Hochschule Esslingen
University of Applied Sciences

Bachelor's Thesis

Decision-Making in a Group of Artificial Intelligences in Different Simulated Environments

Faculty of Information Technology
Software Engineering and Media Informatics

Philipp Erler

Time period: September 1, 2013 to January 31, 2014
First examiner: Prof. Dr.-Ing. Reinhard Schmidt
Second examiner: Prof. Dr.-Ing. Andreas Rößler

Company: Chasing Carrots KG
Adviser: Dominik Schneider

Chasing Carrots
Delicious Game Creations

Affidavit

I hereby declare that this bachelor’s thesis has been written only by the undersigned and without any assistance from third parties. Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Esslingen, January 20, 2013 ________________________________

Foreword

The work on the artificial intelligence for Chasing Carrots and Cosmonautica was and still is a very interesting task for me. It was only possible because several people gave me this opportunity.

Therefore, I want to say “thank you” to:

- my parents for giving me the chance to study without a lot of distraction
- my proofreaders, Katherine Fraser and Friedrich Brilling
- the team of Chasing Carrots for offering this interesting work and supporting me

Reading about the behavior of characters in a computer game hopefully gives you the knowledge to create your own behaviors. However, you will most likely need to see the game running in order to understand all the thoughts behind this work.

Therefore, I encourage you to sign up at the Cosmonautica forum at http://cosmo-nautica.com/ and play the current pre-alpha version. Chasing Carrots is always looking for testers!

Abstract

Natural behavior of a single artificial intelligence or even entire groups is a great challenge but also necessary for the immersion in video games. The object of this bachelor thesis is to find and implement the best technique for natural artificial intelligence behavior in “Cosmonautica”, Chasing Carrots’ next game.

The well-known finite state machines and the more sophisticated hierarchical finite state machines are simple but can hardly be re-used. Goal-oriented behavior (GOB) shines when used for planning a sequence of actions in the goal-oriented action planning technique. The utility functions used in GOB are relatively simple and useful for comparing many options. Another considered technique is fuzzy logic, which uses so-called degrees of memberships to represent the states of game objects. These memberships are well suited for perceptions like feeling cold or endangered. Rule-based systems can become complicated but support fine-grained behaviors. Behavior trees work with a tree structure and several node types to decide which actions are triggered. They are very flexible and can be modified very fast to be used in completely different situations.

The comparison of the different decision-making techniques shows that behavior trees combined with utility functions are the best choice for “Cosmonautica” due to their overall composition of power, performance, complexity, re-usability, and maintainability.

Now, behavior trees are used in “Cosmonautica” to control the behavior of crewmembers in the player's space ship as well as the enemy ships in space fights. They determine e.g. in which situations the crew members work or when a need has to be satisfied. Utility functions are used to compare all possible actions to decide which work task or activity needs to be done.

This combination of behavior trees and utility functions has proved to be an easy and powerful way to control the behavior of artificial intelligences. Because of its qualities, it can and should replace many state machines in object-oriented programs.

Key words: Artificial intelligence, Decision-making, Behavior, Behavior tree, Utility function, Finite-state machine, Fuzzy logic, Goal-oriented behavior, Rule-based system

Contents

Figures

Tables

List of abbreviations

AI	Artificial Intelligence
BT	Behavior Tree
DAG	Directed Acyclic Graph
FPS	Frames Per Second
FSM	Finite-State Machine
GOB	Goal-Oriented Behavior
GUI	Graphical User Interface
HSM	Hierarchical State Machine
NPC	Non-Player Character
UML	Unified Modeling Language

1. Overview

The requirements of software projects can vary with the situation of the involved companies. The practical part of this thesis is used in a project of a very small development team and therefore has to respect some special requirements. Because of this, the overall situation of the company and the project is important for understanding this thesis. Cosmonautica[1] is the second game of the small independent game development studio Chasing Carrots KG[2] in Stuttgart, Germany. It is currently in a very early stage of development. Some parts such as the crewmember-class do already exist but their behaviors are just placeholders. The new decision-making system has to fit between the existing classes. Some work has been done but there is much more to do. Therefore, fast development is not only very important for Cosmonautica but also for the game development in general. This means for the development staff that it focuses more on fast results than on good documentation. As a result, only a few unit tests are made and these are just for the core systems like the behavior trees. These unit tests are also meant as a kind of a tutorial showing how to use the system. The core systems are part of the code base that Chasing Carrots will use for further games.

Cosmonautica is a game combining Sims[3]-like management of the crewmembers of a space ship with trading and fighting in space. Crewmembers choose their actions on their own based on their skills and needs, instead of being chosen directly by the players. This decision-making process is the main object of this thesis. The same system will also be used for different situations such as crewmembers being on ground missions or being on the ship in space fights.

The natural and logical behavior of non-player characters is crucial for the immersion of most games. As the player has no direct control over the decisions of their crewmember, it is even more important to create behaviors without “brain failures”. Deciding to repair the wrong room could be such a dead wrong decision, which would lead to the frustration of the players.

Decision-making is a part of the artificial intelligence. Academic AI usually shows how intelligent a system can be. Game AI in contrast has the aim to create good-looking behavior with emphasize on “looking”. It can happen that a more stupid behavior enables a better game experience. On the pages 19 and 20 in (Millington & Funge, 2009) is described how the relatively stupid behavior of the ghosts in Pac-Man can look intelligent in a group. However, the behaviors in Cosmonautica will require more effort than the semi-random decisions of Pac-Man. Intelligent game characters are relevant for the game design as they may affect the difficulty.

[1] http://cosmo-nautica.com/
[2] http://chasing-carrots.com/
[3] http://www.thesims.com/

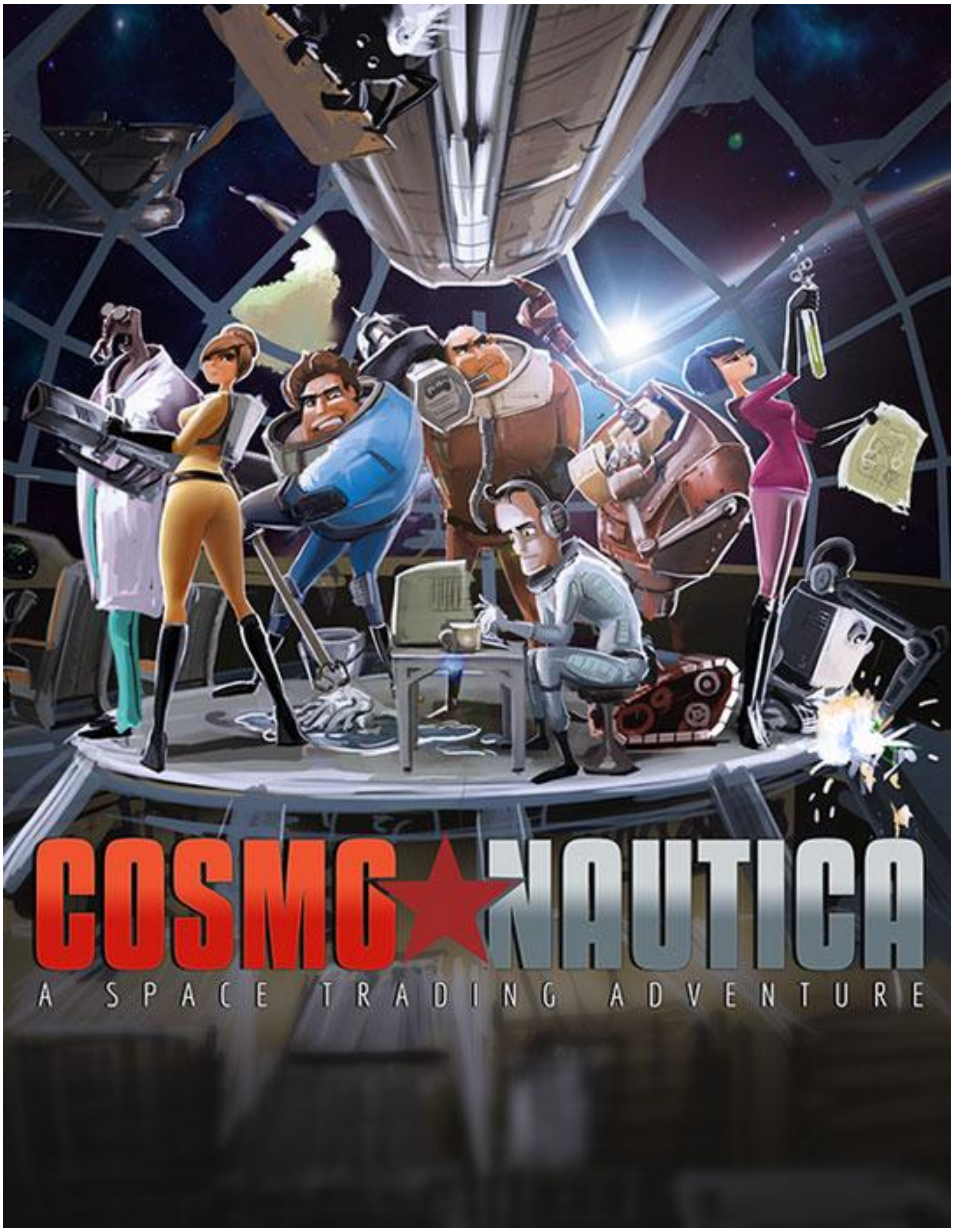


Figure 1: Key visual of Cosmonautica

This concept art shows several parts and possible features of Cosmonautica: crewmembers on a space ship attacking another ship with a torpedo; characters doing tasks like hacking, science, repairing, and cleaning; robots; a spider-like critter; and an extraterrestrial.

2. Task and Scope

To prepare the practical part of this thesis, all decision-making techniques that come into question have to be compared in order to find the best one for Cosmonautica. Afterwards, the chosen technique has to be implemented with re-usability in mind, re-usability in general and the re-usability of behaviors in particular. As the planned system will be a part of games with reasonable high hardware requirements, it has to be implemented in C++ just like the games themselves. As a part of the Chasing Carrots code base, the decision-making core system has to be independent from individual games. The behaviors for Cosmonautica have to be in the Cosmonautica base code what means that they must be independent from the game's engine.

Many decision-making processes need knowledge about all available options, which will change regularly with the game design. Therefore, the decision-making system needs to be flexible enough to support every change. Although no game designer is modeling the behaviors right now, it can be necessary in the future. This means that easy modification of the behavior by non-programmers is an aim, which should be kept in view. Maybe the behaviors have to be controlled by scripts in the future to keep the modification easy for the game designers.

The decision-making system is meant mainly for the crewmembers of the player’s space ship. It is planned to choose the work they do or how to satisfy their needs. Then, the system should control how they do what they do to ensure everything looks natural. In short, the decision-making system must be powerful enough to support Sims-like behavior and it must be flexible enough for any possible change in the future. Crewmembers like these cannot wait to receive some intelligence:

Figure 2: Random-generated crewmember portraits

3. State of the Art

Several techniques for decision-making are available but the games industry is mostly using state machines. Only a few bigger developers work with advanced techniques like goal-oriented behavior. Behavior trees are experiencing an increasing popularity right now. Fuzzy logic is "[…] largely discredited within the mainstream academic AI community" (Millington & Funge, 2009, p. 371). Nevertheless, it was used in several games. Although rule-based systems are relatively popular as a technique for expert systems, they are hardly used for games.

STRIPS and Hierarchical Task Networks are not described here because they are planning algorithms. As planning is not necessary in Cosmonautica and these algorithms are usually quite complicated, they are not considered. Blackboard architectures are also not described, as they are only useful in cases where multiple decision makers search a solution together. Such a cooperative artificial intelligence is not planned for Cosmonautica. Most of the information in this chapter is extracted from chapter 5 "Decision Making" of (Millington & Funge, 2009). Another great source of information about artificial intelligence for games is AiGameDev.com, a commercial website by Alex J. Champandard.

3.1. State Machines

Millington and Funge (2009) say on page 309 about the popularity of state machines that they are used in the "vast majority of decision-making systems used in current games." They are also widely used for the low-level programming of embedded systems such as digital watches or vending machines. As they are intuitive, they are also used for normal applications. Games often have state machines for both game managers and common game objects like non-player characters.

Modeling is usually done in the UML notation. Several tools exist for aiding engineers to model and program. Some of these tools, such as QM by Quantum Leaps[4] even convert the model into a code skeleton, which is typically C-code. State machines can be combined with other techniques, for example fuzzy logic, decision trees, and Markov systems.

As stated in (Millington & Funge, 2009) on page 315, state machines in general are slow because of "many virtual method calls". "They are also difficult to maintain [...]", (Millington & Funge, 2009, p. 318). This applies to both finite-state machines and hierarchical state machines.

3.1.1. Finite-State Machines

Finite-state machines can have exactly one state at a time. Their states are connected by transitions. Those transitions can have conditions and actions. States can execute code on "enter" and "leave" events. A state may also trigger code every time it is updated while running. (cf. Millington & Funge, 2009, p. 309/310)

Here is an example that shows how a simple guard behavior in a fantasy game context can be modeled with a finite-state machine. This guard starts with the patrol

[4] http://www.state-machine.com/qm/

state. When the patrol state is entered or left, the guard reports to his sergeant. As long as he is patrolling, he follows his path. As soon as he notices an enemy, he switches to the “engage enemy” state what causes him draw his weapon. He follows and attacks the enemy. When no enemy is in sight, no matter if he has been defeated or just disappeared, the guard puts away his weapon and returns to the patrol state.

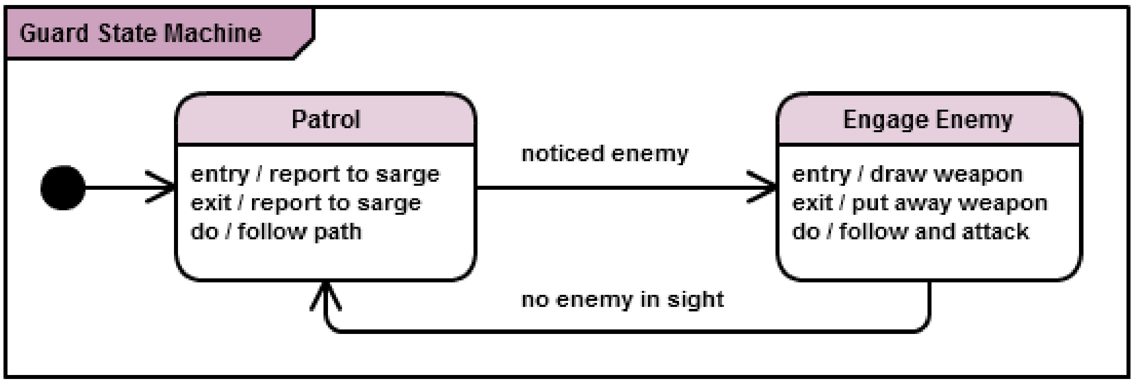


Figure 3: Simple guard behavior as a finite-state machine

Now, the guard’s behavior is to be extended by drinking a healing potion if he is hurt. Therefore, a “drink potion” state is required. This state must be connected not only with the “engage enemy” state but also with the “patrol” state. Otherwise, he would be hobbling on his patrol after fighting. After drinking the potion, the guard should return immediately to the state he just left. Therefore, two “drink potion” states are required because a normal state does not remember from where it was entered.

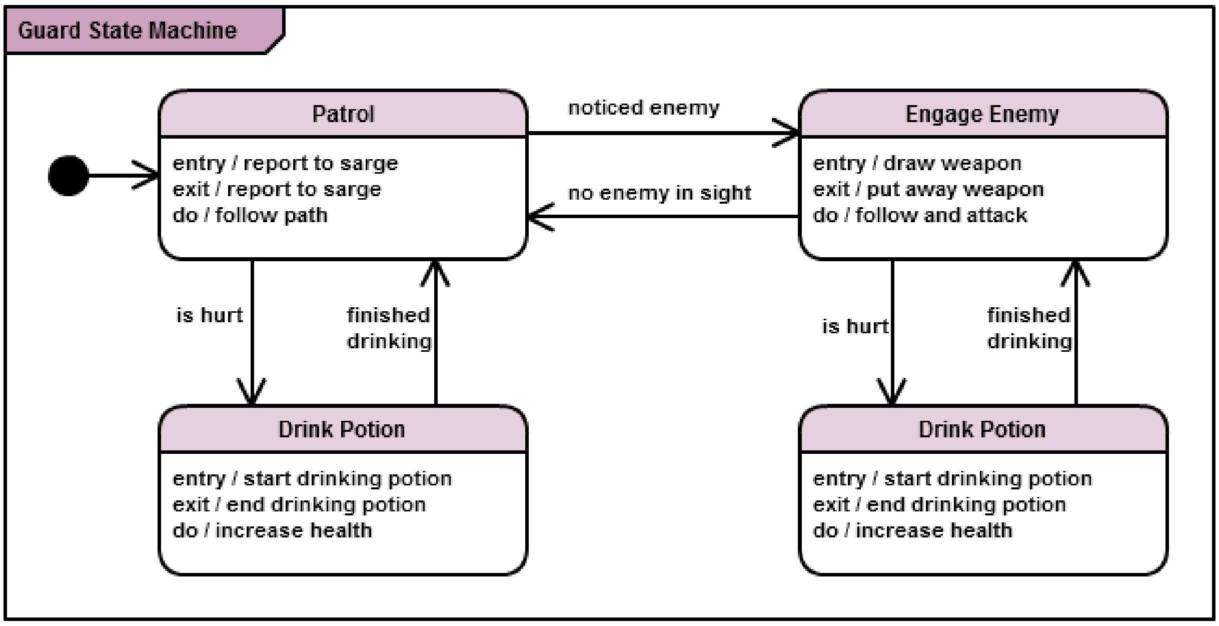


Figure 4: Extended guard sample

Another solution could be to make only one “drink potion” state but with a transition to both the “patrol” and the “engage enemy” state guarded by special condition checks. This solution has the drawback that a transition has to be added every time another state has been added. This way, the state machine will become less and less re-usable. No matter which solution is used, the finite-state machine will be unnecessarily complicated, bad to maintain, and probably redundant. “Meta-states” such as the “drink potion” state can be entered from many other states and they should return typically to the previous state. As they always cause trouble for finite-state machines, hierarchical state machines are designed to deal with them.

Meta-states are not the only problem of finite-state machines. Champandard also criticizes: "FSM don’t provide ways for reusing logic in different contexts, which leaves you with a choice of two evils: redundancy or complicatedness." (Champandard, The Gist of Hierarchical FSM, 2007)

3.1.2. Hierarchical State Machines

The next model shows how the extended guard sample would be modeled for a hierarchical state machine. Obviously, the redundancy is removed. The “patrol” and “engage enemy” states are now sub-states of “guard”. The “drink potion” returns to a history node within the “guard” state. This history node remembers from which sub-state the composite state has left and returns to it when the composite state is entered again.

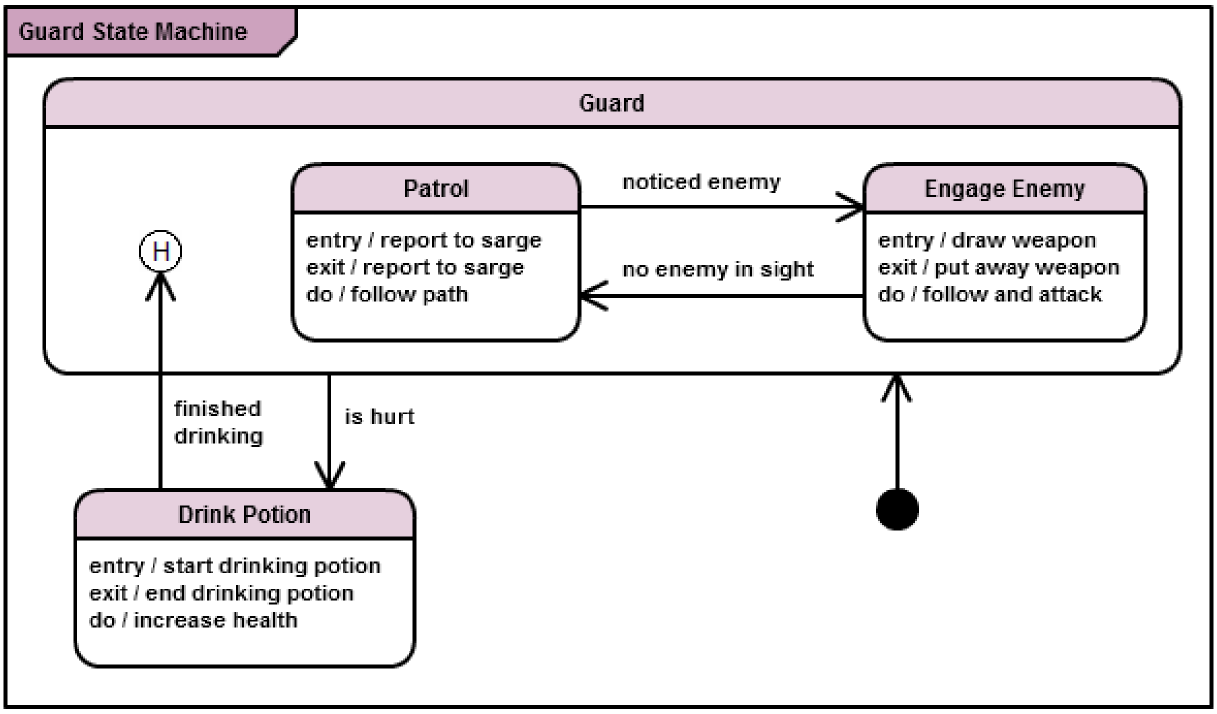


Figure 5: Extended guard sample as a hierarchical state machine

Each state can only have one active sub-state. The number of hierarchy layers is only limited by the complexity a human can handle. Hierarchical state machines are evaluated recursively. Millington and Funge (2009) describe how on pages 321 and 322: each state asks its current state to return its hierarchy. If a state is the lowest state in the hierarchy, it returns itself. If not, it returns itself plus the state from its own current sub-state. The update works similarly. Every current state in the hierarchy does a transition if the condition is fulfilled. If a transition is done, other transitions lower in the hierarchy are not considered anymore.

As shown in the example, the hierarchical state machines can express the same behavior of a finite-state machine even without redundancy. Their implementation and modeling are somewhat more complicated than finite-state machines.

3.2. Fuzzy Logic

Fuzzy logic is a representation of information that is less mathematical but more human. In contrast to most other decision-making techniques, fuzzy logic does not work with the crisp clear logical values “true” and “false”. Instead, it includes a blurry line between those two values and deals with it. Therefore, fuzzy logic lends itself to work with the perceptions of non-player characters.

Fuzzy logic is based on the predicate logic. These predicates can be something like “has ammo”. Everything that has ammo is in a so-called set. Fuzzy logic adds a value to the predicates. Here, the value shows how much ammo something has. This value is also called degree of membership of a set. A zero degree of membership means that the predicate owner is no part of the set in the classical logic and one means that it is a full member. However, values in between these extremes only have a meaning in the fuzzy logic (cf. Millington & Funge, 2009, p. 372). Although they can be handled like them, fuzzy values are not the same as probabilities, as said in (Millington & Funge, 2009) on page 372.

An object can be a member of multiple fuzzy sets. No set makes it impossible to become a member of another set. Therefore, an object could be a member of both “feeling cold” and “feeling hot” fuzzy sets. The process of translating a value to degrees of membership is called “fuzzyfication” in (Millington & Funge, 2009) on page 373.

One kind of information can be translated into several values. For example, one value of temperature can be converted into different values for feeling cold, warm, and hot.

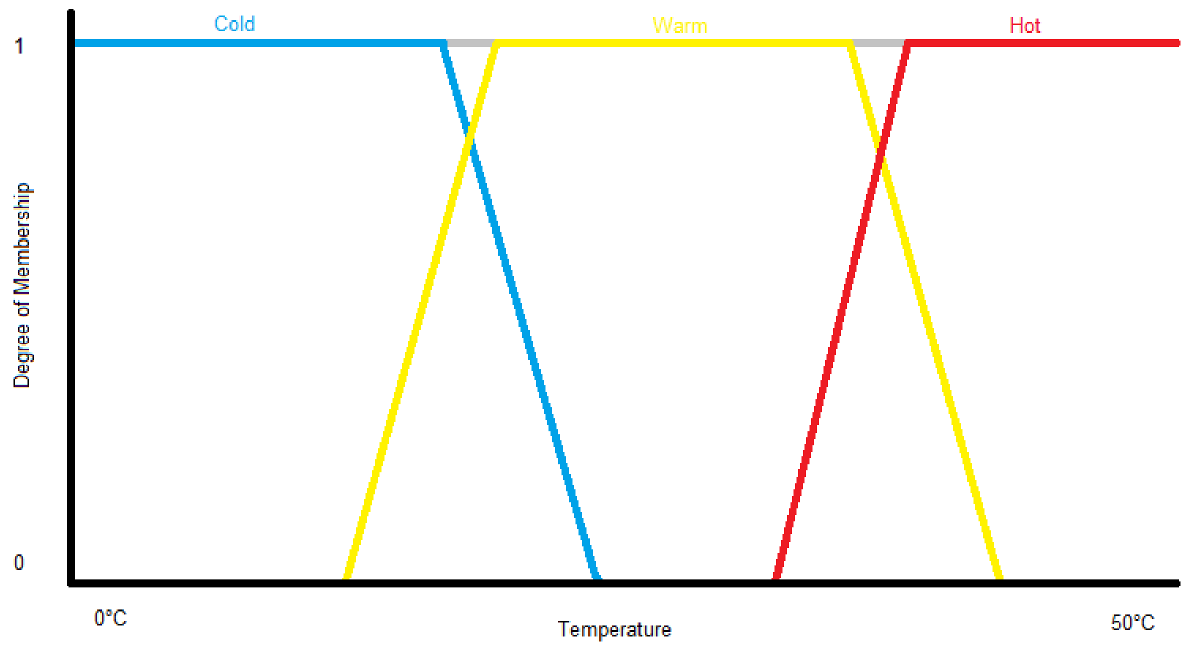


Figure 6: Fuzzy logic temperature example

Note how neighboring fuzzy sets overlap each other. In this example, the object cannot be a member of both the “cold” and “hot” fuzzy sets, but this is excluded by the fuzzyfication functions and not by the fuzzy logic in general. These functions can be as complicated as needed for the application. However, they have to be updated every time the value changes which can cost a lot of computation time.

A character may simply use these degrees of membership to make its decisions. For example, a character could go nearer to a campfire if he has at least a 0.8 degree of membership of the “cold” fuzzy set. With this approach, there will be no blending between “next to the campfire” and “normal”. To improve this, all related fuzzy sets have to be “defuzzyficated” into a single value. The “center of gravity” method from page 377 of (Millington & Funge, 2009) could be used for instance.

So-called fuzzy logic operators can be used to combine several values. Those fuzzy logic operators are similar to traditional logic operators. The following table is from page 380 of (Millington & Funge, 2009):

Expression	**Equivalent**	**Fuzzy Equation**
NOT A		$1 - m_A$
A AND B		$\min(m_A, m_B)$
A OR B		$\max(m_A, m_B)$
A XOR B	NOT(B) AND A	$\min(m_A, 1 - m_B)$
B XOR A	NOT(A) AND B	$\min(1 - m_A, m_B)$
A NOR B	NOT(A OR B)	$1 - \max(m_A, m_B)$
A NAND B	NOT(A AND B)	$1 - \min(m_A, m_B)$

Table 1: Fuzzy logic operators according to (Millington & Funge, 2009)

Fuzzy rules exist to calculate new membership values out of existing ones using fuzzy logic operators. In this example, the membership of a “is at campfire” fuzzy set is calculated. The character is at the campfire if he feels cold and is not guarding the camp:

$$m_{(\text{Is at campfire})} = \min(m_{(\text{feeling cold})} > 0.8, 1 - m_{(\text{is guarding})})$$

With all this fuzzyfication, defuzzyfication, and processes involved, the entire fuzzy logic technique is clearly more complicated than e.g. a state machine. This technique has another drawback: it must calculate all set memberships instead of just considering the likely important values.

3.3. Markov Systems

A fuzzy state machine can be in several states at the same time. They can be described better using the Markov processes. Here is a short introduction from (Millington & Funge, 2009), page 398: “In mathematics, a first-order Markov process is any probabilistic process where the future depends only on the present and not on the past. It is used to model changes in probability distribution over time.”

The state vector, also known as the distribution vector, contains a value for each state. These values can represent e.g. the number of enemies at a place, or more general the degrees of membership of each state. Instead of a state vector, a probability vector can be used. This probability vector contains the percentage of enemies at a place. Probability vectors sum up to one. Both vector types contain only non-negative numbers.

The transition matrix determines the probability for a transition from every state to every state, including the origin state. The state vector can be multiplied with the transition matrix to calculate the state vector of the next step. This multiplication can be repeated as often as needed. Waner (2004) offers an easy explanation of Markov systems with a complete example. However, this site contains only the mathematic basics, no real-world applications.

The degrees of membership of the states in a fuzzy state machine can be put into a state vector and then multiplied with a transition matrix. There can be different transition matrices for different situations. For example, each transition can apply a different transition matrix to the state vector. Now the transitions belong to the entire state machine, not only to single states. Given that, a character can switch its fuzzy states over time or in an instant using this Markov state machine.

Here is an example for using the Markov state machine consisting of just two states:

1. Walk carefully
2. Walk fast

The initial state vector is $(0.25 \quad 0.8)$ meaning that the character is a 25% member of the "Walk carefully" state. It is also an 80% member of the "Walk fast" state.

Two transitions are sufficient for this example:

1. "Under fire" with the transition matrix $\begin{pmatrix} 2 & 0.5 \\ 0 & 0 \end{pmatrix}$
2. "Got hurt" with the transition matrix $\begin{pmatrix} \frac{damage}{maxHealth} & \frac{maxHealth}{2*damage} \\ \frac{damage}{maxHealth} & \frac{maxHealth}{2*damage} \end{pmatrix}$

The character is patrolling and then suddenly, it is hit by an enemy sniper taking half of his health. Therefore, the "got hurt" transition is triggered, which causes the character's state vector to be multiplied with the transition matrix:

$$(0.25 \quad 0.8) * \begin{pmatrix} 0.5 & 1 \\ 0.5 & 1 \end{pmatrix} = (0.525 \quad 1.05)$$

As a degree of membership greater than 1 makes no sense, the resulting state vector is clamped to $(0.525 \quad 1)$. After he was hit, the character will now run for his life but will take cover more often. Sometimes, it can be necessary to add some numbers directly within a transition matrix, not just to multiply with it. Extending the transition matrix by one dimension, just like homogenous coordinates, can be one solution to achieve this.

The entire Markov state machine system can be quite complicated to understand and implement, especially as it requires at least parts of the fuzzy logic system. Depending on the size of the transition matrices, this approach might not be applicable with "single instruction multiple data" chips, which would cost even more computation time. It also scales quite badly with a growing number of states because every time a state is added, all existing transition matrices have to be updated. Markov state machines, like all other state machines, can hardly be reused for similar characters because states have to be exchanged and again all transition matrices must be updated. Markov state machines share some other drawbacks with the fuzzy logic it is based on. For example, they also deal with degrees of membership, not with probabilities.

Although Markov state machines are an interesting technique, they are hardly used in games. This is most likely because the developers do not know them or try to avoid the implementation effort. However, if a character is blending its behavior based on perceptions, Markov state machines might be something to consider.

3.4. Goal-Oriented Behavior

All other techniques described here do mostly react to events. Goal-oriented behavior is different in this regard. It allows characters to follow their own goals and desires. As said in (Millington & Funge, 2009) on page 402, goal-oriented behavior is not a special algorithm but a rather vague category of techniques. Many techniques can be used for characters to seem as if they had goals but the goal-oriented techniques really simulate these goals.

According to page 401 of (Millington & Funge, 2009), goal-oriented behavior is used in “The Sims”. In this game, there are usually less than ten characters active at the same time. Each of them has personal needs, called motives, such as hunger or hygiene. There is a value for every motive, which is called insistence. This insistence value determines how much the motive should influence the behavior. Each Sim tries to satisfy his or her needs with the available actions, for example cooking. Individual Sims can also be told to do a specific action. These actions are offered by the fitments of their house, which can be bought for the money the Sims earn.

A short example: A character has the motives “hunger” and “sleep”. “Hunger” has the highest insistence. Therefore, an action is to be found which satisfies the hunger best. The telephone offers the action “order pizza” what will lower the insistence of “hunger” by 0.5, and the boiling plate offers “cook spaghetti”, which will lower the insistence by 0.6. The character decides to cook himself because of the action’s effect. Note that this simple approach takes neither the delivery time of the pizza or the cooking time nor the side effects such as money into account. Utility systems are made for these somewhat more complicated decisions.

One implementation can allow goals to be fully satisfied. Another might just set the goal’s insistence to zero. The availability of actions may also vary. Some actions may be available anytime and others may depend on the state of the game. Actions can be “consumed” after doing them and have to be created again, or they can persist permanently to be done by the same character again or by another. Often a series of actions is required until the effect is applied. For instance, a character might need to cook, serve the meal, and eat in order to satisfy his hunger. Goal-oriented action planning can make such decisions.

Goal-oriented behavior in general has the disadvantage that all available actions have to be compared which diminishes the advantage of being able to consider a large number of options for a single decision.

3.4.1. Utility Systems

The word “utility” in this context is derived from the domain of economics theory. It is part of Utilitarianism based on the ideas of Jeremy Bentham and John Stuart Mill. It is a value for how good an action or a situation is for an entire society or just an individual.

Discontentment describes how much all goals of a character are fulfilled. It may be just the sum of all insistencies or it may sum their squares or even higher potencies, or it might be squared after summing up. Millington and Funge (2009) say on page 406, “From our experimentation, squaring the goal value is sufficient.” In the following example, it becomes clear that squaring the insistence of each goal and then summing up is meant.

Utility systems are used to calculate the utility of actions. The utility value is not restricted to the insistence values of the actions. Instead, it can include side effects and time issues. If several actions share the highest utility, the decision can be made randomly.

Side effects can be money costs, consumed energy of any kind, ammunition, or an effect on other goals, for example. Each side effect needs a conversion into a utility value. This conversion may be multiplying with a factor or even a special function.

Different kinds of times and durations may be considered in the utility functions. These times can be for instance: The time to carry out the particular action, the time for the whole chain of actions, and the time for the character to come to the action’s location. Some times can be approximated. For example, the walk time can be calculated out of a straight-line from the character to the action. The utility function may also consider how the goals change while the character walks to do the action. The overall time for the action can be calculated into the utility function for something like utility per second, or the utility can be decreased for each second the entire action chain takes. According to page 409 of (Millington & Funge, 2009), the overall time can also be incorporated into the discontentment, or a character can simply “[…] prefer actions that are short over those that are long, with all other things being equal.”

As said on page 408 of (Millington & Funge, 2009), the algorithm is “[…] $O(n * m)$ in time, where n is the number of goals, and m is the number of actions […].” Therefore, the utility calculation of each action should be kept as simple as possible.

Based on the experience with Cosmonautica, the exact utility values do not necessarily have a meaning. They are just used to compare actions in order to choose the best.

3.4.2. Goal-Oriented Action Planning

The previous parts of goal-oriented behavior aim for finding a single best action for a particular character. As the name says, goal-oriented action planning aims for planning actions meaning that this technique is made to find the best sequence of actions for a character.

The simple approach in finding a sequence of actions is to try all possible combinations of actions until the best is found. As said in (Millington & Funge, 2009) on page 413, such an algorithm would be $O(n * m^k)$ in time with k being the number of steps, also called the planning depth. “N” is still the number of goals and “m” is the number of actions. Such a bad performance characteristic is usually not acceptable. Therefore, three algorithms for speeding-up the process with heuristics are mentioned in (Millington & Funge, 2009): depth-first search (page 415), A* (page 418) and IDA* (also page 418).

The structure resulting from combining all actions is a rooted directed tree with game states as nodes and actions as edges. The current game state is the root node. A* algorithms can base the heuristic function on the action’s effect on the current motive to find a fast way through the tree. When a game state is found where the current need is satisfied, the algorithm has found most likely at least a good solution. If such a solution is found, possible remaining planning steps may be omitted. Instead of searching for a satisfied need, the algorithm can also search the tree for the lowest overall discontentment at the last node or the accumulated highest utility value.

As GOAP tries to simulate the game based on the current game state and the considered actions, it will need some sort of world representation, which includes at least the character and its environment with the actions. Depending on the game, this can require a lot of memory. As (Millington & Funge, 2009) states on page 413 and 414, it is not necessary to store the entire world’s information for each character. Instead, storing a list of differences to the current state is sufficient.

Although goal-oriented behavior is not too easy to understand and implement, it is used in several games. According to (Champandard, Special Report: Goal-Oriented Action Planning, 2008), it is used in the “No One Lives Forever”, “S.T.A.L.K.E.R.”, and “F.E.A.R.” series as a few examples.

On page 418 of (Millington & Funge, 2009), it is recommended to use depth-first for the search for the overall discontentment and A* for the search for the best plan for a particular motive.

3.4.3. Smelly Goal-Oriented Behavior

This quite simple approach was used in “The Sims” (cf. (Millington & Funge, 2009), p. 426). Here, motives are represented by smells. Available actions produce a smell for the motives they affect. These smells are simulated like fogs. They cannot go through walls but spread around corners. Therefore, a part of the path finding is transferred to the action provider making the path finding of the characters slightly faster. The intensity of the smell can reduce over its travel distance. Then, the characters seek the smell source with the highest intensity at their current position.

This approach leaves enough possibilities for value tweaking. For example, the emitted smell intensity can depend on the effect of the action. How fast the smell faints can be determined linear or quadratic with the distance. A threshold for smell perception might be necessary to prevent characters from walking through the whole level. The diffusion of the smell simulated quite easily on a field-based level. Each field will need an intensity value for each motive.

3.5. Rule-Based Systems

Rule-based systems are a well-known technique for artificial intelligences. According to page 427 of (Millington & Funge, 2009), "They have been used off and on in games for at least 15 years, despite having a reputation for being inefficient and difficult to implement." Rule-based systems allow the reasoning of characters about the game world in contrast to the other considered techniques.

A rule in this context is made of a condition, the "if" part (also called "pattern"), and an action, which represents the "then" part. If the condition is met, the rule is triggered. Only a triggered rule can be fired. If it is fired, its action is finally executed. As said on page 428 of (Millington & Funge, 2009), "Many rule-based systems also add a third component: an arbiter that gets to decide which triggered rule gets to fire."

Rules need a database for their condition checks. The data within the conditions is structured equally to the data in the database. Conditions can combine data from the database with Boolean operators. A single datum in the database contains an identifier and a content. The content can be a list of datum objects or a single value. Wild cards are needed for rules that check if any character has a specific property.

Millington and Funge (2009) suggest on page 441 and 442 the following arbitration strategies:

- **First applicable:** rules are in a fixed order. The triggered first rule in the list gets to fire.
- **Least recently used:** rules are stored in a data structure for example a linked list. The fired rule is removed from its position and added to the end of the list.
- **Most specific conditions:** the rule with the highest number of matched clauses gets to fire. This gives highly specific rules a better chance to be fired.
- **Dynamic priority arbitration:** the priority of each rule can be changed at run-time to fit e.g. the motives of a character. This is the most flexible but it also needs the most time for its calculation.

Every possible exception to the normal behavior has to be caught or may lead to a "brain failure" of the character. This can be done by adding more general rules. To find all possible exceptions can be very difficult and time-consuming, rule-based systems are therefore quite prone to errors.

3.5.1. Rete

Rete is an algorithm for matching rules against a database. It is the standard of the artificial intelligence industry. Faster algorithms are available but may be patented. According to page 446 of (Millington & Funge, 2009), "Most commercial expert systems are based on Rete, and some of the more complex rule-based systems we’ve seen in games use the Rete matching algorithm.”

Rete is not only the name of the algorithm but also the name of its data structure. This data structure is a directed acyclic graph. “Each node in the graph represents a single pattern in one or more rules. Each path though the graph represents the complete set of patterns for one rule. At each node we also store a complete list of all the facts in the database that match that pattern.” (Millington & Funge, 2009, p. 446)

There are three types of nodes in a Rete graph:

- **Pattern nodes:** these nodes represent individual clauses in a rule.
- **Join nodes:** these nodes work like a Boolean “and” operator. They combine the results of several pattern nodes. They can also work like “xor” or “or”.
- **Rule nodes:** these nodes can be fired.

Due to this structure, Rete shares patterns between rules and therefore “[…] doesn’t duplicate matching effort” (Millington & Funge, 2009, p. 447). As said on page 455 of (Millington & Funge, 2009), the high memory usage of Rete gives the performance advantage over the simple rule-based system. However, the rule sets can grow too large, even for Rete. Rule sets can be then divided into rule groups. Those rule groups can be turned on and off if needed.

3.5.2. Expert Systems

A variant of the rule-based systems, the expert systems, were extremely popular several years ago, and many of them are still in use. “Expert system” is the name of an application, which combines an algorithm, usually a rule-based system’s algorithm, with the knowledge of an expert. Therefore, expert systems are artificial intelligences performing the job of an expert. (cf. Millington & Funge, 2009, p. 457)

An expert system can keep track of every datum modified in the database by any rule. This information can be used for debugging, as an example.

3.6. Behavior Trees

Behavior trees have become increasingly popular within the last few years. Not only big game studios such as Bungie (Halo) use them but also smaller independent teams like Klei Entertainment (Don’t Starve). As said on page 334 of (Millington & Funge, 2009):

> *They are a synthesis of a number of techniques that have been around in AI for a while: Hierarchical State Machines, Scheduling, Planning, and Action Execution. Their strength comes from their ability to interleave these concerns in a way that is easy to understand and easy for non-programmers to create.*

These advantages can be improved even further by graphical tools. The behavior trees from the domain of artificial intelligence have nothing to do with the behavior trees from requirements engineering.

Although sharing the same structure with decision trees, behavior trees are different in many ways. “Behavior trees have a lot in common with Hierarchical State Machines [...]” (Millington & Funge, 2009, p. 334), probably more than with decision trees. Decision trees are directed acyclic graphs made of only two types of nodes: decisions and actions. Decisions represent a question such as “Is the enemy visible?” and their answers are represented by their connected edges. The node at the fitting answer is checked next. If it is an action node, the decision-making is finished and this action node is executed (cf. chapter 5.2).

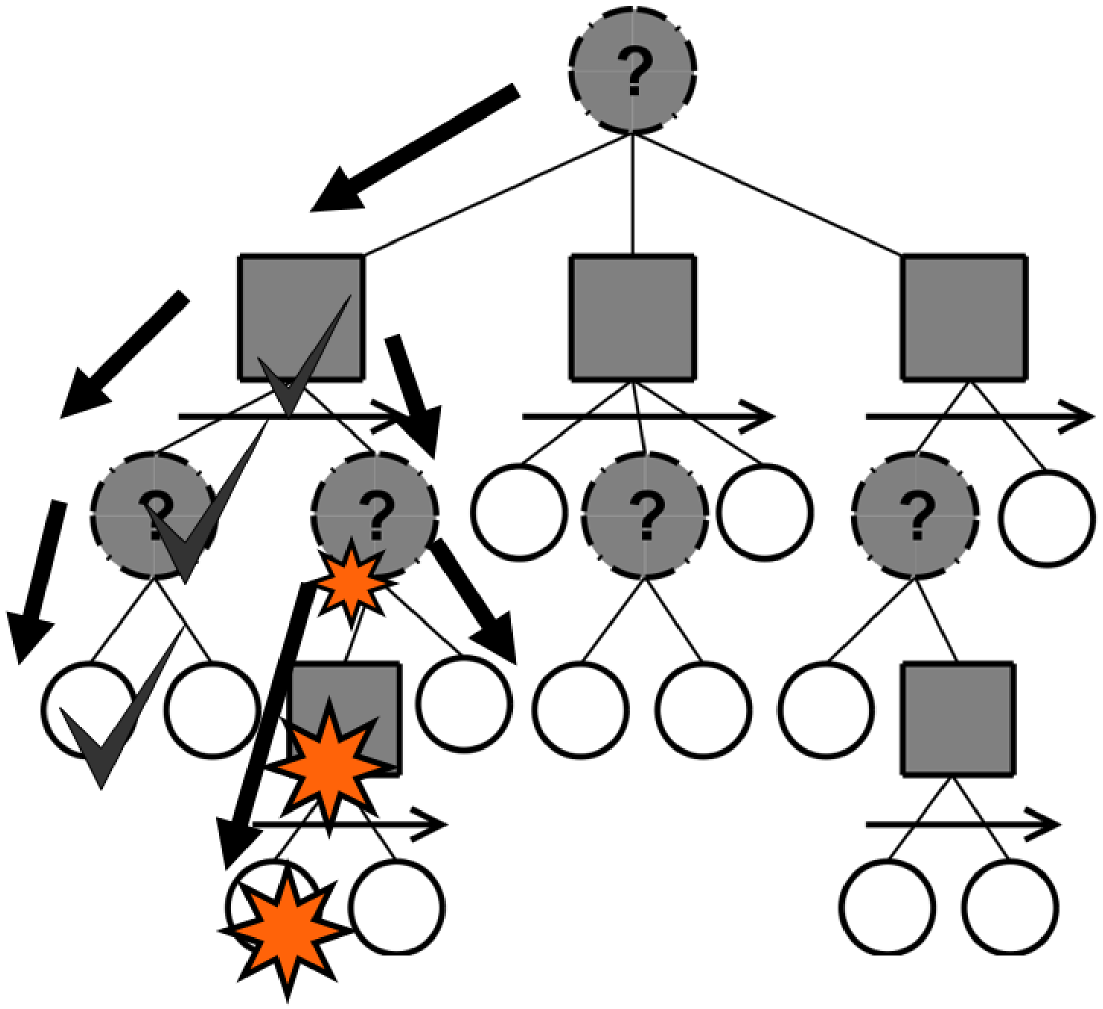


Figure 7: Behavior tree evaluation
Source: AiGameDev.com, Behavior Trees for Next-Gen AI (Slide 41), http://files.aigamedev.com/insiders/BehaviorTrees_Slides.ppt

Behavior trees are also directed acyclic graphs. As all graphs, they are made of nodes and edges. Nodes know only their child nodes. This leads in combination with being a rooted tree to the hierarchical structure of the behavior trees. All nodes share the same interface, making it easy to exchange nodes, which in turn is the reason for their flexibility. A behavior tree can be started very simple as a placeholder and easily extended later on with working states in between.

Behavior trees are evaluated from the root every time. In contrast to decision trees, they are not simply traversed from the top to one leaf node. Instead, each node tries to run through all of its child nodes. The child nodes themselves also try to run through all of their children. Nodes report their evaluation results back to their parent nodes. The parents themselves base their evaluation on the child's result. The behavior tree can be designed in such a way that the root node's result indicates if the behavior tree found a behavior for the situation, in which its controlled object currently is.

Millington and Funge (2009) suggest the following behavior tree node types:

- **Leaf nodes:** These nodes have no child nodes. They are at the end of the behavior tree. They can also be implemented in script code for easier modification.
    - **Action:** These nodes alter the state of the game or game object.
    - **Condition:** These nodes check a fact in the game.
- **Composite nodes:** These nodes have multiple child nodes. They base their own evaluation on the return values of their children.
    - **Sequence:** These nodes try to evaluate their child nodes until one returns something else than "successful".
    - **Selector:** These nodes try to evaluate their child nodes until one returns something else than "failure".
    - **Parallel:** These nodes try to evaluate their child nodes until a specific number of "successful" or "failure" is returned. In contrast to the other composite nodes, parallel nodes do not commence their evaluation at the last "running" node.
- **Decorators:** These nodes have only one child node. They change the way the child is evaluated. Many different types of decorators are imaginable. They can for example change the child's return value or act as breakpoint.

As said on page 340 of (Millington & Funge, 2009), "Behavior trees implement a very simple form of planning [...]. Selectors allow the character to try things, and fall back to other behaviors if they fail." Even this basic planning can make characters more believable. Behavior trees tend to do nothing if they fail instead of running into errors or carrying them into future evaluations. Some parts of behavior trees can be chosen randomly in order to improve the diversity of the behaviors.

3.7. Comparison

Now, that several decision-making techniques have been explained, they have to be compared in order to find the best for Cosmonautica. As a reminder, the technique has to support the behavior of characters like in "The Sims".

Here is a rule of thumb for how to think when using a technique:

- **State Machines:** What is the character doing now and what will he be doing next?
- **Fuzzy Logic / Markov Systems:** In which states and with which degree of membership is the character?

- **Goal-Oriented Behavior:** What is the best action for the situation in which the character is right now? What is the best chain of actions for the character to achieve a goal?
- **Rule-Based Systems:** In which situation should a character do what?
- **Behavior Trees:** What should the character do now?

Therefore, fuzzy logic is not fitting as several states at once are probably not required and the non-crisp logic might make the behavior unclear to the players. When for example the best action has to be found, all techniques except goal-oriented behavior and rule-based systems will grow too large to stay maintainable.

This is a collection of technique characteristics:

- **Simplicity:** how easy the technique is to understand and to implement. The creation of a behavior is included as well.
- **Separation of the work of game designers and programmers:** how good the game designers can focus on writing behaviors and how good the programmers can focus on the systems without interfering each other.
- **Flexibility:** how easy an existing behavior can be modified and extended. How easy it is to modify the behavior system in order to enable behaviors that are more detailed.
- **Behavior Quality:** how detailed a behavior can be. The effort making these behaviors is not included in this.
- **Efficiency:** how efficient a decision can be made by the technique. Must all possibilities be considered or just the most likely ones? Can an option be checked for validity and can it be modified at once?

The following table contains excerpts from the summary tables in (Vassos, 2013) as well as some possibly subjective anticipations and experiences. A '+' represents remarkably more, '-' less, and '0' the average of a characteristic.

	State Machines	**Fuzzy Logic / Markov Systems**	**Goal-Oriented Behavior**	**Rule-Based Systems**	**Behavior Trees**
Simplicity	+	-	0	-	+
Separation of Game Design and Programming	+	0	0	+	+
Flexibility	-	0	0	0	+
Behavior Quality	-	0	0	+	+
Efficiency	+	-	-	+	0
Note	Very simple	The only non-crisp logic here	Great for planning	Limits not reached yet	High flexibility

Table 2: Comparison of Decision-Making Techniques

However, which characteristics are important for Cosmonautica? As the speed of development is crucial at Chasing Carrots, simplicity is necessary for the technique. The separation of game design and programming is not too important right now but may become important in the future. This is because scripting is not yet included and the game designers are mostly working on the graphical user interface. Flexibility is very important because Cosmonautica is not fully planned now and will most likely change often. As the behavior of the crewmembers is a main asset of Cosmonautica, the chosen technique has to support a fairly advanced behavior. Therefore, the possible behavior quality of the technique must be over a certain lower limit but does not need to simulate a real human being. Cosmonautica will be available for mobile devices. Therefore, the hardware requirements, especially the computation time have to be kept in view. However, having the artificial intelligence as a main feature gives it a decent amount of computation time, which nevertheless must not be overused.

Putting it all together makes it clear, that behavior trees are basically the best technique for Cosmonautica. Only "basically" because they will cause trouble for example when choosing the best action to satisfy a need. These possible actions are added, modified, or removed while playing. Behavior trees could be changed at run-time to support this, causing them to grow very large. However, such behavior trees will be difficult to maintain and debug.

Finding the best possible option out of a pool of options can be done using the algorithm from the utility systems. As it considers all options, it is not efficient. However, the number of options is limited and the calculation of their utility values will stay simple. Therefore, the computation time will probably stay within its limits. These "choose the best option"-actions will be implemented as action nodes within the behavior trees.

4. Behavior Trees in Detail

As described in the previous chapter, behavior trees are a great technique for behavior control. Although simple, they are powerful and flexible. Behavior trees really shine when used with a graphical modeling tool.

Behavior trees are gaining more and more popularity in game development. Several sources are available, the best ones being chapter 5.4 of the book "Artificial Intelligence for Games" by Millington and Funge and several articles on AiGameDev.com by Alex J. Champandard.

This part is about how behavior trees work and what their precise parts are. Some of the elements described here may be insufficient or inapplicable for other programs. Luckily, the entire behavior tree technique can be extended and modified very simply. The number of possible node types and decorators is literally infinite. Several behavior tree implementations are available. To keep it simple, only the parts that are absolutely necessary or have proved useful in Cosmonautica are described in depth.

4.1. Tree Structure

Behavior trees can be described by the mathematical graph theory. A mathematical graph is an ordered pair of nodes, also known as vertices and edges. As stated in (Millington & Funge, 2009) on page 306, behavior trees are directed acyclic graphs (DAGs). It is directed because only the parent nodes call their child nodes' evaluation method. Acyclic means that it has no cycle but gains a cycle if a single edge is added to it.

To be more exact, behavior trees are rooted and directed trees. The root is a normal node but it adds a hierarchy to the tree. The hierarchy level of each node is described by the number of edges between it and the root. At an edge, the node that is higher in the hierarchy is the parent of the other node, the child node. Connections between nodes of the same level of hierarchy are not possible in the standard behavior trees. Nevertheless, any connection can be achieved by a special node implementation allowing them to manipulate a user-defined node.

4.2. Node States

Each node sets its own state every time it is evaluated, then it returns this state to its parent node. Which states are possible and useful? Alex J. Champandard says in his talk about behavior trees (Champandard, Understanding the Second Generation of Behavior Trees, 2012) that he tried many possibilities but these states are the best: Invalid, Success, Failure, Running, and Aborted. During the development of Cosmonautica, no situation occurred that required different states. In fact, the whole behavior tree system of Chasing Carrots is based heavily on the behavior tree starter kit from AiGameDev.com[5].

The node states are:

[5] http://aigamedev.com/ultimate/release/behavior-tree-starter-kit-source-release/

- **Invalid:** The state invalid is used for nodes, which have never been initialized. This state is set in the constructor of the base node class.
- **Successful:** Success is set when a node evaluated successfully. For example, a node that teleports the player character to another place could use this state to show that the player has arrived.
- **Failure:** Failure is the opposite. Using the previous example, this state could be set if the teleport fails because the player lacks the resources.
- **Running:** The running state is a bit special as it may persist through several evaluations of the node. In the teleport example, this state would show that the teleport takes time to open a portal. Once the portal is opened, the teleport node would set its state to “successful”.
- **Aborted:** Another node can cause the teleport node to reset. Then, the “aborted” state would be set. This might happen when the character received damage while opening the portal.

There are two events reacting on the states of a node. The on-initialize-event is triggered before the evaluation but only if the node's state is not "running". The on-terminate-event is triggered after the evaluation but before returning its state, only if it is not "running". Both events may execute custom code. In the teleport example, the on-initialize-event could start an animation, the node evaluation would check if the teleport preparation is finished, and the on-terminate-event would set the caster’s new position if successful.

4.3. Base Nodes

Base nodes are the nodes, which must be included in every behavior tree system. Special nodes needed for a particular game can be derived from these nodes.

Different implementations have different node types. One implementation is RAIN{Indie}[6], an artificial intelligence engine for Unity3D. It offers the following node types: Action, Condition, Decorator, Iterator, Parallel, Random, Selector, Sequence, Timer, and Yield[7].

Iterator, timer, and yield nodes do not need to be base nodes because they can be implemented as decorators.

The condition node evaluates its child node only if a pre-defined condition is met. It is quite convenient for the design of behavior trees but its implementation can be difficult in particular when the performance is really important. An overridden evaluation method with the build-in condition will do the job, too. It requires the inheritance of a base node, what will affect the performance much less than a scripted condition. The scopes of game designers and programmers will also overlap at this point.

[6] http://rivaltheory.com/rainindie/

[7] http://support.rivaltheory.com/rainindie/api/class_r_a_i_n_1_1_behavior_trees_1_1_b_t_node.html

Another version of the condition node evaluates its child node only if another child node returned successfully. This is how it works in some other behavior tree systems, for example the system of “Don’t starve”[8]. In this game, the behavior trees are used for the control of monsters and animals. Its entire artificial intelligence system is implemented in Lua. The condition check of such a condition node could be simply a part of the first child node instead of being a separate node. This will speed up the computation but will decrease the modularity. The version of Don’t Starve has also the advantage that the action will not run into “exceptions” and leave its controlled object in a broken state.

Another variant of the condition node is implemented in RAIN. This condition node is almost like an action node except it just reads the game data and instead of modifying it. Its condition is a script string that must be parsed. An approach with better performance is when conditions can by assembled from a pre-defined set of operators and some parts of the game data. This way, it can be used by scripting languages without slowing down the game like a fully scripted condition.

The behavior tree system of Don’t Starve has a special node, the priority node. It evaluates its child nodes in the order of their priority values. To achieve the same effect, the behavior tree designer can just switch the order of the child nodes since the child index is an implicit priority value. Changing the indices at run-time will cause at least swapping costs, but may also lead to inconsistency. In contrast, priority nodes have the advantage that their stored priorities can be changed easily at run-time without affecting anything else. Another special node is the event node. It evaluates its child nodes as soon as an event occurs. This event is not limited to the time the behavior tree is evaluated. That is why event nodes can be unpredictable.

Several behavior tree implementations contain random selectors and random sequences. Random decisions in general improve the diversity but may also make debugging more difficult, especially if the random decision is the base of more detailed decisions. Those random decisions can be used when a decision does not matter to the game logic but to the player. For example, a bot choosing a random taunt animation can be modeled with a random selector. The random sequence is used when all actions have to be done but their order does not matter. For example, a smoking NPC could choose randomly whether to get the cigarettes or the lighter first.

The behavior tree starter kit of AiGameDev.com is limited to the nodes that are common among all examined systems except for the active selector node. This active selector is a usual selector but terminates the previously running node if another node is running or successful in this evaluation step.

As used in the Chasing Carrots implementation, the absolutely necessary nodes are:

[8] http://www.dontstarvegame.com/

- **Action nodes:** the leaves of the behavior tree. They really manipulate or control the game objects. In this implementation, they also act as condition nodes, meaning that they can run checks on game data.
- **Sequence nodes:** evaluate their children until one returns something else than successful.
- **Selector nodes:** evaluate their children until one returns something else than failure, contrary to the sequence node.
- **Parallel nodes:** needed when several nodes are running at the same time. Depending on the implementation, the parallel node may evaluate one up to all of its child nodes. Its return value depends on the number of failed and successful children.
- **Decorator nodes:** intended to modify the way its single child node is evaluated. The details follow in the next chapter.

4.4. Decorators

Millington and Funge (2009) explain the origin of the decorators on page 345:

> *The name 'decorator' is taken from object-oriented software engineering. The decorator pattern refers to a class that wraps another class, modifying its behavior. If the decorator has the same interface as the class it wraps, then the rest of the software doesn't need to know if it is dealing with the original class or the decorator.*

Decorator nodes behave just like this, even in behavior trees.

Decorators are a kind of a behavior tree node. They work only with a single child node. They modify the way their child is evaluated. This can be the number of times the child is evaluated or how its return value is modified. New decorators can be developed just by inheriting the decorator base class and overriding its evaluation method as needed. The list of existing decorators is quite long and the list of possible decorators is practically infinite.

The behavior tree system of Chasing Carrots offers the following decorator nodes:

- **Repeat:** evaluates its child node a set number of times each time the decorator is evaluated. Note that this decorator might cause trouble if the child needs too much performance especially since the child node might be initialized and terminated several times. This number might be changed by another node of the behavior tree.
- **Limit runs:** every time this decorator is evaluated it decrements a counter. Then it evaluates its child if the counter is zero or greater. The counter must be reset by another node of the behavior tree.
- **Return value modifier:** changes the return value of its child node before giving back the changed value. There are several possibilities for the value changing itself.
- **Breakpoint:** breaks the execution of the code like a usual breakpoint for debugging.
- **Logger:** writes the return value of its child node each time it is evaluated into a log. Depending on the game, this log might be just a C++ console window or a

channel of a profiling tool such as Deja Insight[9]. This decorator proved to be almost worthless because usually the entire tree is to be viewed, not only single nodes. A separate recursive logging function is much better suited for this. See the chapter "Debugging" for the details. If every node has a logger as parent, the resulting log will be in wrong order. The first logging entry will be the last evaluated leaf node and the last entry will be the root node.

Another interesting decorator is described in (Millington & Funge, 2009) on page 348 und 349. It is shown how resources can be guarded by this special decorator. It works like a semaphore from the domain of multi-threading. For example, it can be used to ensure that only one animation at a time is running on a character. This decorator is currently not necessary for Cosmonautica because the animations are not controlled by behavior trees. Therefore, the semaphore is not implemented.

4.5. Policies

Policies offer variations of nodes without having to write a new class by deriving and overriding. This matters because every derived class adds performance costs through the run-time type information system. Policies are only used for quite common variations like how many failing child nodes will cause the parallel node to fail, too. They are an option for all variations that have only a limited number of meaningful possibilities. Policies are usually implemented as an enumeration of variations or a number of occurrences that are given to the node via its constructor. They do not need to be modified at run-time since the policies are part of the behavior tree design.

The Chasing Carrots implementation offers policies for several nodes and decorators:

- Parallel node: The state of a parallel node depends on the numbers of successful and failed child nodes. There can be different policies for the success and failure requirements of a single parallel node. The behavior tree starter kit has an enumeration for the policies require all and require one. Integral values are very similar in the implementation but offer more possibilities for the designers. By doing so, the behavior tree designer must take care of these values to assure that they work with the current number of child nodes of the parallel node. The work on Cosmonautica shows that the "all or one" enumeration is sufficient.
- Return value modifier: Always invalid, always running, always failing, always successful, and switch failure with successful. Modifying the aborted state seems useless.
- Breakpoint: Break always, break on invalid, break on running, break on successful, break on failed. The implementation must ensure that breakpoints are not included in the release version of the program.

4.6. Modeling

The notation for AI behavior trees should not be confused with the notation for requirement behavior trees. That notation is completely different and serves other

[9] http://www.dejatools.com/dejainsight

purposes. The deficient unified notation in the AI domain is most likely caused by the lacking interest in documentation of game developers in general.

Alex J. Champandard suggests in his talk about behavior trees (Champandard, Behavior Trees for Next-Gen AI, 2008) a notation for modeling which is inspired by an HTN notation. The symbol for sequences is a rectangle. There is an arrow through the edges between the sequence node and its child nodes. The symbol for selectors is a circle with a dashed borderline and a question mark in its middle. Decorators are shown as a rhombus. Leaf nodes are white circles, whereas actions and conditions have different fill colors if they are modeled explicitly. Champandard has a node for lookup dependencies, which is also a rhombus, but gray filled and contains an ‘L’. His parallel node is a rounded rectangle filled with gray and containing a ‘P’.

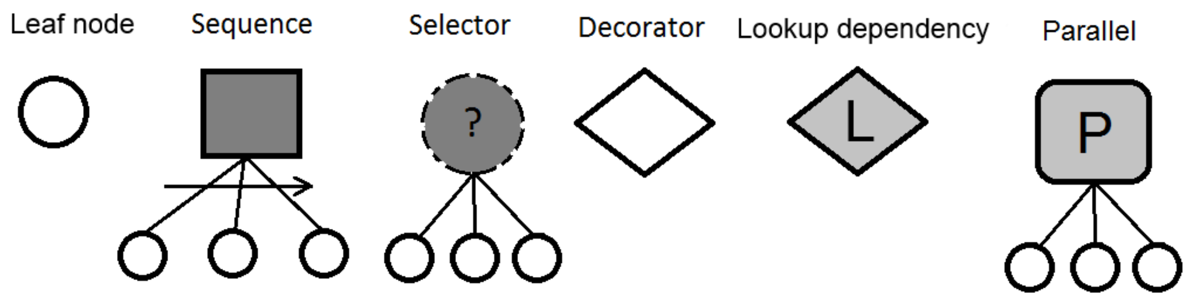


Figure 8: Behavior tree notation according to Champandard

In “Dynamic Difficulty Adjustment Using Behavior Trees” (Kenneth, Olsen, & Phan, 2011) another notation is used. It bases on the notation by Champandard (see page 11) but includes more node types. They use a triangle for the root, but their root is like a decorator without any action. It is just a marker for the beginning of the behavior tree. The action node is the same as the white circle in Champandard’s notation. Here, the selector is a rectangle having its smaller sides rounded. The probability selector has additional probabilities at the edges to its child nodes. The sequence node is not changed. It is still a rectangle. A pentagon indicates the condition node. Like the action node, it is meant to be a leaf node. In contrast to the action node, it has a condition instead of an action. The decorator node is still a rhombus. The link node is also a rhombus but with an ‘L’ in its middle. This node connects different behavior trees introducing more modularity and reusability (see page 13).

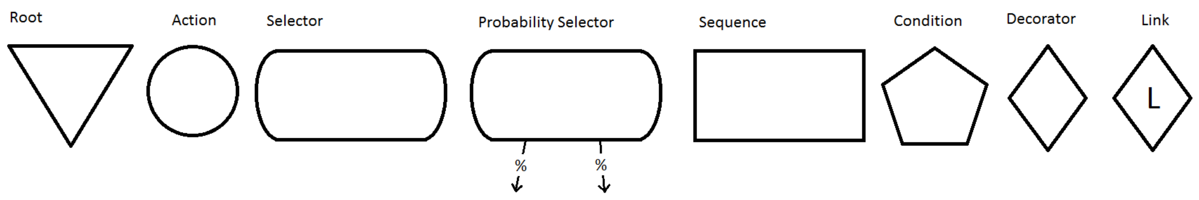


Figure 9: Behavior tree notation according to Sejrsgaard-Jacobsen, Olsen, and Phan

Some parts of this notation are not necessary. For example, the root does not have a purpose despite of being an optical marker. It should not be included into the final working behavior tree. The root might be necessary for their behavior tree system, but such implementation specific details should not be part of a model. The condition and action nodes do not need to be separate nodes, as shown in Cosmonautica.

The last notation mentioned here is the notation of (Millington & Funge, 2009). Due to the popularity of this book, it is probably the most used behavior tree notation. This notation is never explicitly explained, it is only shown in several models. If the models on the pages 336, 345, and 347 are put together, the following notation can be found:

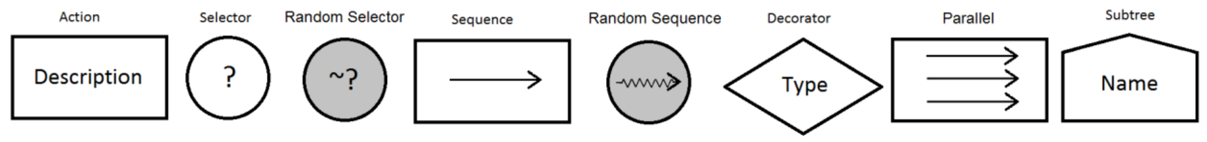


Figure 10: Behavior tree notation according to Millington and Funge

This notation is similar to the notation by Champandard. Strangely, the random sequence is a circle instead of a rectangle like the normal sequence. This is most likely a mistake as it is shown only in figure 5.28.

5. Artificial Intelligence in Cosmonautica

5.1. Workflow

5.1.1. Text-Based Notation

Every notation described in the previous chapter has the same problem: their symbols are exchangeable. This means that the rhombus of the decorator can be used for any other node type as well. This is quite basic for symbols as they stand for something else than they are. However, having a relation between the symbol's appearance and the thing it represents would help for remembering the notation. Furthermore, most symbols are not self-explanatory which means that a person who does not know the notation will not understand it. However, those who know such a notation can understand a model really quickly. It seems like the contact with people who do not know these notations is much more probable.

These are the main reasons why Chasing Carrots uses a completely different approach. Since symbols have no advantage, a text-based notation will be self-explanatory at least. This is the notation for one node:

Node type (policies): Description or identifier

Example:

Parallel (Require all for success, require one for failure): Update crewmember

If there are no policies for this node, they can be omitted including the brackets.

Example:

Action: Update morale

Information about initialization and termination of nodes can be added. This is how it can look like:

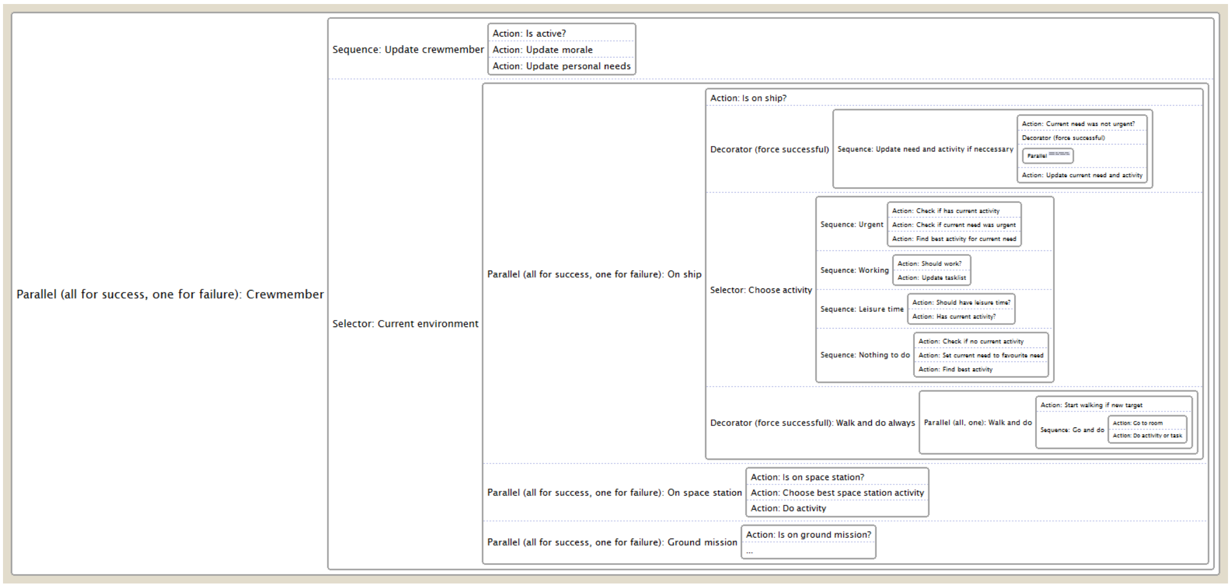
Node type (policies): Description or identifier {Initialize: init code; Terminate: term code}

The node type is the most important information about a node. When this is the first part in the notation, the type can be recognized almost as fast as a symbol. The policies are the next because they give a more detailed description of the node. The third part is intended to tell what the purpose of the node is. If the node's identifier gives enough information, it can be used as a description, too. The description part is optional because the function of nodes like sequences or selectors is clear. Nevertheless, a description at this point can clarify the purpose of the node and its children. For example a description as in "Sequence: Forge sword" is like a comment in source code, it clarifies the purpose of this part on a higher level of abstraction.

One of the major advantages of this approach is that it is not limited to the pre-defined node types. Anyone can add more of them, making it flexible enough to deal with all the possible nodes, especially decorators. This notation was created out of the need to talk about a behavior with people who were not involved in the behavior trees. As mentioned in the introduction, working code is more important than documentation at Chasing Carrots. So the modeling itself and its representation in graphs must not take too much time.

5.1.2. Modeling Tool

Another great advantage of the text-based approach over the symbol notations is that the modeling of behavior trees is much faster and easier. No special tool is needed. Instead, a simple text editor can be enough. “TreeSheets”[10] is used for the modeling at Chasing Carrots. It is a tool for any tree-like data organization. It is intended as a “replacement for spreadsheets, mind mappers, outliners, personal information managers, text editors and small databases” (Oortmerssen). The users can view their entered information in different styles such as a spreadsheet-like and a tree view. However, it is not intended for the modeling of behavior trees as it does not support the modelers with e.g. node templates, but it supports the necessary tree structure and is a fast writing tool.

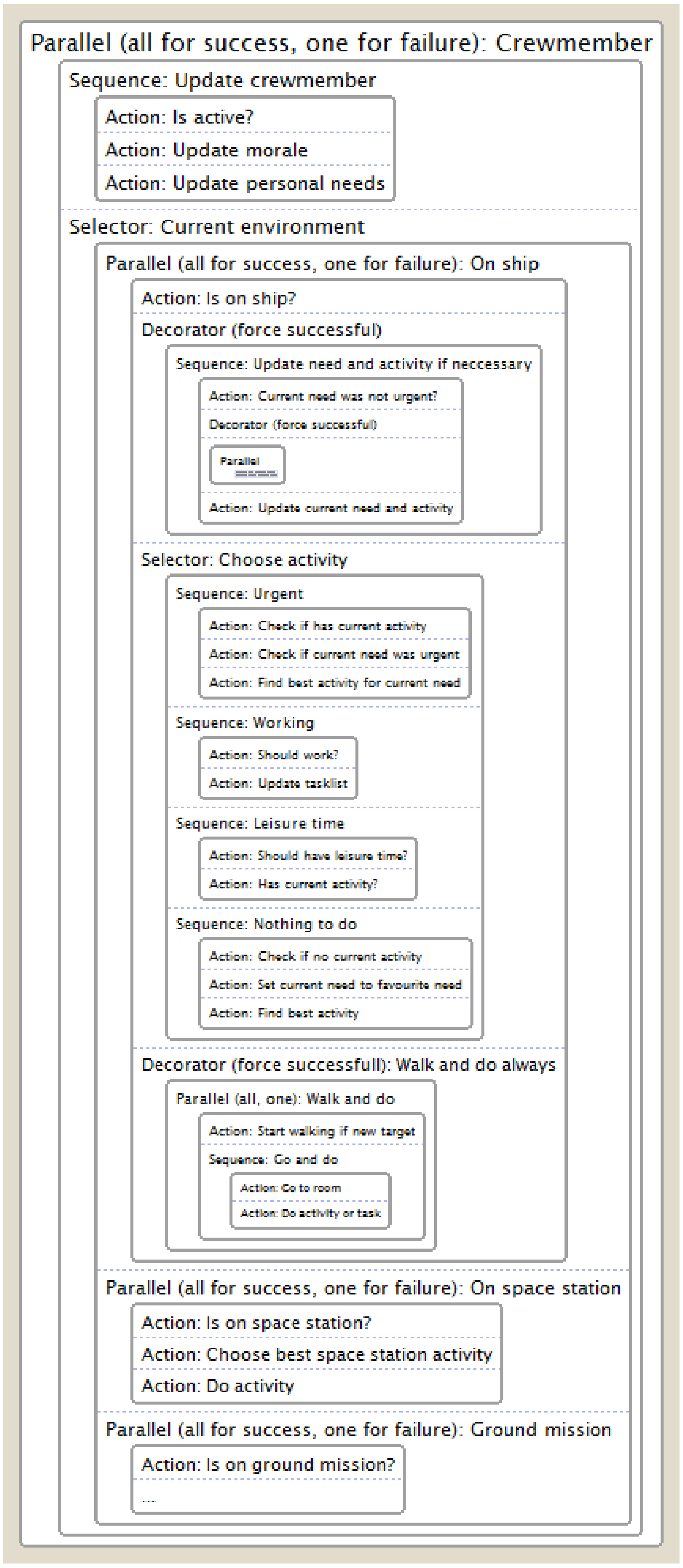


Parallel (all for success, one for failure): Crewmember
Sequence: Update crewmember
Action: Is active?
Action: Update morale
Action: Update personal needs
Selector: Current environment
Parallel (all for success, one for failure): On ship
Action: Is on ship?
Decorator (force successful)
Sequence: Update need and activity if neccessary
Action: Current need was not urgent?
Decorator (force successful)
Parallel
Action: Update current need and activity
Selector: Choose activity
Sequence: Urgent
Action: Check if has current activity
Action: Check if current need was urgent
Action: Find best activity for current need
Sequence: Working
Action: Should work?
Action: Update tasklist
Sequence: Leisure time
Action: Should have leisure time?
Action: Has current activity?
Sequence: Nothing to do
Action: Check if no current activity
Action: Set current need to favourite need
Action: Find best activity
Decorator (force successfull): Walk and do always
Parallel (all, one): Walk and do
Action: Start walking if new target
Sequence: Go and do
Action: Go to room
Action: Do activity or task
Parallel (all for success, one for failure): On space station
Action: Is on space station?
Action: Choose best space station activity
Action: Do activity
Parallel (all for success, one for failure): Ground mission
Action: Is on ground mission?
...

Figure 11: Tree view of a behavior tree

This tree view becomes overloaded very quickly, but it is more understandable for smaller models.

[10] http://strlen.com/treesheets/



- Parallel (all for success, one for failure): Crewmember
  - Sequence: Update crewmember
    - Action: Is active?
    - Action: Update morale
    - Action: Update personal needs
  - Selector: Current environment
    - Parallel (all for success, one for failure): On ship
      - Action: Is on ship?
      - Decorator (force successful)
        - Sequence: Update need and activity if neccessary
          - Action: Current need was not urgent?
          - Decorator (force successful)
          - Parallel
          - Action: Update current need and activity
      - Selector: Choose activity
        - Sequence: Urgent
          - Action: Check if has current activity
          - Action: Check if current need was urgent
          - Action: Find best activity for current need
        - Sequence: Working
          - Action: Should work?
          - Action: Update tasklist
        - Sequence: Leisure time
          - Action: Should have leisure time?
          - Action: Has current activity?
        - Sequence: Nothing to do
          - Action: Check if no current activity
          - Action: Set current need to favourite need
          - Action: Find best activity
      - Decorator (force successfull): Walk and do always
        - Parallel (all, one): Walk and do
          - Action: Start walking if new target
          - Sequence: Go and do
            - Action: Go to room
            - Action: Do activity or task
    - Parallel (all for success, one for failure): On space station
      - Action: Is on space station?
      - Action: Choose best space station activity
      - Action: Do activity
    - Parallel (all for success, one for failure): Ground mission
      - Action: Is on ground mission?
      - ...

Figure 12: Spreadsheet-like view of a behavior tree

This spreadsheet-like view is quite convenient for editing the behavior tree.

TreeSheets is still more like a workaround for behavior tree modeling. In the future, Chasing Carrots may develop its own graphical modeling tool, which might also generate code, most likely Lua code.

5.2. Behavior Tree System

5.2.1. Implementation

The structure of behavior trees can be represented with container classes. As the number of child nodes does not usually change at run-time, a vector has the best performance characteristics. Every node has a vector of pointers to its child nodes. Exceptions are decorators, which have only a single child node. Nodes do not need to know their parent nodes. This is how the directed graph is implemented.

Sometimes binary trees are used to increase the performance of the behavior trees. Those binary trees can only have up to two child nodes. This limitation makes it unnecessary to use container classes. This way, a lot of overhead e.g. from iterators and function calls is avoided. Often, more than two child nodes are required. Therefore, a binary tree will tend to have more nodes in total than a usual tree. The advantages and disadvantages of using binary trees have to be weighed carefully, especially as they cause more modeling effort.

Amongst other features, the behavior tree starter kit by AiGameDev.com contains an example of a memory-optimized variant. With this, only one instance of each single node class is needed for the entire program. Instead of assembling the behavior trees out of the nodes classes, so-called tasks are taken. These tasks are lightweight versions of the nodes. Each of them has a state for itself but may share other data e.g. a blackboard.

Behavior trees can be evaluated using multi-tasking. For example, sub-trees may be evaluated in separate threads what will require the previously mentioned semaphore decorators.

There are different ways how behavior tree nodes can get their data from the program. One approach is to use a blackboard architecture. Another one is to give pointers to game objects into the node constructors and let the special node implementations collect their needed data. This approach needs additional treatment in the memory-optimized version.

Behavior trees make heavy use of inheritance to enforce a common interface for all nodes. Therefore, all nodes are derived from an abstract base node class. The action node class is directly derived from this base class and has an overridden evaluation method. It is also possible for another action implementation to derive from the base node class and execute a callback method on a given object. Decorators are also derived from the base node class. They extend it by its child node, which is evaluated in the decorator evaluation method. Selector, sequence, and parallel nodes are derived from an abstract composite node, which in turn is derived from the base node. The composite node holds a list of the child nodes and does its management. The evaluation itself is implemented in the selector, sequence, and parallel node’s evaluation method.

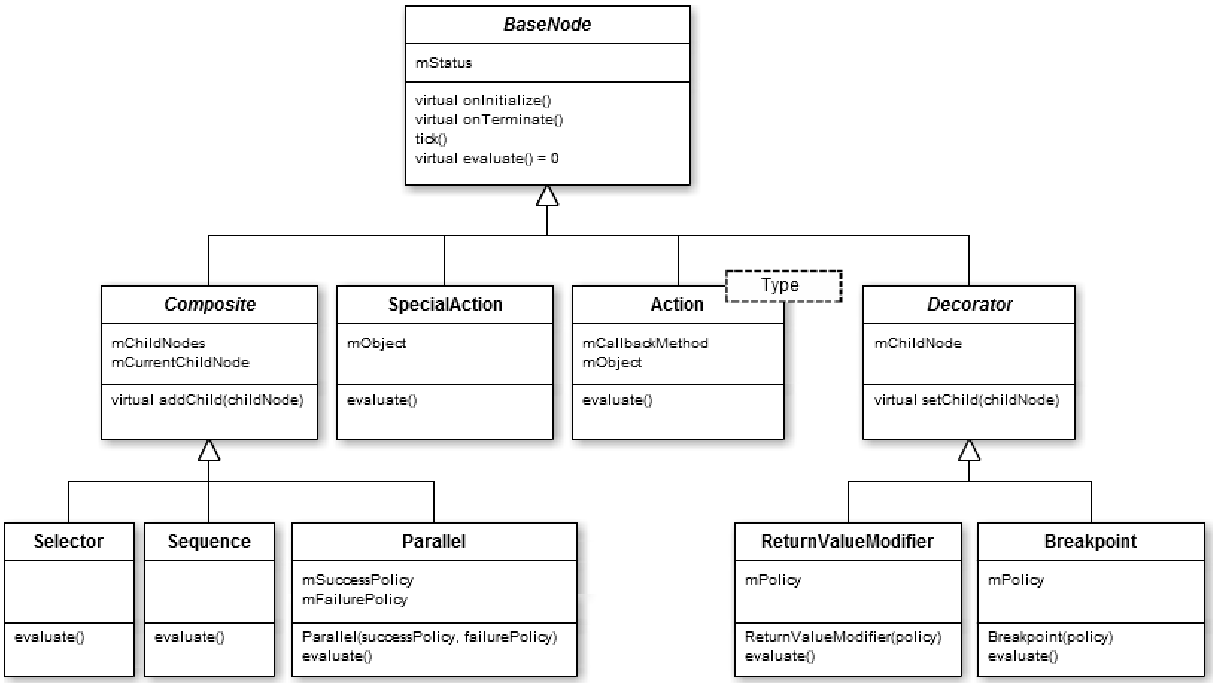
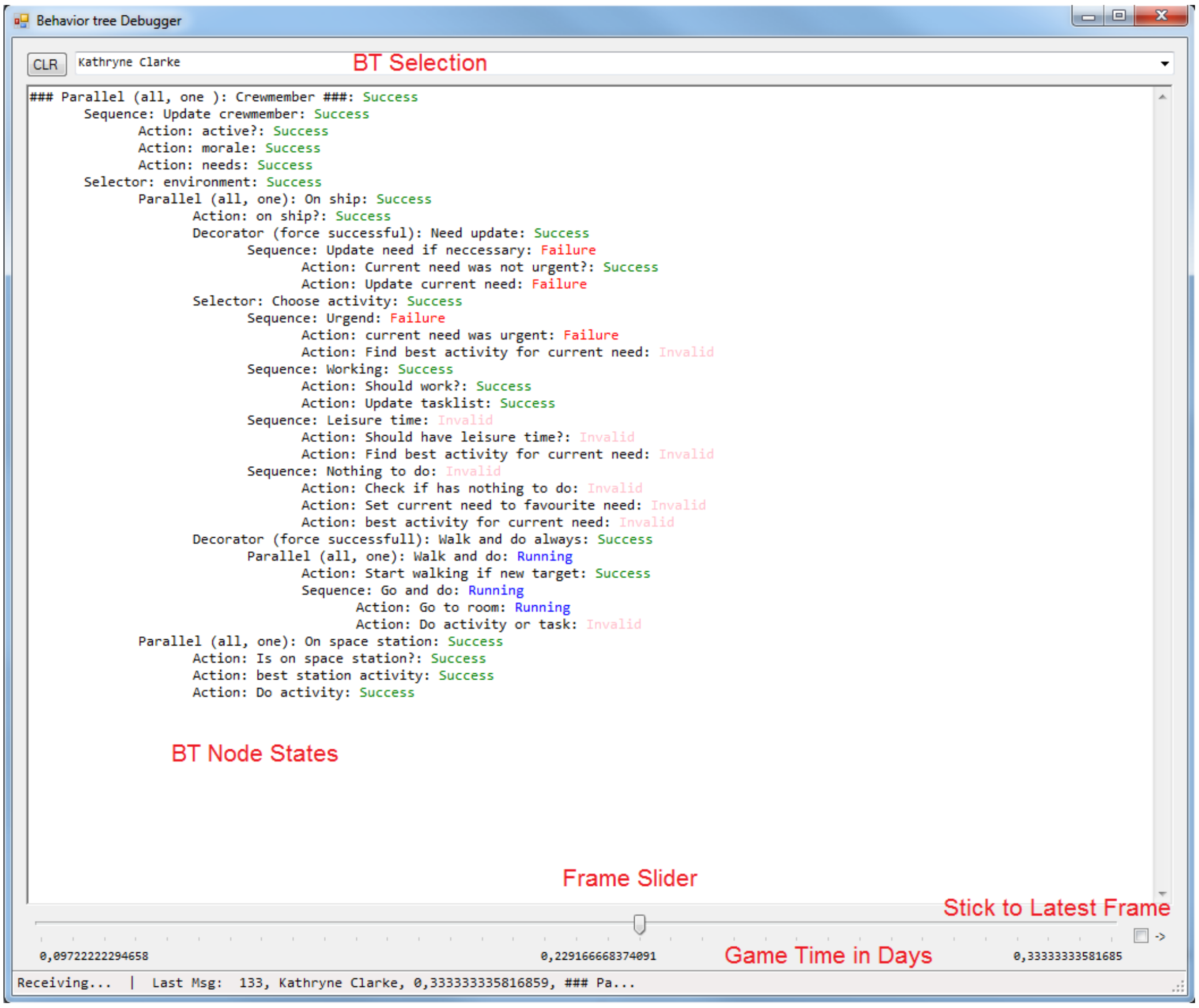


Figure 13: Class diagram of the behavior tree system at Chasing Carrots

5.2.2. Debugging

Behavior trees have a major disadvantage for debugging: their call stacks show only the parent nodes of the current node and are therefore quite useless. Debugging behavior trees will be much more effective if the entire tree with all its nodes' states can be seen in every frame. This is why Chasing Carrots has written a special debugging tool. It is fed with the data of the behavior tree nodes from the game via UDP. The data is sent after the entire behavior tree is evaluated. It stores for each behavior tree the states from the last three in-game days. However, this tool does not show whether a node has been accessed in the last evaluation or not. Older states may simply stay. Note that this tool is not a complete solution for all debugging problems. It is made to aid debugging, especially when wrong behaviors appear after a sequence of frames. Other games may require other information in such a tool.



```
### Parallel (all, one ): Crewmember ###: Success
	Sequence: Update crewmember: Success
		Action: active?: Success
		Action: morale: Success
		Action: needs: Success
	Selector: environment: Success
		Parallel (all, one): On ship: Success
			Action: on ship?: Success
			Decorator (force successful): Need update: Success
				Sequence: Update need if neccessary: Failure
					Action: Current need was not urgent?: Success
					Action: Update current need: Failure
			Selector: Choose activity: Success
				Sequence: Urgend: Failure
					Action: current need was urgent: Failure
					Action: Find best activity for current need: Invalid
				Sequence: Working: Success
					Action: Should work?: Success
					Action: Update tasklist: Success
				Sequence: Leisure time: Invalid
					Action: Should have leisure time?: Invalid
					Action: Find best activity for current need: Invalid
				Sequence: Nothing to do: Invalid
					Action: Check if has nothing to do: Invalid
					Action: Set current need to favourite need: Invalid
					Action: best activity for current need: Invalid
			Decorator (force successfull): Walk and do always: Success
				Parallel (all, one): Walk and do: Running
					Action: Start walking if new target: Success
					Sequence: Go and do: Running
						Action: Go to room: Running
						Action: Do activity or task: Invalid
		Parallel (all, one): On space station: Success
			Action: Is on space station?: Success
			Action: best station activity: Success
			Action: Do activity: Success
```

```
0,09722222294658          0,229166668374091          0,33333333581685
Receiving...  |  Last Msg:  133, Kathryne Clarke, 0,333333335816859, ### Pa...
```

Figure 14: Screenshot of the debugging tool for behavior trees

The state string of the whole behavior tree is assembled recursively. The base node returns only its own state string. Each node needs a string for its description, e.g. "Action: Go to room". The node state has to be converted into a string and added to the full string. The hierarchy can be shown through indentation. For this purpose, a tab-character is added for each level of hierarchy.

```
string BaseNode::getStateString(int hierarchy)
{
	return string(hierarchy,'\t') + mNodeName + ": " +
			mStatus.toString() + '\n';
}
```

Composite nodes add the state string of their nodes to their own state string. They call the getStateString()-method with their own level of hierarchy plus one. Decorators do the same, but only with their single child.

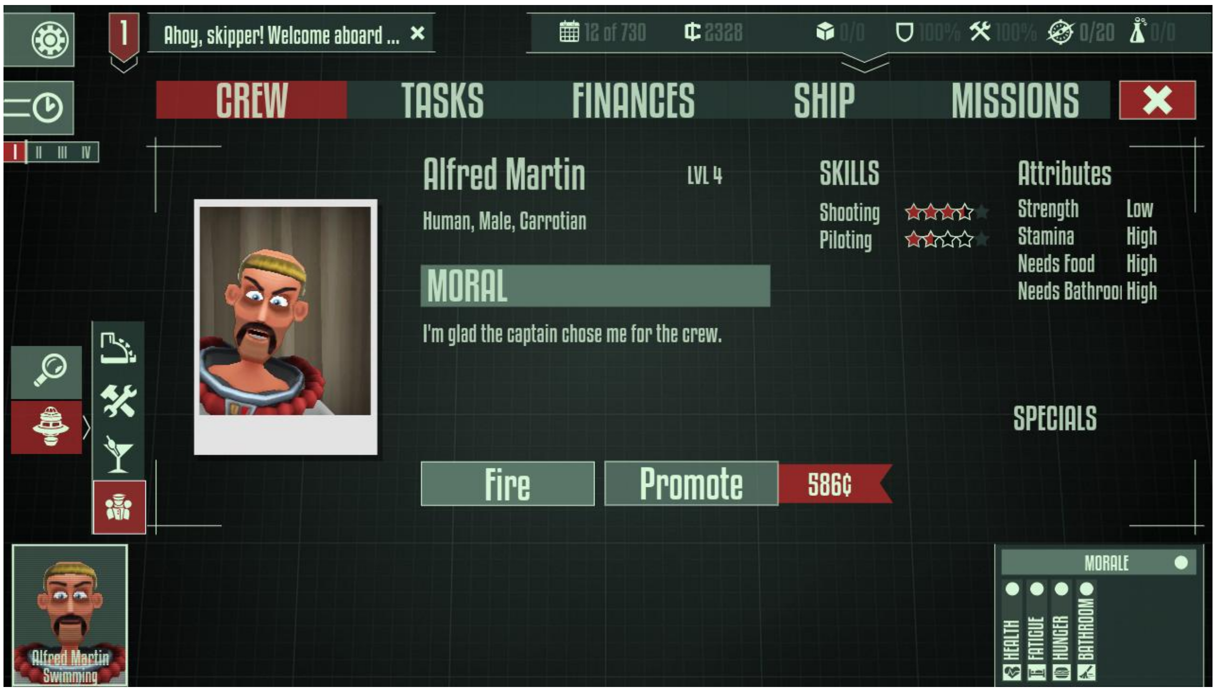
After the full state string of the behavior tree has been assembled, it can be sent via UDP. It can be important, that this string may grow too large for a single UDP-packet.

5.3. Crewmember Behavior

The crewmembers are a main part of Cosmonautica. The players will watch them very often. Therefore, their behavior has to be fine grained enough to be convincing and entertaining. As a main part, the crewmembers in Cosmonautica are placed within a network of many other classes. Some of these classes already existed when the behaviors where added. Therefore, some behaviors had to be made fitting into the systems. Here is a simplified overview of the important classes for the behaviors:

- **Player:** owns the crewmembers and the space ship. This class has the information in which situation the crew currently is. Such situations are for example on space station, or travelling.
- **SpaceShip:** contains information about the overall state of the ship, e.g. cargo, shields whether it is in an orbit or travelling to a planet. It owns the TaskProviders for flight and fight.
- **RoomStructure:** contains information about where which room is and where the elevators are. Rooms are placed within a two-dimensional grid. This grid has its X-coordinate going to the right and Y down just like the pixel-coordinates on the screen. Those coordinates are equal to the indices of a two-dimensional array of room-slots.
- **Room:** has a corridor included. Crewmembers can therefore walk horizontally from the front of the ship to its rear. If they want to go vertically, they need lift-tubes. Rooms offer activities to the crewmembers and they offer tasks to be done by crewmembers. Each room has only few activities and tasks. The cockpit for example offers the piloting-task. Like every other room, it also has the repair and clean tasks as well as bad sleeping and toilet activities.
- **Activity:** modifies the change of the needs of the crewmembers while they do it. Activities may satisfy one need quite fast but may also increase other needs. Activities can be something like eating a snack, which satisfies the hunger a bit but increases the need for toilet. Activities are always available at a room as long as the room is operable. They are not consumed by doing them but they can be done by only a single crewmember at once. If more than one crewmember tries to do the same activity at the same time, the first starts doing it and all following crewmembers wait in a queue near the room.
- **Task:** When crewmembers are working on tasks, they generate something useful for the player. This can be for example medic points enabling the “get healing” activity for hurt crewmembers. They have usually only one assigned activity. This activity makes the crewmember for example hungrier while repairing.
- **NeedSystem:** belongs to a crewmember. It contains the information about which needs the crewmember has. Not everyone has every need. Instead, crewmembers have a chance to have some secondary needs such as entertainment or fitness. All crewmembers have needs like the needs for food, toilet, and sleep. The need system also keeps track of the current need and activity and provides information for the visual representation of the crewmembers in order to make them walk to the activities and start the doing animations.

- **Need:** has the information about how urgent it is. The need is equal to the goals from the goal-oriented behavior and its urgency value is similar to the insistence of these goals. It also defines, what happens if the need has a specific urgency level. For example, a strong need for food will lower the health of a crewmember. Each need can have a standard change over time. The hunger, for instance, increases on its own and the health will slowly regenerate if the crewmember is not too hurt. Needs contain two different urgency values. The first one is a volatile value indicating the current state of a need. If this value grows too big, the behavior system tries to satisfy the need. The second value is damped and based on the first value. It represents the state of the need over the last few game-days. This second value triggers the urgency effects and influences the morale of the crewmember.
- **TaskProvider:** is a list of all possible tasks in the current situation. The current situation can be for example while travelling, on space station, or in space fight.
- **ActivityProvider:** is a list of all possible activities in the current situation.
- **Skills:** determine which tasks a crewmember can do. Each task has a required skill. Some skills like “gunning” are required for several different tasks. Although it is currently not used, tasks may require a minimum skill level.



Alfred Martin LVL 4
Human, Male, Carrotian

MORAL
I'm glad the captain chose me for the crew.

SKILLS
Shooting ★★★★
Piloting ★★

Attributes	
Strength	Low
Stamina	High
Needs Food	High
Needs Bathroo	High

SPECIALS

Figure 15: Information about a crewmember in the GUI

The Crewmember class is the center of all the effort. It contains the information in which shift the crewmember should work. It also has a list of assigned tasks for every situation, like travelling or in a fight. This list can be edited by the players to modify the priority of these task types for a single crewmember or tasks can be removed from the task list to have them done by another crewmember. Every instance of the crewmember-class has its own need system. If a crewmember starts doing a task, he sets the associated activity on his need system. Crewmembers have a morale value, which they update by considering the need values in their need system. When the morale becomes too low, the crewmember will be on strike and then refuse to work. Crewmembers improve their skill level while working on a task, which requires this particular skill. They also have a reference to the current task and activity provider.

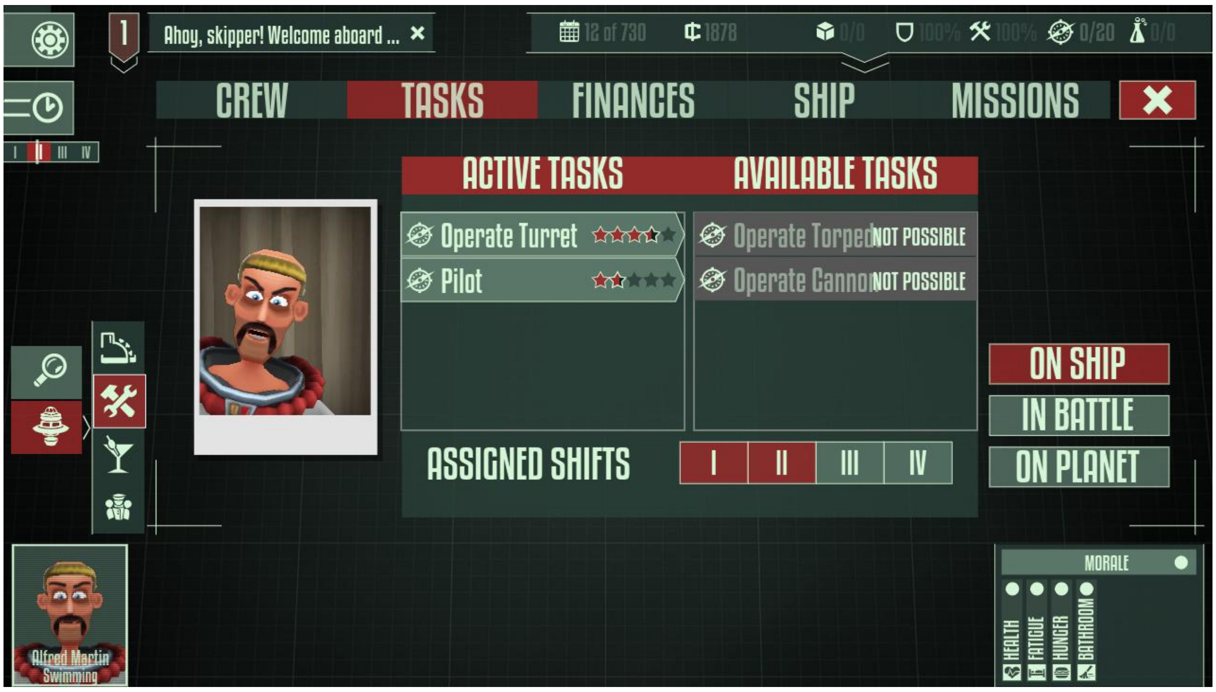


Figure 16: Task lists of a crewmember in the GUI

The Crewmember-class owns its behavior tree nodes. It manages the entire behavior tree, which means that the construction, evaluation, and de-construction are done by the crewmember. The leaf nodes of the behavior tree need access to data and methods of the crewmember or his need system. Therefore, methods for checking and modifying are added to the crewmember and the need system. These methods perform a check on a member datum or modify it, and then return a Boolean representing the success of the action. Such methods can be relatively simple like the check if the crewmember should work now. They can also be quite complex like the method, that searches for the best activity for the current need of the crewmember. If that method finds an activity, it sets the current one to this activity and returns true.

5.3.1. Behavior Tree for Crewmembers

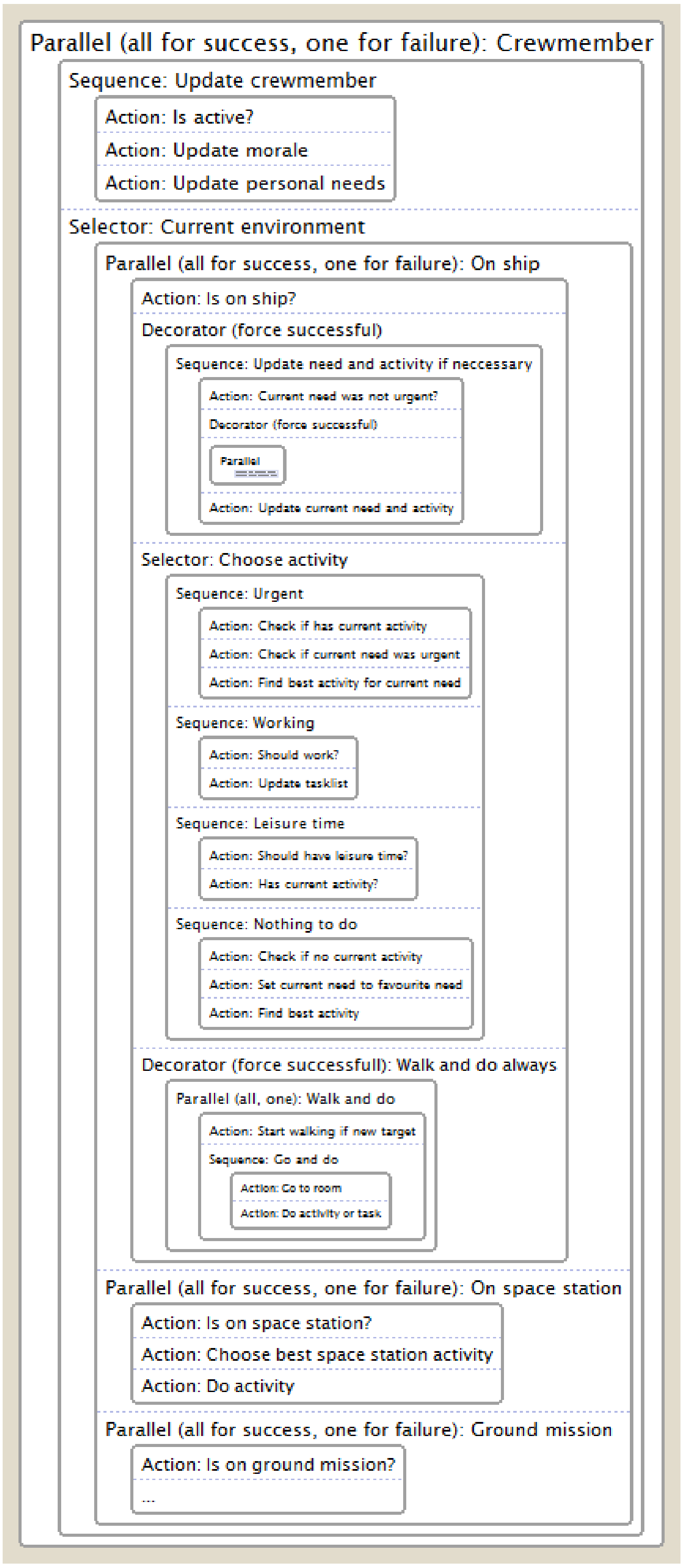

Parallel (all for success, one for failure): Crewmember
Sequence: Update crewmember
Action: Is active?
Action: Update morale
Action: Update personal needs
Selector: Current environment
Parallel (all for success, one for failure): On ship
Action: Is on ship?
Decorator (force successful)
Sequence: Update need and activity if neccessary
Action: Current need was not urgent?
Decorator (force successful)
Parallel
Action: Update current need and activity
Selector: Choose activity
Sequence: Urgent
Action: Check if has current activity
Action: Check if current need was urgent
Action: Find best activity for current need
Sequence: Working
Action: Should work?
Action: Update tasklist
Sequence: Leisure time
Action: Should have leisure time?
Action: Has current activity?
Sequence: Nothing to do
Action: Check if no current activity
Action: Set current need to favourite need
Action: Find best activity
Decorator (force successfull): Walk and do always
Parallel (all, one): Walk and do
Action: Start walking if new target
Sequence: Go and do
Action: Go to room
Action: Do activity or task
Parallel (all for success, one for failure): On space station
Action: Is on space station?
Action: Choose best space station activity
Action: Do activity
Parallel (all for success, one for failure): Ground mission
Action: Is on ground mission?
...

Figure 17: Behavior Tree Model for Crewmembers

Here is how the behavior tree of the crewmembers works in detail. The entire evaluation of the behavior tree is successful if both the updating of the crewmember and the finding of the current activity are successful. A parallel node with the policies “all for success” and “one for failure” is used to get a behavior like a sequence that begins its evaluation always with its first child node. Updating the crewmember means that his or her morale is updated as well as all the needs in the need system, but only if the crewmember is active now. Crewmembers can be inactive when they die or leave the crew for another reason such as low morale. The crewmember-objects are not destroyed while playing. Instead, their data is overridden if they are exchanged or reset to inactive if they leave the crew.

If the update part is successful, the behavior tree lets the crewmember choose the current activity for the current environment. As the crewmembers will be on the ship most of the time, this environment is checked first in the behavior tree. To ensure that this order of the environments is used every time the behavior tree is evaluated, a parallel node with the policies “one for success” and “all for failure” is used. It works mostly like a selector that begins its evaluation always with its first child node.

The part of the behavior tree for the ship environment checks at first if the crewmember is on the ship right now. If not, the next environment is checked. If he or she is on the ship, the current need and activity are updated if necessary. They are updated at once for performance reasons because they share most of the condition checks. It is only necessary if the current need has never been urgent since the crewmember was assigned that need as his current need. This is important because crewmembers should completely satisfy an urgent need. Without this check, the crewmember would stop satisfying the previous urgent need as soon as it is not urgent anymore. If this check for urgency is successful, the current need and activity are updated. This is the process of finding the most important need, that can be satisfied and finding the best activity for it. Afterwards, the current task is reset if existing. It has to be reset here und set again later in the working-part of the behavior tree or this reset-node has to appear in multiple places within the tree. Finding out whether a crewmember should not work anymore is much more effort than always resetting and setting only when necessary. This “reset task if existing“ part is the child of a force successful-decorator because more update nodes can be added to the on ship updates. These new nodes should be independent from the success of the task reset.

The same reason is there for the force successful decorator as parent of the entire on ship update part. Choosing the current activity and doing it should be done always, no matter if a need was urgent or not. If the current need was urgent, it must not change until it is satisfied but the activity might be changed. This is important because a crewmember might be very tired and therefore sleeping on the ground in a room. As soon as a bed is not occupied anymore, he should move to this very bed and sleep more efficient.

If the current need was not urgent, it is checked if the crewmember should work. This depends on his assigned shifts, the alarm state of the crew and other modifiers such as being on strike. If he should work, his list of tasks is updated as well as the current task and the assigned activity of this particular task is set as the current activity of the need system. If the crewmember should not work, he might have leisure time. Crewmembers on strike or working without anything to do are treated as if they had leisure time. Afterwards it is only checked if the crewmember has a current activity. He might not have an activity if none of his needs is beyond a certain threshold or if there is no useful activity available. In such cases, the crewmember would do anything, not even idling. Therefore, the "nothing to do" part is added. If the crewmember has no current activity, this part sets the current need to the favorite need and finds the best activity for it.

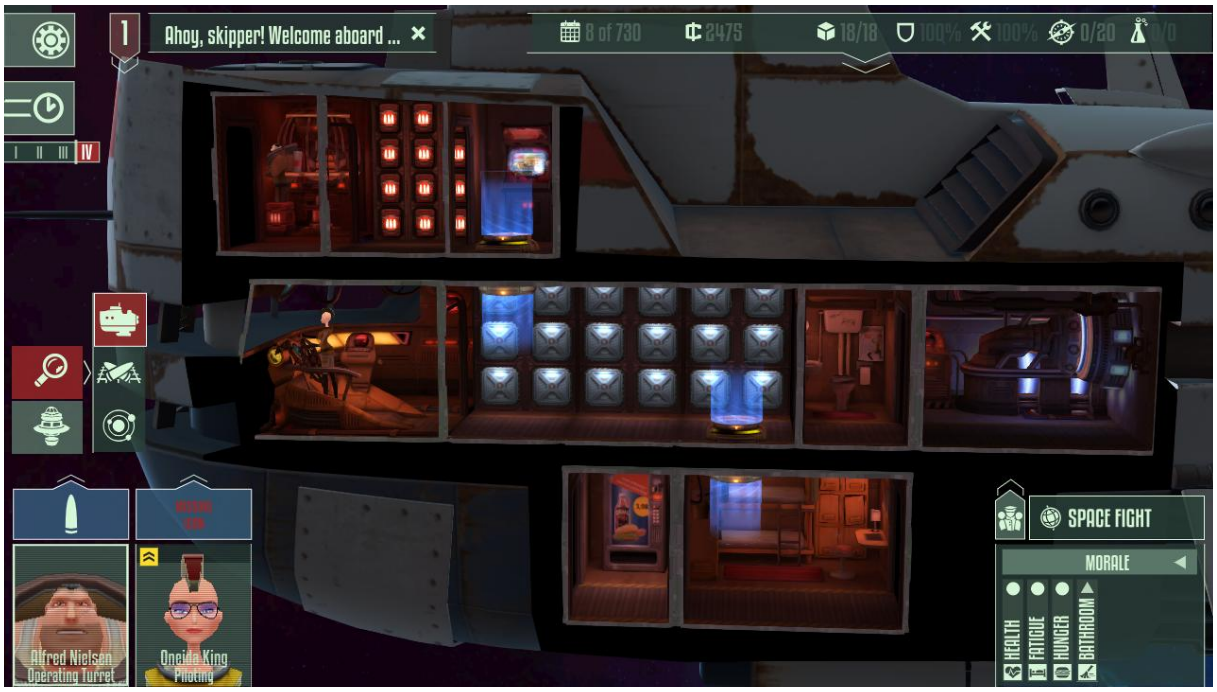


Figure 18: Alarm on a space ship

After the crewmember has been updated and found a current activity to do, he starts walking to that activity in case it has changed. The "go to room"-action just checks if the crewmember has arrived at the activity. It returns "running" while he is walking or standing in a queue. The walk speed depends on the urgency of the current need if it is not an activity of a task. The speed is also influenced by the crew's alarm state and the weight the crewmember is carrying. After having reached the activity, the crewmember does his current activity or task. This means that the current activity modifies the current need values and the task generates e.g. science points. The "start walking if new activity" node returns only successful if the activity has changed and the crewmember started walking. If this return value is inverted, the "walk and do" parallel node fails what resets the "go and do" sequence. This is necessary to ensure that crewmembers walk always to the new activity before they do it.

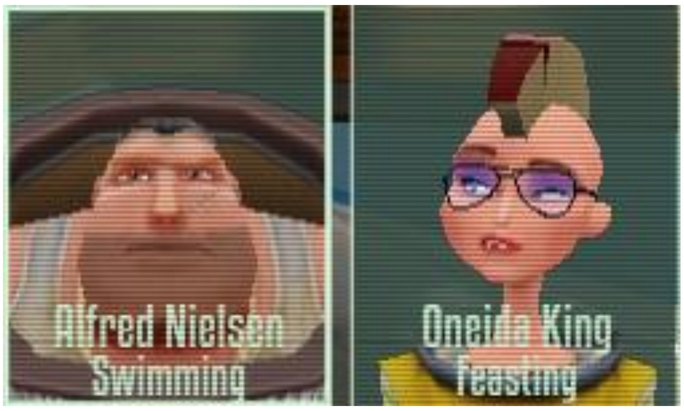


Figure 19: All the information players get about their crewmembers on space stations

Crewmembers who are not on the ship can only be on a space station, because ground missions are not yet implemented and any other environments are not planned. They are not visible on space stations, only their current activity's name appears in the GUI. Therefore, the behavior for this environment can be much simpler than on the ship. The space station part of the behavior tree just checks if the crewmember is on a space station, chooses an activity, and then does this activity. Crewmembers have automatically leisure time on the space station as soon as the ship docks. Space stations have superior activities for all needs and cause no costs for the player.

It should be noted that there is no distinction between travelling and fighting within the behavior tree. Both situations are handled in the "on ship" part. If the situation is changed, the crewmembers get a different task provider and a different activity provider assigned by the player-object. The search for the best activity and task can consider only the activities and tasks of the current providers. The task provider for travelling with the ship contains for example "cleaning", whereas the task provider for space fights contains different gunner tasks for each turret, instead.

5.3.2. Finding the Best Activity for Crewmembers

How the search for the best activity fits into the behavior should have become clear by now. The search itself has not been explained, yet. This is what follows here.

There are several reasons why the concrete search for the best activity is not done in the behavior tree but within a BT action node. The first reason is that activities are added by placing rooms and they are removed when the providing room is sold or destroyed. This would require adding nodes to the tree at run-time. If all of the approximately hundred rooms of a very big ship are placed, the behavior tree would then grow by at least each two hundred tasks and activities times the number of possible management nodes. This would be simply too much to stay within the budget for the computation time. A second reason why the activity search is not modeled in the behavior tree is that not all possible activities are known now and the activities are planned to be modifiable by the players.

Four more or less different searches are required for the crewmembers. One search tries to find the current need and the best activity for it. The second search tries to find only the best activity for a already known current need. The third search tries to find the best task for a crewmember. The last and fourth search is the search for an activity on a space station.

In order to find the current need, all needs of a need system are considered beginning with the most urgent. A need can only be the current need if an activity exists that can satisfy this need. This requires checks for the occupation and availability of the activity.

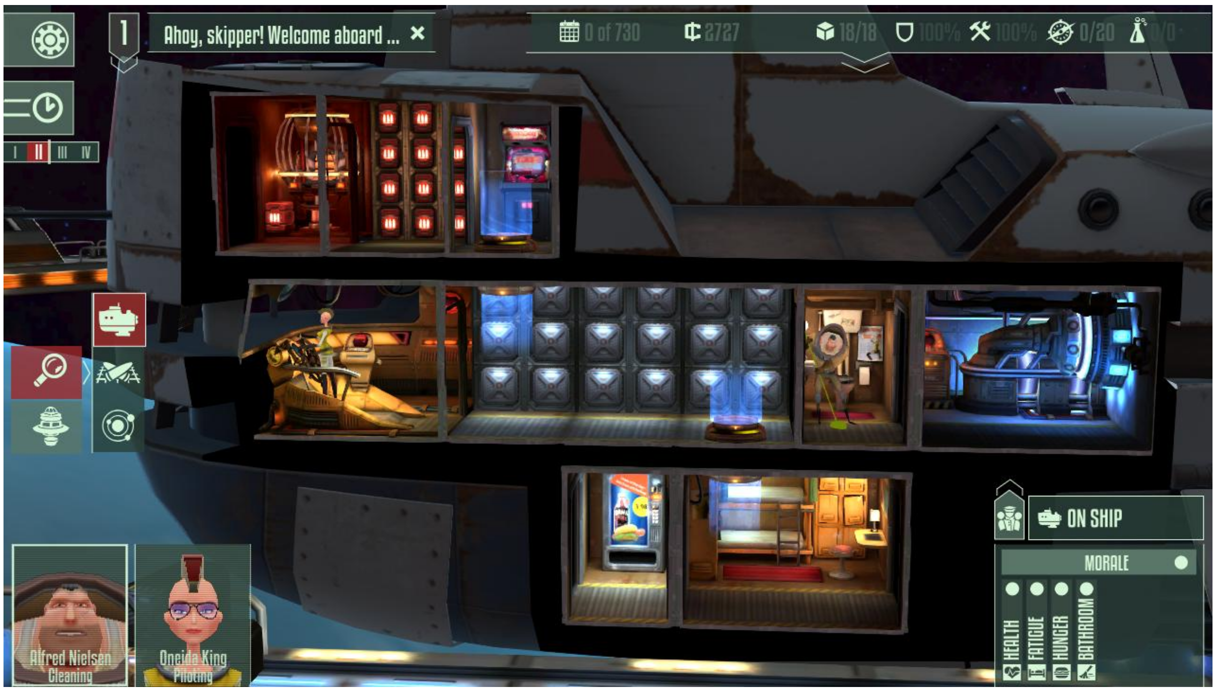


Figure 20: Crewmembers working in a fully loaded small space ship

The search for the best activity has the same prerequisites for the possible activities. Therefore, both searches are pulled together if possible to decrease the calculation time. To find the best activity for the most urgent need, all activities, which lower this need's value, are looked at. Each of these activities gets its utility value calculated. This is done with the technique of the utility systems from the category of the goal-oriented behavior.

In the case of the crewmember activities, the utility is calculated out of this data:

- the **distance** between the crewmember and the possible activity
- the **effect** of the activity on the current need as well as the effect on other needs
- both **urgency** values of the current need with different weightings
- whether the possible activity is the **current** activity
- the length of the **queue** in front of the activity
- how much **dirt** and **damage** the activity does on the room of the activity

Each of these values has a factor, which defines how much the value influences the overall utility of the activity. The distance between the crewmember and the possible activity is meant to decide between equal activities in different rooms. It should only decide for a less effective activity over a more effective if the latter is at the other end of a big ship. The walk speed or the required time to satisfy a need by doing the activity are not considered anymore because it led to a hard-to-control non-linear change of the utility value over time.

The calculation of the distance between the crewmember and the activity consumes a lot of computation time. In the first simple implementation, a simplified A*-path search was used for every utility calculation. This meant that this path search was done up to eight times for the crewmembers multiplied by about ten for the possible activities times about three for the behavior tree updates in the time-lapse-mode, all in a single frame. It lowered the frame-rate in this situation even under one FPS. To solve this problem, two changes were made: the path-finding results are cached for the current position of the crewmember and the path finding is only done if the straight line between the crewmember and the possible activity is shorter than the actual distance to the best activity so far. The path-finding results are cached for each crewmember's current position as the starting point. If he or she moves, the cache is emptied. This is probably not the best solution but it has an acceptable trade-off between computation time and memory usage. Because the crewmembers can only walk horizontally through the rooms and vertically through lifts, the distance is calculated like this: $dist = abs(x_{target} - x_{origin)} + abs(y_{target} - y_{origin)}$. Here, the path-finding algorithm can only find longer paths than the straight line. Knowing that, activities with exactly the same utility values without the distance part can be omitted if the straight line is not shorter than the shortest path so far. This may sound like a rare occasion, but in fact, it brings back the computation time within reasonable limits.

If several activities have exactly the same utility value, one is chosen randomly to improve the diversity of the behavior. In a concrete case, this makes crewmembers choose randomly the lift weights or running on treadmill activities of the fitness room, instead of choosing one of them only if the other is occupied. Both activities have equal effects but have different animation. Therefore, the decision does not matter for the need system but it does matter for the players.

Finding the best task is a little different. Tasks have a priority value stored instead of a utility value. This priority is calculated in a similar way but it is independent from a certain crewmember. Instead, it depends on the room providing the task. These values are updated by the room structure-class which owns the task providers which in turn owns the tasks. The crewmembers take the tasks they have the skills for from their current task provider. They set these tasks ordered by their priority in the crewmembers' task list. Then, they set the task with the highest priority in both lists as their current task. The priority values are calculated differently for each task type. The priority of the repair task for example is based on the damage of the providing room and its importance. This importance value causes crucial rooms like the cockpit to be repaired more often than e.g. cargo rooms. Other tasks such as the man turret tasks have simply the maximum priority value of one but only if the crew is in a space fight. However, each task type needs an own algorithm for the priority calculation.

The last search for activity is the search while on a space station. Crewmembers find their current need just like on the ship by choosing the most urgent need which can be satisfied. As space stations have activities for all needs, the current need is simply the most urgent one. Afterwards, crewmembers choose a random activity, which satisfies the current need. Therefore, only the effect of the activity and the need's urgency are considered for the space station activities. Using a utility function like on the ship would cause crewmembers to do the activities with the greatest effect again and again exactly in the same order. Choosing the activities randomly improves the diversity of the behaviors a lot.

It turned out through the use of utility functions that the utility values themselves are not important. Only their relation to each other is relevant for ordering the activities. As mentioned before, utility systems need to compare all actions with every motive what leads to $O(motives * actions)$ in time. By choosing the most urgent need before searching an activity, the complexity is decreased to $O(motives + actions)$ which is significantly less and might make it possible to get Cosmonautica running on mobile devices.

Although activities can be occupied, they are not excluded from the activity search. Instead, the utility function gives a penalty for the utility if the possible activity is not the current activity. This prevents crewmembers from switching their current activity to an equal one in the same room but it allows crewmembers to switch activities with others and keeps the path-finding exclusion method working.

5.4. Artificial Intelligence of Space Ships in Fights

Although the development of the space fights is not finished and the diplomacy system is in the concept phase, it should be described here to show for what behavior trees and utility functions can be used without a lot of modification.

Space fights in Cosmonautica are triggered if the player is attacked by a pirate for example. Then, both the player ship and the pirate stay on the trajectory of the attacked ship to its destination planet. Only the relative velocities are important for the fight. Players can choose special actions from crewmembers doing tasks like piloting or operating a turret. Pilots provide different maneuvers such as a circle maneuver or fleeing. Turret gunners enable turrets to fire the corresponding projectiles or torpedoes.

The behavior tree for the enemy ships is meant to simulate the player's decisions as well as simplified crewmembers on the ship. As the crewmembers of the AI ships do not really exist for performance reasons, the re-loading times and tasks with the special actions have to be updated in a different way. The enemy ship has a room structure and room with tasks just like the player's ship. However, its tasks are not done by crewmembers but updated by the ship behavior tree. In short, the artificial intelligence for the enemy ships plays the same role as the player: it triggers special actions if useful and makes simple plans like starting a fly-by maneuver only if the broadside is ready to fire.

During the fight, both the player and the AI ship can make offers and demands via the diplomacy screen to settle the fight without destroying the enemy. Destroying it would cause most of its goods to be destroyed as well what makes the diplomacy demands give much better rewards. These demands have to be accepted by the recipient, then goods are exchanged and the fight ends. The ship AI has to be able to make demands and accept incoming demands if they are acceptable. How much can be demanded depends on the information of the demanding party about the other ship and its interior. The AI should not be too easy to be taken in by the player and it should demand the player to pay a painful amount of goods without destroying him. Offers can consist of money, crewmembers, possible rooms, freight, truce, and ammunition from both parties. For such a wide range, a utility function will be needed which is not ready yet.

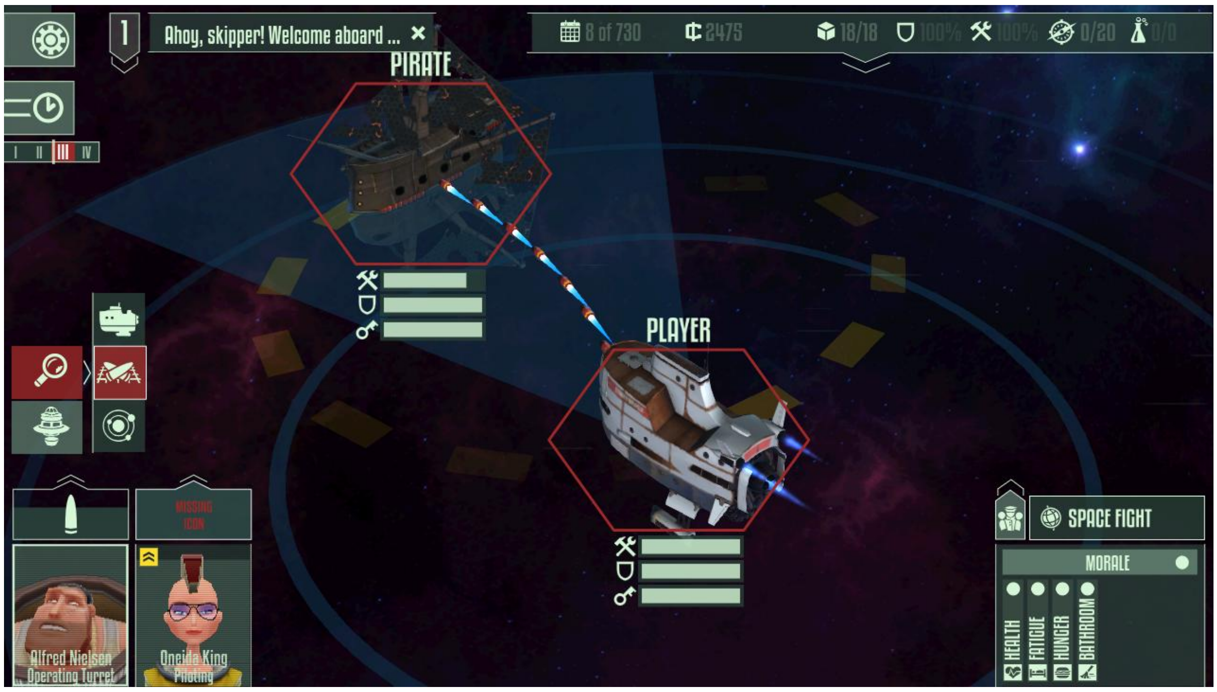
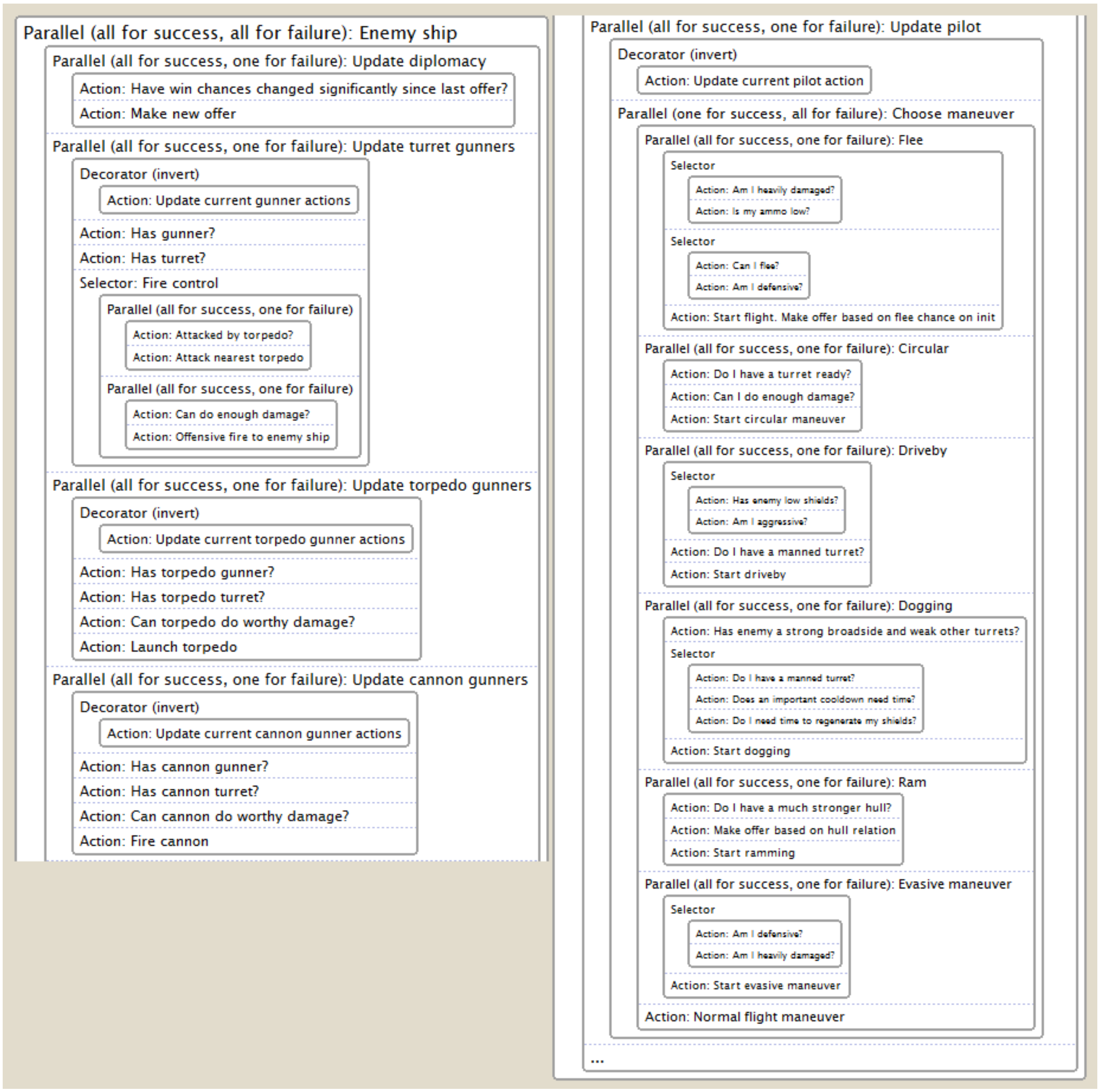


Figure 21: Firing a gun turret at a pirate ship in a space fight while in a circular maneuver

The root node of the enemy ship's behavior tree is a parallel node with the policies all for success and all for failure. These policies are used, because the tree should try to update all parts but some might not be available. For example, the AI could have a ship without torpedo turrets or all turret gunners have died, causing the turret part to fail. At first, the diplomacy system is updated meaning that a new offer should be made if the win chances have significantly changed. This "make offer" node will use a utility function to rate the situation and make demands.

The next parts are the parts for the different types of turrets. They are structured similar: at first, they update running action if existing, if not, they check whether all prerequisites are met and finally start a new action if it is useful. The update actions node is used to update the tasks of the AI ship. The gun turret can be used for attack and for defense. The defense fire tries to destroy an incoming torpedo. All turret parts check if the action, e.g. fire broadside can do an amount of damage which is worthy to the cost of the projectile. This will be calculated in utility functions evaluating the hit probability for example.



- Parallel (all for success, all for failure): Enemy ship
  - Parallel (all for success, one for failure): Update diplomacy
    - Action: Have win chances changed significantly since last offer?
    - Action: Make new offer
  - Parallel (all for success, one for failure): Update turret gunners
    - Decorator (invert)
      - Action: Update current gunner actions
    - Action: Has gunner?
    - Action: Has turret?
    - Selector: Fire control
      - Parallel (all for success, one for failure)
        - Action: Attacked by torpedo?
        - Action: Attack nearest torpedo
      - Parallel (all for success, one for failure)
        - Action: Can do enough damage?
        - Action: Offensive fire to enemy ship
  - Parallel (all for success, one for failure): Update torpedo gunners
    - Decorator (invert)
      - Action: Update current torpedo gunner actions
    - Action: Has torpedo gunner?
    - Action: Has torpedo turret?
    - Action: Can torpedo do worthy damage?
    - Action: Launch torpedo
  - Parallel (all for success, one for failure): Update cannon gunners
    - Decorator (invert)
      - Action: Update current cannon gunner actions
    - Action: Has cannon gunner?
    - Action: Has cannon turret?
    - Action: Can cannon do worthy damage?
    - Action: Fire cannon
  - Parallel (all for success, one for failure): Update pilot
    - Decorator (invert)
      - Action: Update current pilot action
    - Parallel (one for success, all for failure): Choose maneuver
      - Parallel (all for success, one for failure): Flee
        - Selector
          - Action: Am I heavily damaged?
          - Action: Is my ammo low?
        - Selector
          - Action: Can I flee?
          - Action: Am I defensive?
        - Action: Start flight. Make offer based on flee chance on init
      - Parallel (all for success, one for failure): Circular
        - Action: Do I have a turret ready?
        - Action: Can I do enough damage?
        - Action: Start circular maneuver
      - Parallel (all for success, one for failure): Driveby
        - Selector
          - Action: Has enemy low shields?
          - Action: Am I aggressive?
        - Action: Do I have a manned turret?
        - Action: Start driveby
      - Parallel (all for success, one for failure): Dogging
        - Action: Has enemy a strong broadside and weak other turrets?
        - Selector
          - Action: Do I have a manned turret?
          - Action: Does an important cooldown need time?
          - Action: Do I need time to regenerate my shields?
        - Action: Start dogging
      - Parallel (all for success, one for failure): Ram
        - Action: Do I have a much stronger hull?
        - Action: Make offer based on hull relation
        - Action: Start ramming
      - Parallel (all for success, one for failure): Evasive maneuver
        - Selector
          - Action: Am I defensive?
          - Action: Am I heavily damaged?
        - Action: Start evasive maneuver
      - Action: Normal flight maneuver
  - ...

Figure 22: Behavior tree for the artificial intelligence of enemy ships

The following part after the turrets is for the pilot. Similar to the turrets, the pilot action is updated as first. If there is nothing to update, the next maneuver is searched. It is chosen by evaluating the hull damages of both ships, the turrets' loading states and some more information. Afterwards, other parts e.g. for hacking might be added.

6. Conclusion

The behavior trees are now in use at Chasing Carrots and will probably stay. They are up and running in Cosmonautica controlling crewmembers as well as the AI ships. Utility functions have proved to be able to handle situations where too many options for behavior trees have to be evaluated.

Behavior trees have shown these advantages:

- The **flexibility** is very high as shown through the usage of behavior trees for AI ships what was not planned in the beginning.
- The **maintaining** of existing behavior trees is fast and easy even without a graphical modeling tool.
- The **code** of BT nodes can easily be **re-used**.
- Entire BT **nodes** can be **re-used**.
- **Node instances** can be **re-used** in the standard implementation but only if they do depend only on the call of their evaluation methods and not the initialization or termination. However, the memory-optimized implementation is recommended, instead of this special case.
- **Parts of the behavior trees** can be easily **re-used**.
- The code for the BT action nodes can **encourage a good data capsulation** between the classes related to the controlled class.
- The **quality** of the behaviors is high enough for the usage of behavior trees in people simulations.
- Action nodes can be **as detailed as needed**. They can contain and control entire state machines.

Nothing is perfect. Behavior trees are no exception as they have shown these disadvantages:

- The complexity of behavior trees can make them **hard to understand**. In order to understand the overall behavior of a character, one must understand the behavior tree as a whole as well as its special action nodes. This is especially difficult if the action node code and the BT modeling is done by different people.
- The **distribution of complexity** between possibly multiple behavior trees for a single character, each behavior tree on its own, and their action nodes' code is tricky. It can be useful to combine or split behavior trees. Putting too much complexity into the action nodes makes the behavior tree harder to understand as well as decreasing its re-usability. Usually, the entire complexity cannot be modeled into the behavior tree and trying it will result in a gigantic and slow behavior tree.
- The action nodes need public or friend methods to work what **deteriorates** the **capsulation** of their controlled classes.
- Too detailed policies, such as a concrete number of required fails for making a parallel node fail, are a **possible source of errors**. This is because the policies have to be changed when the number of child nodes of the parallel node changes.

- Splitting up similar things, like checking all activities and then choosing one, might be required to re-use nodes but cause **code duplication** and are **inefficient** if both nodes are executed in a row.
- **Debugging** a behavior tree with the usual tools is **difficult**, mostly because of the useless call-stack. It is very helpful to see every node's state of the entire behavior tree with one glance. Therefore, a debug print or an external tool is recommended.

Here are some other experiences gained through Cosmonautica:

- Utility functions are great as action nodes for behavior trees where a normal node structure is not applicable or needs too many resources.
- Behavior trees are very powerful. More and more advanced behaviors might not be good for the game. Crewmembers in Cosmonautica for example do not avoid each other anymore during activities. Instead, they wait in a queue in front of the activity giving the player a chance to see the bottlenecks in the ship.
- Regarding parallel nodes:
  - Parallel nodes with the policies all for success and one for failure are like sequences always beginning their evaluation at their first child.
  - Parallel nodes with the policies one for success and all for failure are like selectors always beginning their evaluation at their first child.
  - Parallel nodes with the policies all for success and all for failure just try to evaluate all their children beginning always at their first child.
  - Parallel nodes with the policies one for success and one for failure seem useless.
- The beginning of the behavior tree development was a bit rushed. Some names and processes in the code are not consistent to the literature what might be confusing.
- State-like behaviors in behavior trees usually require a small state machine controlled by action nodes. This decreases the complexity of these state machines to the absolute minimum and therefore increases their re-usability. It also makes the entire behavior tree harder to understand. However, this is still much better than a single big state machine.

It has been shown that behavior trees are a good replacement for state machines in many cases. Especially if state machines would be very big or change their states very often, the behavior control should be done with a behavior tree instead. Behavior trees need an object-oriented programming language for the node classes with their common interface. Behavior trees are still useful even when state machines are needed by action nodes.

7. Outlook

Although the crewmember and ship behaviors already work quite well, there is still much more work to do on Cosmonautica. The crewmembers do not react to each other now. Creating likes and dislikes amongst them will be kept rather simple. A first idea is that their relationships should change over time when being near each other or working together as two examples. The change of the relationship could be determined by special skills and modifications as well as their current relationship and similarities, which can be e.g. race, religion, gender, and skills.

Depending on the progress of the development and the remaining time, ground mission might be added to Cosmonautica. The player will be able to select crewmembers to send them onto a planet or let them board another space ship. There, they will act as a squad fighting enemy characters or gathering resources and artifacts.

Space fights were already mentioned. Some parts of them are still missing: cyber warfare, space fighters, more maneuvers and weapon types, energy shields, different enemy ship types with different behavior characteristics, and many more. The diplomacy system is also not implemented now. The main difficulty of this will be to estimate the win chances correctly and then make offers or demands without deceiving the player.

Many more side-missions, which are much more complex, will be added. Some of these missions will require the player to support special needs of critters or guests, for example. Such additional characters might get another, maybe simplified, version of the crewmembers' behavior tree.

Now, the behavior tree system can only be used from within the C++ code. Although the game designers might be able to modify the behaviors directly in the code, it will not be as efficient as they need it. Therefore, a way has to be found to give them the possibility to modify the behaviors with scripts or a graphical tool. The first possibility could be to expose C++ functions to Lua, which generate new nodes and attach them to a behavior tree. Such scripts have to be executed before the controlled object instances, e.g. crewmembers, are constructed. A graphical tool could then create such scripts automatically, based on the modeled behavior trees.

The system for artificial intelligence at Chasing Carrots consisting of behavior trees and utility functions has proved to be useful and will most likely stay for a while. The behaviors in Cosmonautica will be increased, modified, and extended. With behavior trees being such a flexible technique, these tasks will be less annoying and more efficient.

Glossary

Bot: an artificial intelligence agent. Bots simulate the player behavior either to act as opponent or teammate in multiplayer games. Bots are a special variant of NPCs.

Cycle (graph theory): a path through a graph with the same node at its beginning and at its end.

Edge (graph theory): the connections between nodes, also known as vertices. In behavior trees, edges connect always one node of a higher hierarchy with a node of a lower hierarchy. The higher one is the parent node and the lower is its child node.

Independent developer: is a game development company, also known as indie dev, that does not depend on money from a contracting entity like a publisher.

Leaf node (graph theory): at the ends of a tree. These nodes are the only ones without child nodes. They stand for an action or a decision made by using a behavior tree.

Loop (graph theory): an edge that has its beginning and end at the same node.

Node: represents a decision or an action in a behavior tree.

Non-player character: an object in the game world. It is usually similar to the player's avatar. In contrast to this, an NPC cannot be controlled by any player. Thus, their behavior is only controlled by the program.

Root node (graph theory): the root of a behavior tree is the top of the hierarchy. Every other node is directly or indirectly a child of this node.

Scrum: an agile development methodology. It is a fast, iterative, and incremental process. It consists of meetings and artifacts to increase productivity but relies on an open working atmosphere.

Unit test: a method of ensuring the functionality of a portion of code. It does not need to be a program on its own. Instead, it can be run by another program, the testing suite.

Appendix

A Behavior Tree Base Nodes

Important parts of the base node implementation:

```
/*
Adapted from the Behavior Tree Starter Kit from AiGameDev.com:
https://github.com/aigamedev/btsk
*/

#ifndef BehaviourTreeNodes_h_
#define BehaviourTreeNodes_h_

#include <vector>

namespace BTStatus{
        enum Enum{
                Invalid,
                Success,
                Failure,
                Running,
                Aborted
        };
}

/* This is the abstract base class for the behavior tree nodes. */
class BTNode
{
public:
        BTNode(): mStatus(BTStatus::Invalid){};

        virtual void onInitialize() {}
        virtual void onTerminate(BTStatus::Enum) {}
        BTStatus::Enum tick()
        {
                if (mStatus != BTStatus::Running)
                        onInitialize();

                mStatus = update();

                if (mStatus != BTStatus::Running)
                        onTerminate(mStatus);

                return mStatus;
        }
        void reset(){mStatus = BTStatus::Invalid;}
        bool isTerminated() const {return mStatus == BTStatus::Success ||
                mStatus == BTStatus::Failure;}
        bool isRunning() const {return mStatus == BTStatus::Running;}
        BTStatus::Enum getStatus() const {return mStatus;}
        void abort()
        {
                onTerminate(BTStatus::Aborted);
                mStatus = BTStatus::Aborted;
        }

protected:
        virtual BTStatus::Enum update() = 0;
        BTStatus::Enum mStatus;
};

template <class ObjectType>
class BTAction : public BTNode
{
public:
        BTAction(ObjectType* object = NULL,
                bool (ObjectType::*evaluationFunction)() = NULL)
        : mEvaluationFunction(evaluationFunction), mObject(object){}

protected:
        virtual BTStatus::Enum update()
        {
                if(mEvaluationFunction && mObject)
                {
                        if((mObject->*mEvaluationFunction)())
                                return BTStatus::Success;
                        else
                                return BTStatus::Failure;
                }
                else
                        return BTStatus::Invalid;
        }

        bool (ObjectType::*mEvaluationFunction)();
        ObjectType* mObject;
};

class BTCompositeNode : public BTNode
{
public:
        virtual void onInitialize()
        { mCurrentChildNode = mChildNodes.begin(); }

        virtual void addChild(BTNode* child)
        { mChildNodes.push_back(child); }

        virtual void onTerminate(BTStatus::Enum)
        {
                for (BTNodeVector::iterator child(mChildNodes.begin());
                        child != mChildNodes.end(); ++child)
                {
                        if((*child)->getStatus() == BTStatus::Running)
                                (*child)->abort();
                }
        }
```

```
protected:
        virtual BTStatus::Enum update() = 0;
        std::vector<BTNode*> mChildNodes;
        std::vector<BTNode*>::iterator mCurrentChildNode;
};

class BTSequence : public BTCompositeNode
{
protected:
        BTStatus::Enum update()
        {
                if (mResetEveryUpdate)
                        this->onInitialize();

                for (;;)
                {
                        BTStatus::Enum s = (*mCurrentChildNode)->tick();

                        if (s != BTStatus::Success)
                                return s;

                        if (++mCurrentChildNode == mChildNodes.end())
                        {
                                this->onInitialize();
                                return BTStatus::Success;
                        }
                }
        }
};

class BTSelector : public BTCompositeNode
{
protected:
        BTStatus::Enum update()
        {
                for (;;)
                {
                        BTStatus::Enum s = (*mCurrentChildNode)->tick();

                        if (s != BTStatus::Failure)
                                return s;

                        if (++mCurrentChildNode == mChildNodes.end())
                        {
                                return BTStatus::Failure;
                        }
                }
        }
};
```

```
class BTParallel : public BTCompositeNode
{
public:
	enum Policies{
		RequireOne,
		RequireAll
	};

	BTParallel(Policies successPolicy = RequireAll,
		Policies failurePolicy = RequireOne) :

	mSuccessPolicy(successPolicy), mFailurePolicy(failurePolicy){}
	virtual ~BTParallel();

	virtual void onTerminate(BTStatus::Enum)
	{
		for (BTNodeVector::iterator it = mChildNodes.begin();
			it != mChildNodes.end(); ++it)
		{
			BTNode& b = **it;
			if (b.isRunning())
				b.abort();
		}
	}

protected:
	Policies mSuccessPolicy;
	Policies mFailurePolicy;
```

```
        BTStatus::Enum update()
        {
                int successCount = 0, failureCount = 0;

                for (BTNodeVector::iterator it = mChildNodes.begin();
                        it != mChildNodes.end(); ++it)
                {
                        BTNode& b = **it;
                        b.tick();

                        if (b.getStatus() == BTStatus::Success)
                        {
                                ++successCount;
                                if (mSuccessPolicy == RequireAll &&
                                        successCount == mChildNodes.size() ||
                                        mSuccessPolicy == RequireOne &&
                                        successCount == 1)
                                        return BTStatus::Success;
                        }
                        if (b.getStatus() == BTStatus::Failure)
                        {
                                ++failureCount;
                                if (mFailurePolicy == RequireOne &&
                                        failureCount == 1 ||
                                        mFailurePolicy == RequireAll &&
                                        failureCount == mChildNodes.size())
                                        return BTStatus::Failure;
                        }
                }
                return BTStatus::Running;
        }
};

class BTDecorator : public BTNode
{
public:
        BTDecorator() : mChildNode(NULL) {}
        BTDecorator(BTNode* child) : mChildNode(child) {}
        virtual BTDecorator& setChild(BTNode* child)
                {mChildNode = child; return *this;}
        virtual void removeChild(){mChildNode = NULL;}

protected:
        virtual BTStatus::Enum update() = 0;
        BTNode* mChildNode;
};
```

```
class BTReturnStateModificator : public BTDecorator
{
public:
        enum Policy{
                AlwaysInvalid,
                AlwaysRunning,
                AlwaysFailing,
                AlwaysSuccessfull,
                InvertFailureAndSuccessfull
        };

        BTReturnStateModificator(BTNode* child = NULL,
                Policy policy = InvertFailureAndSuccessfull) :
                BTDecorator(child), mPolicy(policy) {}

        void setPolicy(Policy newPolicy){mPolicy = newPolicy;}
        Policy getPolicy(){return mPolicy;}

protected:
        Policy mPolicy;

        virtual BTStatus::Enum update()
        {
                BTStatus::Enum result(BTStatus::Invalid);
                if (mChildNode)
                        result = mChildNode->tick();

                switch(mPolicy)
                {
                case AlwaysInvalid:
                        return BTStatus::Invalid;
                        break;
                case AlwaysRunning:
                        return BTStatus::Running;
                        break;
                case AlwaysFailing:
                        return BTStatus::Failure;
                        break;
                case AlwaysSuccessfull:
                        return BTStatus::Success;
                        break;
                case InvertFailureAndSuccessfull:
                        if (result == BTStatus::Failure)
                                result = BTStatus::Success;
                        else if (result == BTStatus::Success)
                                result = BTStatus::Failure;
                        return result;
                        break;
                default:
                        return BTStatus::Invalid;
                }
        }
};

#endif // BehaviourTreeNodes_h_
```

Literature

AngryAnt. (2013). *The Behave project*. Retrieved from AngryAnt: http://angryant.com/behave/

Byte56. (2013, March 25). *Decision Tree vs Behavior Tree*. Retrieved from Game Development Stack Exchange: http://gamedev.stackexchange.com/questions/51693/decision-tree-vs-behavior-tree

Champandard, A. J. (2007, September 5). *The Gist of Hierarchical FSM*. Retrieved from AiGameDev.com: http://aigamedev.com/open/article/hfsm-gist/

Champandard, A. J. (2007, September 6). *Understanding Behavior Trees*. Retrieved from AiGameDev.com: http://aigamedev.com/open/article/bt-overview/

Champandard, A. J. (2008, December 28). *Behavior Trees for Next-Gen AI.* Retrieved from AiGameDev.com: http://files.aigamedev.com/insiders/BehaviorTrees_Slides.pdf

Champandard, A. J. (2008, October 31). *Special Report: Goal-Oriented Action Planning.* Retrieved from AiGameDev.com: http://files.aigamedev.com/members/Goal-Oriented-Action-Planning.pdf

Champandard, A. J. (2012, February 12). *Behavior Tree Starter Kit Source Release*. Retrieved from AiGameDev.com: http://files.aigamedev.com/AiGD-BehaviorTreeStarterKit-1.1-src.zip

Champandard, A. J. (2012, February 26). *Understanding the Second Generation of Behavior Trees.* Retrieved from AiGameDev.com: http://files.aigamedev.com/members/SecondGeneration_Slides.pdf

Champandard, A. J. (2012, March 21). *Understanding the Second Generation of Behavior Trees*. Retrieved from AiGameDev.com: http://aigamedev.com/premium/tutorial/second-generation-behavior-trees/

Chasing Carrots KG. (2013, October 29). *Chasing Carrots About.* Retrieved from Chasing Carrots: http://chasing-carrots.com/pdfs/company.pdf

Chasing Carrots KG. (2013). *Cosmonautica*. Retrieved from Cosmonautica: http://cosmo-nautica.com/

Deja Tools LLC. (2013). *Deja Insight*. Retrieved from Deja Tools: http://www.dejatools.com/dejainsight

Electronic Arts Inc. (2013). The Sims Official Site. *The Sims 3*. Electronic Arts Inc. Retrieved from http://www.thesims.com/en-us

Hecker, C. (2009, April 12). *My liner notes for spore/Spore Behavior Tree Docs*. Retrieved from Chris Hecker's Homepage: http://www.chrishecker.com/My_Liner_Notes_for_Spore#Behavior_Tree_AI

Isla, D. (2005, March 11). *GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI*. Retrieved from Gamasutra: http://www.gamasutra.com/view/feature/2250/gdc_2005_proceeding_handling_.php

Kenneth, S.-J., Olsen, T., & Phan, L. H. (2011). *Dynamic Difficulty Adjustment Using Behavior Trees.* Aalborg: Aalborg University.

Keßler, D. (2008). *Parallel Agent Models for a Realtime Strategy Game.* Kassel: University of Kassel. Retrieved from http://www.uni-kassel.de/eecs/fileadmin/datas/fb16/Fachgebiete/PLM/Dokumente/Master_Bachelor_Diplom/bachelorfinal.pdf

Klei Entertainment Inc. (2012). Don't Starve. Retrieved from http://www.dontstarvegame.com/

Millington, I., & Funge, J. (2009). *Artificial Intelligence for Games, 2nd Edition.* Boca Raton: CRC Press.

Oortmerssen, W. v. (n.d.). *TreeSheets*. Retrieved from Wouter's Wiki: http://strlen.com/treesheets/

Rival {Theory}. (2013, October 29). *RAIN User Manual: Nodes*. Retrieved from Rival{Theory}: http://rivaltheory.com/wiki/doku.php?id=behavior:behaviortrees:nodes:start

Russell, S., & Norvig, P. (2004). *Künstliche Intelligenz: Ein moderner Ansatz.* München: Pearson Studium.

Vassos, S. (2013). *Introduction to STRIPS Planning and Applications in Video-games.* Retrieved from Department of Computer, Control, and Management Engineering Antonio Ruberti at Sapienza University of Rome: http://www.dis.uniroma1.it/~degiacom/didattica/dottorato-stavros-vassos/AI&Games-Lecture1-part2.pdf

Waner, S. (2004, January). *Summary of Chapter 9 in Finite Mathematics / Finite Mathematics & Applied Calculus, Topic: Markov Systems*. Retrieved December 11, 2013, from Finite Mathematics & Applied Calculus: http://people.hofstra.edu/stefan_waner/RealWorld/Summary8.html