DECISION-MAKING IN COSMONAUTICA

Achieving a natural behavior of NPCs is a great challenge but also necessary for immersion. Philipp Erler explains the artificial intelligence in Cosmonautica.

Philipp Erler just finished his studies of Software Engineering at the University of Applied Sciences in Esslingen.

Philipp was a programmer at Chasing Carrots while writing his bachelor's thesis »Decision-Making in a Group of Artificial Intelligences in Different Simulated Environments«. He worked on the prototype of a racing game before moving on to work on »Cosmonautica«. More info on the game at www.cosmo-nautica.com.

The bachelor's thesis and the behavior tree core system: www.pplusplus.lima-city.de/bachelorsthesis.html

Cosmonautica is the second game of the small indie studio Chasing Carrots KG in Stuttgart, Germany. The game combines »Sims«-like management of the crew members in the player's spaceship with trading and space fights. Instead of getting direct orders from the player, the NPC crew members choose their actions on their own within certain limits. The AI system must therefore make these decisions. Since the crew members' behaviors are a main part of the game and crucial for the player's success, the decision-making system must support quite complex behaviors. In short, it is meant to choose the work which the crew members do or how they satisfy their needs. Then, the system should control how they do it to ensure that everything looks natural.

Cosmonautica was in an early stage of development when the work on the decision-making system started. Some behavior related systems, like the need system for example, did already exist; but the behaviors were just placeholders. Therefore, the new decision-making system had to fit between these systems. The decision-making core system is meant to be a part of the code base Chasing Carrots will use for future games.

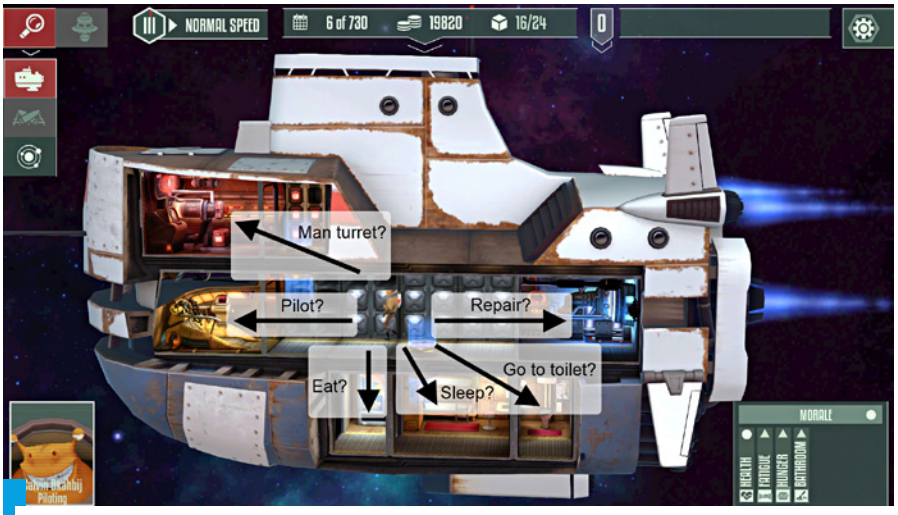
Behavior Trees

State machines are very simple but can hardly be re-used. Fuzzy logic or Markov systems as extension for state machines or stand-alone are not necessary for Cosmonautica because real probabilistic behaviors and multiple states are not desired. Goal-oriented action planning is a great technique for action planning, which is not required. But utility functions, a part of the goal-oriented behaviors, made their way into Cosmonautica. Behavior trees combine simplicity with good behaviors and awesome flexibility.

Tree Structure

Behavior trees are directed, acyclic graphs made of nodes and edges. To be more exact, behavior trees are rooted and directed trees. Nodes need to know only their child nodes, which leads to a hierarchy. All nodes share the same interface, making it easy to exchange them, which in turn is the reason for the flexibility of the behavior trees. A behavior tree can be started very simple as a placeholder

Cosmonautica combines »Sims«-like crew management with trading and space fights.



What should Calvin (lower left-hand corner) do? Moreover, where? Should he work or satisfy his needs? How long and how often? These are difficult questions for humans and even more so for a computer.

and can be easily extended later on with working states in between.

Node States

Each node sets its own state every time it is evaluated, then it returns this state to its parent node. Alex J. Champandard says in his talk »Understanding the Second Generation of Behavior Trees« that he tried many possible state combinations but the following are the best:

- **Invalid** – This state is set in the constructor of the base node class. Therefore, the invalid state indicates that the node's evaluation method has never been called. It can also be set to show that an exception occurred.
- **Successful** – This state 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** – Any node may cause the teleport node to be reset. Then, the aborted state is used. This could happen when the character received damage while opening the portal.

	State Machines	Fuzzy Logic / Markov Systems	Goal-Oriented Behavior	Rule-Based Systems	Behavior Trees
Simplicity	+	-	o	-	+
Separation of Game Design and Programming	+	o	o	+	+
Flexibility	-	o	o	o	+
Behavior Quality	-	o	o	+	+
Efficiency	+	-	-	+	o
Conclusion	Very simple	Probabilistic	Great for planning	Limits not reached yet	High flexibility

This overview compares different decision-making techniques and their characteristics. Note that behavior trees might not be the best choice for your own decision-making system. As Millington and Funge say in »Artificial Intelligence for Games«, it can be cumbersome to achieve some behaviors with behavior trees. This applies especially to state-like behaviors.

The crew members' behavior is a huge part of the game, so the decision-making has to feel natural to the player.

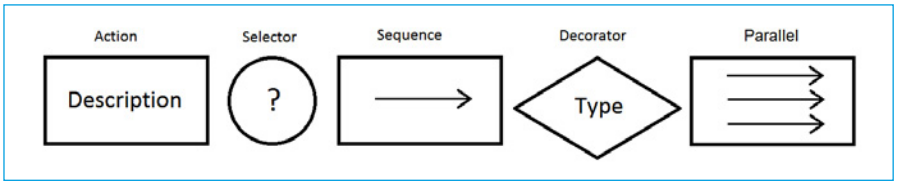
There are two events reacting on the states of a node. The »on initialize«-event is triggered before the evaluation but only if the node's state is not running. The »on terminate«-event is triggered after the evaluation but before returning its state, only if it is not running. Both events may execute 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 and stop the animation.

During the development of Cosmonautica, no situation occurred that required different states. In fact, the whole behavior tree system applied by Chasing Carrots is based heavily on the behavior tree starter kit by AiGameDev.com.

Basic Node Types

Millington and Funge suggest the following behavior tree node types:

- **Leaf Nodes** – They are at the end of the behavior tree and have no child nodes.
  - **Action** – These nodes alter the state of the game or a game object.
  - **Condition** – They check a fact in the game.
- **Composite Nodes** – These nodes can 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 other than successful.
  - **Selector** – They try to evaluate their child nodes until one returns other 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. Instead, they start always with the first child.
- **Decorators** – These nodes have only one child node. They change the way this child is evaluated. Many different types of decorators are imaginable. For example, they can change the child's return value or act as a breakpoint.



The most important node types of the behavior tree notation used by Millington and Funge.

Behavior Tree Evaluation

Behavior trees are evaluated from the root every time the tree is updated, which can happen with a significantly lower frequency than the normal frame rate. In Cosmonautica, the update of the crew members' behavior trees is done every ten in-game minutes, which equals about two real seconds. This relieves the CPU quite a lot but leads to a more difficult optical representation of the crew members.

Each node of the behavior tree tries to run through all of its child nodes. The behavior tree can be designed in such a way that the root node's result indicates if the tree found a fitting behavior for the situation of the object it currently controls.

Implementation

Once the behavior tree has been assembled, there are usually no more changes as long as the game runs. Therefore, a vector of pointers to the child nodes is the best way to achieve the special tree structure. The only exceptions are decorators, which only have a single child node and leaf nodes without any children.

Whether the crew member should work is controlled inside the crew member class through his assigned tasks and the game time. This class has a small internal state machine for the walking and doing state.

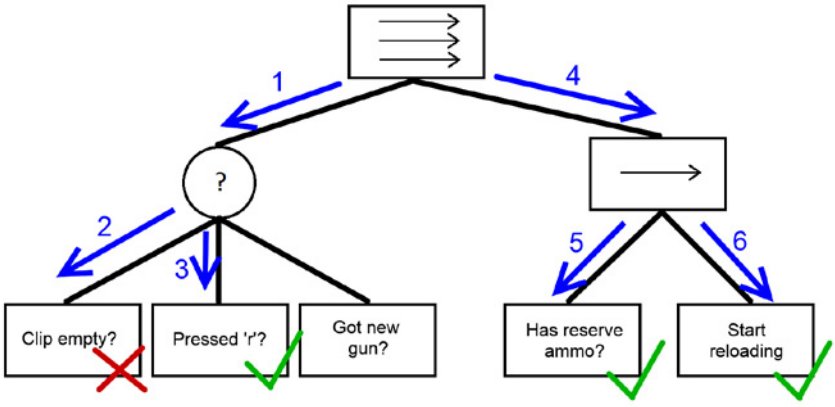
Finding the Best Activity through Utility Functions

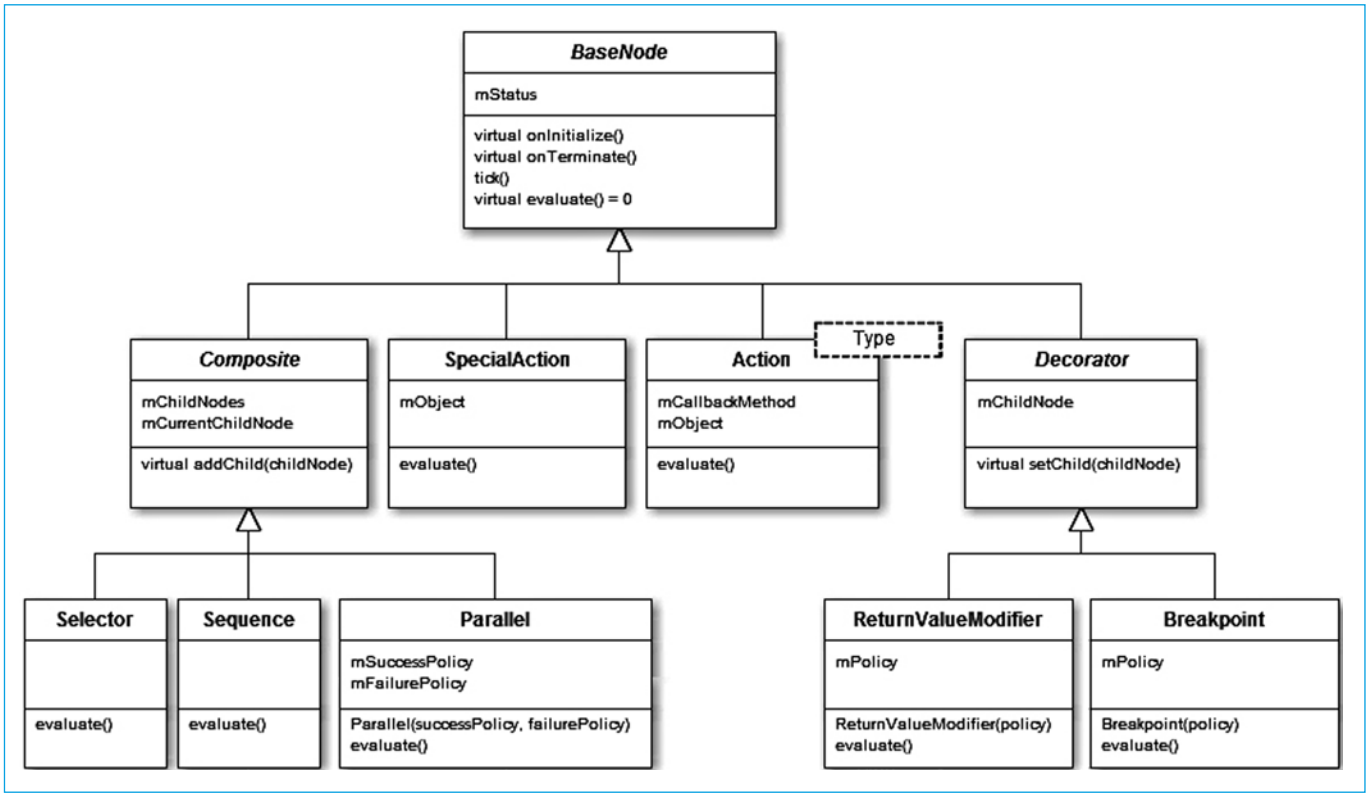
There are several reasons why the actual search for the best activity is not modeled into the behavior tree but is done within action nodes. The first reason is that the activities in Cosmonautica are added by placing rooms and they are removed when the providing room is sold or destroyed. To model this into the behaviour tree would require a regular modification of the tree while the game runs, which might have nasty side effects. If all approximately hundred rooms of a big ship were placed, the behavior tree would then grow by at least each two hundred tasks and activities plus many management nodes. This would be simply too much to keep a good performance. Another reason why the activity search is not modeled in the behavior tree is that not all possible activities are already known and the activities are planned to be changed by mods. Therefore, maintaining the behavior tree will be almost impossible. With the approach of putting the searches into action nodes, only utility functions have to be added for every new type of activity or task.

In order to find the current need, all needs of a crew member are considered beginning with the most urgent. A need can only be the current need if an activity exists, which can satisfy this need. The search for the best activity has the same conditions as the search for the current need. Therefore, both searches are pulled together to decrease the calculation time. To find the best activity, all activities lowering the current need's urgency are looked at. Each of these activities gets its utility value calculated. The calculation itself is done with the utility-system technique from the category of the goal-oriented behavior.

Example: Evaluation of a Behavior Tree

The parallel root node runs its first child, a selector, which looks for a reason why the gun should be reloaded. This node looks for a »successful« child node and returns successful if one is found. Now, the root node runs its second child, a sequence node, which checks if the player has reserve ammo. If this check is successful, the sequence node starts the reloading. The parallel node here has the policies »all for successful« and »one for failure«, which means that its second child is not evaluated if the first child returns »failure«.





All node classes are derived from the abstract base node. This class manages that the »initialize« and »terminate« events are triggered, calls the evaluation method, sets its result as the current node state and returns it to the caller. The »Composite« class manages the vector of its child nodes. The »SpecialAction« class enables custom »initialize« and »terminate« events and provides special member variables.

The AI system makes sure that the crew members work or satisfy their needs when it makes sense. The crew members also choose the room for these activities with some intelligence.

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

- The **distance** between the crew member and the possible activity
- The **effect** of the activity on the current need as well as the effect on other needs
- The **urgency** value of the current need
- Whether the possible activity is the **current** activity or the target activity
- The length of the **queue** in front of the activity
- How much **dirt** and **damage** the activity does to the room of the activity

Each of these values has a factor assigned, which defines how much the value influences the overall utility of the activity. For example, the distance between the crew member and the possible activity is mostly meant to decide between equal activities in different rooms. Therefore, its factor is balanced to decide for a less effective activity over a more effective only if the latter is at the other end of the 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 nonlinear change of the utility value over time.

If several activities have exactly the same utility value, one is chosen randomly by the behavior tree to improve the diversity. In the case of the fitness room, this makes the crew members choose randomly between lifting weights and running on the treadmill. Both activities have the same effects on the character, but are animated differently.

Prioritizing Tasks

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 crew member. It depends only on the room providing the 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 engine room 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 ship is in a space fight. However, each task type needs a separate priority calculation.

Space Station Activities

The search for an activity on a space station is another case. Crew members find their current need just like on the ship by choosing the most urgent need which can be satisfied. Because space stations have activities for all needs, the current need is simply the most urgent one. Afterwards, crew members choose a random activity, which satisfies the current need. Therefore, only the type of the activity and the need's urgency are considered for the space station activities. Choosing the activities randomly improves the diversity of the behaviors a lot. The behaviors and decisions on the space stations can be much simpler because all the information the players get about it is a short text in the GUI.

Utility Functions

Our experience shows that the utility values themselves are not important at all. Only their relation to each other is relevant for ordering the activities. Utility systems usually need to compare all activities with every need what leads to »O(needs*activities)« in time. By choosing the most urgent need before searching an activity, the complexity is decreased to »O(needs+activities)«.

Conclusion

The AI system makes sure that the crew members work or satisfy their needs when it makes sense. The crew members also choose the room for these activities with intelligence. They wait in a queue for a reasonable time and they try to do their assigned tasks as good as they can.

Bibliography

AiGameDev.com (Champandard, Alex J.)
»Behavior Tree Starter Kit«
https://github.com/aigamedev/btsk
»Understanding the Second-Generation of Behavior Trees - #AltDevConf«
http://aigamedev.com/insider/tutorial/second-generation-bt/

Millington, Ian and Funge, John
»Artificial Intelligence for Games, 2nd Edition«
Boca Raton: CRC Press, 2009.

The behavior trees are up and running in Cosmonautica and they are controlling the crew members. The utility functions have proven to be useful for situations where behavior trees would have become too big, for example the search for an activity.
Behavior trees have shown these advantages:

- The **flexibility** is very high as shown through the usage of behavior trees for the AI ships in space fights, which was not planned at the beginning.
- The **maintaining** of existing behavior trees is fast and easy even without a graphical modeling tool.
- The **code** of the BT nodes can easily be re-used as well as parts of the behavior trees.
- The **quality** of the behaviors is high enough to support everything we need.

Nothing is perfect. Behavior trees are no exception and showed these disadvantages:

- The **distribution of complexity**, especially between the behavior tree itself and its 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 and decreases 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.
- Splitting up similar things, like checking all activities and then choosing one, might be required to re-use nodes but causes **code duplication** and **inefficiency**.
- **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.

We gained valuable experiences through Cosmonautica. We learned that utility functions are great as action nodes where a normal node structure is not applicable. We also came to understand how powerful behavior trees are. On the other hand, we noticed that more and more advanced behaviors can also harm the game. Crew members 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. And we learned that 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. But it can also make the entire behavior tree harder to understand. However, this is still much better than just a single big state machine.

Philipp Erler

The Crew members' Behavior Tree

The crew members' behavior tree consists of several parts:

1. Update the morale and all needs if the crew member is alive.
2. Find the best activity for the current environment.
3. Stick with the current need if it is urgent or was urgent since the current activity started.
4. Update walking or do the activity if the crew member has arrived. Then start walking to the new activity if the current activity has changed.