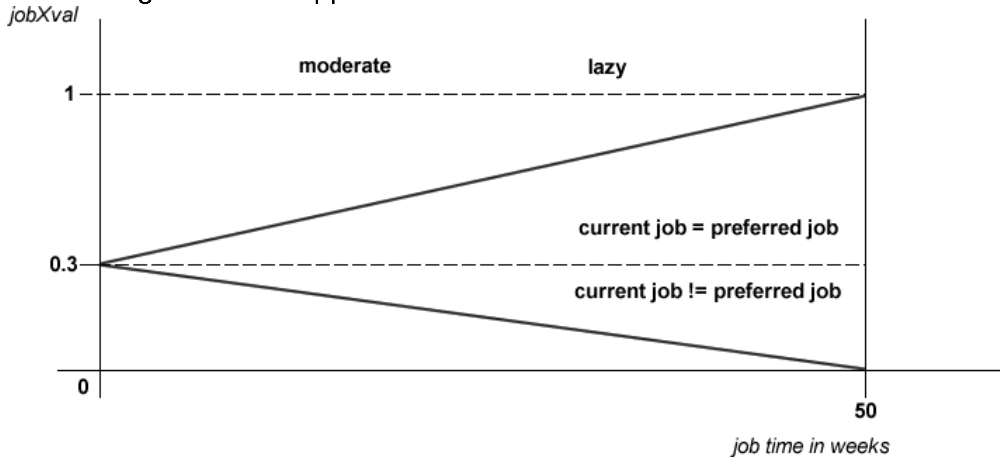


Figure 5-1 - calculation of the jobXval value

Appendix R shows the membership functions of the individual factors. The fuzzy trapezoid and fuzzy grade function are based on [D. M. Bourg and G. Seeman, 2004, p. 194-197]. The result of each is a value between 0 and 1.

2. Weight the results in order to get the final membership to a fuzzy set, based on the table shown in Appendix P:

```
happyDeg = 0.5*jobWeight+0.25*housingWeight+0.25*foodStoreWeight;
```

Example 5-2 – calculating the resulting happiness degree

The equation in Example 5-2 ensures that even though the happy degree consists of three factors, its value will stay between 0 and 1.

The other two fuzzy sets (normal and sad) are calculated in a similar way. Appendix Q shows full bodies of the fuzzification functions.

5.2.2. Activity Fuzzy Sets

The activity consists of three fuzzy sets: active, moderate and lazy. Similarly to happiness fuzzy sets, their values are calculated as a weighted average of fuzzified values based on factors age and movement time. The movement time can have a value between 0 and 50, similarly to the time spent in a job (discussed in 5.2.1. above). Appendix Q shows the membership functions.

During the game play, the happiness tends to change in levels, i.e. the movement of the happiness bar is not smooth. This is because trapezoids were used to calculate the happiness. On the other hand, the activity tends to drop down smoothly when a unit moves. Appendix R shows the difference in activity membership functions – their shape is gradual, therefore the value goes down and up smoothly.

5.3. Impact of the Inner State

After the Inner Sensor class fuzzifies the activity and happiness, other functions in other agent's classes use the values to adjust the agent's behaviour.

5.3.1. Motion Speed

The motion is speed is calculated by the agent's Motion Engine. In primary implementations the motion speed was fixed. Later, an algorithm was added so that the speed varies based on the activity level. To adjust the motion speed, one activity value is needed instead of three membership degree values. The value is then used in an equation for calculating speed. The Inner Sensor engine is able to return *activity weight,* i.e. a number between -1 and 1 based on the three activity fuzzy sets. Appendix S shows how the value is calculated. The algorithm is based on the defuzzification principle discussed in section 2.4.3. However, outcome of this algorithm is not an exact number needed for calculating motion speed. Instead the value is normalised between -1 and 1 so that the function can be reused in different places.
Example 5-3 shows how the motion speed can be calculated using a defuzzified value of activity. The speed can have a value from the range <0.6;1.2>.

```
double temp = m_pAgent->GetInnerSensorEngine()->GetActivityWeight();
m_fSpeed = 0.9 + temp*0.3;
```

Example 5-3 – using the activity weight to calculate motion speed

Furthermore, an agent can get to the 'crazy' state which means that the movement speed is higher but the obstacle avoidance is less effective. This happens if agent is sad but has high activity level. Fuzzy rule with the AND axiom [D. M. Bourg and G. Seeman, 2004, p. 200] is used to test whether this behaviour should be triggered.

5.3.2. Work Effectiveness

The Behaviour Engine is responsible for processing a current task. It queries the Inner Sensor engine about the current happiness in order to adjust the farming and mining speed. Similarly to how the Motion Engine calculates the speed, Behaviour engine uses *happiness weight* to determine the gathering speed. Based on the way fuzzification works (section 5.2.1.) an agent gathers faster if it likes the job it does and if it works for a longer time. On the other hand, doing unpreferred job type slows down the gathering speed.

There are two parameters which are affected by the defuzzified happiness when it comes to breeding: a number of clones agent is going to produce and the amount of days needed to do so. Both are calculated when the breeding task is given to an agent and don't change during breeding. However, happiness level still drops or goes up (based on if the agent likes the task or not) which affects the two breeding parameters if the task is given to an agent again. This means that if a player identifies agents which like to breed, giving them this task again and again will speed up the population growth.

5.3.3. Amount of Food Needed

The amount of food agent needs per week depends both on happiness and activity. Therefore, both weights are considered. Example 5-4 shows the calculation of the combined weight.

```
double activity = m_pInnerSensorEngine->GetActivityWeight();
double happiness = m_pInnerSensorEngine->GetHapinessWeight();
double temp = - 0.4*happiness - 0.6*activity;//-(0.4*activity) +
0.6*happiness;
m_iFoodPerWeek = ceil((2 + (temp*1)));
```

Example 5-4 – calculating the amount of food needed each week

5.3.4. Life Length

As it was mentioned in section 4.5 above, the death time depends on agent's age and current amount of food in store. After the Inner Sensor engine was implemented, additional rule could be applied to the death time which keeps a unit alive if it is happy. If other rules determine that the agent should die, the *happiness scale* is tested:

```
if (m_pInnerSensorEngine->GetHapinessScale() < 0.6)
```

Example 5-5 – testing the happiness scale

The happiness scale can have values between 0 and 1. The defuzzification principle is the same as with the happiness weight (Appendix S), however the values are of a different range. This is simply to make the condition more understandable.

5.4. Representation of the Inner States

The inner state of an agent has a major impact on its behaviour. Therefore, it is very important to let the player know about the current state, preferably in more than one way. Alien Farm units change their speech and sounds according to their happiness and have little state bars shown next to them when they are selected.

5.4.1. State Bars

Figure 5-2 shows the unit's happiness (green) and activity (red) state bars when the happiness is moderate and activity high. The state bars are updated each frame to reflect the current state. Similarly to calculating if conditions are met for an agent to die (section 5.3.4), calculation of the bars' length uses *happiness and activity scales* (i.e. numbers from range <0;1>)

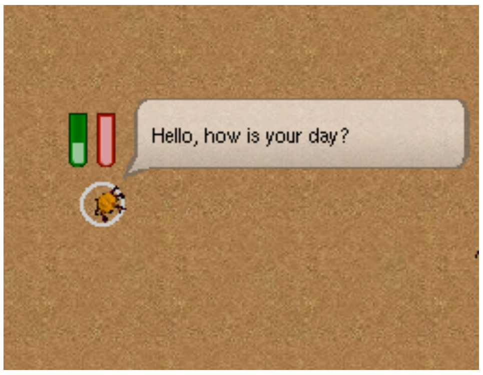


Figure 5-2 – agent's state bars and speech bubble

5.4.2. Speech

Figure 5-2 shows the agent's speech bubble as well. During the earlier implementation, the text was used for reporting what task agent was given or what task it was performing. Sentences like 'I am going to gather some grain' or 'I am going to produce 3 clones in 90 days" were used. The sentences were stored in a 2D array and referenced according to integer value of the current task and the task seed (implemented for making the text vary for the same task type). The value was obtained by converting the task's name of enumeration type into integer.
In the latest implementation version, the speech changes according to agent's happiness. Another dimension was added to the speech array which represents the happiness level. The algorithm which selects the right sentence looks both at the current task and happiness level and forwards an appropriate message to agent's speech bubble which displays it. Please refer to Appendix T to read more about the mechanism of initiating the speech array and displaying the right sentence.

There is a 'shout' feature implemented in the agent which is triggered when a different than usual message needs to be displayed in the speech bubble. This is used when an agent wants to directly respond to the player's interaction with. For example, when an agent is sent to gather grain but then it discovers the farm is already occupied by someone else, its state is set to idle. However, rather than displaying a normal idle message it informs the player that the farm is occupied. Such a message is displayed both of a unt is selected or unselected until the agent is given a different task or is selected again. Also, the bubble disappears automatically after couple of seconds if the agent is not selected till then.

5.4.3. Sound

In strategy games, units usually play a sound when they are selected or given a task. Alien Farm implements the same feature, however the sounds were selected so that user can identify when an agent is irritated (sad). Normally agent is able to play three sounds which are selected randomly. Another set of sounds is used when the happiness scale drops below 0.3. (i.e. an agent is sad).

5.5. Testing and Evaluation

The following tests were done during the implementation process. A number of iterations were used to produce the final version of the current system and a number of minor adjustments were done during this process.

First, the membership degrees of all fuzzy sets were hard-coded into the Inner Sensor Engine to test the inner state representation. All possible actions were given to an agent for each of the degrees in order to test if the text in the Speech bubble is displayed correctly and whether the state bars have a correct length.

When it was clear that the inner state is represented appropriately, the hard-coded values from the Inner Sensor Engine were taken off and replaced by correct numbers during the game play.

Debug information was displayed by the game interface for each selected alien, listing the values of fuzzy sets membership degrees and of factors which affect them, including weeks spent working, preferred job, weeks spent in movement, etc. A number of games were played and the debugging information was compared to what was happening in the game. This way, the reactions of the inner state could be compared to the game play in real-time.

Once it was sure that the inner state changes correctly, the focus was moved on what the inner state affects – movement speed, gathering speed, etc. These values were also a part of the debug information. The effect of happiness and activity on various behavioural factors was changed a number of times until the results were satisfactory and the game play was smooth, but not too easy.

The debug information was hidden after the testing was completed but can still be turned on by altering the agent's code in case that further adjustments or additions will need to be done in the future.

5.5.1. Inner States and the Colony Information Panel

The Colony Information panel at the bottom of the interface displays the overall happiness and activity (apart from other information).
Testing overal happiness value: all agents were doing what their preference was therefore there were all 100% happy. The average happiness was 100%.
In a different situation, one agent out of 3 was not doing the right job and its happiness went down to 0% while the happiness of others was 100%.The expected and displayed average was 66%.
This was done with 3 and 6 agents.
The same applies for testing the overall activity.

5.6. Conclusion

Implementing the inner state into agents made them behave differently from each other and react on tasks they were given based on their individual preferences. This made the game play much more interesting and managing the colony harder. The main reason behind this was to address the current problem with almost all strategy games – units do what player makes them to but do not display any kind of impact on them.

The inner state consists of two variables: happiness and activity. Both have three fuzzy sets associated with them. The fuzzification process takes place in the agent’s Inner Sensor Engine. The engine is able to provide defuzzified values of happiness and activity to other classes like the Motion or the Behaviour engines. They use the values in order to adjust agent’s behaviour.

The shapes of fuzzy membership functions have a great impact on the resulting values and need to be chosen carefully. Although fuzzy logic is relatively easy to implement, it takes a lot of time to modify both the fuzzification and defuzzification processes in order to get a playable game and interesting behaviour.

In the next chapter, we will have a look at how Alien Farm was completed and how players reacted on the unstable behaviour of the units.

6. Completing a Strategy Game

Once all the functionality is in place, there is time to make something out of it. Players want a variation first of all - they really want to play a new unique game when they click on the ‘New game’ button. The chapter will provide information of how a random map is constructed and what constraints are there when doing so.

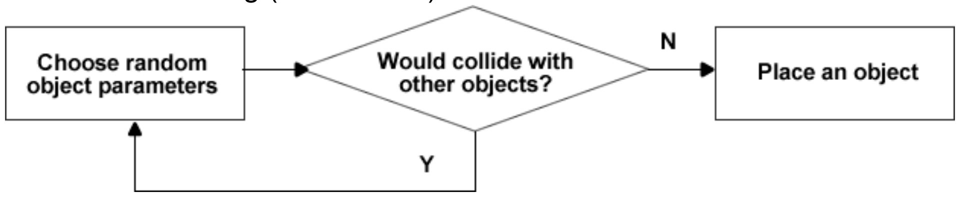
The most important aspect of each game is winning and getting rewards for what you do. No one will play a game unless there is a sense of achievement and clear objectives are specified. Alien Farm has a main goal of the game but also gives the player rewards in forms of god powers when they achieve something non compulsory along the way.

Testing a game before its release is also inevitable. A developer knows what to do and understands the game completely, but how will other people react to the environment and will they be able to find everything they need? The final sections of the chapter will report on testing the game by other people and will list the final adjustments which needed to be done in order to complete the artefact.

6.1. Random Map Creation

As it was mentioned in section 3.7.2., the obstacle avoidance algorithm has a few flaws including agents not being able to avoid walls of objects and obstacles of other than radial shape. Therefore, these constraints needed to be incorporated in the algorithm for creation of maps.

When a game starts, the CTerrain creates terrain objects (rocks, swamps and trees) and crystals. These have a texture which reminds of a circular shape and are understood as circles (they have a centre and diameter). Figure 6-1 shows how the algorithm which places new objects works. First a random position from ranges <-2000; 2000> for X and <-1500; 1500> for Y and diameter (from a certain range individual to an object type) is chosen. Than arrays of existing objects are iterated through and the newly created object parameters are tested to make sure the object will be placed on a free space, similarly to the test performed when player attempts to build a building (section 4.1.).


Choose random object parameters
Would collide with other objects?
N
Place an object
Y
Figure 6-1 algorithm for creating a random object

However, while in the case of building the equation of circle is used (since only a point, i.e. mouse location needs to be tested against), here both objects are compared based on the bounding circles collision detection. Example 6-1. shows the algorithm which compares one object to another. If the function returns true for any of existing objects, a new random position is chosen and the process repeats until a free space for the new object is found. The advantage of choosing another random position rather than just moving an object sideways is that there is less chance that the objects will be next to each other and the map looks more random.

```
HitTestBothSpaceAround(int x_, int y_, int w_, int extraSpace_){
    bool hit = false;
    //------------- calcuate real distance of 2 centres:
    float deltaX = abs(((x_) - (m_fWorldX)));
    float deltaY = abs(((y_) - (m_fWorldY)));
    //--- calculate distance when touching, add extra space to each
radius:
    float distance = sqrt(deltaX*deltaX  + deltaY*deltaY);
    float sumDiam = (w_/2+extraSpace_) + (m_fw/2+extraSpace_);
    //-------------- compare:
    if (distance < sumDiam){
          hit = true;
    }
    return hit;
}
```

Example 6-1 - hit test of object with an object

Three units are placed on a fixed location so that they are concentrated around the centre of the map. Therefore, when terrain objects and crystals are being created the algorithm also checks they are not inside a square of size 120x50 with its middle in the centre of the map:

```
if (newPositionFree && abs(x)-(r/20) > 60 && abs(y)-(r/2) > 50){
    ... // create an object
}
```

Example 6-2 - test against a free space in the middle of the map

The above described implementation of random map creation was very satisfactory since it was able to create maps where agents were able to avoid obstacles mostly effectively. Each map is completely new which creates interesing environments. Appendix U shows 10 examples of randomly created maps.

6.2. Rewards to the Player

Important part of a game play is not only trying to reach the main goal but also to receive rewards as you progress. Therefore God Powers were added to the game to give player an advantage if he accomplishes a side objective. The objectives are listed in the Objectives panel at the bottom of the interface. The following is a list of God Powers:

God Power	**Condition**	**Effect**
Silo	Have 15 aliens	Builds a silo for free
Energy boost	Accumulate (different from gather) 500 crystals	All units have a maximum activity for 30 weeks
Happiness boost	Build 40 houses	All units are 30% more happy for 20 weeks

The following additions were done to the CStatePlayClass in order to make the god powers work,:

1. a Boolean array which flags powers already won – this ensures that a power is not activated twice
2. allHappyCount and allActiveCount float variables which are first set to 0. When a God Power is used they are set to 20 and 30 weeks. Afterwards, the counts decrease each week and the Happiness and Energy boosts work until the value of these floats is 0 again.

Also, the CPlayInterface class has a record of unlocked powers, i.e. powers which were won. This ensures that when a game is saved, the record of available powers is saved as well and loaded later on.

6.3. The Win and Lose States

The most important part of any game is winning. In Alien Farm you can win by having a population of 30 and keeping the overall happiness of aliens higher than 70%. This means player has to keep expanding the colony (have more aliens being born than dying), distribute the tasks so that agents do their preferred jobs, build enough houses and have enough food in store to keep them happy.

The CStatePlay class tests whether these two conditions are met each frame. If so, the game is paused and a dialogue about winning the game is displayed. Player can start a new game or exit. Similarly, the class also tests whether all agents are dead. If this is the case, a dialogue about losing the game is displayed and player can restart or exit.

6.4. Testing and Evaluation

6.4.1. The Game Play Length

A target was to make the game play longer than 20 minutes. At least 5 games were played and the following factors mostly affecting the game length were identified:

- speed of breeding
- number of clones agent produces
- target population

These factors were adjusted in order to increase the game play until the results were satisfactory.

6.4.2. Game Play and Interface

Evaluation results in the following sections are based on 5 other players testing the final game implementation. The age of the game testers was between 16 and 25 years. They spend between 4-50 hours a week playing games, mostly first person shooters, strategies and simulations. They were instructed to play the game between 5-10 times and then fill in a questionnaire where they could rate individual features, game AI and the interface. Also, they were able to comment on various aspects of the game. Please refer to Appendix V where the individual questionnaires are presented.

The tutorial usefulness scored somewhere in the middle – the interface was described sufficiently, however, important bits like where to build buildings and why units die were missing. The players generally knew what to do in the game provided that they read the tutorial slides. There was one tester who was not sure what to do - according to his words he didn't see the tutorial at all.

The stability of the game was rated well, however testers who used the save/load features reported that the game would crash after a while when a scenario was loaded. The solution of this problem is described in section 6.5.2.

6.4.3. Agents' AI

The path finding was rated generally well – there were only a few occasions when agents would wander around trying to find their way. They didn't bump into obstacles most of the time.

The behaviour of units (gathering speed, breeding, sounds and speech) was rated of moderate diversity. This result is satisfactory, however better score was expected. The testers considered the diversity good because it made the game more interesting, but they complained about inability to plan their actions (the amount of new clones differed, when the obstacle avoidance failed it was difficult to get a unit to a farm which resulted in loosing food).

Also, it was sometimes difficult to figure out the job preference of an agent. This problem was not addressed as the point of the game is to try and find out what the aliens want.

The general impression of the game seems to be good and a number of testers said that they liked the diversity in behaviour of units. Section 6.5. lists improvements which were done based on the questionnaires in order to make Alien Farm a more enjoyable experience.

6.4.4. Hardware Requirements

The game was created on a PC with the following specifications: AMD Phenom II 2.80 GHz, 3GB RAM, NVidia GeForce 7600 GS. It was tested by the developer on a laptop with AMD Turion 64 800 Mhz processor, 1GB of RAM and ATI Mobility Radeon X700 graphics card.
The laptop ran the game well so its specifications were listed in the ReadMe file as minimal hardware requirements. However, one of the testers with a stronger PC reported that he could not play the game because it was too slow. Another person reported that the game used approximately 50% of 3GHz Athlon X2 CPU.
The main reason which slows the game down is the Update function of an agent where all agent engines do their work each loop. The CPU requirement goes up with the amount of units. Further tests would need to be done in order to specify the hardware requirements more precisely. Also, the obstacle avoidance algorithms could be made more efficient by filtering the array of tested objects, but the time constraints didn't allow for this.

6.5. Minor additions and Improvements

6.5.1. Tutorial

Since the testers said that they could not understand why some things were happening, additional information was added to the tutorial slides. The list of additions is the following:

1. inform about building restrictions - objects cannot be built close together and farms have to be built on swamps
2. inform the player that the aliens need to eat every week, therefore food needs to be collected all the time
3. explain that the number of clones and number of days needed to multiply vary based on an alien's current happiness
4. explain what the state bars are for
5. let the player know how activity affects the movement

6.5.2. Crashing Problem with a Loaded Game

All the testers complained about the game crashing after a few minutes when the loading function was used. By investigating the problem, it was found out that the XML tree was missing inevitable information: it stored the number of agents but it didn't store the amount of live agents. The loading algorithm therefore set the amount of live agents equal to the amount of all. This was causing an array-out-of-bounds exception when a dead agent was

removed after a while of displaying its grave. The problem was fixed by decreasing the number of live units by one for each agent which was flagged as dead.
Another flaw of the loading process caused an agent to pass its action's target array id as the minimum integer value. This caused a run-time error when the agent was freeing its target resource (the agent was dead already). The exact reason of this problem could not be found. However, an extra IF statement was added to the CTerrain's method which returns a building based on its array position - if the position is smaller than zero, the method returns the first building in the array (being one of the crystals since they are always constructed first). When a dead agent tries to free the building where the array position is smaller than zero, it means that it didn't work with this building anyway (the position would be greater than 0 if it did). When it sends a request to the 1st building about stopping working there, the algorithm first checks whether agent works there already and proceeds only if this is true. Since the method will never return true in this case, the solution of this problem does not affect the building or other agents anyway.

6.5.3. Making of an Installation File

When the game was first tested on a different PC, it wouldn't run because of missing OpenGL libraries it uses. This was solved by making an installation file which installs the game on a location selected by the user and also copies necessary libraries into the Windows directory. If the libraries already exist, the setup programme asks the user what to do. The libraries are not uninstalled with the game.
Install Maker 1.2. was used to create the setup file. Install Maker is a freeware created by Clickteam in 1999.

6.5.4. The Game Speed

Usually, different machines have different CPU speed which means calculations are done in different times. Most of all, this affects movement speed of objects. The way this problem is be overcome is by measuring delta time between programme frames and using a constant delta time in movement calculations, for example when a certain number is being added to object's X position each frame.
Even though the CTimer class was used for measuring the delta time and the float dt variable was used in calculating agent's speed, the result was that on different machines it travelled at a different speed. The reason for this could not be found.
To deal with this problem an integer global variable GameTimeSpeed was used. The CGame class still uses delta time to calculate the game speed. However, the GameTimeSpeed value is calculated only by the CGame class and is than picked up by other classes to perform calculations.

```
g_iGameTimeSpeed = 100*dt;
```

Example 6-3 - calculating the game speed

This also means that the Update function of all classes doesn't need to have the extra delta time parameter. Instead, classes use a global time speed which makes creation of new classes faster.

6.6. Conclusion

This chapter described how Alien Farm was completed as a strategy game. A map creation algorithm was developed, following the obstacle avoidance constraints (having circular objects, avoiding objects to be close together).
Player is given rewards in form of God Powers for completing side objectives. This makes the game more interesting because there is more than one target to achieve. The God Powers give advantages to the player like being able to build a free silo and raising the energy and happiness level of agents for a certain time. The main target of the game is to have a certain population and overall happiness level. The objective was adjusted during alpha testing in order to optimise the average game length.
After the implementation was finished, five people tested the game and filled in an evaluation questionnaire. Alien Farm scored generally well, although there were some problems with the information provided and the save/load feature. All these problems were addressed to deliver a final improved game.

7. Conclusion and Future Work

The project consisted of research and experimentation with the currently used games AI techniques including obstacle avoidance in a continuous environment, finite-state machines, task-oriented behaviour, task processing and fuzzy logic.

The project artefact is a 2D strategy game called Alien Farm. Units in the game are able to collect resources and breed. To move they use an obstacle avoidance algorithm for continuous environments based on Bourg's and Seeman's work (2004). The algorithm is mostly able to deal with single radial obstacles which put restrictions into the creation of maps for the game. The units are finite-state machines which have three basic states: idle, moving towards and performing. They use a number of engines which process various tasks, most importantly the Behaviour and the Motion engines.

Units have an individualised inner state which reflects their recent actions and whether they liked them or not. The inner state consists of happiness and activity and affects various aspects of the behaviour, including work effectiveness and the movement speed. Fuzzy logic was used to evaluate the inner state and to adjust the attributes of units. The result is individualised behaviour which makes a difference in game play compared to other strategy games. Player needs to monitor happiness and activity of the units in order to win the game. Right units need to be picked up for certain types of jobs, otherwise their work is not effective.

A number of people tested the game and said they found the individualised behaviour interesting but harder to control. The project proved the point that making units behave more like humans is a positive addition to a game. Further adjustments and additions which could be added to Alien Farm would surely make it a popular commercial game.

Additional features could include more types of structures and resources which would create more tasks for the aliens to do. Instead of displaying the speech bubbles, voices could be recorded and played to make the game look more professional. Also, there is space for adding extra game features like able to 'play' with the aliens in order to raise their happiness for a certain amount of time. This feature was in fact planned to be implemented but the time constraints didn't allow for it.

Probably the most interesting addition would be to implement a colony of NPC units or a multi-player mode playable over the network. In both cases, it would be interesting to create units which like to go to war and are very efficient when fighting in comparison to others. However, they would probably have to have a disadvantage like slow movement or higher consummation of food in order to balance the game.

Another point which this work made is that fuzzy logic is very good for evaluation of the current situation since it helps to split classification of the input from calculation of the output. This is very useful in an open-ended project like this since adding additional parameters and behaviour is made easier.

Various parts of the Alien Farm code could be re-used for other projects. This includes especially the Behaviour engine which is independent from the classes outside the Agent's boundary. Also, the Inner Sensors engine provides useful fuzzy logic functions and methods able to pre-defuzzify values for usage by other classes.

Another useful class is the XmlHandler which is an extension of TinyXML, a set of open-source libraries for handling xml documents. [TinyXML]. The Handler class provides useful functions for work with xml tree nodes, is independent from other project classes and provided as a singleton.

This project was useful to the author for two reasons: firstly a broad research into games AI needed to be undertaken in order to deliver the project artefact. Secondly, project management and software development techniques like creating a project plan and using iterations of testing and implementation had to be used. This provided a solid ground for future development work. The project did not go well all the time - some features needed to be taken off the plan so that the final artefact was delivered as a complete packaged solution and on time. However, the main features, most importantly the inner states, were implemented successfully and all project aims and objectives have been satisfied.

References

AI Depot: http://ai-depot.com/articles/artificial-intelligence-in-games/

AI Game Dev page A: aigamedev.com/site/best-of-2007

AI Game Dev page B: aigamedev.com/reviews/top-ai-games

Bourg D.M., Seeman G., 2004. AI for Game Developers. 1st edition. O'Reilly Media Inc.

Blobs in Games: simblob.blogspot.com/2005/12/black-and-white-2-ai.html

DaniWeb: http://www.daniweb.com/forums/thread68587.html

Gamasutra page A: www.gamasutra.com/php-bin/news_index.php?story=16798

GameAI page A: www.gameai.com/polls/061703_081503.html
GameAI page B: http://www.gameai.com/polls/010701_020501.html
GameAI page C: http://www.gameai.com/polls/100101_110901.html

Game Dev: www.gamedev.net/community/forums/topic.asp?topic_id=441493

Game FAQs: http://www.gamefaqs.com/computer/doswin/data/11255.html

GameZone: www.gamezone.com/news/06_07_02_04_18PM.htm

Gaming Today: http://news.filefront.com/half-life-2-episode-2-release-date-confirmed-october-9/

Ikeda Kenji's web site: http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml

Laird J.E., Lent M., 2001. Human-Level AI's Killer Application. AI Magazine, 22: 15-25

Lee J.R., Williams A.B., 2004. Behaviour Development through Task Oriented Discourse. Computer Animation and Virtual Worlds, 15: 327–337

Levy S., 1993. Artificial Life: The Quest for a New Creation. Penguin
Liu B., Choo H., Lok S., Bona S., Lee S. C., Poon F. P., Tan H., 1994. Finding the Shortest Route Using Cases, Knowledge, and Dijkstra's Algorithm. National University of Singapore. IEEE Intelligent Systems, October '94, p. 7-11

MMu partnership server: http://www.partnership.mmu.ac.uk/cme/Geometry/CoordGeom/Circles/CirclesEqns/EqnsCircles.html

Planet Crap: www.planetcrap.com/topics/288/

Ringdahl O., 2003. Path Tracking and Obstacle Avoidance Algorithms for Autonomous

Forest Machines. T. Hellström: Umeå University.

Shack News web site: http://www.shacknews.com/onearticle.x/55781

Slashdot forum: ask.slashdot.org/article.pl?sid=06/05/09/2251228

Spronck P., Ponsen M., Sprinkhuizen-Kuyper I., Postma E., 2006. Adaptive game AI with dynamic scripting. Machine Learning, 63: 217–248

Thomason L, 2006. Tiny XML for C++. http://www.grinninglizard.com

TinyXML: http://www.grinninglizard.com/tinyxmldocs/index.html

University of Georgia web site: http://www.cs.uga.edu/~potter/kbs/AOM-Quick-Start.ppt

W3schools: http://www.w3schools.com/Xml/xml_usedfor.asp

Wikipedia Games AI: http://en.wikipedia.org/wiki/Game_artificial_intelligence
Wikipedia Dijkstra: http://en.wikipedia.org/wiki/Dijkstra's_algorithm
Wikipedia Halo 3: http://en.wikipedia.org/wiki/Halo_3
Wikipedia WOW: http://en.wikipedia.org/wiki/World_of_Warcraft#Ongoing_gameplay
Wikipedia XML: http://en.wikipedia.org/wiki/XML

Appendices

Appendix A: Games that brought innovations into AI

Diablo

Blizzard released the first Diablo in 1997. Focus of AI developers included areas like response time of NPC reactions, path finding and knowledge acquisition. Diablo implemented dynamical interaction with player and extended virtual world [J. E. Laird and M. Lent, 2001].

Black and White

The game was released by Lionhead Studios in 2001. It implements machine learning (you have a creature pet which can be trained) and A-life (evolving world of NPCs) [AI Game Dev, page B]. Also, the way opponents attack your base is very realistic and forces players to change their strategy throughout a game [Blobs in Games].

Age of Mythology

Age of Mythology was developed by Ensemble Studios and published by Microsoft Game Studios in 2002. Apart from being a great strategy with opponents which adapt to your game, the developers added scripting which makes the AI even more interesting. You can use the XS (Expert System) language in order to script behaviour of your opponents [University of Georgia web site] which is perceived as a great advantage in today's games [GameAI page B].

The game won a number of awards including IGN PC's 'Best Strategy Game award' and 'Best of Show Real-Time Strategy' from Wargamer [Game Zone].

World of Warcraft

This game was released by Blizzard Entertainment in 2004 and became a hit considerably fast. Most of the game play is based on receiving quests from NPCs that usually involve killing a number of monsters or finding items [Wikipedia WOW]. Behaviour of NPCs is realistic in a way that opponents tend to chase you if you start running away, but only if they have enough health. This makes them seem as if they were afraid to die but still wanted to kill a player if possible. Also, enemies create invading groups if they are close together and you attack one of them.

According to Shack News [Shack News web site], US Army is considering using MMO games like World of Warcraft for training purposes.

Half-Life 2 Episode 2

The game released by Valve in 2007 [Gaming Today] received 'Best AI in a Mainstream Game' reward from Gamasutra, a web site which specializes in AI in the same year. The main improvements to AI were highly interactive NPCs with scripted behaviour as well as high level of coordination between player's enemies [Gamasutra page A]

Halo 3

Bungie released Halo 3 in 2007 exclusively for Xbox 360 [Wikipedia Halo 3]. The game players voted for this game to have the 'Most popular AI in a game' in 2007. The reasons were interesting group behaviour (group of NPCs behaves differently when its members die) and AI's awareness of combat mechanics [Gamasutra page A].

Appendix B: A tile-based vs. continuous environment

Tile-based

A tile-based environment consists usually of squares with constant size which cover the whole map. Position of an object is usually represented as row and column numbers. Movement is done in a way that one tile represents one step and an object can move in one of the 8 directions: north, north east, east, south east, south, south west, west and north west. Figure AP1 shows the implementation of such terrain type in a game. The triangle represents current position of a NPC and darkened areas stand for obstacles.

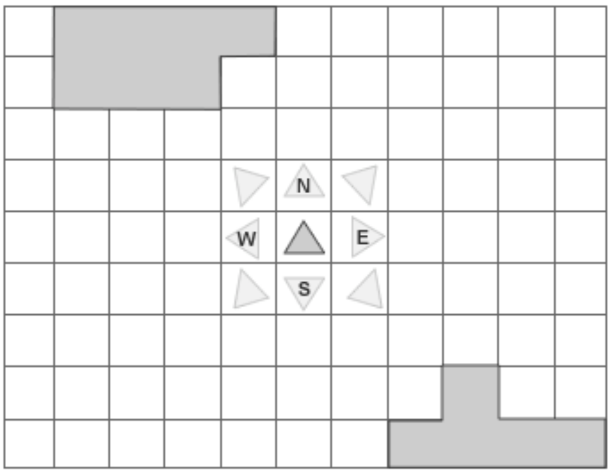


Figure AP1: Tile-based environment

Continuous

A continuous environment is not divided into tiles and position of objects is represented by real numbers relative to a point on a map (usually one of the map corers or its middle). Movement is much more realistic in this type of environment because an object can go in any direction. Figure AP2 shows how a continuous environment can be used in a game.

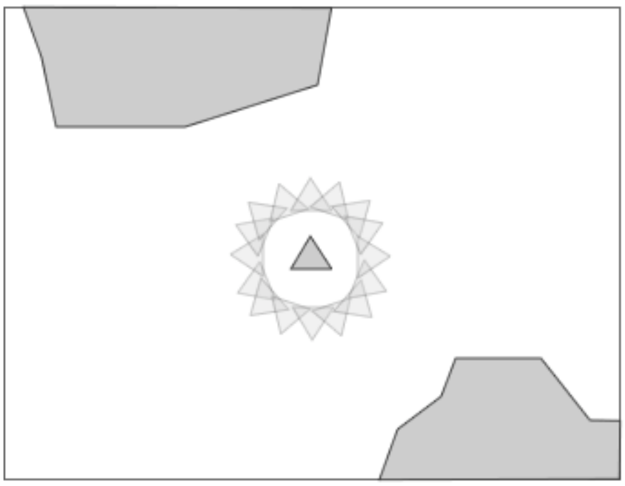


Figure AP2: Continuous environment

Appendix C: Entity in a continuous environment and its movement

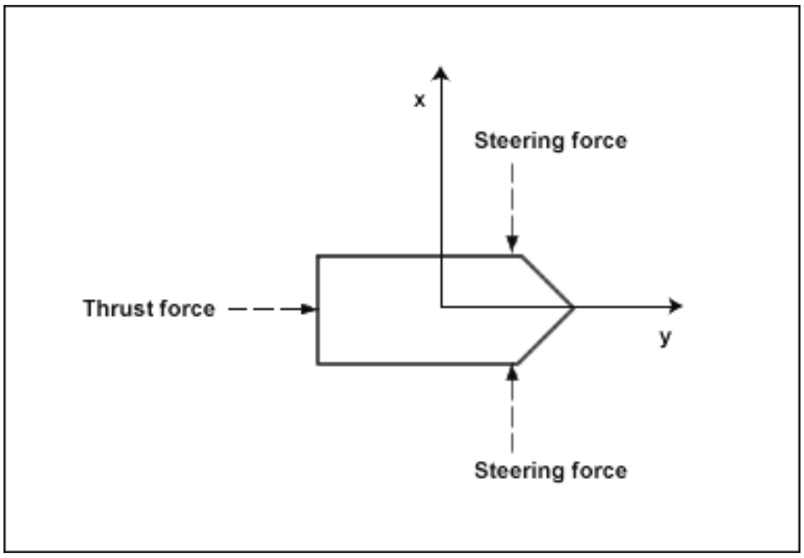


Figure AP3: Forces applied to a vehicle.
[D. M. Bourg and G. Seeman, 2004, p. 17]

There are two types of forces applied to an entity. The thrust force makes it move and can be positive (forward movement) or negative (backward movement). Alternatively, additional thrust force can be applied in the front of the entity to make it move backwards. The steering forces make the entity turn right or left. The turning radius is a function of speed which means that the turning angle gets bigger when the speed is higher. [D. M. Bourg and G. Seeman, 2004]

Appendix D:
Pure pursuit method for obstacle avoidance
[O. Ringdahl, 2003]

Figure AP4 shows the geometrical description of movement in the pure pursuit algorithm. It consists of the following steps:

1. Determine current location of the vehicle
2. Find a carrot point - point where the vehicle should get
3. Transform vehicle's and carrot point's location into vehicle's local coordinates We can use the following equation mentioned by D.M. Bourg and G Seeman (2004):

   ```
   x = X cos θ + Y sin θ
   y = -X sin θ + Y cos θ
   ```
   Where x and y are object's local coordinates, X an Y the global coordinates and θ is the object's orientation relative to the global coordinate system.
4. Calculate curvature c which is inverse of the radius of path a vehicle should follow
5. Determine steering angle

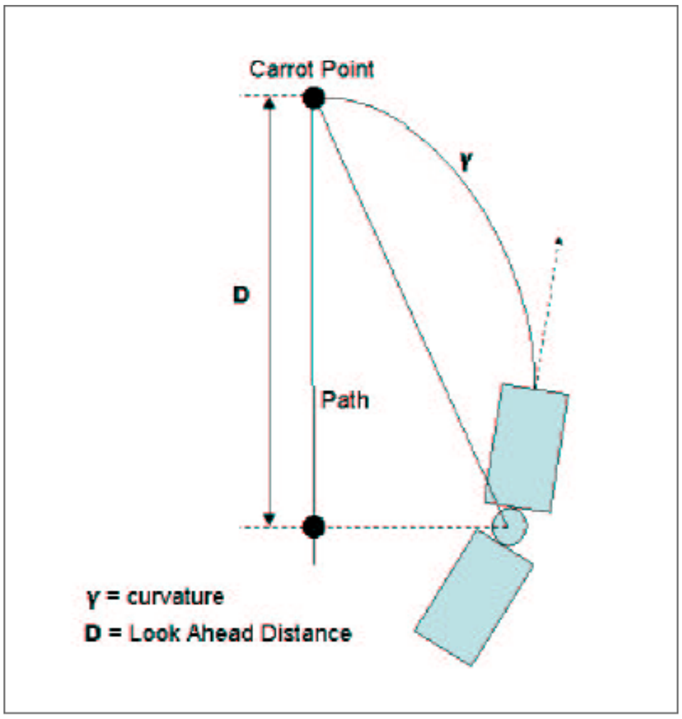


Figure AP4 - Pure Pursuit method
[O. Ringdahl, 2003]

This will produce a realistically-looking movement where vehicle tends to steer towards the target during the movement rather than turning towards a target and than going forward.

Appendix E: Various methods used for obstacle avoidance

Waypoints

Points placed on the terrain, which identify free space between obstacles [D. M. Bourg and G. Seeman, 2004]. They can be understood as nodes of a tree where two connected nodes don't have an obstacle between them (Figure AP5). The waypoints' connections can represent roads where each connection has its weight. Weight can mean both distance and applicability of a route [Bing Liu et. al, 1994] - e.g. major terrain type of a road.

This approach has the following advantages:

a. Movement is easy to manage and alter
b. Testing is generally straight forward

Disadvantages of this approach are:

a. A developer has to plot the nodes offline each time the environment changes
b. Positions of waypoints as well as their connections must be stored in the memory

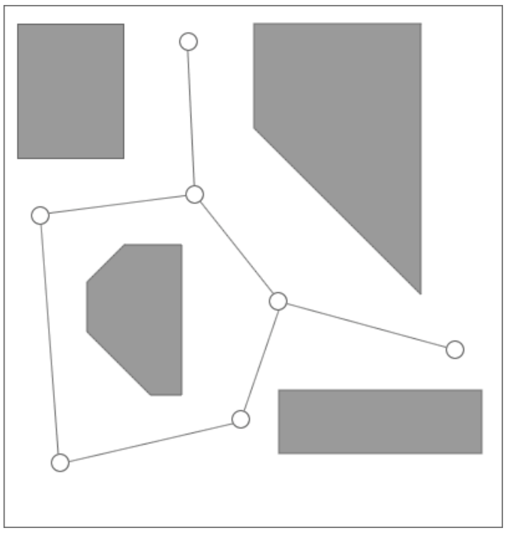


Figure AP5 - Waypoints placed between obstacles

To plan a route, an agent needs to find the shortest way from waypoint A to waypoint B. The A* algorithm [D. M. Bourg and G. Seeman, 2004] or Dijkstra's algorithm [Bing Liu et. al, 1994] can be used to efficiently search tree of waypoints for the shortest route.

The A* algorithm is usually used in tiled environments where each tile has its score. The score is a sum of tile's movement cost (from a starting tile) and a heuristic value. The heuristic value is estimated as a movement cost of a straight path from a current tile towards the target (i.e. we don't take obstacles between the tiles into account). The A* algorithm is able to find the least expensive path through the environment by examining tiles in a way to the target.

The Dijkstra's algorithm is a variation of the A* algorithm and works in a similar way using a tree of connected nodes where each connection has a weight. The algorithm uses a Distance list, a Previous vertex list and a Visited list to iterate through the nodes and find the shortest path [K. Ikeda's web site, Wikipedia Dijkstra].

Vector field histogram (VFH)

VFH consists of 3 levels of data [O. Ringdahl, 2003]:

a. 2D histogram grid that stores information about where obstacles are in certain radius around a vehicle. The grid is divided into N sectors and is updated via vehicle's sensors each time it moves. The sensors are able to detect range and size of obstacles around the vehicle.
b. 1D polar histogram - the 2D histogram grid is mapped onto a graph so that each sector has its polar obstacle density value. This value represents how many obstacles there are in a sector. A smoothing function can be applied in order to make neighbour sectors less attractive as well.
c. Output of the VFH algorithm - steering angle. The angle is determined by choosing one of the sectors with low obstacle density from the polar histogram. The winner sector is the one closest to the object's ideal direction (i.e. direction toward the target)

Using this approach an AI entity is able to plan the path around obstacles in a certain radius around the vehicle. However, the algorithm requires a lot of computations and therefore it is not suitable for agent systems with high amount of entities [O. Ringdahl, 2003].

Potential function

The function calculates the potential energy of any object and determines attraction and repulsion forces between two entities [D. M. Bourg and G. Seeman, 2004]. Computer-controlled entity moves towards or from an object according to the calculated force.

The advantage of this approach is that it can be used in many ways including chasing, evading, swarming and obstacle avoidance. Also, if parameters are set properly the movement appears to be very realistic.

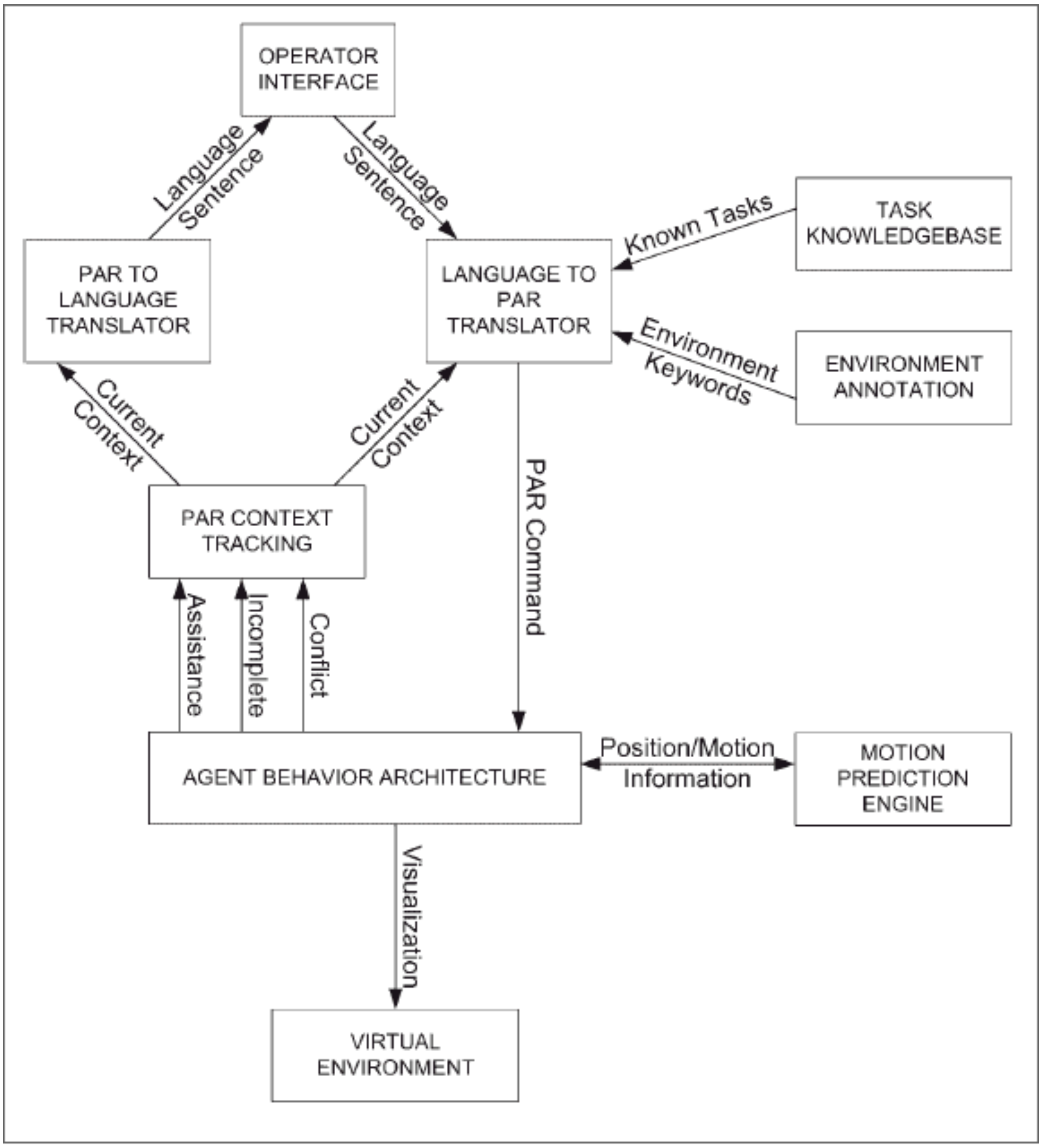
The main disadvantage is that the function can be quite CPU-intensive, especially for a large number of game units.

Tracing algorithm

The algorithm makes an AI entity follow straight lines or use any other path finding algorithm until an obstacle is found. The AI entity tries to go around an obstacle by following its walls until an original path can be followed again. This approach is usually used in tiled indoor environments but can be adapted also to continuous ones by using line-tracing and point content functions [D. M. Bourg and G. Seeman, 2004].

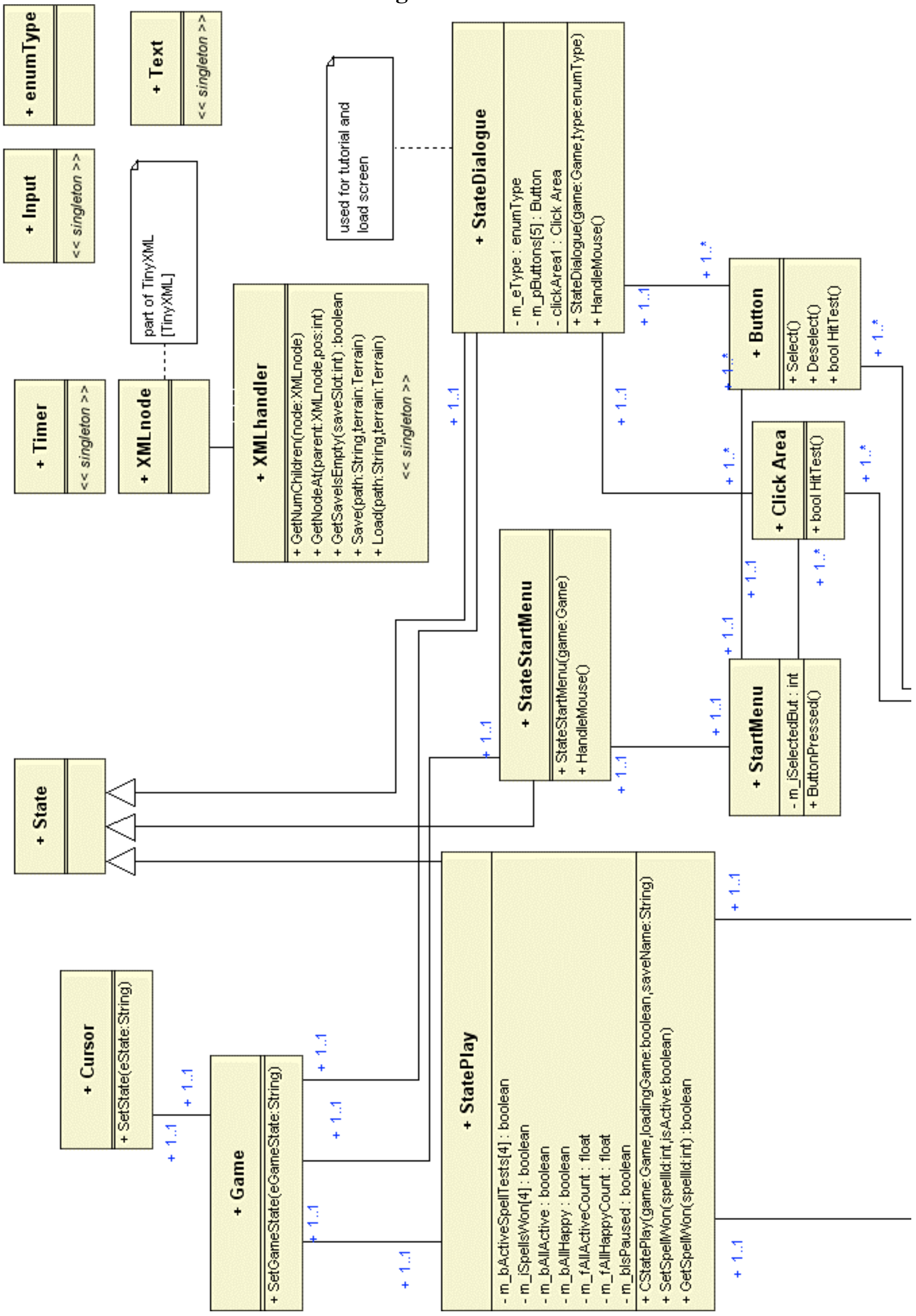
The advantage of this approach is that no path data needs to be stored and computations are quite straight forward. However, the movement is not as much realistic as when using the algorithms mentioned above.

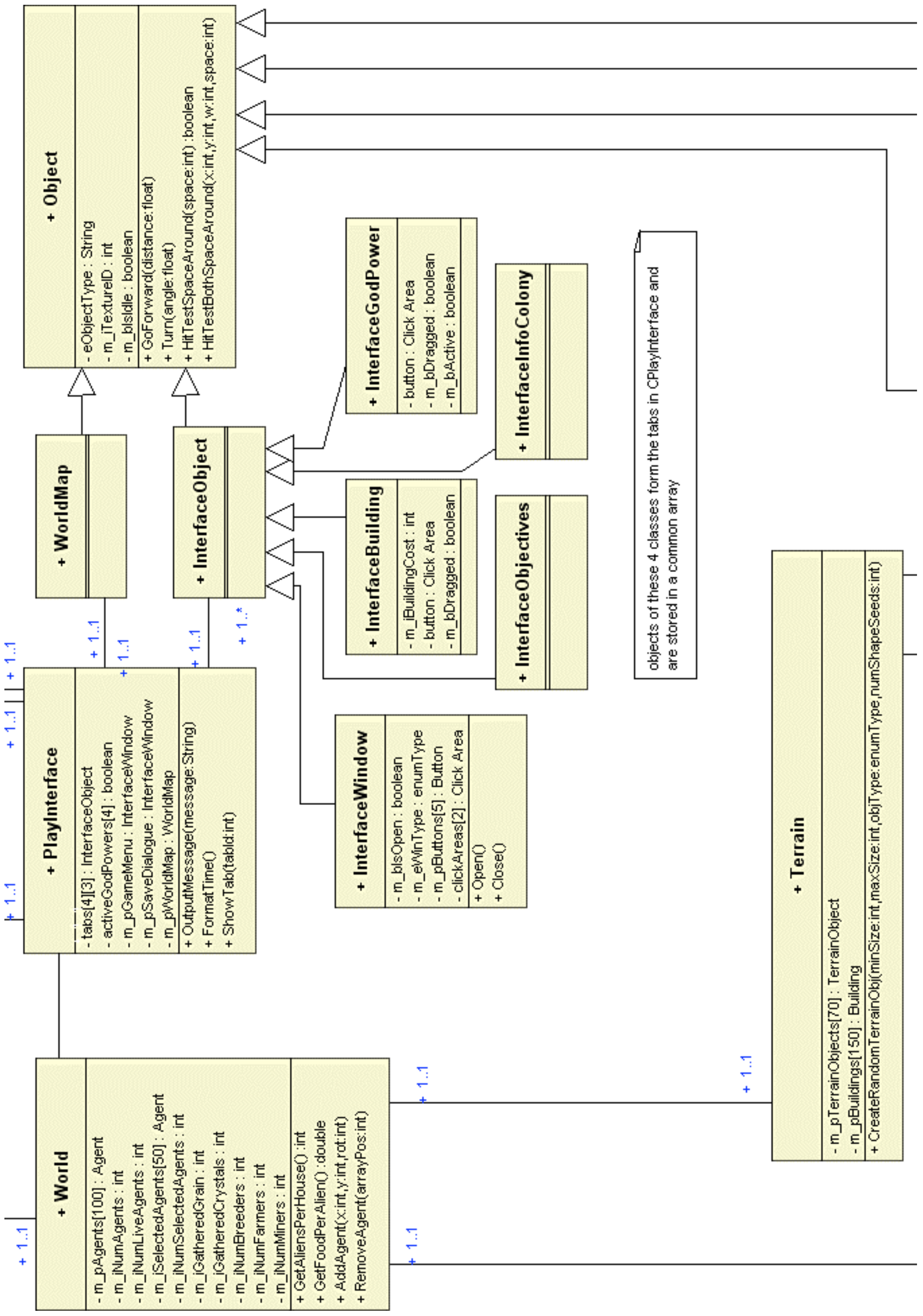
Appendix F:
VSRC Task Discourse Architecture

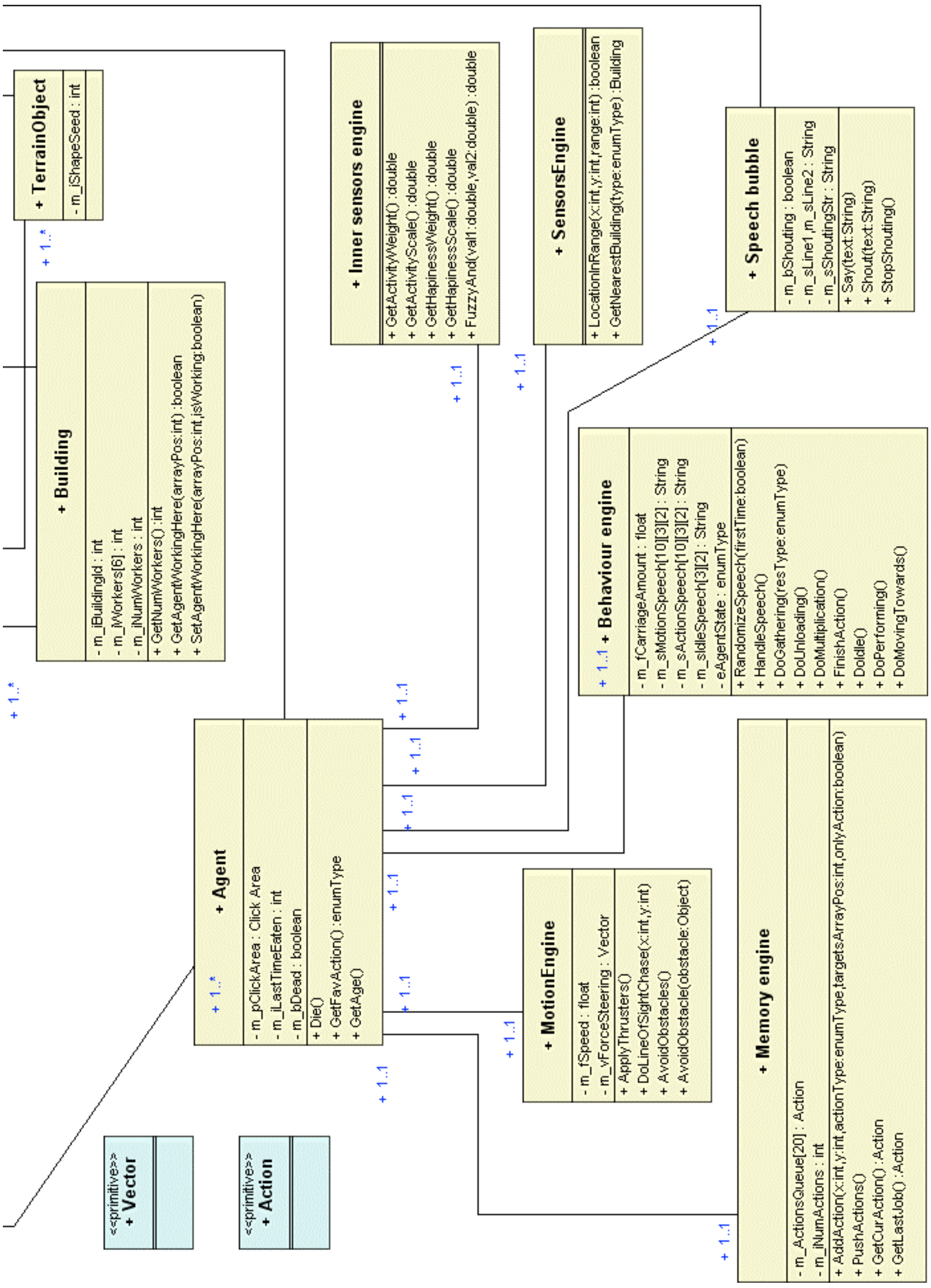


[J.R. Lee and A.B. Williams, 2004]

Appendix G: UML diagram of the artefact

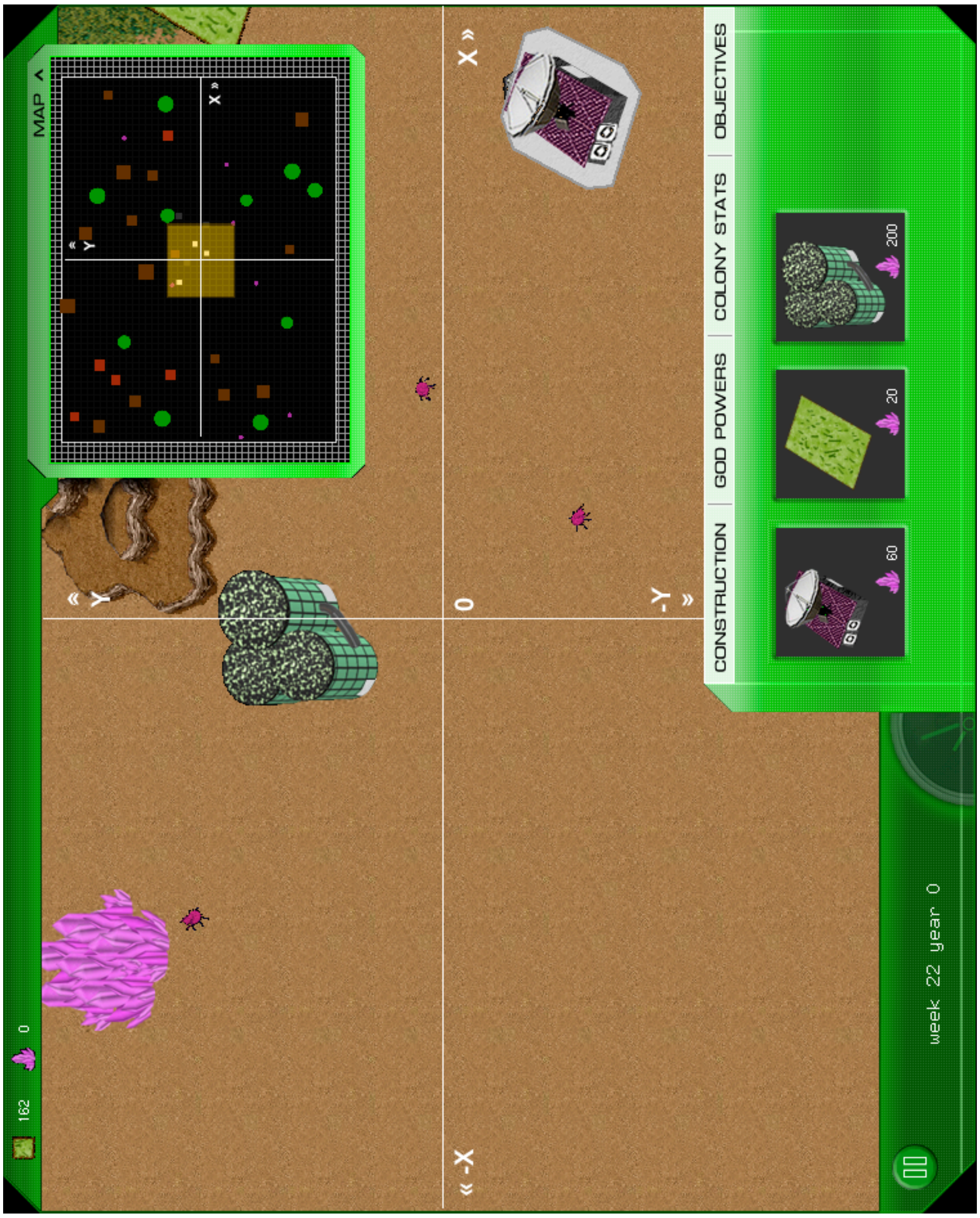

+ enumType
+ Input
<< singleton >>
+ Text
<< singleton >>
used for tutorial and load screen
+ StateDialogue
- m_eType : enumType
- m_pButtons[5] : Button
- clickArea1 : Click Area
+ StateDialogue(game:Game,type:enumType)
+ HandleMouse()
part of TinyXML [TinyXML]
+ Timer
<< singleton >>
+ XMLnode
+ XMLhandler
+ GetNumChildren(node:XMLnode)
+ GetNodeAt(parent:XMLnode,pos:int)
+ GetSaveIsEmpty(saveSlot:int) :boolean
+ Save(path:String,terrain:Terrain)
+ Load(path:String,terrain:Terrain)
<< singleton >>
+ Button
+ Select()
+ Deselect()
+ bool HitTest()
+ Click Area
+ bool HitTest()
+ StateStartMenu
+ StateStartMenu(game:Game)
+ HandleMouse()
+ StartMenu
- m_iSelectedBut : int
+ ButtonPressed()
+ State
+ Cursor
+ SetState(eState:String)
+ Game
+ SetGameState(eGameState:String)
+ StatePlay
- m_bActiveSpellTests[4] : boolean
- m_iSpellsWon[4] : boolean
- m_bAllActive : boolean
- m_bAllHappy : boolean
- m_fAllActiveCount : float
- m_fAllHappyCount : float
- m_bIsPaused : boolean
+ CStatePlay(game:Game,loadingGame:boolean,saveName:String)
+ SetSpellWon(spellId:int,isActive:boolean)
+ GetSpellWon(spellId:int) :boolean
+ 1..1
+ 1..*


+ Object
- eObjectType : String
- m_iTextureID : int
- m_bIsIdle : boolean
+ GoForward(distance:float)
+ Turn(angle:float)
+ HitTestSpaceAround(space:int) :boolean
+ HitTestBothSpaceAround(x:int,y:int,w:int,space:int)
+ WorldMap
+ InterfaceObject
+ InterfaceGodPower
- button : Click Area
- m_bDragged : boolean
- m_bActive : boolean
+ InterfaceInfoColony
+ InterfaceBuilding
- m_iBuildingCost : int
- button : Click Area
- m_bDragged : boolean
+ InterfaceObjectives
objects of these 4 classes form the tabs in CPlayInterface and are stored in a common array
+ PlayInterface
- tabs[4][3] : InterfaceObject
- activeGodPowers[4] : boolean
- m_pGameMenu : InterfaceWindow
- m_pSaveDialogue : InterfaceWindow
- m_pWorldMap : WorldMap
+ OutputMessage(message:String)
+ FormatTime()
+ ShowTab(tabId:int)
+ InterfaceWindow
- m_bIsOpen : boolean
- m_eWinType : enumType
- m_pButtons[5] : Button
- clickAreas[2] : Click Area
+ Open()
+ Close()
+ Terrain
- m_pTerrainObjects[70] : TerrainObject
- m_pBuildings[150] : Building
+ CreateRandomTerrainObj(minSize:int,maxSize:int,objType:enumType,numShapeSeeds:int)
+ World
- m_pAgents[100] : Agent
- m_iNumAgents : int
- m_iNumLiveAgents : int
- m_iSelectedAgents[50] : Agent
- m_iNumSelectedAgents : int
- m_iGatheredGrain : int
- m_iGatheredCrystals : int
- m_iNumBreeders : int
- m_iNumFarmers : int
- m_iNumMiners : int
+ GetAliensPerHouse() :int
+ GetFoodPerAlien() :double
+ AddAgent(x:int,y:int,rot:int)
+ RemoveAgent(arrayPos:int)
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..*
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1

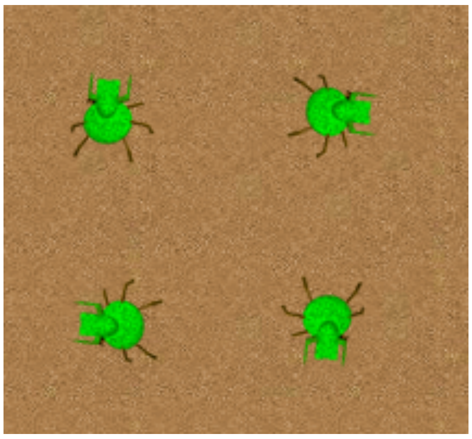

+ TerrainObject
- m_iShapeSeed : int
+ Building
- m_iBuildingId : int
- m_iWorkers[6] : int
- m_iNumWorkers : int
+ GetNumWorkers() :int
+ GetAgentWorkingHere(arrayPos:int) :boolean
+ SetAgentWorkingHere(arrayPos:int,isWorking:boolean)
+ Inner sensors engine
+ GetActivityWeight() :double
+ GetActivityScale() :double
+ GetHapinessWeight() :double
+ GetHapinessScale() :double
+ FuzzyAnd(val1:double,val2:double) :double
+ SensorsEngine
+ LocationInRange(x:int,y:int,range:int) :boolean
+ GetNearestBuilding(type:enumType) :Building
+ Speech bubble
- m_bShouting : boolean
- m_sLine1,m_sLine2 : String
- m_sShoutingStr : String
+ Say(text:String)
+ Shout(text:String)
+ StopShouting()
+ Behaviour engine
- m_fCarriageAmount : float
- m_sMotionSpeech[10][3][2] : String
- m_sActionSpeech[10][3][2] : String
- m_sIdleSpeech[3][2] : String
- eAgentState : enumType
+ RandomizeSpeech(firstTime:boolean)
+ HandleSpeech()
+ DoGathering(resType:enumType)
+ DoUnloading()
+ DoMultiplication()
+ FinishAction()
+ DoIdle()
+ DoPerforming()
+ DoMovingTowards()
+ Agent
- m_pClickArea : Click Area
- m_iLastTimeEaten : int
- m_bDead : boolean
+ Die()
+ GetFavAction() :enumType
+ GetAge()
+ MotionEngine
- m_fSpeed : float
- m_vForceSteering : Vector
+ ApplyThrusters()
+ DoLineOfSightChase(x:int,y:int)
+ AvoidObstacles()
+ AvoidObstacle(obstacle:Object)
+ Memory engine
- m_ActionsQueue[20] : Action
- m_iNumActions : int
+ AddAction(x:int,y:int,actionType:enumType,targetsArrayPos:int,onlyAction:boolean)
+ PushActions()
+ GetCurAction() :Action
+ GetLastJob() :Action
<<primitive>>
+ Vector
<<primitive>>
+ Action
+ 1..*
+ 1..*
+ 1..*
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1
+ 1..1

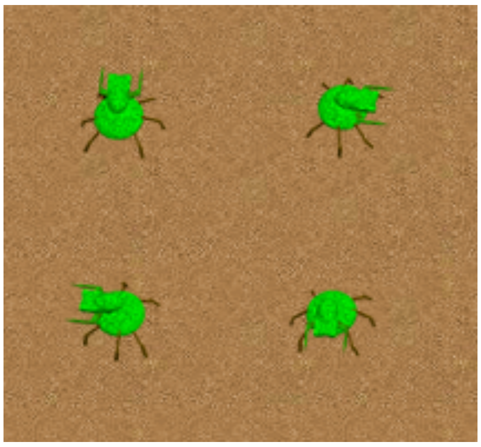
Appendix H:
World axes in Alien Farm and the mini map

The view is centred in the beginning of the game. The axes emerge from this point. The mini map displays all world objects in a magnified view: the white dots are the aliens, the brown squares are hills, the green circles swamps, red squares are trees, the purples squares represent crystals and the gray squares are buildings.

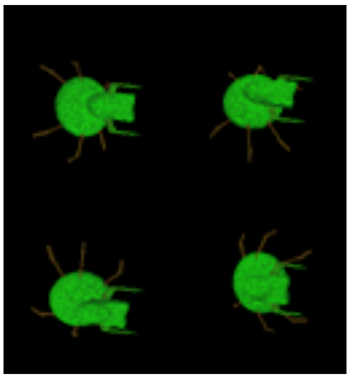


Appendix I:
Adjusting the texture based on unit's rotation, implementation 1


Simply rotating an agent model would result in awkward display


The result of applying one four different textures based on rotation

X = 0 X = 1
Y = 1



Y = 0

The image on the left is the original texture used for an agent – each of the four corners corresponds to a certain rotation. The texture coordinates are calculated based on the current rotation each loop frame:

```
//------------set texture according to the rotation:
if (m_fRotation >= 45 && m_fRotation<225){
    textureAreaTopR_x = 0.5;
} else {
    textureAreaTopR_x = 1;
}
if (m_fRotation >= 135 && m_fRotation<315){
    textureAreaTopR_y = 0.5;
} else {
    textureAreaTopR_y = 1;
}

//-------------- prepare texture coordinates for rendering:
m_fX1=textureAreaTopR_x;
m_fY1=textureAreaTopR_y;
m_fX3=textureAreaTopR_x-0.5;
m_fY3=textureAreaTopR_y-0.5;

//-------------- use the texture coordinates when doing rendering:
glRotatef(m_fRotation,0,0,1);
glBegin(GL_POLYGON);
    glTexCoord2f(m_fX1, m_fY1);
```

```
        glVertex2f(0.5f, 0.5f);
        glTexCoord2f(m_fX3, m_fY1);
        glVertex2f(-0.5f, 0.5f);
        glTexCoord2f(m_fX3, m_fY3);
        glVertex2f(-0.5f, -0.5f);
        glTexCoord2f(m_fX1, m_fY3);
        glVertex2f(0.5f, -0.5f);
    glEnd();
```

Note that since we do not change the rotation itself when the texture coordinates are calculated, the individual four images need to be pre-rotated so that they are displayed appropriately when applied in real-time.

Appendix J:
Steering towards a target when it is right at the back of an agent

The problem with the algorithm used for navigation towards a target occurs when a target is right at the back of an agent and the resulting steering force is 0. To overcome this problem, the following test is done after initial calculation of the steering force:

```
//----- create a normalised vector which stores agent’s absolute
        orientation:
CVector3d orient = GetOrientationVector();

//-------- test if the steering force was calculated as very low:
if (fabs(m_vMovingForce.m_fx) < 3){
    bool turn180 = false;
    /*
    The GetAbsoluteWorldSideHoriz() and GetAbsoluteWorldSideVert()
       functions return enumeration values such as eWSnorth,
       eWSsoutEast, etc.
    */
    eWorldSide horizSide = m_pAgent->GetSensorEngine()-
>GetAbsoluteWorldSideHoriz(x,y);
    eWorldSide vertSide = m_pAgent->GetSensorEngine()-
>GetAbsoluteWorldSideVert(x,y);

    //-------- horizontal test:
    if ((horizSide == eWSwest && orient.m_fx > 0) || (horizSide ==
eWSeast && orient.m_fx < 0) ){
          turn180 = true;
    //-------- vertical test:
    } else if ((vertSide == eWSnorth && orient.m_fy < 0) || (vertSide ==
eWSsouth && orient.m_fy > 0) ){
          turn180 = true;
    }

    //---------- if target is at the back, initiate rapid turn:
    if (turn180){
          m_vForceSteering.m_fx += 10;
    }
}
```

Appendix K:
Testing against collision with other agents in the Idle state

To solve the problem of two and more agents ending their movement on top of each other, a simple algorithm was added to the agent’s behaviour which checks against all agents which have a higher array position. If collision occurs agent chooses a random location close to its original position.

```
for (int i=m_pAgent->GetArrayPos()+1;i<m_pAgent->GetWorld()-
>GetNumAgents();i++){

    //---- check if other agent is around the same location:
    if (m_pAgent->GetSensorEngine()->LocationInRange(m_pAgent-
>GetWorld()->GetAgent(i)->GetWorldX(),m_pAgent->GetWorld()->GetAgent(i)-
>GetWorldY(),20)){

          //----- create a new random position:
```

```
          int actionX = m_pAgent->GetMemoryEngine()-
>GetRandomDirection(60);
          int actionY = m_pAgent->GetMemoryEngine()-
>GetRandomDirection(50);

          //-----move towards the position (add a GoTo action as a
                 new task for self:
          m_pAgent->GetMemoryEngine()->AddAction(m_pAgent->GetWorldX() +
actionX,m_pAgent->GetWorldY() + actionY,eActionGoTo,-1,true);
    }
}
```

To create a new random location, both X and Y coordinates are adjusted using the GetRandomDirection (distanceFromOriginal) method:

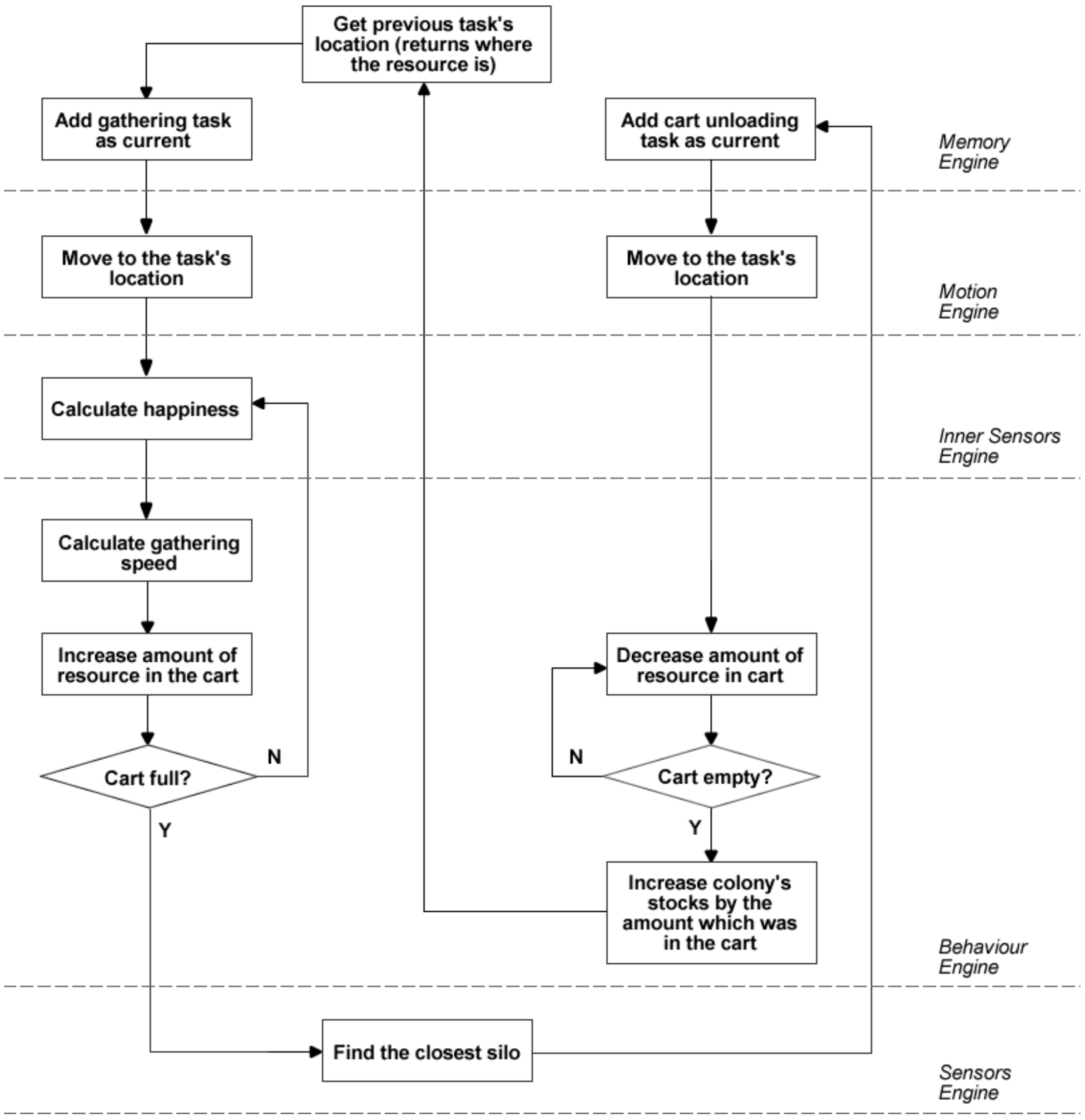
```
float CAgentMemoryEngine::GetRandomDirection(float maxDistance_){

    int range=maxDistance_*2;
    //--------- range <- maxDistance; maxDistance > [DaniWeb]:
    float xDiff = ((range*rand()/(RAND_MAX+1.0)))- maxDistance_;

    //---------- limit agent's own space <-40;40>:
    if (xDiff > - 40 && xDiff < 40){
          //----- use recursion if number within this space:
          xDiff = GetRandomDirection(maxDistance_);
    }
    return xDiff;
}
```

Appendix L: Extension to the obstacle avoidance algorithm

The following algorithm is a part of the obstacle avoidance algorithm in the CAgentMotionEngine class. It deals with the problem where agent would circle around an obstacle when target of its task in inside it

```
void CAgentMotionEngine::AvoidObstacle(CObject*obstacle_){
    ...  //obstacle avoidance algorithm

//---- test if should stop in front of obstacle if target inside it:
    eActionType curActionType = m_pAgent->GetMemoryEngine()-
>GetCurActionType();
    bool stopWhenNear = true;
    if (obstacle_->GetObjectType() == CObject::eTypeBuilding){
          if (curActionType == eActionGatherCrystals){
                if (m_pAgent->GetTerrain()->GetBuilding(obstacle_-
>GetArrayPos())->GetBuildingType() != CBuilding::eBuildingCrystals){
                      stopWhenNear = false;
                }
          } else if (curActionType == eActionUnload){
                if (m_pAgent->GetTerrain()->GetBuilding(obstacle_-
>GetArrayPos())->GetBuildingType() != CBuilding::eBuildingSilo){
                      stopWhenNear = false;
                }
          }
    }

    if (stopWhenNear){
          int curActionX = m_pAgent->GetMemoryEngine()->GetCurActionX();
          int curActionY = m_pAgent->GetMemoryEngine()->GetCurActionY();

          if (obstacle_->HitTest(curActionX,curActionY) && obstacle_-
>HitTestBothSpaceAround(m_pAgent->GetWorldX(),m_pAgent-
>GetWorldY(),50,0)){
                m_pAgent->GetBehaviourEngine()-
>SetState(eAgentPerforming);
                m_pAgent->SetForcedStop(true);
          }
    }
}
```

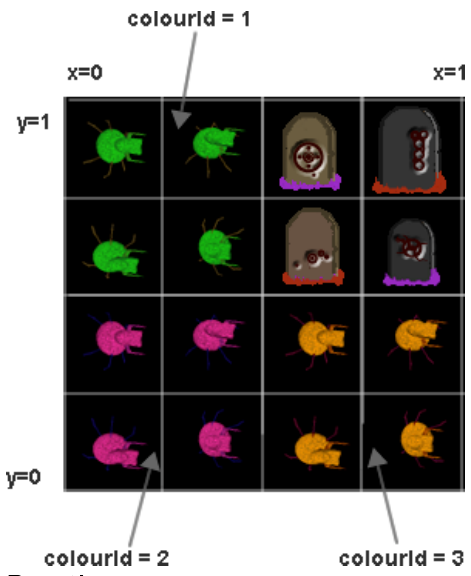
Appendix M: Graphical representation of the gathering algorithm


Get previous task's location (returns where the resource is)
Add gathering task as current
Add cart unloading task as current
Memory Engine
Move to the task's location
Move to the task's location
Motion Engine
Calculate happiness
Inner Sensors Engine
Calculate gathering speed
Increase amount of resource in the cart
Decrease amount of resource in cart
Cart full?
N
Y
Cart empty?
N
Y
Increase colony's stocks by the amount which was in the cart
Behaviour Engine
Find the closest silo
Sensors Engine

Appendix N:
Improved texture for an agent – additional colours and graves

The following algorithm shows how the texture coordinates are being calculated after implementation of three distinct colour of agent:

```
//------------set texture according to the rotation:
if (m_fRotation >= 45 && m_fRotation<225){
    textureAreaTopR_x = 0.25;
} else {
    textureAreaTopR_x = 0.5;
}

if (m_fRotation >= 135 && m_fRotation<315){
    textureAreaTopR_y = 0.75;
} else {
    textureAreaTopR_y = 1;
}
//--------- adjust the texture coordinates based on agent’s colourId:
if (m_iColorId == 3){
    textureAreaTopR_x += 0.5;
}
if (m_iColorId > 1){
    textureAreaTopR_y -= 0.5;
}

//-------------- prepare texture coordinates for rendering:
m_fX1=textureAreaTopR_x;
m_fY1=textureAreaTopR_y;
m_fX3=textureAreaTopR_x-0.25;
m_fY3=textureAreaTopR_y-0.25;
```



The image on the left shows the texture for this implementation. The individual images are pre-rotated for the reasons explained in Appendix I.

Death

The top right set of images are graves. No matter what colour an agent had, their original texture coordinates need to be converted to the ones matching the top right corner of the texture. This is done in the Die() function using the following algorithm:

```
if (m_iColorId < 3){
    m_fX1 += 0.5;
    m_fX3 += 0.5;
    }
if (m_iColorId > 1){
    m_fY1 += 0.5;
    m_fY3 += 0.5;
}
```

As we can see the grave image is chosen according to agent’s current orientation (the original alien texture is chosen according to the rotation and when agent is dead, the texture coordinates are increased by a set amount: 0.5.). However, the grave image doesn’t depend on agent’s original colour.

Appendix O:
An example of XML structure for a saved game

```
<?xml version="1.0" ?>
<planet>
  <gameInfo  gatheredGrain="174" gatheredCrystals="0" timeElapsed="986"
spell1won="0" spell2won="0" spell3won="0" allHappy="0" allActive="0"
allHappyCount="0" allActiveCount="0" spell1active="0" spell2active="0"
spell3active="0" />

  <terrain rocks="15" swamps="10" trees="5">
    <tree  x="-1699" y="1380" w="195" shapeSeed="2" />
    <tree  x="-1299" y="922" w="205" shapeSeed="2" />
    <tree  x="1362" y="361" w="241" shapeSeed="1" />
    <tree  x="-1242" y="333" w="243" shapeSeed="1" />
    <tree  x="-1136" y="1103" w="242" shapeSeed="1" />
    <rock  x="70" y="272" w="187" shapeSeed="2" />
    <rock  x="1531" y="-1108" w="295" shapeSeed="2" />
    <rock  x="293" y="1247" w="290" shapeSeed="1" />
    <rock  x="958" y="830" w="311" shapeSeed="2" />
    <rock  x="435" y="743" w="222" shapeSeed="2" />
    <rock  x="1801" y="1000" w="184" shapeSeed="2" />
    <rock  x="-1801" y="1099" w="277" shapeSeed="3" />
    <rock  x="-1464" y="-253" w="249" shapeSeed="3" />
    <rock  x="-1533" y="708" w="244" shapeSeed="2" />
    <rock  x="-1067" y="-155" w="199" shapeSeed="2" />
    <rock  x="-497" y="1439" w="332" shapeSeed="3" />
    <rock  x="-125" y="584" w="330" shapeSeed="1" />
    <rock  x="918" y="518" w="221" shapeSeed="2" />
    <rock  x="-1426" y="-688" w="274" shapeSeed="1" />
    <rock  x="123" y="-975" w="193" shapeSeed="2" />
    <swamp  x="485" y="354" w="256" shapeSeed="3" />
    <swamp  x="1696" y="375" w="290" shapeSeed="1" />
    <swamp  x="-1725" y="411" w="297" shapeSeed="1" />
    <swamp  x="-1765" y="-657" w="290" shapeSeed="1" />
    <swamp  x="699" y="1116" w="293" shapeSeed="1" />
    <swamp  x="651" y="-503" w="223" shapeSeed="3" />
    <swamp  x="965" y="-1001" w="293" shapeSeed="1" />
    <swamp  x="-892" y="825" w="243" shapeSeed="1" />
    <swamp  x="-680" y="-944" w="225" shapeSeed="3" />
    <swamp  x="759" y="-1249" w="273" shapeSeed="2" />
  </terrain>

  <buildings farms="1" crystals="7" houses="1" silos="1">
    <crystals  x="-273" y="306" />
    <crystals  x="-1686" y="-974" />
    <crystals  x="1330" y="826" />
```

```
        <crystals  x="-1928" y="-446" />
        <crystals  x="401" y="-359" />
        <crystals  x="-255" y="-608" />
        <crystals  x="1042" y="-288" />
        <silo  x="-24" y="148" />
        <house  x="388" y="-59" />
        <farm  x="488" y="233" />
    </buildings>

    <units units="3">
        <Feena x="238" y="85" rotation="32" dead="0" lifeTime="141" nextNumChildren="0" carriageAmount="0" movementActivity="0" daysInJob="0" favAction="1" colourId="2">
            <action  type="0" targetsArrayId="0" x="0" y="0" />
            <lastJob  type="0" targetsArrayId="-842150451" x="-842150451" y="-842150451" />
        </Feena>
        <Tolen x="-241" y="227" rotation="141" dead="0" lifeTime="141" nextNumChildren="0" carriageAmount="6" movementActivity="7" daysInJob="45" favAction="2" colourId="2">
            <action  type="3" targetsArrayId="0" x="-288" y="285" />
            <lastJob  type="3" targetsArrayId="0" x="-288" y="285" />
        </Tolen>
        <Tiree x="20" y="-30" rotation="0" dead="0" lifeTime="141" nextNumChildren="0" carriageAmount="0" movementActivity="0" daysInJob="0" favAction="2" colourId="2">
            <action  type="0" targetsArrayId="0" x="0" y="0" />
            <lastJob  type="0" targetsArrayId="-842150451" x="-842150451" y="-842150451" />
        </Tiree>
    </units>
</planet>
```

Appendix P: Calculation and the use of the inner state

Inner state variables

	Affected by	Value range of item from column 2
Happiness	- if doing a preferred job (affecting 50% of happiness)	Gather grain, mine, breed, do nothing
	- number of room in houses (affecting 25% of happiness)	<1;N>, where N=number of aliens
	- amount of food in store (affecting 25% of happiness)	<0;MAX_INT>
Activity	- movement time (affecting 60% of activity)	<0;50> weeks
	- age (affecting 40% of activity)	<0;4> years

Affected Attributes

	Value range of an attribute	Affected by
Farming speed	<0.018;0.18>	- Happiness: the happier, the greater the gathering speed
Mining speed	<0.002;0.02>	- Happiness: the happier, the greater the gathering speed
Number of clones wants	[2,4,6]	- Happiness: the happier, the more children wants
Amount of days needed to clone	<100;300>	- Happiness: the happier, the smaller the amount
Movement speed	<0.6;2>	- Activity: the lower activity the slower the motion
Units of food needed per week	<1;3>	- activity (by 60%): the more tired the more food required
		- happiness (40%): the happier the less food required
life length in years	<1;4>	- amount of food in store - happiness

Appendix Q: Fuzzification algorithms

```
//--------------------------------- HAPINESS
/* hapiness depends on:
- 50% on the job (based on personal job preferences),
- 25% on number of houses (compare # houses with population to get a
ratio R: population/numHouses*4)
    - if R <= 1 => good, aliens happy
    - if R > 1 => bad, getting depressed
    - limit : R = 3;
- 25% amount of food in store
*/

double CAgentInnerSensorEngine::GetHappyDeg(){
    double jobXval = GetJobFuzzyXvalue();
    double jobWeight = FuzzyTrapezoid(jobXval,0.6,0.8,1,1.1);
    int aliensPerHouse = m_pAgent->GetWorld()->GetAliensPerHouse();
    double housingWeight;
    if (aliensPerHouse == 1){
          housingWeight = 1;
    } else {
          housingWeight = 0;
    }
    double foodPerAlien = m_pAgent->GetWorld()->GetFoodPerAlien();
    double foodStoreWeight = FuzzyGrade(foodPerAlien,40,50);
    double happyDeg;
    happyDeg = 0.5*jobWeight + 0.25*housingWeight + 0.25*foodStoreWeight;
    //------------------ test if happiness spell was used:
    if (m_pAgent->GetWorld()->GetStateWrapper()->GetAllHappy()){
          happyDeg += 0.3;
          if (happyDeg > 1){
                happyDeg = 1;
          }
    }
    return happyDeg;
}

double CAgentInnerSensorEngine::GetNormalDeg(){
    double jobXval = GetJobFuzzyXvalue();
    double jobWeight = FuzzyTrapezoid(jobXval,0.2,0.4,0.6,0.8);
    int aliensPerHouse = m_pAgent->GetWorld()->GetAliensPerHouse();
    double housingWeight;
    if (aliensPerHouse == 2){
          housingWeight = 1;
    } else {
          housingWeight = 0;
    }
    double foodPerAlien = m_pAgent->GetWorld()->GetFoodPerAlien();
    double foodStoreWeight = FuzzyTrapezoid(foodPerAlien,20,30,40,50);
    double normalDeg = 0.5*jobWeight + 0.25*housingWeight +
0.25*foodStoreWeight;
    return normalDeg;
```

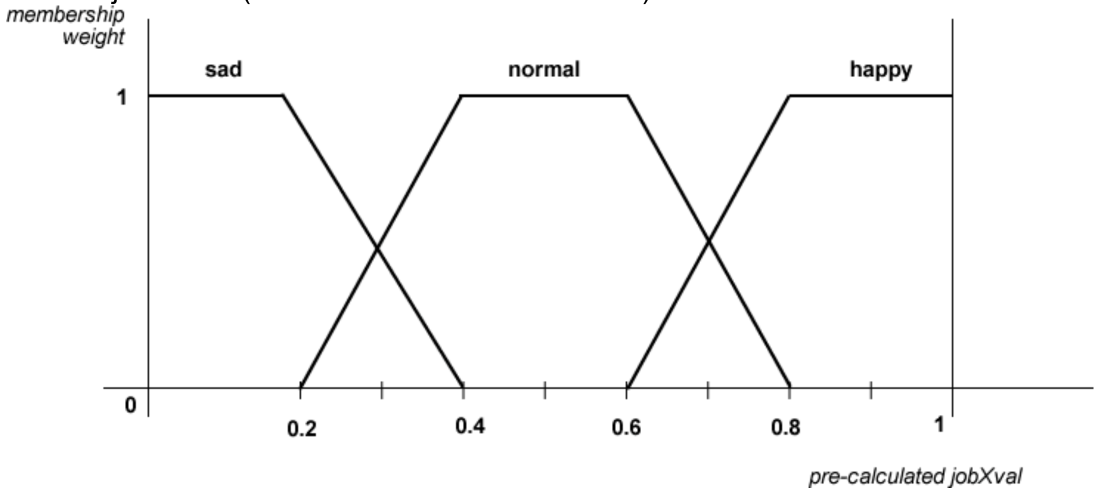
```
}

double CAgentInnerSensorEngine::GetSadDeg(){
    double jobXval = GetJobFuzzyXvalue();
    double jobWeight = FuzzyTrapezoid(jobXval,-0.1,0,0.2,0.4);
    int aliensPerHouse = m_pAgent->GetWorld()->GetAliensPerHouse();
    double housingWeight;
    if (aliensPerHouse >= 3){
          housingWeight = 1;
    } else {
          housingWeight = 0;
    }
    double foodPerAlien = m_pAgent->GetWorld()->GetFoodPerAlien();
    double foodStoreWeight = FuzzyTrapezoid(foodPerAlien,-1,0,20,30);
    double sadDeg;
    sadDeg = 0.5*jobWeight + 0.25*housingWeight + 0.25*foodStoreWeight;
    //------------------ test if happiness spell was used:
    if (m_pAgent->GetWorld()->GetStateWrapper()->GetAllHappy()){
          sadDeg -= 0.3;
          if (sadDeg < 0){
                sadDeg = 0;
          }
    }
    return sadDeg;
}


double CAgentInnerSensorEngine::GetJobFuzzyXvalue(){
    double daysInJobPercent = m_pAgent->GetMemoryEngine()->
GetWeeksInJob()/m_pAgent->GetNumRememberedWeeks();
    eActionType curActionType =  m_pAgent->GetMemoryEngine()->
GetCurJobType();
    eActionType prefActionType = m_pAgent->GetFavAction();
    double jobTemp;

    /* cur job evaluation graph:

    sad        normal       happy
    ----\    /--------\   /-----------
         \  /          \ /
          \             \
         / \           / \
    ----|--|--|-------|---|----------------
      0.2 0.3 0.4    0.6 0.8


      - 0.3 is a hasRightJobLimit, if does , value on X increases from
0.3, if not, it decreases from 0.3
      - action goTo or None, stay at the same level
      */
    if (curActionType == prefActionType){
          jobTemp = 0.3 + 0.7*daysInJobPercent;
    }
    else if (curActionType == eActionNone || curActionType ==
eActionGoTo){
          Action lastJob = m_pAgent->GetMemoryEngine()->GetLastJob();
          if (lastJob.type == prefActionType){
                jobTemp = 0.3 + 0.7*daysInJobPercent;
          } else {
                jobTemp = 0.3 - 0.3*daysInJobPercent;
          }
    } else {
          jobTemp = 0.3 - 0.3*daysInJobPercent;
    }
    m_dLastJobTemp = jobTemp;
    // || curActionType == eActionNone || curActionType == eActionGoTo
    return jobTemp;
}
//---------------------------------- ACTIVITY
/*
    activity affected by
50% time spent in job - the time goes gradually down when agent is out of
job. The time is set to 0 when a new job.
50% motion time
    */
double CAgentInnerSensorEngine::GetActiveDeg(){
    double daysMoving = m_pAgent->GetBehaviourEngine()->
GetMovementActivity();
    double ageWeight = FuzzyTrapezoid(m_pAgent->GetAge(),-0.1,0,1,2);
    double movementWeight = FuzzyTrapezoid(daysMoving,-1,0,0,14);
    double activeDeg;
    //------------------ test if activity spell was used:
    if (m_pAgent->GetWorld()->GetStateWrapper()->GetAllActive()){
          activeDeg = 1;
    } else {
          activeDeg =  0.4*ageWeight + 0.6*movementWeight;
    }
    return activeDeg;
}
double CAgentInnerSensorEngine::GetModerateDeg(){
    double daysMoving = m_pAgent->GetBehaviourEngine()-
>GetMovementActivity();
    double movementWeight = FuzzyTrapezoid(daysMoving,0,14,14,35);;
    double ageWeight = FuzzyTrapezoid(m_pAgent->GetAge(),1,2,3,4);
    double activeDeg;
    activeDeg = 0.4*ageWeight + 0.6*movementWeight;
    return activeDeg;
}

double CAgentInnerSensorEngine::GetLazyDeg(){
    double daysMoving = m_pAgent->GetBehaviourEngine()-
>GetMovementActivity();
    double movementWeight = FuzzyGrade(daysMoving,14,35);
    double ageWeight = FuzzyGrade(m_pAgent->GetAge(),3,4);
    double activeDeg;
    //------------------ test if activity spell was used:
```

```
    if (m_pAgent->GetWorld()->GetStateWrapper()->GetAllActive()){
          activeDeg = 0;
    } else {
          activeDeg =  0.4*ageWeight + 0.6*movementWeight;
    }
    return activeDeg;
}
```

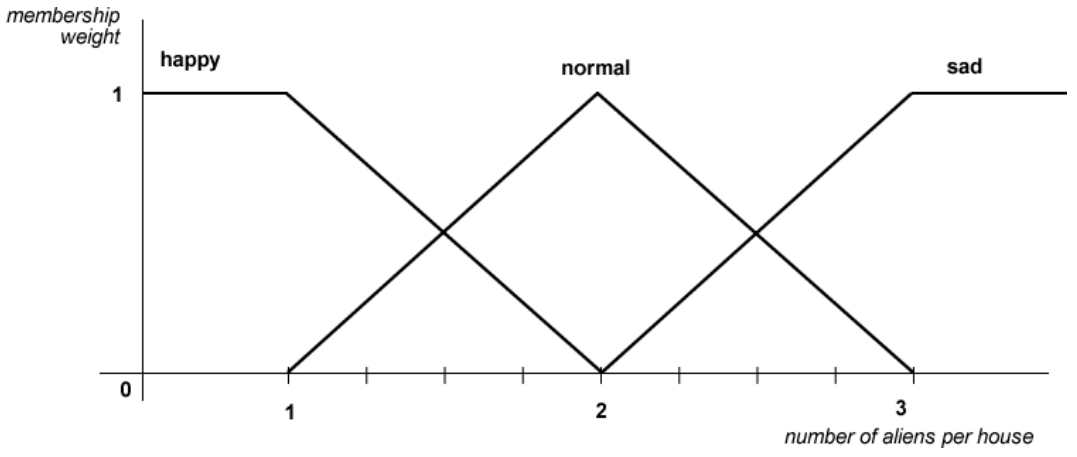
Appendix R: Fuzzification membership functions

Happiness fuzzy sets

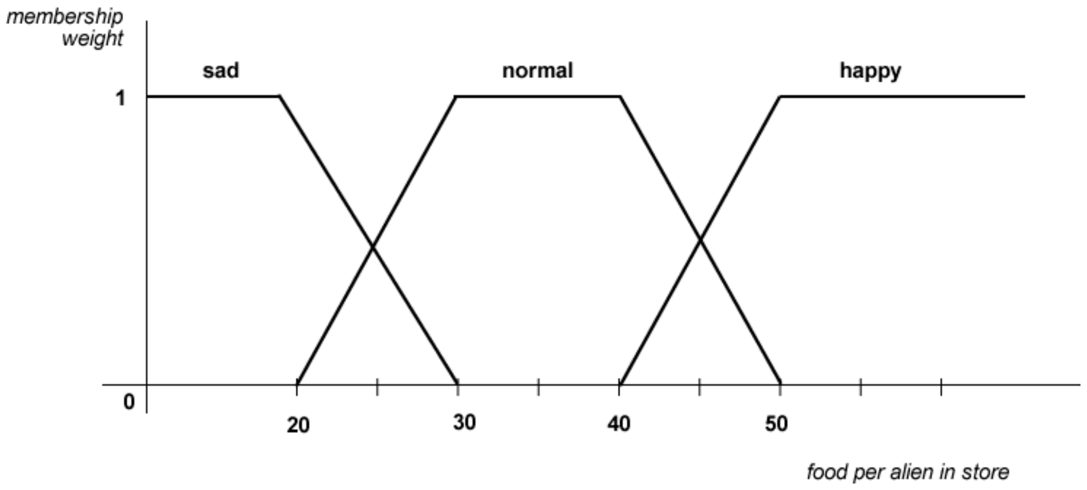
Current job factor (normalised value between 0-1):



Houses factor:

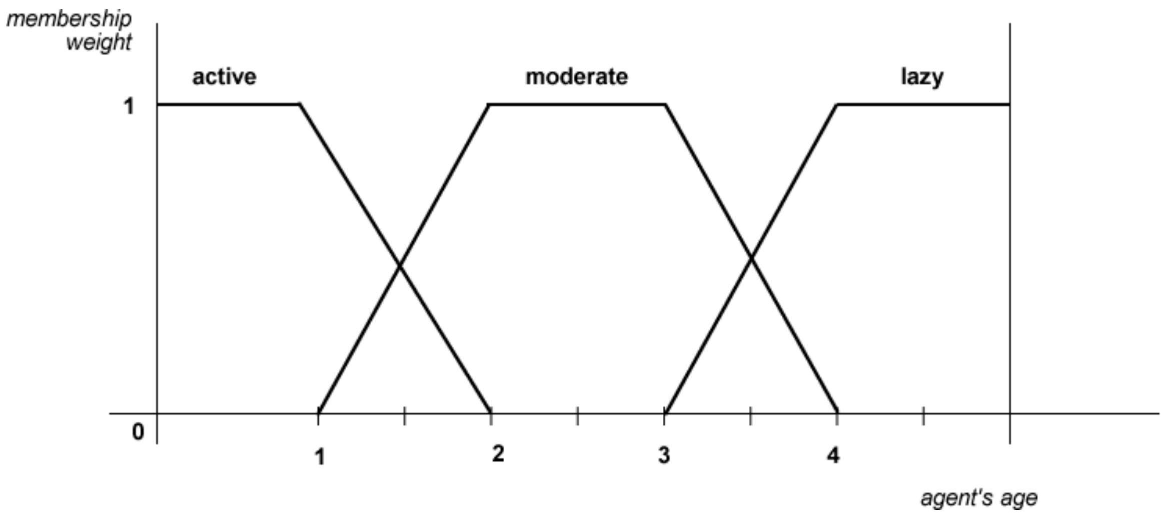


Food factor:

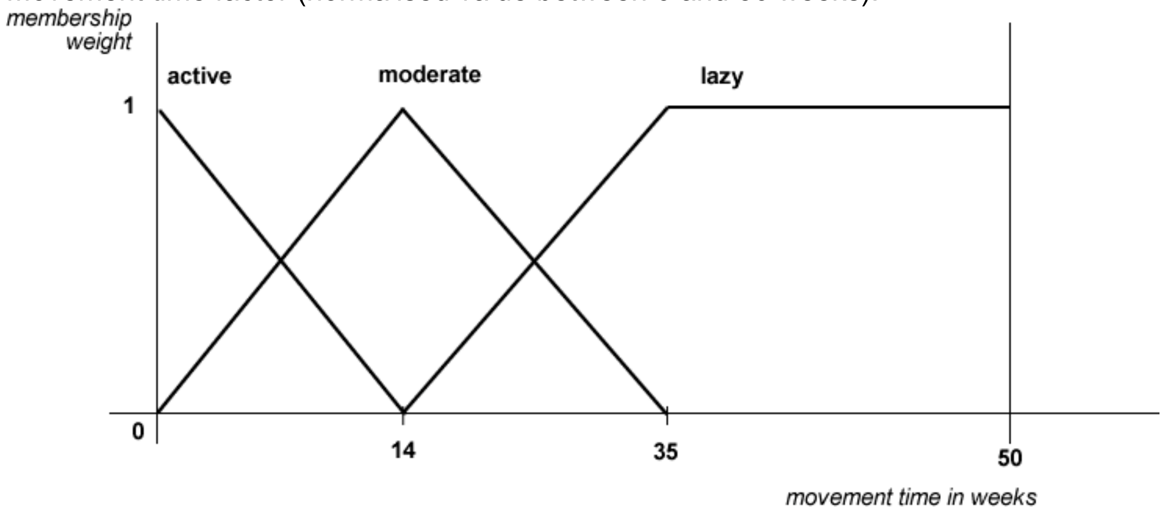


Activity fuzzy sets

Age factor:



Movement time factor (normalised value between 0 and 50 weeks):



Appendix S:
Defuzzification of happiness and activity

```
/*
    GetActivityWeight()
    - represent overal activity by a number between -1 and 1
    - used when getting crisp fuzzy output for afffected functions
*/
double CAgentInnerSensorEngine::GetActivityWeight(){

    /* EXAMPLE: use fuzzy logic to calculate speed which can have values
<0.8;2>:
    middle of speed values = 1.4,
    therefore MODERATE has 1.4 weight, LAZY 0.8, ACTIVE 2,
    speedAddition = lazyDeg*0.8 + moderateDeg*1.4 + activeDeg*2

    => GENERALISED FUNCTION: determine weight of adding/removing from a
middle of possible values towards max/min
          1. calulate result TEMP if possible values were <-1;1>, with
middle 0.1
          2. transfer a number from this scale to needed scale, e.g.
<0.8,2> by finding the middle M of needed scale and value V from the
middle to both ends. Than multiply V by TEMP and
add it to M.
    */

    double lazyDeg = GetLazyDeg();
    double moderDeg = GetModerateDeg();
    double activeDeg = GetActiveDeg();
    double temp = lazyDeg*(-1) + moderDeg*0.1 + activeDeg*1;
    return temp;
}

/*
    GetActivityScale()
    - represent overal activity by a number between 0 and 1:
    - used mostly for the activity bar
*/
double CAgentInnerSensorEngine::GetActivityScale(){
    double weight = GetActivityWeight(); //values = <-1;1>
    //------- now convert to <0;1>:
    double scale = (weight+1)/2;
    return scale;
}

double CAgentInnerSensorEngine::GetHapinessWeight(){
    double happyDeg = GetHappyDeg();
    double normalDeg = GetNormalDeg();
    double sadDeg = GetSadDeg();
    double temp = sadDeg*(-1) + normalDeg*0.1 + happyDeg*1;
    return temp;
}
```

```
double CAgentInnerSensorEngine::GetHapinessScale(){
    double weight = GetHapinessWeight(); //values = <-1;1>
    //------- now convert to <0;1>:
    double scale = (weight+1)/2;
    return scale;
}
```

Appendix T: Creating speech

During initiation of the speech array, it is filled with sentences of the String type. The sentences have place holders for numeric values denoted by '%d' characters. For example, to initiate sentences for the breeding action, the following code was used:

```
//--------------------- sad
m_sActionSpeech[int(eActionMultiply)][0][0] = "Another %d useless lives
will be created in %d days";
    m_sActionSpeech[int(eActionMultiply)][0][1] = "Do I have to clone
myself into %d waggans again?";
//--------------------- normal
m_sActionSpeech[int(eActionMultiply)][1][0] = "There will be %d more
waggans here in %d days";
    m_sActionSpeech[int(eActionMultiply)][1][1] = "I shall make %d clones
of me in %d days.";
//--------------------- happy
m_sActionSpeech[int(eActionMultiply)][2][0] = "I am looking forward to
see my %d new clones in %d days!!";
    m_sActionSpeech[int(eActionMultiply)][2][1] = "My %d new clones will
be definitelly perfect";
```

There are two things needed to be mentioned about the sentences: the first is that the place holders for integer values need to be in the same order because they are replaced in an algorithm executed later on. For example, in the 0th sentence seed (the seed is the 3rd dimension of the array) the number of clones is always referenced first, followed by the number of remaining days.
Secondly, instead of referencing the 1st dimension of the array (the one related to the task type) by a hard-coded integer number, conversion to integer from enumeration task type was used. This ensures that if the enumeration order changes, right sentences will still be displayed.

When an agent is given a task or is selected, it changes its speech text. The function RandomizeSpeech is called. This function sets the sentence id (based on the current task), sentence happiness id and sentence seed to an appropriate number:

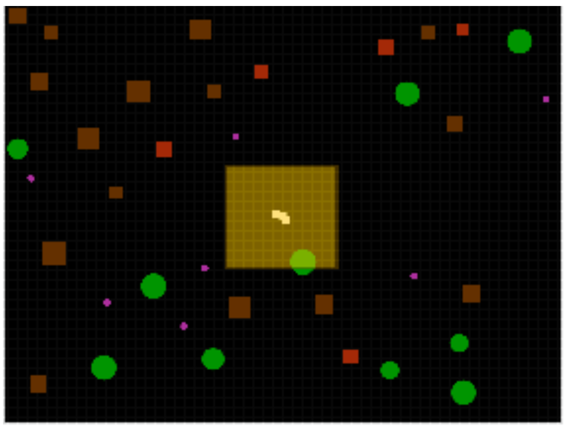
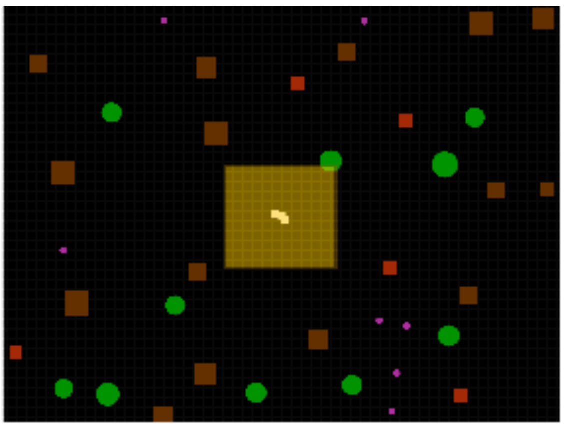
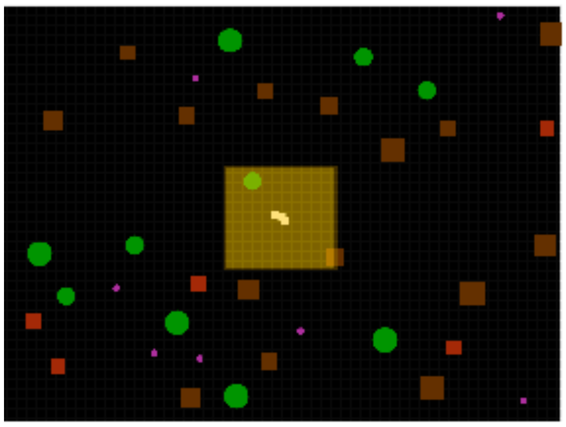
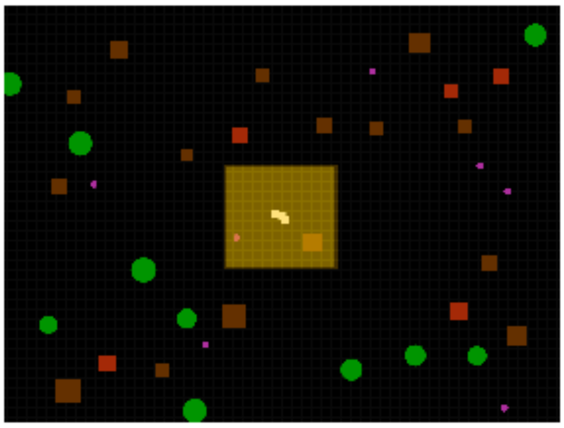
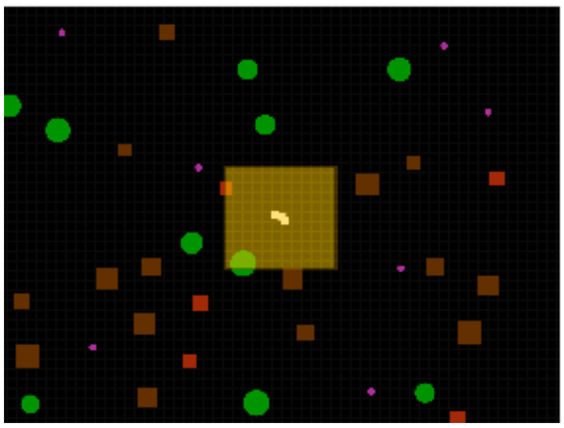
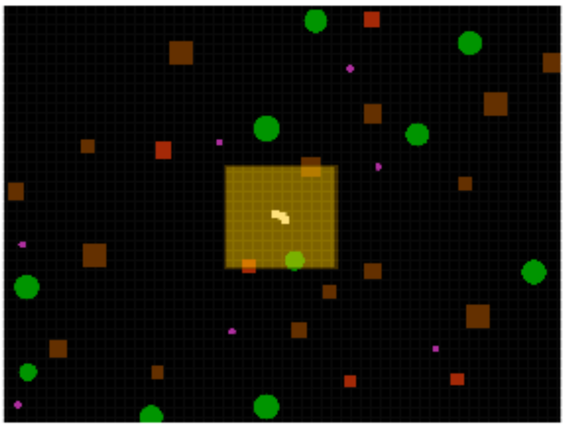
```
void CAgentBehaviourEngine::RandomizeSpeech(bool firstTime_){
    eActionType type = m_pAgent->GetMemoryEngine()->
GetCurAction().type;
    m_iSentenceId = int(type);
    //---------------------- find out what happiness:
    /*
          evaluate happiness from scale <0;1> whee 0 = 100% sad:
    */
    double happiness;
    if (firstTime_){
          happiness = 0.5;
    } else {
          happiness = m_pAgent->GetInnerSensorEngine()->
GetHapinessScale();
    }
```

```
        if (happiness < 0.3){
              //----- is sad:
              m_iSentenceHappinness = 0;
        } else if (happiness > 0.7){
              //------ is happy:
              m_iSentenceHappinness = 2;
        } else {
              //------ is normal:
              m_iSentenceHappinness = 1;
        }

        //---- choose a random sentence for a particular action: [DaniWeb]
        int range=2;
        m_iSentenceSeed = int(range*rand()/(RAND_MAX+1.0));
    }
```

Finally, the Handle Speech method fills a selected sentence with an appropriate integer value and sends the text to the Speech bubble for displaying. The body of this function is too long to be fully listed here. The following code shows how an integer value is put into a string from the Speech array if the current task is breeding and the sentence seed is 0 (the case discussed above):

```
int num1 = int(m_iNextNumOfChildren);
int num2 = ceil(m_fCarriageAmount);

//------- since the sprintf function only works with char* and prints
into a char[] variable, they need to be created first:
char outp[100] = "";
const char*  sentence =
m_sActionSpeech[m_iSentenceId][m_iSentenceHappinness][m_iSentenceSeed].c_
str();

//------------- inserting the integers into the sentence:
sprintf(outp,sentence,num1,num2);
//--------- convert back to a string:
string output = outp;

//-------------- send to the speech bubble:
m_pAgent->GetSpeechBubble()->Say(output);
```

Appendix U:
Random maps screenshots

Legend:

🟢 swamp 🟫 rock 🟥 trees 🟪 crystal ☐ alien

The yellow square represents the part of map which is currently being viewed.

Appendix V: Evaluation questionnaire answers

TESTER 1

ABOUT YOU

1. How many hours per week/month do you spend playing computer or console games (please fill in one of the boxes)?

20	per week	OR		per month

2. Put a numbers 1-3 next to three of the listed game genres, where 1 means the one you play the most:

3	First person shooter (Max Payne, Battlefield 1943)	2	Strategy (Warcraft, Age of Empires)
2	RPG (Diablo, Dungeons & Dragons)	1	MMOG (Lords of Everquest, World of Warcraft)
2	Racing game (Need for Speed, Top Gear)	1	Adventure (Zork, Escape from Monkey Island)
1	Action-adventure (GTA, Prince of Persia)	3	Simulation (F22, Orbiter)
1	Life simulation (Sims, Jurrasic Park: Operation genesis)		

GAME PLAY AND INTERFACE

Always tick only one box in the following questions. There are 5 possible answers in each question and the first and last answer values are described above the tick boxes. You can 'tick' a box by putting 'X' inside it.

1. How much did you learn in the tutorial as opposed to the actual game play?

Most in the tutorial — most during the game

	X			

2. Rate stability of a game - did it crash?

Unstable — Stable

			X	

3. How many times didn't you know what to do with the interface (e.g. you couldn't find a button or information)?

I was confused all the time The interface was easy to understand

			X	

4. How many times didn't you know what to do in the game (e.g. how to make aliens happier, how to build, etc)?

I was confused all the time It was always clear what I needed to do

				X

PATH FINDING

1. How many times did an alien get to a target location in the way you would have expected (i.e. the most effective way)?

Almost never Most of the time

			X	

2. How many times did an alien bump into an obstacle or went straight through it?

Almost never Most of the time

		X		

BEHAVIOUR OF ALIENS

1. How much did speech of aliens differ from each other?

Not at all Very much

			X	

2. How much did sounds of aliens differ from each other?

Not at all Very much

	X			

3. How much did a way aliens performed their task differ from each other (e.g. speed of gathering, movement, number of days they needed to breed)?

Not at all Very much

		X		

4. In overall, do you consider the differences in behaviour to be good for the game play?

Bad / disturbing				Good
☐	☒	☐	☐	☐

Explain why:

It is impossible to have an exact plan - e.g. alien changes its path, it takes more time to get to a farm which means all aliens die...

5. How many times did aliens behave unexpectedly?

Never				Very often
☐	☐	☐	☒	☐

Write more about any unexpected behaviour:

Sometimes when it bumped into an obstacle, it changes the direction by 180 degrees (not considering where the target was), then it bumped into another obstacle and only after that found the right way

TESTER 2

ABOUT YOU

3. How many hours per week/month do you spend playing computer or console games (please fill in one of the boxes)?

   [20] per week OR [ ] per month

4. Put a numbers 1-3 next to three of the listed game genres, where 1 means the one you play the most:

1	First person shooter (Max Payne, Battlefield 1943)	3	Strategy (Warcraft, Age of Empires)
	RPG (Diablo, Dungeons & Dragons)		MMOG (Lords of Everquest, World of Warcraft)
	Racing game (Need for Speed, Top Gear)		Adventure (Zork, Escape from Monkey Island)
	Action-adventure (GTA, Prince of Persia)		Simulation (F22, Orbiter)
2	Life simulation (Sims, Jurrasic Park: Operation genesis)		

GAME PLAY AND INTERFACE

Always tick only one box in the following questions. There are 5 possible answers in each question and the first and last answer values are described above the tick boxes. You can 'tick' a box by putting 'X' inside it.

5. How much did you learn in the tutorial as opposed to the actual game play?

   Most in the tutorial — most during the game

   [ ] [ ] [ ] [x] [ ]

6. Rate stability of a game - did it crash?

   Unstable — Stable

   [ ] [ ] [ ] [ ] [x]

7. How many times didn't you know what to do with the interface (e.g. you couldn't find a button or information)?

   I was confused all the time — The interface was easy to understand

   [ ] [ ] [x] [ ] [ ]

8. How many times didn't you know what to do in the game (e.g. how to make aliens happier, how to build, etc)?

I was confused all the time				It was always clear what I needed to do
X				

PATH FINDING

3. How many times did an alien get to a target location in the way you would have expected (i.e. the most effective way)?

Almost never				Most of the time
			X	

4. How many times did an alien bump into an obstacle or went straight through it?

Almost never				Most of the time
X				

BEHAVIOUR OF ALIENS

6. How much did speech of aliens differ from each other?

Not at all				Very much
	X			

7. How much did sounds of aliens differ from each other?

Not at all				Very much
			X	

8. How much did a way aliens performed their task differ from each other (e.g. speed of gathering, movement, number of days they needed to breed)?

Not at all				Very much
			X	

9. In overall, do you consider the differences in behaviour to be good for the game play?

Bad / disturbing				Good
			X	

Explain why:

It makes it more enjoyable. Though sometimes it appeared to be quite random

10. How many times did aliens behave unexpectedly?

Never				Very often
	x			

Write more about any unexpected behaviour:

Sometimes they slowed down quite a lot when running around obstacles.

TESTER 3

ABOUT YOU

5. How many hours per week/month do you spend playing computer or console games (please fill in one of the boxes)?

4-5	per week	OR		per month

6. Put a numbers 1-3 next to three of the listed game genres, where 1 means the one you play the most:

1	First person shooter (Max Payne, Battlefield 1943)	2	Strategy (Warcraft, Age of Empires)
	RPG (Diablo, Dungeons & Dragons)		MMOG (Lords of Everquest, World of Warcraft)
	Racing game (Need for Speed, Top Gear)		Adventure (Zork, Escape from Monkey Island)
	Action-adventure (GTA, Prince of Persia)	3	Simulation (F22, Orbiter)
	Life simulation (Sims, Jurrasic Park: Operation genesis)		

GAME PLAY AND INTERFACE

Always tick only one box in the following questions. There are 5 possible answers in each question and the first and last answer values are described above the tick boxes. You can 'tick' a box by putting 'X' inside it.

9. How much did you learn in the tutorial as opposed to the actual game play?

Most in the tutorial				most during the game
		X		

10. Rate stability of a game - did it crash?

Unstable				Stable
		X		

11. How many times didn't you know what to do with the interface (e.g. you couldn't find a button or information)?

I was confused all the time				The interface was easy to understand
				X

12. How many times didn't you know what to do in the game (e.g. how to make aliens happier, how to build, etc)?

I was confused all the time				It was always clear what I needed to do
☐	☐	☐	☐	X

PATH FINDING

5. How many times did an alien get to a target location in the way you would have expected (i.e. the most effective way)?

Almost never				Most of the time
☐	☐	☐	☐	X

6. How many times did an alien bump into an obstacle or went straight through it?

Almost never				Most of the time
X	☐	☐	☐	☐

BEHAVIOUR OF ALIENS

11. How much did speech of aliens differ from each other?

Not at all				Very much
☐	☐	☐	X	☐

12. How much did sounds of aliens differ from each other?

Not at all				Very much
☐	☐	☐	X	☐

13. How much did a way aliens performed their task differ from each other (e.g. speed of gathering, movement, number of days they needed to breed)?

Not at all				Very much
☐	X	☐	☐	☐

14. In overall, do you consider the differences in behaviour to be good for the game play?

Bad / disturbing				Good
☐	☐	X	☐	☐

Explain why:

good because it makes the game harder, bad because it made any planning impossible

15. How many times did aliens behave unexpectedly?

Never				Very often
		X		

Write more about any unexpected behaviour:

numbers of new waggans when breeding, preference of their jobs

More comments:
Tutorial: I found out there how to control the game, there was everything there about that. On the other hand I couldn't find out where to build buildings (especially farms) which was not in the turorial and I pretty much didn't find it out even during the game :) - sometimes I could have two next to each other so that they were touching, sometimes I could put maximum 1 farm on the green land. Also sometimes it wrote that I couldn't build houses and silos on a certain place and I have no idea why and I don't know if there is a connection between their life length, happiness and the food they have, maybe I would figure out somehow after longer time but there were too many of them.

Stability of the game: when I saved it and then ran a saved position it always switched off after about a minute. But when I started a new game it didn't crash at all, the problems were caused by loading saved games. Also bearing in mind the official hardware requirements it was needed quite a lot of CPU power, it used approximately 50% of it (I have 3GHz Athlon X2)

Interface was well-arranged and well described in the tutorial, I didn;t have any problems with it (if I don't consider it is not working under a high resolution)
Path finding: they moved more or less straight, they avoided the buildings, walked only through farms, they went slower across the green fields, overall they didn't have problems with movement.

Behaviour: as opposed to other games (e.g. age of empires) the talking and sounds quite differed, it was obvious that it repeats but as much often to look monotonous. the movement speed was more or less the same, gathering speed as well (of one type, the gathering speed on a farm was quite higher than close to a crystal), however the number of breeding days was in fact always different (in a certain interval)

In terms of judging the differences, it made planning quite hard and sometimes it looked more like an artificial chaos than artificial intelligence :) for example it often happened that a waggan said it would make a number of clones and than there was a different number of them there (in the beginning the problem was mainly that when this happened I had a problem to keep up with the food). But on the other hand at least it was not too easy, in most of the strategy games it is enough to figure out one strategy and just keep it up. The only problem was that in the beginning when I had three of them they all wanted to go to one farm, or when some of them didn't want to do anything. It would be easier if there was a job preference shown after clicking on a unit so that you can find out what it would like to do, otherwise I cannot figure it out, only after I give it a task.

TESTER 4

ABOUT YOU

7. How many hours per week/month do you spend playing computer or console games (please fill in one of the boxes)?

20	per week	OR		per month

8. Put a numbers 1-3 next to three of the listed game genres, where 1 means the one you play the most:

3	First person shooter (Max Payne, Battlefield 1943)	2	Strategy (Warcraft, Age of Empires)
	RPG (Diablo, Dungeons & Dragons)	1	MMOG (Lords of Everquest, World of Warcraft)
	Racing game (Need for Speed, Top Gear)		Adventure (Zork, Escape from Monkey Island)
	Action-adventure (GTA, Prince of Persia)		Simulation (F22, Orbiter)
	Life simulation (Sims, Jurrasic Park: Operation genesis)		

GAME PLAY AND INTERFACE

Always tick only one box in the following questions. There are 5 possible answers in each question and the first and last answer values are described above the tick boxes. You can 'tick' a box by putting 'X' inside it.

13. How much did you learn in the tutorial as opposed to the actual game play?

Most in the tutorial				most during the game
			X	

14. Rate stability of a game - did it crash?

Unstable				Stable
			X	

15. How many times didn't you know what to do with the interface (e.g. you couldn't find a button or information)?

I was confused all the time				The interface was easy to understand
		X		

16. How many times didn't you know what to do in the game (e.g. how to make aliens happier, how to build, etc)?

I was confused all the time — It was always clear what I needed to do

	X			

PATH FINDING

7. How many times did an alien get to a target location in the way you would have expected (i.e. the most effective way)?

Almost never — Most of the time

			X	

8. How many times did an alien bump into an obstacle or went straight through it?

Almost never — Most of the time

	X			

BEHAVIOUR OF ALIENS

16. How much did speech of aliens differ from each other?

Not at all — Very much

	X			

17. How much did sounds of aliens differ from each other?

Not at all — Very much

		X		

18. How much did a way aliens performed their task differ from each other (e.g. speed of gathering, movement, number of days they needed to breed)?

Not at all — Very much

		X		

19. In overall, do you consider the differences in behaviour to be good for the game play?

Bad / disturbing — Good

			X	

Explain why:

because it is at least a bit more difficult to play, otherwise everything would be the same...

20. How many times did aliens behave unexpectedly?

Never				Very often
X				

Write more about any unexpected behaviour:

I would allow for more time before aliens die because when I played it for the first couple of times they all died in a while and I had to start again.. and I read the whole tutorial but I couldn't figure out what to do with that.

TESTER 5

ABOUT YOU

1. How many hours per week/month do you spend playing computer or console games (please fill in one of the boxes)?

50	per week	OR		per month

2. Put a numbers 1-3 next to three of the listed game genres, where 1 means the one you play the most:

2	First person shooter (Max Payne, Battlefield 1943)	1	Strategy (Warcraft, Age of Empires)
3	RPG (Diablo, Dungeons & Dragons)	3	MMOG (Lords of Everquest, World of Warcraft)
2	Racing game (Need for Speed, Top Gear)	2	Adventure (Zork, Escape from Monkey Island)
3	Action-adventure (GTA, Prince of Persia)	1	Simulation (F22, Orbiter)
3	Life simulation (Sims, Jurrasic Park: Operation genesis)		

GAME PLAY AND INTERFACE

Always tick only one box in the following questions. There are 5 possible answers in each question and the first and last answer values are described above the tick boxes. You can 'tick' a box by putting 'X' inside it.

1. How much did you learn in the tutorial as opposed to the actual game play?

Most in the tutorial				most during the game
			x	

2. Rate stability of a game - did it crash?

Unstable				Stable
	x			

3. How many times didn't you know what to do with the interface (e.g. you couldn't find a button or information)?

I was confused all the time				The interface was easy to understand
			x	

4. How many times didn't you know what to do in the game (e.g. how to make aliens happier, how to build, etc)?

I was confused all the time				It was always clear what I needed to do
		x		

PATH FINDING

1. How many times did an alien get to a target location in the way you would have expected (i.e. the most effective way)?

Almost never				Most of the time
	x			

2. How many times did an alien bump into an obstacle or went straight through it?

Almost never				Most of the time
x				

BEHAVIOUR OF ALIENS

1. How much did speech of aliens differ from each other?

Not at all				Very much
		x		

2. How much did sounds of aliens differ from each other?

Not at all				Very much
x				

3. How much did a way aliens performed their task differ from each other (e.g. speed of gathering, movement, number of days they needed to breed)?

Not at all				Very much
				x

4. In overall, do you consider the differences in behaviour to be good for the game play?

Bad / disturbing				Good
			x	

Explain why:

Because I can use them more effectively, they do more work !!!

5. How many times did aliens behave unexpectedly?

Never				Very often
	x			

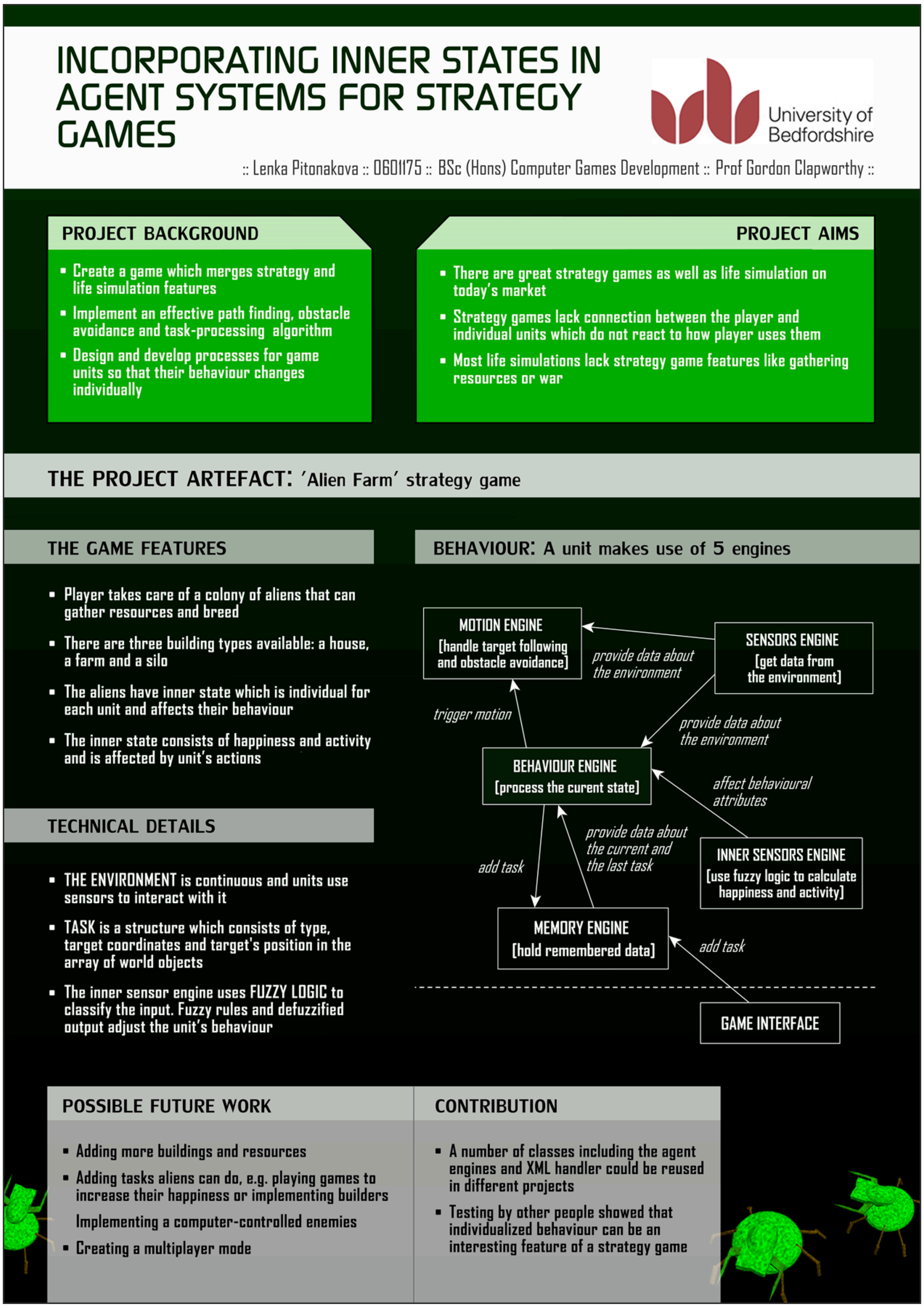
Write more about any unexpected behaviour:

About 3 times an alien stopped moving and started blinking, once all 3 died in the beginning after about 15 seconds.

They go slower through those grass fields, they should avoid them, it would be more effective, the energies tab can't be switched back to the buildings menu. And it read my save and switched the game off and I couldn't continue the game.

Moving the map could work with mouse too and display population number!!!!! otherwise a great idea, expand it !!! (: thanks for the game!

Appendix W:
The Project Poster



INCORPORATING INNER STATES IN AGENT SYSTEMS FOR STRATEGY GAMES

University of Bedfordshire

:: Lenka Pitonakova :: 0601175 :: BSc (Hons) Computer Games Development :: Prof Gordon Clapworthy ::

PROJECT BACKGROUND

- Create a game which merges strategy and life simulation features
- Implement an effective path finding, obstacle avoidance and task-processing algorithm
- Design and develop processes for game units so that their behaviour changes individually

PROJECT AIMS

- There are great strategy games as well as life simulation on today's market
- Strategy games lack connection between the player and individual units which do not react to how player uses them
- Most life simulations lack strategy game features like gathering resources or war

THE PROJECT ARTEFACT: 'Alien Farm' strategy game

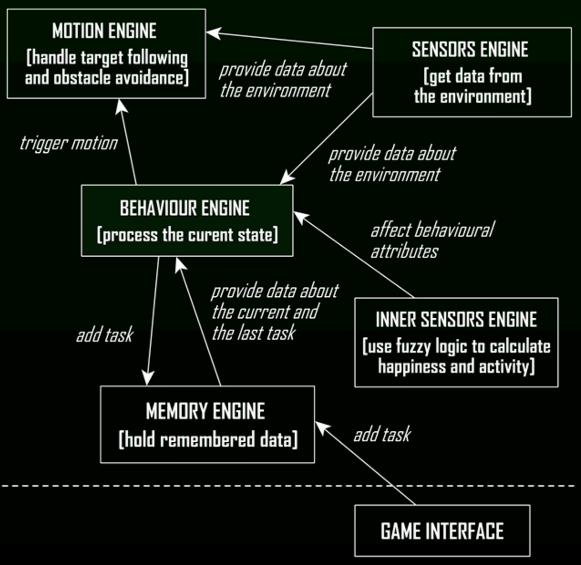
THE GAME FEATURES

- Player takes care of a colony of aliens that can gather resources and breed
- There are three building types available: a house, a farm and a silo
- The aliens have inner state which is individual for each unit and affects their behaviour
- The inner state consists of happiness and activity and is affected by unit's actions

TECHNICAL DETAILS

- THE ENVIRONMENT is continuous and units use sensors to interact with it
- TASK is a structure which consists of type, target coordinates and target's position in the array of world objects
- The inner sensor engine uses FUZZY LOGIC to classify the input. Fuzzy rules and defuzzified output adjust the unit's behaviour

BEHAVIOUR: A unit makes use of 5 engines


MOTION ENGINE [handle target following and obstacle avoidance]
SENSORS ENGINE [get data from the environment]
provide data about the environment
trigger motion
provide data about the environment
BEHAVIOUR ENGINE [process the curent state]
affect behavioural attributes
provide data about the current and the last task
add task
INNER SENSORS ENGINE [use fuzzy logic to calculate happiness and activity]
MEMORY ENGINE [hold remembered data]
add task
GAME INTERFACE

POSSIBLE FUTURE WORK

- Adding more buildings and resources
- Adding tasks aliens can do, e.g. playing games to increase their happiness or implementing builders
  Implementing a computer-controlled enemies
- Creating a multiplayer mode

CONTRIBUTION

- A number of classes including the agent engines and XML handler could be reused in different projects
- Testing by other people showed that individualized behaviour can be an interesting feature of a strategy game