

Title:

Perceived Challenge

Project theme:

Evaluating the appropriateness of behavior trees

Project period:

Spring Semester 2010,
February 1st to June 4th

Project group:

d614a

Participants:

Morten Justesen
Dan Leinir Turthra Jensen
Kim Jung Nissen

Supervisor:

Zeng Yifeng

Print run:

5

Pages:

118 (total with appendices 190)

Abstract:

The Game AI concept Behavior Tree (BT) is presented and formalized.

A method for evaluating BT diversity and challenge is devised. The BT system SMARTS[16] is revised for performance and stability, and expanded by adding a general perception system. The SMARTS integration into the Gluon game engine is completed. A game, The Quantum Sea, is constructed to allow evaluation of these.

Evaluation shows that the measurements are useful, and indicates that the revised SMARTS system with scheduler and perception system functions better than the previous version.

The Qt-based Gluon framework is shown to be useful, if currently lacking in certain features.

Appendices (number, type):

1 CD-ROM with source and binaries

The contents of the report are freely available, however publishing (with source) must happen only by agreement with the authors.

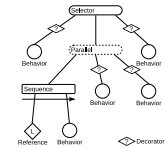
The source code of the Gluon SMARTS system is subject to the terms of the GNU Public License, version 3 available from <http://www.gnu.org/licenses/gpl.html>.

The Quantum Sea is available under the Attribution-Noncommercial-Share Alike 3.0 version of the Creative Commons license, available from <http://creativecommons.org/licenses/by-nc-sa/3.0/>.

Behavior Trees are a concept used in game AI, but which until very recently was disregarded in academia - presumably not out of spite, but rather because there simply was no ground work done on the topic, such as a formalization which would allow for calculations to be performed on the structure.

To assist with this, we set out to explore what measures might be useful in measuring the appropriateness of a behavior tree. To arrive at this, we need to perform a formalization, create a software toolchain to support the calculations in a real-world example, construct a test bed game based on an actual concept in which those measurements can be investigated, and finally the evaluation of the various methods described throughout the report.

Thus we start the report by describing behavior trees as a tree structure containing logic, and describe both the look and the meaning of each of the six possible building blocks which make up a behavior tree: Behaviors, Links, Sequences, Selectors, Parallels and Decorators.



With the description complete, a more strict formalization can be conducted, in which is described not only the language structure and the syntax, but also the execution of the tree. Before describing the measurements, methods for optimizing behavior trees are described. These allow game designers to construct their trees however they want, and then simply trust that the tree executes rapidly and correctly.

Once this has been completed, the measurements are constructed which allow for the evaluation of the appropriateness of any behavior tree, when viewed at a higher level. Based on work done by Yannakakis and Hallam[28], measurements are defined which describe the appropriateness through the challenge a tree poses to a player and the diversity of the behavior it represents. Two methods are devised, based on local and global paths.

To allow for this work to be conducted, the SMARTS framework developed through the course of the project Where Game AI Meets Academic AI[16] is revisited, and revisions performed on it:



The method of execution is changed from a recursive function calling in which each node executes its own children and replaced by a system in which a scheduler ensures the timely execution of behaviors. This system allows for behavior trees to be run without needing a threading system for parallels, which turned out to be a large problem previously when running tests at high speed.

Further, the SMARTS system is expanded to include a general perception system, which uses the character concept from the SMARTS behavior tree system to view the world in an adjusted manner. This system closes the missing gap in the SMARTS system, which with this addition becomes a complete game AI system, as defined by Daniel Sanchez-Crespo Dalmau[13].

Finally, the work to create an implementation into a specific game engine, where the Qt-based Gluon game creation and distribution framework is chosen as an extreme example (due to its nature of forcing scripting based implementations) is continued and completed. In the previous project this work was proved to be possible, and in this project that work is completed to the extent that the integration becomes usable for the creation of actual games.



To be able to perform tests on a real world example, the game The Quantum Sea is constructed, based on a premise of colliding particles in the sub-atomic world and gathering points on the basis of these collisions. This game, though simple in concept, turns out to be very difficult to master and thus fits the basic concept of what a casual game should be.



As such, it is a good candidate for performing an evaluation of a behavior tree which controls the game in place of the player. This hands-off approach to creating a behavior tree allows for the minimizing of variables during construction, which while in a scenario in a true real world scenario would be less advantageous, is good for us due to the nature of this report: We wish to evaluate the methods themselves, rather than truly create a player for The Quantum Sea.

The evaluation chapter contains descriptions of three evaluations: Evaluating the measurements, evaluating the tree construction method from the previous report[16] and finally the perception system.



During the evaluation of the measurements, both the global and local method is discovered to be applicable in varying situations. This encourages the suggestion that both methods could be used in conjunction, and suggestions for each method to be used in the situations where they make sense. The design method is found to be both thorough and intuitive, and the perception system is found to be, while cumbersome in its current incarnation, technically usable.

Through the course of this project, our adviser Zeng Yifeng has been very engaged and very engaging, and has been very active in helping us achieve our goals. We would thus like to extend special thanks to him.

We would also like to thank the Oxygen team (<http://oxygen-icons.org/>) for their kind contribution of some of the images used for the Part pages, and Eugene "it-s" Trounev for his design of the logos for both SMARTS and The Quantum Sea.

A CD-ROM is attached to this report which contains the source of the implemented software. Should you be viewing this report without that CD-ROM, the contents of that disc along with updated errata and other information can further be found on the project website at <http://leinir.dk/perceived-challenge/> and at the following pages on gitorious:

SMARTS <http://gitorious.net/gluon-bt/>

Gluon <http://gitorious.net/gluon/>

Full homepage: <http://gluon.gamingfreedom.org/>

The Quantum Sea <http://gitorious.net/gluon-bt/thequantumsea/>

Kim Jung Nissen

Student of Science in Computer Science

with Specialisation in Game and Engine Programming (stud.sc)

Dan Leinir Turthra Jensen

Student of Science in Information Technology, Software Development

with Specialisation in Game and Engine Programming (stud.it)

Morten Justesen

Student of Science in Engineering (Software) (stud.polyt)

1. Introduction	1
1.1. Purpose of This Project	1
I Formalizing Behavior Trees	3
2. Introduction to Behavior Trees	5
2.1. Building Blocks	5
2.1.1. Behaviors	6
2.1.2. Links	6
2.1.3. Sequences	6
2.1.4. Selectors	7
2.1.5. Parallels	7
2.1.6. Decorators	7
2.2. A Small Example	8
2.3. Conclusion	8
3. Behavior Trees Formalized	9
3.1. Formal Definition of Behavior Tree	9
3.1.1. Behavior Tree Syntax	10
3.1.2. Behavior Tree Execution	11
3.2. Conclusion	12
4. Evaluating Interesting Behavior Trees	13
4.1. Measuring Behavior Trees	13
4.1.1. Behavior Diversity	13
4.1.2. Diversity of Sequences	17
4.1.3. Challenge Level	19
4.1.4. Target Specific Challenge Level	22
4.1.5. Challenge Level of Sequences	23
4.2. Optimization of Behavior Tree Structure	23
4.2.1. Sequence	23
4.2.2. Probabilistic Selector	23
4.2.3. Random Selector	24
4.2.4. Prioritized Selector	25
4.2.5. Parallel	25
4.2.6. Conclusion	25
4.3. Optimization of Interest Value	26
4.3.1. Three Variable Interest Optimization	26
4.3.2. Optimal Weight Generalization	29
4.3.3. Optimizing Entire Behavior Trees	31
4.4. Conclusion	34

II	SMARTS Revisited	35
5.	SMARTS Library	37
5.1.	Analysis	37
5.1.1.	Tree structure	37
5.1.2.	Parallels and Threading	38
5.2.	Re-Design	38
5.2.1.	Over-all Design	38
5.2.2.	btNode	39
5.2.3.	btCharacter	40
5.2.4.	Scheduler	40
5.3.	Implementation Details	43
5.3.1.	Queues	43
5.3.2.	Execution	44
5.3.3.	btProbSelectors	47
5.3.4.	btParallels	47
5.4.	Conclusion	49
6.	Perception	51
6.1.	Inspiration	51
6.1.1.	Pac-Man™	52
6.1.2.	Quake™	52
6.1.3.	Thief: The Dark Project™	52
6.1.4.	Age of Empires™ and Warcraft™	53
6.2.	Design	54
6.2.1.	Setting Up the Criteria	54
6.2.2.	Library Design	55
6.3.	Implementation Details	56
6.3.1.	btPerception	57
6.3.2.	btPerceptionViewcone	57
6.3.3.	btPerceptionAtoms and btPerceptionInfo	58
6.4.	Conclusion	59
7.	SMARTS in Gluon	61
7.1.	Analysis	61
7.1.1.	Sound in Gluon	61
7.1.2.	The SMARTS Layout	62
7.2.	Design	62
7.2.1.	Scriptability	63
7.3.	Implementation Details	63
7.3.1.	Registration with Gluon	63
7.3.2.	Behavior Trees in Gluon	65
7.3.3.	Behavior Tree Execution in Gluon	67
7.3.4.	Perception in Gluon	69
7.4.	Conclusion	70
III	The Quantum Sea	71

8. Introducing The Quantum Sea	73
8.1. Game Design	73
8.2. A Small Level	74
8.2.1. Level Design	74
8.3. Conclusion	75
9. Particle Perception	77
9.1. Particles	77
9.1.1. Defining the Particles	77
9.2. Player	80
9.2.1. High Level Behavior	81
9.2.2. Possible Activities	81
9.2.3. Low-level Actions	81
9.2.4. Division Into Behaviors	82
9.2.5. Goal Identification and Ordering	85
9.3. Conclusion	87
10. Quantum Construction	89
10.1 Level Building	89
10.1.1 The Anatomy of a Particle	89
10.1.2 Putting Together Levels	90
10.2 Behavior Trees	91
10.2.1 Working with SMARTS Designer	91
10.2.2 Creating the Actions	93
10.3 Conclusion	94
IV Evaluation	95
11. Evaluation	97
11.1 Testing Procedure	97
11.1.1 The Quantum Sea	97
11.1.2 Constructed Example	100
11.2 Evaluation of the Methods	101
11.2.1 Local Method	101
11.2.2 Global Method	102
11.2.3 Constructed Example	103
11.3 Comparison of the Methods	104
11.4 Evaluation of Behavior Tree Design Process	104
11.5 Evaluation of Perception System	105
11.6 Conclusion	106
V Concluding Matters	107
12. Conclusion	109
13. Discussion and Future Work	113
13.1 The Quantum Sea	113

13.2	Metrics and Methods	113
13.3	Extensions to SMARTS Behavior Trees	114
13.4	Adjustments to SMARTS	114
13.4.1	Perception System	115
13.4.2	Measurements	115
VI	Appendix	119
A.	Introducing Gluon	121
A.1.	The Gluon Packages	121
A.2.	Gluon Engine	123
A.3.	Gluon Creator	124
A.4.	Gaming Freedom	125
A.5.	Gluon Player	127
B.	Design for The Quantum Sea	129
C.	Game Item Definitions	135
C.1.	Non-collidable Items	135
C.2.	Particle Definition Sheets	135
C.2.1.	Isotopes	135
C.2.2.	Particles	138
C.2.3.	Sub-particles	140
D.	Test Results	141
D.1.	The Quantum Sea Results	141
D.1.1.	Local Method	141
D.1.2.	Global Method	159
D.2.	Constructed Example	188
D.2.1.	Local Method	188
D.2.2.	Global Method	189

Introduction

Through the years, game AI has been neglected by academia, not out of malice, but because it has a different goal than the AI techniques usually applied in academia: When academics speak of artificial intelligence, or machine intelligence, the thoughts are that AI should be designed to search for the optimal solution to a problem. When Kasparov played against Deeper Blue in 1997 and lost¹, this is the kind of AI he worked against.

However, one problem with such AI is that they are designed specifically to win. This soon becomes a problem when you look at what the word "game" means. Roger Caillois describes this as an activity which must be fun, separate, uncertain, non-productive, governed by rules and fictitious. While this seems self-evident, in particular the first item here often sits at odds with the goals of classic machine intelligence; a game which you play, but in which your artificial opponent always wins, simply is not fun.

So, game AI instead centers around the goal of making a game as enjoyable as possible. Towards this goal, classic AI will sometimes do fine, but most often it is more the case that it works against the will of the game designer. Game AI thus aims to make it possible for game designers to construct game characters which are believable, and act in accordance with the game world's premise.

Behavior Trees as defined by Alex J. Champandard[9] aim at dealing with some of the problems found in techniques used in the industry to solve the game designer's problems, such as hierarchical finite state machines, scripting or planners. In the report *Where Game AI Meets Academic AI*[16] the first steps towards pulling Behavior Trees into the realm of academia were made, and this report is a continuation of that effort.

1.1. Purpose of This Project

Based on the work done in *Where Game AI Meets Academic AI*[16], we will start by formalizing behavior trees, and constructing a method for evaluating their appropriateness. Also based on the tests of metrics used for defining challenge and behavioral diversity in *Where Game AI Meets Academic AI*[16], we will define two new measures for these two metrics. We will also present a way of calculating which weights and probabilities in a behavior tree will give the most interesting game experience.

Refactoring of the SMARTS Library presented in the report mentioned above is necessary to accommodate some of the design and implementations issues in the first version, including both basic design issues as well as features lacking in the

¹<http://www.research.ibm.com/deepblue/>

system itself, which will allow the testing of methods on behavior trees in a more straight forward manner than was possible previously.

We thus expand the library with new functionality in the form of a perception system, to enable the creation of a wider variation of game AI which perceive the world in a more sensible way than simply reading the information directly out of the game world.

Furthermore, we take the preliminary integration of the SMARTS Library into Gluon through its plugin system created in the same report, and expand this into a fully functioning implementation in the spirit of Gluon.

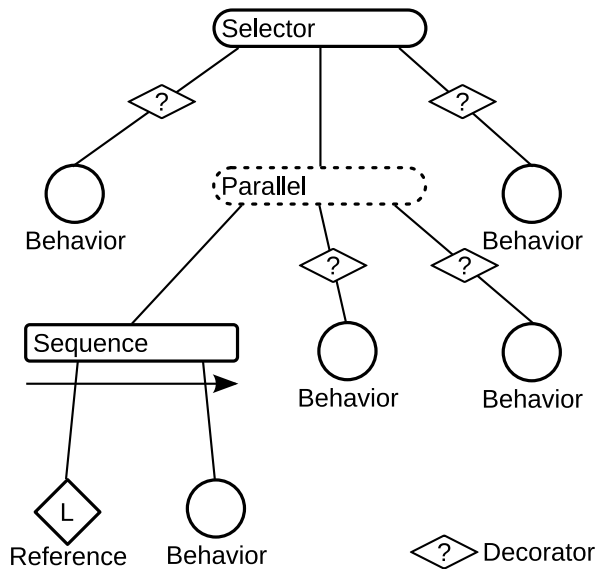
For testing the implementation together with the challenge and behavioral diversity measures, we will create a game using Gluon, and run our tests through that game. We then evaluate the tests and results to determine if the measures are adequate for creating an interesting game. We further use the test bed application from Where Game AI Meets Academic AI[16] to compare the new measurements with those employed in the previous report.

In summary, this project will consist of work based on the following items:

- Formulate a method for evaluating the appropriateness of behavior trees
 - Formalize behavior trees
 - Define measures for challenge and behavior diversity
- Revisit the SMARTS Library
 - Analyze the SMARTS Library.
 - Redesign with the power of hind-sight
- Integrate the SMARTS Library into Gluon through plugins
- Create a game using Gluon based on the design presented in January
- Evaluate the measures for challenge and behavior diversity

Part I

Formalizing Behavior Trees



Behavior Trees have their origin in the game industry, and in this part we perform a formalization and define measurements for behavior diversity, challenge level and interest.

Introduction to Behavior Trees

In this section we will describe behavior by explaining each part of them in some detail. This is done to demonstrate how the different components work together, and what kind of behaviors can be built with behavior trees.

We describe the individual components that make up a behavior tree and then put these into context of a small example, which is only described superficially. For a further exploration of this particular behavior, please see Section 9.2 on page 80. It should be noted that this example is only one way to create this behavior, and that there are other ways to accomplish the same thing.

Figure 2.1 shows this real-world example from the game *The Quantum Sea* of what a behavior tree looks like, and the following section shows the symbolic relevance of the various parts of the tree.

2.1. Building Blocks

The following sections are adapted from *Where Academic AI Meets Game AI*[16]. There are six different types of components available when creating behavior trees. This section will cover the use and functionality of these components, as well as show the symbols used in diagrams when designing behavior trees.

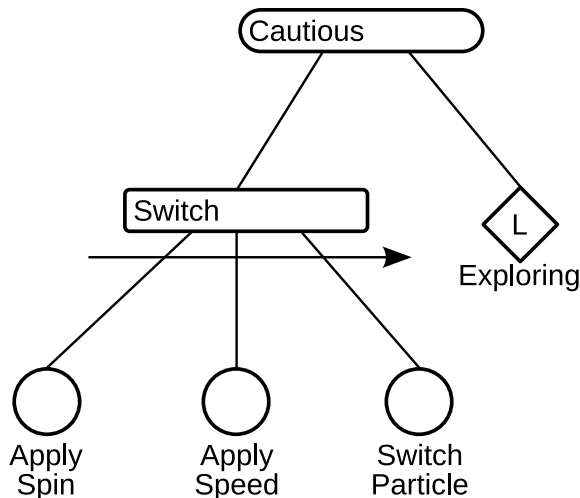


Figure 2.1.: A sample behavior tree from the game *The Quantum Sea*

2.1.1. Behaviors

Behaviors in a behavior tree can contain actions which the game AI has to perform or be subtree of the behavior tree. These actions are provided by the game programmer as black boxes, which allows the AI designer to use them when constructing behavior trees.



Furthermore each behavior will end execution with a termination status which could be successfully completed execution, execution failed cleanly or some unexpected error occurred.

As seen in the image, a behavior is symbolized by a circle, with the name of the behavior underneath. Optionally, in the case of single-letter names, the name can be written in the center of the circle in stead of underneath. In this case, a comment can be written underneath which does not influence the semantics of the tree (in other words: meta information).

2.1.2. Links

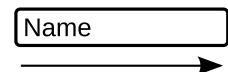
A link node is a specialized type of behavior which references another behavior tree. This is what allows for the abstraction and consequent re-use of behaviors. A function in programming languages can be called from many places, returning the control flow to where they were called. A link node functions in the same way, in that many link nodes can call the same behavior tree, which then only knows its children, and not the caller. This allows for the modular building of behavior trees.



Links are symbolized by a diamond with the letter L inside. The name of the behavior tree which is pointed to is written underneath.

2.1.3. Sequences

As the name indicates, a sequence is a sequence of behaviors executing in a specified order, and is thus a composite. The sequence implicitly defines dependencies between the behaviors executed, from the execution order. A behavior executes, and maybe changes the value of a number of variables, which are later used by another behavior.



A sequence will react differently depending on the termination statuses received from its children. So if a child fails it will bail out and when a child succeeds it will continue to execute the next child behavior until the sequence is done. When the last child has succeeded, the sequence will return with a success.

The sequence composite is symbolised by a square box with the name of the node written inside, and an arrow underneath it. This arrow should stretch over the lines pointing to all the children of the sequence, as seen on the Switch sequence in Figure 2.1 on the preceding page.

2.1.4. Selectors

While sequences only execute behaviors in a given order, the selector composite is used for selecting a given behavior for execution. It is possible to use any kind of AI algorithm to take care of the decision making of the selector. Thereby the selection of a behavior could be done using e.g. probability, priority or complete non-determinism. As a sequence, the selector has to handle the termination statuses from its children, and though the reactions resembles those of a sequence there are some vital differences.



When a child fails the selector will try to select another child as long as there are new ones. The first child to succeed will cause the selector to return with a success. This is the logical opposite of a sequence.

As an example of a variant of the selector we take a probability selector. This type uses probabilities on the children for choosing the next child for execution. Besides the probabilities this type of selector behaves as an ordinary selector which only run each child once and fails if all children has failed.

A sequence is symbolized by a pill shape (that is, a rectangle where the rounding is complete at both ends), with the name of the node written inside it.

2.1.5. Parallels

With the components described so far it is possible to create a behavior tree. However, the execution of the tree will be very linear, and not supporting any form of concurrency. This is where the composite Parallel come into the picture.



Parallels are capable of running all of their children simultaneously, and are responsibly for all of their child behaviors, insuring they terminate cleanly when the parallel itself terminates. There are different ways of deciding when the parallel has to terminate, depending on the context in which it is used; e.g. is it possible to specify which and how many behaviors should succeed to count as a success or how many are allowed to fail before it becomes a failure.

A parallel is symbolized by a pill shape whose border is dotted rather than solid as seen in the selector.

2.1.6. Decorators

Behaviors, links, selectors, sequences and parallels are the corner stones when creating a behavior tree. At some point, however, the need will arise for adding features or extending a behavior or a subtree, without changing the behavior tree drastically or modifying the implementation of a behavior. It is here decorators come into the picture.



Decorators can be inserted almost everywhere in a behavior tree but with one restriction: they cannot be leaf nodes. A decorator can extend any behavior with extra functionality without the behavior having any knowledge about the decorator. Furthermore a decorator cannot branch, as it only has one child behavior.

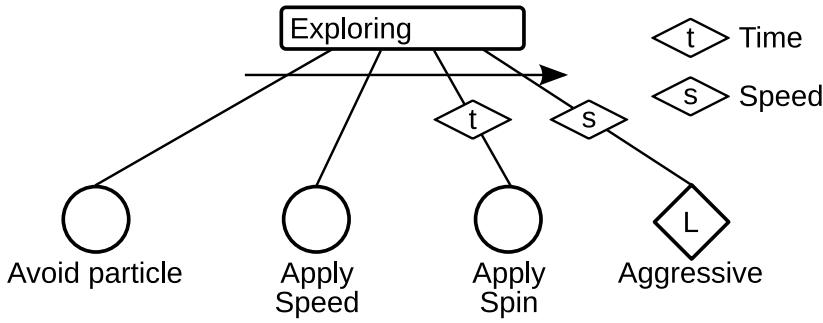


Figure 2.2.: Another sample behavior tree from the game The Quantum Sea

A decorator is symbolized by a flattened diamond with a single letter inside, indicating which decorator is being used. They are put on the connecting line going to a behavior, as seen in Figure 2.2. It is further suggested, but not required, that the full name of the decorator be listed on the diagram somewhere, also in the style shown on the figure.

2.2. A Small Example

Figures 2.1 and 2.2 show most of the various possibilities of behavior trees, and Figure 9.7 on page 86 shows the only item not used in those two (that is, a parallel composite node). In this section we sketch out what would happen in the first of these, named Cautious, to give an idea of how behavior trees function:

The selector Cautious will first attempt to run the Switch sequence, which in turn will run each of its children in order. If either of the behaviors Apply Spin, Apply Speed or Switch Particle fails, Cautious will attempt to execute the next behavior, the link to the Exploring behavior tree. However, if all three succeed, the sequence will return true, and the selector will have succeeded and not try to execute the linked behavior.

2.3. Conclusion

In this chapter we went through a recap of what behavior trees are and which kinds of building blocks can be used to create a behavior tree, as well as a short description of how one should interpret the syntax and control flow of a behavior tree.

Behavior Trees Formalized

Behavior trees are practical structures used to design behaviors in games.[7][8] Behavior Systems that resemble behavior trees have been used in many different areas such as modeling behavior of intelligent agents.[5] In fact there are several behavior systems that are not quite behavior trees but are very similar in structure.[2]

In order to evaluate what a good behavior tree is and potentially how to optimize them, we need a formal way of describing Behavior Trees.

Behavior Trees can be seen as a form of programming language designed specifically for behaviors. Lets look at the three main parts of a programming language and how they work in a behavior tree:

Any programming language need to support three different elements: elementary building blocks, means of combination and means of abstraction.[1]

The elementary building blocks of behavior trees is the action and the condition, an action does something (and possibly checks something) while a condition solely checks some condition.

The second element is means of combination and the basic behavior tree have two (extended versions with parallels can be made) called sequences and selectors, these allow combination of behavior trees which include everything, note that these means of combinations are closed in the sense that all elements that can be combined are behavior trees and all combinations are also behavior trees. This property is what allow us to build complex behavior trees.

Then there are means of abstraction, this is something not always described in behavior tree literature[9], but we support it through behavior tree references. This allow the designer to used whole behavior trees as simple actions without much thought on how the are implemented. Abstraction is also an integral part of behavior trees as a whole as actions are really abstractions over arbitrary code.

As with any programming language we can specify the syntax (the look) and the semantics (the meaning) of behavior trees.

3.1. Formal Definition of Behavior Tree

We use a recursive definition to describe the structure of a behavior tree. We use the definition of the individual nodes as defined in the previous chapter. A behavior tree is a rooted tree (connected acyclic graph).

A behavior tree is a tree of behavior nodes, these nodes can have types such as action, condition or composite.

A **Behavior Tree** can be any of the following:

1. A **simple action** (action or condition), which is a piece of arbitrary code.

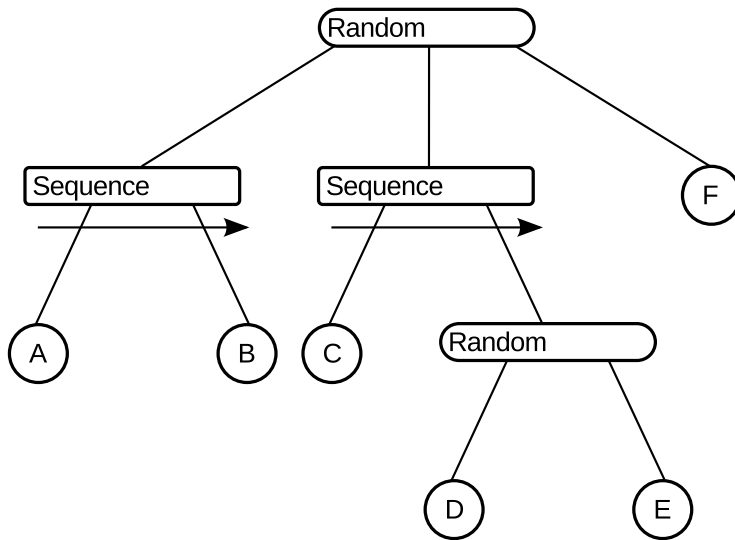


Figure 3.1.: Structure of a Behavior Tree

2. or A **composite** (either selector or sequence) with one or more behavior trees as children.
3. or A **decorator** with exactly one behavior tree as a child.

This defines a tree with composites as internal nodes and simple actions as leaf nodes. More generally, a behavior tree represents a decomposition of behavior in individual subtasks with each their own goal. And any behavior tree can be used as a sub behavior of a larger behavior tree. [12][9]

The original inspiration for behavior trees is Hierarchical Finite State Machines [15]. But behavior trees introduces the limitation that the hierarchy have to be a tree. The states in a behavior tree is no longer real states as they cannot change to arbitrary other states[9].

3.1.1. Behavior Tree Syntax

The syntax for behavior trees is graphical and not textual as traditional programming languages. It is build around a tree structure. Even though behavior trees are graphical structures it is not necessary to create behavior trees graphically (we have used tree views) as they could just as well be designed in a traditional written language such as in ABL[21][22] which otherwise is similar to behavior trees. The graphical nature is for the benefit of the behavior designers such that they do not have to be programmers to create behaviors.

3.1.2. Behavior Tree Execution

We use simple pseudo code to define the semantics of the execution of a behavior tree. This is a very simplified view on what a behavior tree does, with details cut for brevity.

```
1 Simple Action A    =>  status = exec code(A); return status
```

```
1 Sequence S        =>
2   for A in actions(S):
3       if A is simple action:
4           status = exec code(A)
5       else:
6           status = exec behavior(A)
7       if status is failure:
8           return failure
9   return success
```

```
1 Selector S, nextaction(S)  =>
2   for A in nextaction(S):
3       if A is simple action:
4           status = exec code(A)
5       else:
6           status = exec behavior(A)
7       if status is success:
8           return success
9   return failure
```

```
1 Decorator D        =>
2   status = exec code(D)
3   if status is failure:
4       return failure
5
6   if action(D) as A is simple action:
7       exec code(A)
8   else:
9       exec behavior(A)
10
11   status = exec code(D)
12   return status
```

The procedure $\text{nextaction}(S) \Rightarrow A$ gives the next action to execute until all have been executed. Typically prioritized or probabilistic order.

This semantics corresponds to the implementation we had in our old testbed[16]. This is a relevant definition as it represent what behavior we want from the behavior tree, but not necessary the best way to implement it in practice, chapter 5 on page 37 handles this subject.

Execution with Scheduler

When using a scheduler the operational semantics change, as the execution behaviors and child behaviors is not done in each behavior but rather in the scheduler. This basically means that a behavior returns an index on a child behavior to the scheduler, which in turn in the next call executes this child. When a child finishes execution the status is then returned to its parent, such that it can process the information and continue its execution if necessary.

3.2. Conclusion

Through this chapter has been conducted a formalization of behavior trees: Firstly as a structure in which a behavior tree is described as a tree of composites, decorators and simple actions, secondly as a graphical language, then finally the semantics for the execution of behavior trees are described.

Evaluating Interesting Behavior Trees

What is a good behavior tree? This is the question we want to come with an answer to. Lets start with a definition of what we want a good behavior tree to be:

Definition 1: Good Behavior Tree

A **good behavior tree** is one that when executed produces a behavior which the player finds interesting to play against.

Note that we limit our definition to behaviors of opponents to players, not allied or neutral entities. This is a fair limitation as most behaviors used in games are for opponents, and that measuring qualities such team-cooperation are outside the scope of this work.

The reason for this definition is that it is important to be able to define what is interesting for the human player. It is also a way of measuring the entertainment value of an AI opponent[28].

From this definition we need to find out what classifies an interest behavior and how we can relate an interesting behavior with the behavior tree that produced it. That is, how do we measure interesting behavior directly on the behavior tree.

Metrics have been developed for measuring interesting behavior by defined on traces of behavior in a running game[28]. We believe that we can do the same directly on the behavior tree structure. If the behavior is interesting the behavior tree is good.

4.1. Measuring Behavior Trees

We want to define how interesting the performance of an agent is by combining the behavioral diversity and the challenge level.

A function is needed which gives us the maximum interest based on challenge and diversity. This function takes a behavior tree, success rates of actions and weights for actions; and returns an interest value.

We want to find the weights that maximizes this function. (The best compromise between challenge and diversity).

4.1.1. Behavior Diversity

Let us define behavior diversity before talking about measuring and calculation of behavior diversity.

Definition 2: Behavior Diversity

Behavior diversity defines how often child nodes are selected compared to the number times a selector makes a selection, the more evenly the distribution the higher the behavior diversity.

Consider the two extreme cases: The case where the diversity is the highest possible and the case where it is lowest. The most diverse behavior is when it tries all possible sub-behaviors with the same probability, i.e. they all have the same weight. The least diverse behavior is when it always perform the same action. This assumes that the behaviors are either simple actions as defined in Section 3.1 on page 9 or all have the same internal diversity.

Definition 3: Canonical Behavior Tree

A **Canonical Behavior Tree** is a behavior tree only consisting of a single selector with only simple actions as children.

It is easy to define the behavioral diversity in terms of a behavior tree. If we consider the case of a single selector with primitive actions (no other composites) the most diverse behavior is when the weight of all actions are the same. The least diverse behavior is when one action is 1.0 and the rest is 0.0 allowing only one action to be performed.

We want a function that satisfies this such that it produces the highest value possible when weights are even and the lowest possible value when the one action is always selected.

Definition 4: Configuration

A **configuration** is a specific list of values for the weights of the behaviors of a selector.

We also want it to satisfy that if we take some configuration and make the weights more even the diversity should go up.

Because the most diverse configuration is the one with even weights, a simple approach to measuring diversity of a configuration is to calculate the distance between the configuration and the optimal one. It is then easy to define using euclidean distance:

$$BD = \sqrt{\sum_{i=0}^n \left(\frac{1}{n} - w_i\right)^2} \quad (4.1)$$

This satisfies our conditions (if we minimize or negate) as the distance is from the optimal one produces a value of 0.0 and the least produces the highest possible distance.

Another approach is to measure the variability of a set of variables, which is calculated as the sum of the squares of the difference from the estimated mean value:[29]

$$\text{Sum of Squares} = \sum_{i=0}^n (X_i - \bar{X})^2 \quad (4.2)$$

Our mean can easily be calculated as $1/N$ where N is the number of weights. Using this mean a measure for behavioral diversity can be written as:

$$BD = \sum_{i=0}^n \left(\frac{1}{n} - w_i\right)^2 \quad (4.3)$$

This turns out to just be another simpler way to look at the same geometric problem.

Another better way to define behavioral diversity is to define it as the amount of uncertainty about a particular configuration. We can also see it as the amount of disorder in the configuration. This measure is called entropy[23] and have been used for measuring diversity before[28].

We can map entropy to diversity such that the least diverse configuration have entropy 0 and rises as the diversity goes up. The entropy of a configuration is defined as:

$$\text{Shannon Entropy} = H(w_1, \dots, w_n) = -k \sum_{i=1}^n w_i \ln(w_i) \quad (4.4)$$

This entropy value is dependent on the number of weights in the configuration, but it is possible to normalize to a value between 0.0 and 1.0 such that the configuration with highest diversity gets 1.0 and the lowest get 0.0. We could do this by scaling it with the maximum entropy for a given number of weights which turns out to be just $\ln n$ where n is the number of behaviors. The other measures can be similarly scaled by dividing with the maximum distance.

We choose Shannon Entropy as the measure for diversity because it produces nicer results when used together with our challenge measure as we shall see later. What makes entropy such a good measure, is that it has a number of properties that are appropriate for diversity as described by Balch [4, 212–213]. In particular properties 4 to 6 describes exactly the properties we want for measuring diversity in behavior trees as described earlier. Other metrics such as Euclidean Distance does not cover the properties 1 to 3 (it is not continuous and does not define recursion) which are important when we want to measure diversity of hierarchies, which is a typical usage in trees.

This leads us to the following definition of behavioral diversity based on Shannon Entropy, dropping the negation and the constant as the function have the qualities we want without them.

$$\text{Behavioral Diversity} = BD(w_1, \dots, w_n) = \sum_{i=1}^n w_i \ln(w_i) \quad (4.5)$$

It is also simple to expand diversity to hierarchies of selectors by noticing that having the same weights per path to the leaf level gives the highest behavioral diversity. If one path has weight 1.0 and the rest have 0.0 this will have the least diverse behavior, as expected.

Using entropy also provides a way to deal with hierarchical diversity. This is done by utilizing Hierarchical Social Entropy[4], where the diversity of sub-groups are calculated individually and combined using a weight for the contribution for each group plus the diversity of the values of the contributions themselves.

$$BD(N_1, \dots, N_n) = \sum_{i=1}^n BD(\text{children}(N_i)) + H(\text{weight}(N_1), \dots, \text{weight}(N_n)) \quad (4.6)$$

This is convenient because we will not have to calculate the paths directly, as we can calculate the diversity of parts of the tree separately.

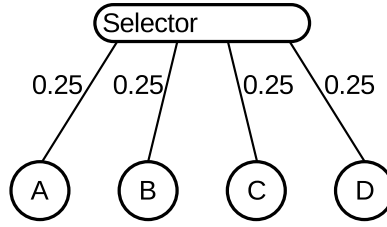


Figure 4.1.: A diverse behavior tree

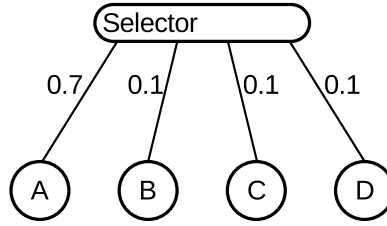


Figure 4.2.: A less diverse behavior tree

This does not handle the case where one or more weights w_i are zero as $\ln(0) = -\infty$, so we have to make sure all weights are positive. We can ensure this by adding a small constant to the weights before calculating the entropy or by ignoring behaviors with weight 0 completely.

Example 4.1.1.1 Measure of Behavioral Diversity

To show how entropy can be used for calculating the behavior diversity, two examples are used and the diversity compared.

$$\begin{aligned} \text{Diversity} &= - \sum_{i=0} w_i \ln w_i \\ &= -(0.25 \cdot \ln 0.25 + 0.25 \cdot \ln 0.25 + \\ &\quad 0.25 \cdot \ln 0.25 + 0.25 \cdot \ln 0.25) \\ &= 1.38 \end{aligned}$$

Figure 4.1 shows a selector with even weights on the child nodes. Then we use the entropy to calculate the diversity of the behavior tree. From these calculations we can see that the diversity of that particular tree is 1.38.

$$\begin{aligned} \text{Diversity} &= - \sum_{i=0} w_i \ln w_i \\ &= -(0.7 \cdot \ln 0.7 + 0.1 \cdot \ln 0.1 + \\ &\quad 0.1 \cdot \ln 0.1 + 0.1 \cdot \ln 0.1) \\ &= 0.9 \end{aligned}$$

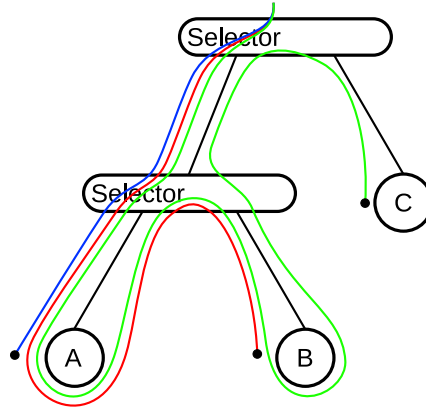


Figure 4.3.: Three possible paths through a hierarchy of selectors

Figure 4.2 on the facing page shows another selector with a different set of weights than Figure 4.1 on the preceding page, and again we apply the entropy calculations on this tree. The diversity is calculated to 0.9.

As we would expect the diversity of the Figure 4.1 on the facing page is greater than that of Figure 4.2 on the preceding page, which also can be seen on the distributed weights for the selectors.

We need to optimize the entropy of the weights of the paths of the behavior tree to get the most diverse behavior.

Two trees with the same diversity per path can have slightly different behavior. This is because a selector creates a relation or partial ordering between its children (actions).

Example 4.1.1.2 Execution Order

If we consider a selector with another selector and a simple action C as children and the child-selector has two simple actions A and B as children. Then if the child-selector is selected for execution first and S is executed first. If S now fails, the next action to be selected is B . This creates a partial ordering, because if the simple actions had been part of just one single selector, then C would be considered after A had failed. See Figure 4.3 for a visual description of this.

This indicates that using entropy on paths might not be enough to measure behavioral diversity as partial orderings create less diverse behavior. We want to find out if this can be ignored when optimizing behavior trees. Calculating diversity as Hierarchical Social Entropy does not differentiate these cases.

4.1.2. Diversity of Sequences

The behavioral diversity of a selector can be calculated as the entropy in a system, for simplicity we minimize the entropy value for maximum diversity allowing us to drop the negation of entropy.

$$Diversity = A \cdot \ln(A) + B \cdot \ln(B) + C \cdot \ln(C) + D \cdot \ln(D) \quad (4.7)$$

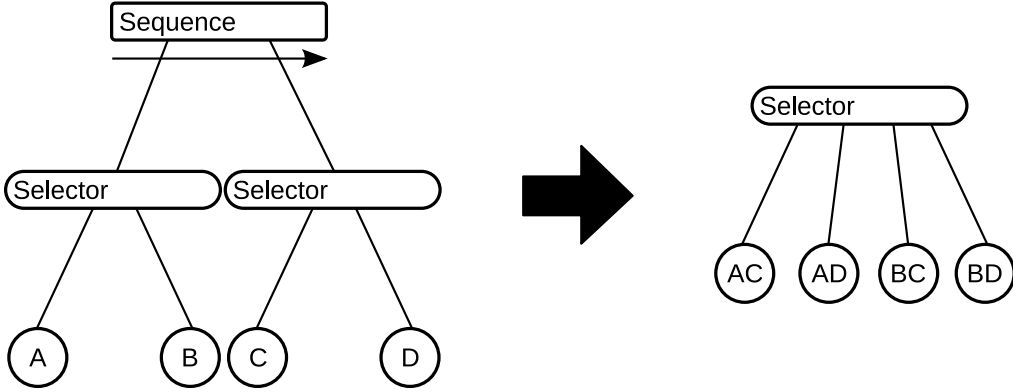


Figure 4.4.: A Sequence of Selectors converted to a single Selector

But this does not let us calculate the diversity of a sequence directly. The intuition behind entropy of a sequence is that a sequence in itself does not contribute to the diversity of a behavior tree. A sequence of simple actions is exactly as diverse as a simple action alone, from an entropy point of view. In other words: A selector does not produce uncertainty about its behavior on its own, as it always runs the same sub-behaviors in the same order.

But if the sequence have selectors as children, then they will contribute to diversity, and the sequence aggregates the diversity in some way. If we assume that the event that an action in a sequence fail is rare, then the diversity is simply a selection between possible combinations actions from the first selector and the second, as illustrated in Figure 4.4. The diversity of this can be written as:

$$Diversity = AC \cdot \ln(AC) + AD \cdot \ln(AD) + BC \cdot \ln(BC) + BD \cdot \ln(BD) \quad (4.8)$$

Rewriting the logarithms allow us to rewrite as such:

$$\begin{aligned} Diversity &= AC \cdot (\ln A + \ln C) \\ &+ AD \cdot (\ln A + \ln D) \\ &+ BC \cdot (\ln B + \ln C) \\ &+ BD \cdot (\ln B + \ln D) \end{aligned}$$

Then factoring out the logarithms yields an expression similar to the diversity for a selector.

$$\begin{aligned} Diversity &= \ln A \cdot (AC + AD) \\ &+ \ln B \cdot (BC + BD) \\ &+ \ln C \cdot (AC + BC) \\ &+ \ln D \cdot (AD + BD) \end{aligned}$$

And because we know that $A + B = 1, C + D = 1$ we can see that the the diversity calculation for a sequence reduces to the diversity for a selector. This mean that

we do not need to calculate all the different possibilities for behaviors, which is much more efficient.

This is a reasonable way of measuring the diversity of sequences as it can be seen as hierarchical entropy of a selector where the weights of all children are 1.0. This will simplify to simply summing over the entropy of the children, as the entropy of the sequence itself is $0.0 (\sum_{i=0}^N 1 \cdot \ln 1)$.

Also, if a simple action is added to a sequence the diversity value of the sequence does not change as there is no uncertainty about the execution and thus the entropy ($\ln 1 \cdot 1$) is zero.

4.1.3. Challenge Level

Challenge is how difficult the agent is to play against. We have defined that an agent using a (from the point of the agent) very successful action would give this action a high challenge level. An agent using a strategy with very low success rate is of lower challenge level, and this leads to the following definition:

Definition 5: Challenge Level

The higher the overall success rate of the agent, the higher the **challenge level**. That is the higher the probability that an agent selects a high success rate strategy, the higher the **challenge level**.

For our purposes challenge is a pure measure of how challenging an agent is. Other literature goes further and defines what a appropriate challenge is (not too hard, not too easy). But for now we want to know how hard a game is.[3][25][28]

The highest challenge level is when the agent always uses the optimal policy or strategy (let us not consider that it would be more challenging if a larger number of equally challenging strategies are used; optimal means that it will win in the vast majority of cases).

It terms of a behavior tree, take a tree with two actions with high success rates and one action with low success rate. Selecting the more successful actions more often will lead to more challenging behavior (this assumes the actions achieve the same goal). By adjusting the weights on these actions we can control how often they are run, thus how challenging the behavior is. We only consider selectors when defining and calculating challenge for now.

Challenge level is a contrast to behavioral diversity. The most diverse is when the weights are even, but the most challenging is when the most challenging behaviors have the highest weight. We believe that by finding the best compromise between challenge and diversity, we can create interesting behaviors. Challenge can be seen as the difficulty level of the agent. We can scale this difficulty by changing the ratio of challenge to diversity in calculating how interesting a behavior is.

We possess the knowledge of how successful an action has been, and on top of that the strategy which is the weights on each of the actions. To optimize the strategy to maximum challenge is thus simply to select the one with the highest success rate. But we want to support several actions with varying success rates, such that a strategy is selected according to success rates.

Measuring Challenge

To measure challenge we need a formulation that capture that a configuration that weights behaviors with high success rates should have a high challenge value. Intuitively it can be seen as a distance problem, where the challenge is the distance to the optimal configuration. For our measure of challenge we need a function that satisfies certain properties. First it should produce the highest value when the weights are the same as the success rates, giving the most successful behaviors the highest weight. Second it should produce the lowest value when the behavior with the lowest success rate is given a weight of 1.0. And third, any change in a configuration toward the success rates of the behaviors should yield a higher challenge.

This should intuitively be captured by the euclidean distance from the optimal to the particular configuration in question:

$$\text{Challenge} = \sqrt{\sum_{i=0}^n (w_i - p_i)^2} \quad (4.9)$$

But this have some problems such that when trying to optimize for a combination of challenge and diversity, the combined expression does not yield an exact analytical solution.

The idea of using a distance metric is not bad, we just need to use another distance metric that does not have these problems.

The measure we chose to use is KL-divergence[24][18] and is defined as such:

$$\text{KL-divergence} = \sum_{i=0}^n w_i \ln \frac{w_i}{p_i} \quad (4.10)$$

KL-divergence is also called relative entropy and is the difference between two probability distributions. It is not a true metric in that the distances $d(p_1, p_2)$ and $d(p_2, p_1)$ are not necessarily the same. Also to have the highest value represent the highest challenge, we have to negate KL-divergence.

Example 4.1.3.1 Measure of challenge

To show how KL-divergence can be used to measure the challenge level of a behavior tree we calculate the challenge level of two very different behavior trees.

Figure 4.5 on the facing page show a selector with even weights. This will produce the highest possible diversity for this example, but it is not the most challenging configuration possible. To calculate the challenge we not only consider the weights but also the success rate of each action. In this case each action is run with the same probability despite the fact that one action has a high success rate of 0.8 and the other actions have low success rates such as 0.05.

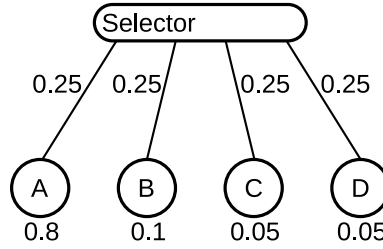


Figure 4.5.: A diverse but not very challenging behavior tree.

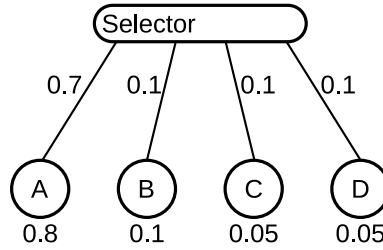


Figure 4.6.: A more challenging behavior tree

$$\begin{aligned}
 \text{Challenge} &= - \sum_{i=0}^n w_i \ln\left(\frac{w_i}{p_i}\right) \\
 &= 0.25 \cdot \ln \frac{0.25}{0.8} + 0.25 \cdot \ln \frac{0.25}{0.1} + \\
 &\quad 0.25 \cdot \ln \frac{0.25}{0.05} + 0.25 \cdot \ln \frac{0.25}{0.05} \\
 &= 0.743
 \end{aligned}$$

As expected, calculating the diversity produces a high value of 0.743 which means low challenge as the value should be as close to zero as possible indicating the most challenging case.

Figure 4.6 shows the same behavior tree but with different weights assigned. In this case the most successful action is assigned a higher weight than the other actions. This should by the definition of challenge give a more challenging behavior and thus a challenge value closer to zero.

$$\begin{aligned}
Challenge &= - \sum_{i=0}^n w_i \ln \frac{w_i}{p_i} \\
&= 0.7 \cdot \ln \frac{0.7}{0.8} + 0.1 \cdot \ln \frac{0.1}{0.1} + \\
&\quad 0.1 \cdot \ln \frac{0.1}{0.05} + 0.1 \cdot \ln \frac{0.1}{0.05} \\
&= 0.045
\end{aligned}$$

As seen in the calculation above, the found challenge value is indeed much closer to zero, as was to be expected.

This demonstrates by example that this measure is a reasonable measure of challenge of a behavior tree.

4.1.4. Target Specific Challenge Level

Instead of trying to optimize the challenge level, it is possible using the definition of challenge as used by Yannakakis[28] to target a specific success rate of the agent.

Example 4.1.4.1 Specific Challenge Level

As an example we can set the challenge level to be approximately 50 (in an 0 to 100 range instead of 0 to 1). A high value gives harder agents and lower values gives easier agents as by the definition of challenge.

Say we have the values (100, 100, 25) as success rates for strategies under a selector. We want to create weights w_1, w_2, w_3 such that $100w_1 + 100w_2 + 25w_3 = 50$. That is to say that we want the success rate over time to be approximate 50.

The weights should sum up to 1 that is $w_1 + w_2 + w_3 = 1$, we also want to make sure that if two values are the same they should have the same weight.

We can solve this exactly by seeing the problem as a geometric problem where we create a vector of the ratios of the values:

$$100/225, 100/225, 25/225 = (0.44, 0.44, 0.11) \quad (4.11)$$

and a vector of the opposite which is

$$(100 - 100), (100 - 100), (100 - 25) = (0, 0, 75), (0/75, 0/75, 75/75) = (0.0, 0.0, 1.0). \quad (4.12)$$

The vector we want is one the line from $v_1 = (0.44, 0.44, 0.11)$ to $v_2 = (0.0, 0.0, 1.0)$ we can create a new vector from the first to the second $v_1 v_2 = (-0.44, -0.44, 0.89)$. We can now describe the vector of weights we need as $v_1 + v_1 v_2 s$ (where s is a scale from 0 to 1). This allows us to create the equation $(0.44 - 0.44s)100 + (0.44 - 0.44s)100 + (0.11 + 0.89s)25 = 50$, the solution to which gives us the scaling factor s . Using this value in $v_1 + v_1 v_2 s$ gives the weights.

This approach, however, has several problems: First of all it only allows one to target a challenge level and not a diversity level. It also does not ensure that non-negative weights are produced. What we want is a general way to optimize both challenge and diversity.

4.1.5. Challenge Level of Sequences

We want to expand the concept of challenge and diversity beyond selectors to sequences of behaviors.

When viewed from above, a sequence is considered a simple atomic action. This action also have a success/fail rate which can be both measured by simulation or calculated by multiplying the individual sub actions, i.e. the chance of the first child succeeding times the chance the second child succeeds gives the chance they both succeed in sequence. This success rate can be used as with simple actions. A sequence does not in itself contribute to behavioral diversity, but it does contribute challenge.

The challenge of a selector can be calculated as a selector with all the weight being one. For diversity, the sequence does not contribute anything here because $1 \cdot \ln 1 = 0$. For challenge this gives a level dependent of the success rates. Similarly to how we calculated diversity, we could convert to a selector and calculate the challenge. But we do not have to as we can calculate directly.

It is possible to use a measure of challenge which is defined as the most appropriate level of challenge and not the highest one possible.[28] In terms of behavior trees we have the hypothesis that the most appropriate level of challenge is around where the agent use the good strategies the most and the bad strategies the least, but balanced such that it does not always choose the best strategy.

Behavior where the player wins more have a lower challenge and behaviors where the player loses more have higher challenge. If we map this to success rates of actions, we want weights such that the combined behavior considering both challenge and diversity is close to winning half the time and losing the other half. And we want to be able to easily adjust this.

4.2. Optimization of Behavior Tree Structure

There are several things that can be done to a behavior tree in order to optimize it. One thing we can do is simplify the trees into equivalent trees with fewer and simpler composites. We can do this for selectors, sequences and parallels.

4.2.1. Sequence

This is the simplest case. If we have a sequence with a sequence as a child we can combine these sequences into one by simply moving the sub-behaviors into the parent sequence (as seen in figure 4.7). This creates one larger sequence that will have the exact same behavior. This makes the execution faster and consumes less memory as less structure has to be stored.

4.2.2. Probabilistic Selector

This is the case where a selector has one or more sub-selectors. If we consider the case using probabilistic selectors we have that the marginal probabilities of selecting a behavior should be the same in the original and the transformed tree. If we consider a tree with a selector with two children with weights 0.5 and 0.5 and

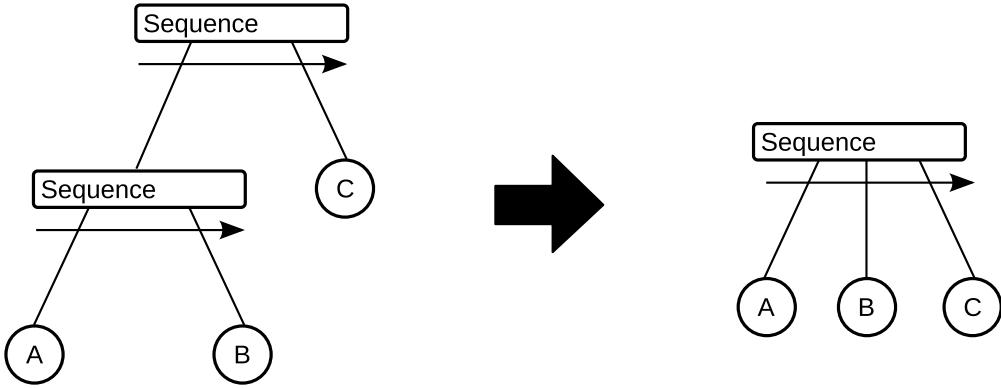


Figure 4.7.: Consolidate Sequences

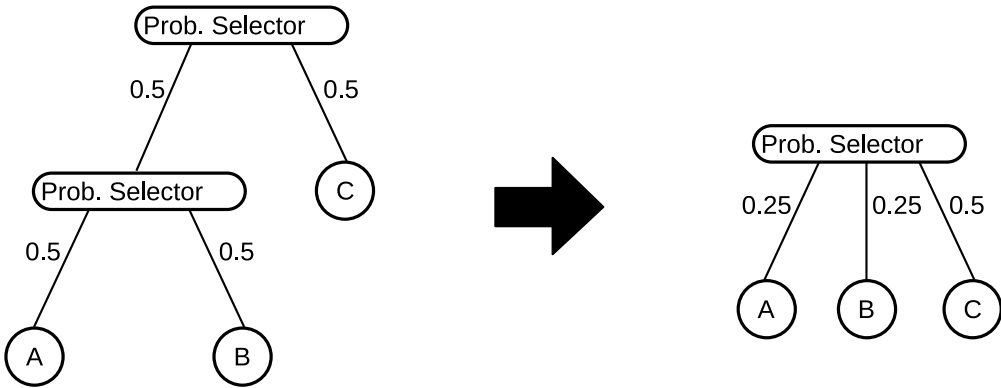


Figure 4.8.: Consolidate Probabilistic Selectors

one of the children is another selector with two children with weights 0.5 and 0.5. This tree would be approximately equivalent to a tree with two children with 0.25 weights and one child with 0.5 weight (the weights are simply multiplied). But it is not exactly the same because having two selectors creates a binding between the actions in each selector. This means that we cannot simply do this conversion and expect the same behavior, but we can argue that we can use the conversion for evaluation of diversity and challenge because as challenge and diversity goes up for one representation it goes up for the other and the opposite.

4.2.3. Random Selector

Now we can simplify probabilistic selectors, but what about random selectors. These can be converted simply by viewing them as probabilistic selectors and giving each child the same weight.

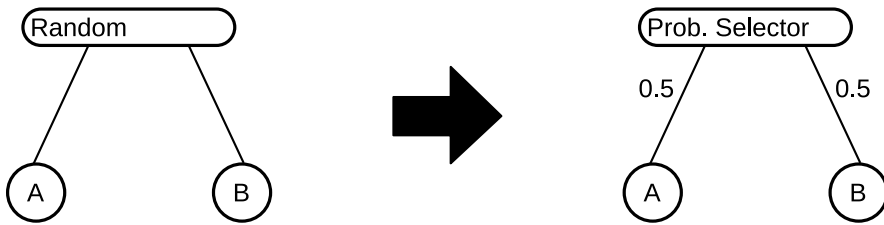


Figure 4.9.: Convert Random Selector to Probabilistic Selector

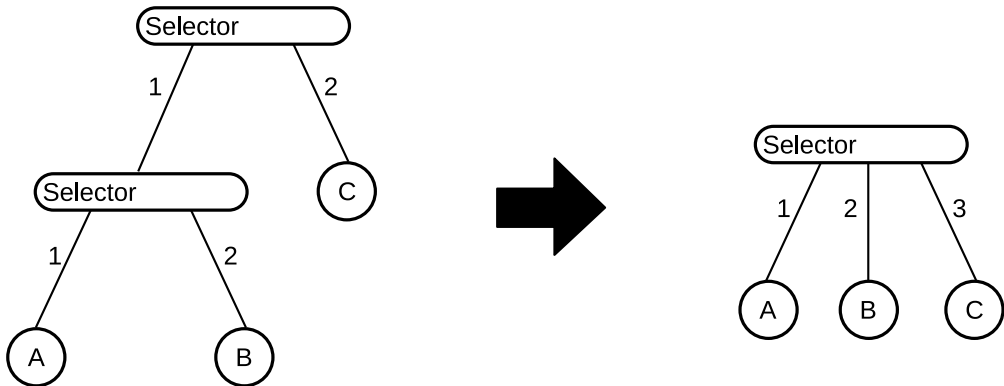


Figure 4.10.: Consolidate Prioritized Selectors

4.2.4. Prioritized Selector

Prioritized selectors can be simplified with the same process as sequences can as they are simply sequences with the change that they return the moment one behavior returns success.

4.2.5. Parallel

As long all parallels involved use certain termination criteria, it is possible to combine parallels in the same way as with sequences. Note, however, that in the cases where at least one uses a termination criteria where the parallel first terminates if all children have failed and first succeeds if all children have succeeded, it is not what this mean if the two parallels were combined.

4.2.6. Conclusion

The reason why these kinds of simplifications should be performed by the computer and not simply writing them in this way from the start is because it is easier for the designers to design behaviors out of logical pieces which can be combined. The fact that these can be combined and simplified should not matter to the designers.

A typical use case is the use of references where one behavior tree could be a sequence of sub-behaviors is used as a reference in another sequence. Here it is not directly evident that it can be simplified because the sequence is hidden in the reference. And more importantly: the designer should not have to know what is actually inside the reference, as this abstraction is an important part of the behavior tree concept, as mentioned in Chapter 2 on page 5.

4.3. Optimization of Interest Value

One major issue with usability in behavior trees is hand tweaking of floating point priorities. This is difficult to do when there are many children. And it makes it hard for a game designer to achieve the exact behavior they want[11].

A paper on complexity in Halo 2[12] refers to this problem as parameter creep, where the number of values needed is too large to manage. They suggest a solution to this problem by allowing inheritance of behaviors such that the complexity can be shared and reused. An approach such as that suggested by Object-Oriented Hierarchical State Machines[20] could be a solution to this problem.

Our solution to this problem is to hide all the values under higher level concepts, such as the challenge and diversity measures described earlier, and let the weights be generated automatically based on those higher level values. Having a way to measure challenge and diversity and the ability to optimize them not only helps solve the problem of parameter creep but also allow the behavior tree designer to instantly see the impact of changing one of the lower level float values.

We need to find weights such that the combined challenge and behavioral diversity is as high as possible. It can be hard to illustrate as there typically are many variables to consider and presenting the function which is to be optimized graphically is impossible as it has too many dimensions. With a selector with only two nodes, the challenge and diversity can be illustrated as in Figure 4.11. The x-axis represents the value of two weights w_1 and $w_2 = 1 - w_1$. The green line represents the challenge level for the combinations of the two weights, with the most challenging point at (0.3, 0.7). The red line represents the diversity with the most diverse point at the midpoint. The orange line represents the combined measure weighting the two measures evenly. The black line is the derivative of the two measures, with the most optimal point at where this line is equal to zero.

It would give the highest challenge level to weigh the highest valued action 1.0 and the rest 0.0, but this would give a behavioral diversity of one, thus giving us an interest of 1.0 (assumed equal weights for challenge and behavioral diversity). Finding the highest interest is a compromise between these two values.

4.3.1. Three Variable Interest Optimization

To demonstrate how we find the optimal weights for optimal interest values, we use an example with three weights. The unknown weights are named w_1 , w_2 and w_3 . Variables p_1 to p_3 are known success rates for each sub-behavior. k_1 and k_2 are

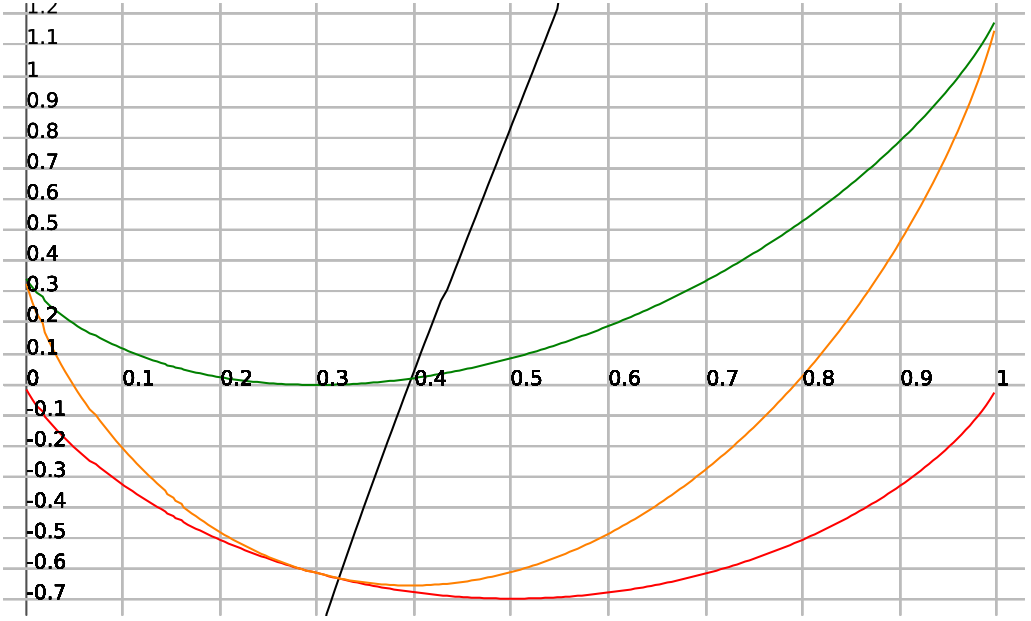


Figure 4.11.: Graphing of diversity and challenge

constants controlling the contribution of the interest of diversity and challenge. k_1 should sum to one, as should w_1 to w_3 , and p_1 to p_3 .

One way to solve the problem of finding optimal values for the weights in both challenge and behavioral diversity levels is to treat it as a geometric problem and using a distance metric between the point (instance of weights) and the optimal one. In the case of behavioral diversity the most diverse configuration is when all weights are the same, we can then calculate how diverse an instance is by the distance from that instance to the optimal one. We scale this such that the optimal configuration gives a value of 1.0 and the configuration furthest away (i.e. the one with one weight 1.0 and the rest 0.0) a value of 0.0.

In the case of challenge, we can use the distance to the case where one weight is 1.0 and the rest is 0.0.

These two values combined gives the final interest (except for weights between them).

But how do we find the optimal one: simple, we just have to find the maximum of the combined function. This can be done analytical by finding the derivative of the combined function and set equal zero, i.e find extrema.

Definition 4.13 describes the interest of a behavior of a selector using a combination of entropy[23] for diversity and KL-divergence [24][18] for challenge.

$$E = k_1(w_1 \ln w_1 + w_2 \ln w_2 + w_3 \ln w_3) + k_2(w_1 \ln \frac{w_1}{p_1} + w_2 \ln \frac{w_2}{p_2} + w_3 \ln \frac{w_3}{p_3}) \quad (4.13)$$

This formulation can be rewritten so that w_3 is written as $1 - w_1 - w_2$ as they

sum to one.

$$E = k_1[w_1 \ln w_1 + w_2 \ln w_2 + (1 - w_1 - w_2) \ln(1 - w_1 - w_2)] \\ + k_2[w_1 \ln \frac{w_1}{p_1} + w_2 \ln \frac{w_2}{p_2} + (1 - w_1 - w_2) \ln \frac{1 - w_1 - w_2}{p_3}] \quad (4.14)$$

Rewrite logarithms from fractions to difference.

$$E = k_1[w_1 \ln w_1 + w_2 \ln w_2 + (1 - w_1 - w_2) \ln(1 - w_1 - w_2)] \\ + k_2[w_1 \ln w_1 - w_1 \ln p_1 + w_2 \ln w_2 - w_2 \ln p_2 \\ + (1 - w_1 - w_2) \ln(1 - w_1 - w_2) - (1 - w_1 - w_2) \ln p_3] \quad (4.15)$$

Factor out $(k_1 + k_2)$ and k_2 .

$$E = (k_1 + k_2)w_1 \ln w_1 + (k_1 + k_2)w_2 \ln w_2 + (k_1 + k_2)(1 - w_1 - w_2) \ln(1 - w_1 - w_2) \\ - [k_2w_1 \ln p_1 + k_2w_2 \ln p_2 + k_2(1 - w_1 - w_2) \ln p_3] \quad (4.16)$$

Find derivative with respect to w_1 and w_2 , and set equal to zero to find extrema.

$$\frac{\partial E}{\partial w_1} = (k_1 + k_2)[\ln \frac{w_1}{1 - w_1 - w_2}] - k_2[\ln p_1 - \ln p_3] = 0 \\ \frac{\partial E}{\partial w_2} = (k_1 + k_2)[\ln \frac{w_2}{1 - w_1 - w_2}] - k_2[\ln p_2 - \ln p_3] = 0 \quad (4.17)$$

Isolate expressions in 4.17 that involve the weights in both equations.

$$\frac{w_1}{1 - w_1 - w_2} = \exp \frac{k_2(\ln p_1 - \ln p_3)}{k_1 + k_2} \\ \frac{w_2}{1 - w_1 - w_2} = \exp \frac{k_2(\ln p_2 - \ln p_3)}{k_1 + k_2} \quad (4.18)$$

Isolate $1 - w_1 - w_2$ in both equations, set them equal and isolate weights in the equation.

$$\frac{w_1}{w_2} = \exp \frac{k_2(\ln p_1 - \ln p_2)}{k_1 + k_2} \quad (4.19)$$

Isolate w_1 in equation 4.18 and replace w_2 using equation 4.19.

$$w_1 = \exp \frac{k_2(\ln p_1 - \ln p_3)}{k_1 + k_2} - \exp \frac{k_2(\ln p_1 - \ln p_3)}{k_1 + k_2} \cdot w_1 \\ - w_1 \cdot \exp \frac{k_2(\ln p_2 - \ln p_1)}{k_1 + k_2} \cdot \exp \frac{k_2(\ln p_1 - \ln p_3)}{k_1 + k_2} \quad (4.20)$$

Isolate w_1 such that we have an expression for w_1 not involving any weights, only the success rates and the k constants.

$$w_1 = \frac{\exp \frac{k_2(\ln p_1 - \ln p_3)}{k_1 + k_2}}{1 + \exp \frac{k_2}{k_1 + k_2}(\ln p_1 - \ln p_3) + \exp \frac{k_2}{k_1 + k_2}(\ln p_2 - \ln p_3)} \quad (4.21)$$

This is symmetric for w_2 and once w_1 and w_2 is determined w_3 is just $1 - w_1 - w_2$. But this only works for three weights in total, we want a to be able to find n weights.

4.3.2. Optimal Weight Generalization

Here we generalize the definition of interest using entropy and KL-divergence to be defined on an arbitrary number of weights w_1 to w_n . Apart from that the definition is the same.

$$E = k_1(w_1 \ln w_1 + \cdots + w_i \ln w_i + \cdots + w_n \ln w_n) + k_2(w_1 \ln \frac{w_1}{p_1} + \cdots + w_i \ln \frac{w_i}{p_i} + \cdots + w_n \ln \frac{w_n}{p_n}) \quad (4.22)$$

As with the three weight case, we can write one of the weights named w_i as 1 minus the sum of all the other weights.

$$E = k_1[w_1 \ln w_1 + \cdots + w_{i-1} \ln w_{i-1} + (1 - w_1 - \cdots - w_{i-1} - w_{i+1} - \cdots - w_n) \ln(1 - w_1 - \cdots - w_{i-1} - w_{i+1} - \cdots - w_n) + \cdots + w_n \ln w_n] + k_2[w_1 \ln \frac{w_1}{p_1} + \cdots + (1 - w_1 - w_2 - w_{i-1} - w_{i+1} - w_n) \ln \frac{(1 - w_1 - w_2 - w_{i-1} - w_{i+1} - w_n)}{p_3} + w_n \ln \frac{w_n}{p_n}] \quad (4.23)$$

We then rewrite the logarithms and factor out the k s as with the three weight case.

$$E = (k_1 + k_2)[w_1 \ln w_1 + \cdots + w_{i-1} \ln w_{i-1} + w_{i+1} \ln w_{i+1} + w_n \ln w_n + (1 - w_1 - w_2 - w_{i-1} - w_{i+1} - w_n) \ln(1 - w_1 - w_2 - w_{i-1} - w_{i+1} - w_n)] - k_2[w_1 \ln p_1 + \cdots + w_{i-1} \ln p_{i-1} + (1 - w_1 - w_2 - w_{i-1} - w_{i+1} - w_n) \ln p_i + w_n \ln p_n] \quad (4.24)$$

We then differentiate, set equal to zero to find the extrema of the functions with respect to w_1 to w_n except for w_i .

$$\begin{aligned} \frac{\partial E}{\partial w_1} &= (k_1 + k_2) \left[\ln \frac{w_1}{1 - w_1 - w_2 - w_{i-1} - w_{i+1} - w_n} \right] - k_2 [\ln p_1 - \ln p_i] = 0 \\ \frac{\partial E}{\partial w_2} &= (k_1 + k_2) \left[\ln \frac{w_2}{1 - w_1 - w_2 - w_{i-1} - w_{i+1} - w_n} \right] - k_2 [\ln p_2 - \ln p_i] = 0 \\ \frac{\partial E}{\partial w_{i-1}} &= (k_1 + k_2) \left[\ln \frac{w_{i-1}}{1 - w_1 - w_2 - w_{i-1} - w_{i+1} - w_n} \right] - k_2 [\ln p_{i-1} - \ln p_i] = 0 \end{aligned} \quad (4.25)$$

We rewrite these $n - 1$ equations to isolate the weights.

$$\begin{aligned} \frac{w_1}{1 - w_{i-1} - w_{i+1} - \cdots - w_n} &= \exp \frac{k_2 (\ln \frac{p_1}{p_i})}{k_1 + k_2} \\ &\vdots \\ \frac{w_{i-1}}{1 - \sum_{j \neq i} w_j} &= \exp \frac{k_2 (\ln \frac{p_{i-1}}{p_i})}{k_1 + k_2} \end{aligned} \quad (4.26)$$

And then isolate $(1 - \sum_{j \neq i} w_j)$ in equations 4.26 and set all these equations equal.

$$\frac{w_1}{\exp \frac{k_2 (\ln \frac{p_1}{p_i})}{k_1 + k_2}} = \cdots = \frac{w_{i-1}}{\exp \frac{k_2 (\ln \frac{p_{i-1}}{p_i})}{k_1 + k_2}} = \cdots = \frac{w_n}{\exp \frac{k_2 (\ln \frac{p_n}{p_i})}{k_1 + k_2}} = w \quad (4.27)$$

Using equations 4.26 and replace the weights with an expression using only the success rates p_1 to p_n multiplied with the factor w which is the ratio between the success rates and the associated weights obtained in equation 4.27. With this we obtain an equation 4.28 for an arbitrary weight w_j .

$$\frac{w \cdot \exp \frac{k_2 \ln \frac{p_j}{p_i}}{k_1 + k_2}}{1 - \sum_{j \neq i} w \cdot \exp \frac{k_2 \ln \frac{p_j}{p_i}}{k_1 + k_2}} = \exp \left(\frac{k_2}{k_1 + k_2} \cdot \ln \frac{p_j}{p_i} \right) \quad (4.28)$$

$$w = 1 - w \cdot \sum_{j \neq i} \exp \frac{k_2}{k_1 + k_2} \ln \frac{p_j}{p_i} \quad (4.29)$$

Rewriting equation 4.28 we get an equation for w .

$$w = \frac{1}{1 + \sum_{j \neq i} \exp \frac{k_2}{k_1 + k_2} \ln \frac{p_j}{p_i}} \quad (4.30)$$

We can now find expressions for arbitrary weights w_1 to w_n by using equations 4.27 and 4.30.

$$w_1 = \frac{\exp \frac{k_2}{k_1 + k_2} \ln \frac{p_1}{p_i}}{1 + \sum_{j \neq i} \exp \frac{k_2}{k_1 + k_2} \ln \frac{p_j}{p_i}} \quad (4.31)$$

Use that the weights should always sum to one.

$$w_i = 1 - \sum_{j \neq i} w_j \quad (4.32)$$

Substitute weight w_1 from equation 4.31 and weights w_2 to w_n similarly. Is further simplified such that the exponential function is no longer needed.

$$w_i = 1 - \frac{\sum_{j \neq i} \frac{p_j}{p_i} \frac{k_2}{k_1 + k_2}}{1 + \sum_{j \neq i} \frac{p_j}{p_i} \frac{k_2}{k_1 + k_2}} \quad (4.33)$$

Rewrite to sum over all successrates including the i th one.

$$w_i = 1 - \frac{\sum_{j=1}^n \frac{p_j}{p_i} \frac{k_2}{k_1 + k_2} - 1}{\sum_{j=1}^n \frac{p_j}{p_i} \frac{k_2}{k_1 + k_2}} \quad (4.34)$$

This yields the final formulation of the optimal weight of an arbitrary action in a selector.

$$w_i = \frac{1}{\sum_{j=1}^n \frac{p_j}{p_i} \frac{k_2}{k_1 + k_2}} \quad (4.35)$$

In the case where k_1 is 1, that is all the weight is on the behavioral diversity and none is on challenge, the whole expression $k_2/k_1 + k_2$ becomes zero and the terms all become 1. Now the sum can be written $1 \cdot n$, thus the whole equation becomes 1 over n . This is by the definition of behavioral diversity the most diverse case as we would want it to.

In the other extreme case where k_2 is 1, all the weight is on the challenge part of

the measure. In this case the expression $k_2/k_1 + k_2$ becomes one and all the terms simply become p_j/p_i , summed over j this becomes $1/p_i$. The expression $1/(1/p_i)$ becomes simply p_i , thus all the weights reduces to the success rate, which is the case we defined as the most challenging case.

Adjusting k_1 and k_2 can then be used to tweak the method to either prefer diversity or challenge.

Note that throughout the rest of the report k_1 refers to the weight of behavior diversity and k_2 to the weight of the challenge.

4.3.3. Optimizing Entire Behavior Trees

Now we have a method to find the optimal weights for a single selector with an arbitrary number of sub-actions. In order for this to be useful in general behavior trees it needs to be possible to determine optimal weights for arbitrary behavior trees including multiple selectors and sequences.

Global Path Based Method

One way to determine the diversity and challenge of an arbitrary behavior tree is to convert the tree into its canonical form with only one selector with a list of simple actions. Now the general solution for optimizing behavior trees can be applied directly.

To do this we start with an arbitrary behavior tree. The selectors of this tree are then combined as described in 4.2.2 to 4.2.4 and shown in algorithm 1. Then we combine the sequences as in 4.2.1, creating a much simpler behavior with approximately the same behavior and more importantly diversity and challenge.

Algorithm 1 Combine Selectors

Require: An arbitrary behavior tree
 $\text{allSelectors} \leftarrow \text{getAllSelectors}(\text{BT})$
for each selector in allSelectors **do**
 if parent is a selector **then**
 for each child of selector **do**
 $\text{parent}(\text{child}) \leftarrow \text{parent}(\text{selector})$
 $\text{weight}(\text{child}) \leftarrow \text{weight}(\text{selector}) * \text{weight}(\text{child})$
 $\text{children}(\text{parent}(\text{selector})) \leftarrow \text{child}$

Ensure: A behavior tree with reduced selectors

But to optimize the weights of the tree for diversity and challenge, we need to eliminate the sequences. We do this by the method described in 4.1.2 which combines possible paths into simple actions allowing us to eliminate the sequences. This is done from the leaf level up to the root level. Algorithm 2 on the next page shows the code for accomplishing this task.

The algorithm first gets all of the sequences in the current behavior tree or subtree. Then it combines any simple behaviors in the sequences followed by then running through every of its children, combining any child selector with any simple

Algorithm 2 Convert Sequences

Require: An arbitrary behavior treeallSequences \leftarrow getAllSequences(BT)**for** each sequence in allSequences **do**combinedSimpleBehaviors \leftarrow combineSimpleSequenceBehaviors(sequence)**for** each child of sequence **do****if** child is a selector **then**combinedSimpleBehaviors \leftarrow combineSimpleAndSelectorBehaviors(combinedSimpleBehaviors, selector)

setChildren(sequence, combinedSimpleBehaviors) setAsSelector(sequence)

Ensure: A behavior tree with reduced selectors

combined behaviors, such that every possible combination is created. When combining two nodes their success are multiplied together to form the combined nodes success rate. Lastly it sets the newly combined behaviors as the children of the sequence and sets a flag on the sequence which defines that the sequence now has been converted to a selector.

We can now combine the newly created selectors with the existing selectors the same we combined selectors before.

Algorithm 3 Canonic Behavior Tree

Require: An Arbitrary Behavior Treeselectors \leftarrow getAllSelectors(BT)selectors \leftarrow restructureSelectors(BT, selectors)selectors \leftarrow convertSequencesToSelectors(BT, selectors)selectors \leftarrow restructureSelectors(BT, selectors)**Ensure:** A Canonical Behavior Tree

Now we have a behavior tree the canonical form, from which we can calculate the optimal weights.

Obviously the behavior tree will not have the same behavior after all these transitions, which is described in subsection 4.2.2 on page 23.

Because we want to use the original behavior for actual execution and not our converted tree. We need to convert the calculated weights back such that they can be used in the original tree.

Algorithm 4 Convert Selector Weights Back

Require: Original and canonical behavior treesselectors \leftarrow sortSelectorsFromBottom(originalBT)simplifiedselectors \leftarrow selectors(canonicalBT)**for** each selector and simplified selector **do**selector.weight \leftarrow sum(simplifiedselector.actions.weight)**for** each action in simplifiedselector.actions **do**selector.action.weight \leftarrow action.weight / selector.weight**Ensure:** A behavior tree with weights adjusted

The method shown in algorithm 4 convert the weights found by the weight optimizer back such that they can be used in the original behavior tree. To do this it need both the converted tree and the original. It should be noted that simplifiedselector contains only the actions corresponding to the actions of the original selector.

Algorithm 5 Convert Sequence Weights Back

Require: Original and canonical behavior trees
 behaviors \leftarrow getAllSimpleBehaviors(origBT)
for each behavior X in behaviors **do**
 $X.weight \leftarrow 0.0$
 for each behavior Y in canonicalBT that contain X **do**
 $X.weight += Y.weight$
 sequences \leftarrow getAllSequenceBehaviors(origBT)
for each sequence X in sequences **do**
 selectors \leftarrow getAllSelectorsBehaviors(sequence)
 for each selector Y in selectors **do**
 normalizeChildrenWeights(selectors)
Ensure: A behavior tree with weights adjusted

Algorithm 5 is similar to 4, but convert weights back to sequences instead of selectors. It uses the calculated weights to determine the probability of a sequence starting with a particular behavior. Afterwards it insures that any selectors get their child weights normalized such that they are in the range between 0 and 1. The main issue with this method is that it is expensive to run because converting sequences to selectors can be expensive. The method also involves a lot of converting back and forth between different tree representations which is also time consuming.

Local Method with Selector Concatenation

Because the method involving sequence to selector conversion is expensive, we propose an alternative approach avoiding the expensive operation. In this approach we only concatenate the selectors ,to allow our optimizer to work over larger behaviors, but we treat the sequences as simple actions and optimize sub-behaviors locally.

We start with an arbitrary behavior tree and combine the selectors as described in 4.2.2 to 4.2.4.

We now run the optimization locally for each of the remaining selectors treating all sequences as simple actions when seen from above in the tree.

We then convert the weights back to the original tree and we are done. This is actually done by using algorithm 4 on the preceding page and running this on each of the remaining probability selectors in the tree. Note that the originalBT will be current selector's original form while the canonical behavior tree is the current selector's canonical form.

Obviously this completely ignores any knowledge about sequences because the intuition behind this method is that sequences does not add to the diversity and challenge directly, only selectors do this as described in subsection 4.1.5 on page 23.

Entirely Local Method

This is the simplest possible method to used in that it only considers local optimizations.

We start with some arbitrary behavior tree and calculate the optimal weights independently for all selectors. This happens directly on the tree and ignores the existence of sequences and without any attempt at global optimization.

With this method there are no need to convert the resulting weights back to the original tree, as no changes have been introduced to the tree.

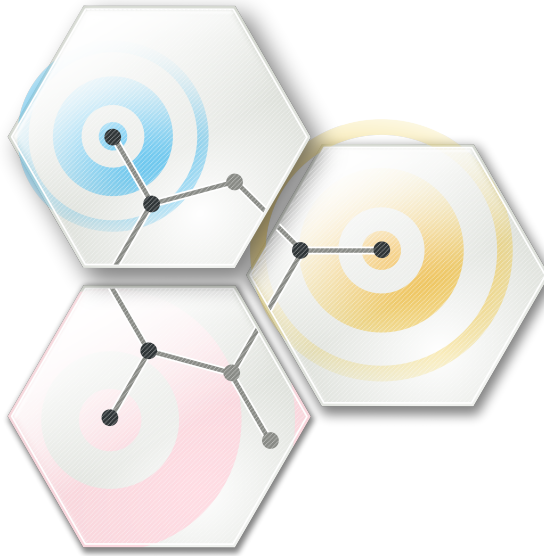
This naive assumption that we can optimize locally and achieve results might be practical to use in some cases, such as weighting different high-level strategies. It also have the advantage that it is cheap to run and scales nicely. But it does not guarantee to be optimal in any way.

4.4. Conclusion

Over the course of the last many pages we have constructed ways of measuring and optimizing behavior trees, and finally how to optimize the interest value of a behavior tree. Three methods for optimizing the value have been suggested, and we must now begin to investigate which are appropriate in which situations.

Part II

SMARTS Revisited



SMARTS is a generic game AI system based on the concept of Behavior Trees, which in this revised version also includes a generalised perception system.

SMARTS LIBRARY

To be able to perform an evaluation of the various methods presented in the previous chapter, an implementation of behavior trees is needed. While the SMARTS Library in the form in which it existed at the end of 2009 did work, its construction was based on various assumptions which later proved to be problematic when using it in a practical application.

As such, we believe that it is advantageous to spend some time conducting a re-evaluation of the system, which will later allow us to use the library in the evaluation process in a more effective manner than it was possible during the evaluation period of Where Academic AI Meets Game AI[16].

In this chapter we will look at issues that were encountered because of the previous design of the SMARTS Library. We will also re-design parts of the SMARTS Library for being able to solve some of these issues.

5.1. Analysis

In this section we will reflect on the issues that the first SMARTS Library design presented us with, as discovered through its use.

5.1.1. Tree structure

The previous behavior tree design[16] used a two layer tree structure, The first layer was the internal structure of the tree, while the second layer contained all the functionality e.g selectors, sequences, and so forth.

This structure gave an unnecessary memory overhead because of the two layer structure, as well as introducing a significant number of context switches and call stack usage because of the internal hierarchical structure when executing the behavior trees. Furthermore problems with function naming arose, in that some of the function names were not as intuitive or describing as they could have been.

The issues regarding memory overhead and naming conventions can be solved by using a single tree structure instead a two layer version used now. Both issues with context switching and call stack usage can be solved having a list instead of the tree structure, though the tree structure is used for graphical and design purposes. Each tree node will point at the locations in the list where its children are placed, thereby reducing the amount of context switches when the character is calling its think functions.

Another way to solve this problem is to manually have a call stack together with keeping the single layer tree structure. This solutions will provide the same advantages as above, but it is not necessary to saving a lot of indices to other nodes, because it is just necessary to pop the top of the stack when a tree node has finished executing.

For being able to do these kinds of execution it would be necessary with an scheduler with ensures the correct node in the tree is executed, together with saving and restore the execution states. A scheduler would also make implementing interrupts and resumption in behavior trees easier

5.1.2. Parallels and Threading

During the implementation of the parallels composite[16, p. 9] we decided to use threads to implement this tree node, for being able to check any conditions and running the subtrees at the same time, this decision proved to be a source to great number of problems.

First of all, we had to create synchronizations points in the implementation, for being able to stop and synchronize all threads in a parallel. Furthermore a lot of deadlocks on mutexes occurred, which made the threads stall and not end their execution, requiring us to have to add a timeout when executing the threads, so that the behavior trees could recover from these incidents. Debugging the code for errors and bugs is almost impossible because of the threads. It is possible to see that something went wrong and where in the code, but tracing the origin of the error is highly complex.

There are several solutions to this problem, where one is to use Qt's signal / slot functionality, such that the behavior tree will only get updates when the conditions has been fulfilled, instead of querying for the values in a thread. Another solution is to force the behavior trees to either run at fixed time or fixed frame execution, such that the conditions will be checked every time the given behavior resumes execution. It is also possible to connect to Qt's event system, and thereby sending and handling events in different parts of the AI system.

Because of the design, each character in the game is run in its own thread. This insures that it is possible to reuse behavior trees, without having a copy of the tree for each of the characters. The same problems as above can arise, where the game and the AI threads have to synchronize. By using a scheduler and a perception system for each character, it is possible to just to pass the character as an argument to the currently running node, without each character having to run in a separate thread. The scheduler would then ensure the correct execution state is used when a character executes its behaviors.

5.2. Re-Design

In this section the re-design of a number of SMARTS Library components are described starting with the new over-all design followed by the `btNode` class together with node types derived from this class. Then the `btCharacter` class is described together with the new scheduler placed in this class' `think` function.

5.2.1. Over-all Design

The SMARTS Library will be changed because of some the issues encountered in the last design. One of the big design changes is the elimination of threads in parallels, and to do this the SMARTS Library is changed to employ a scheduler

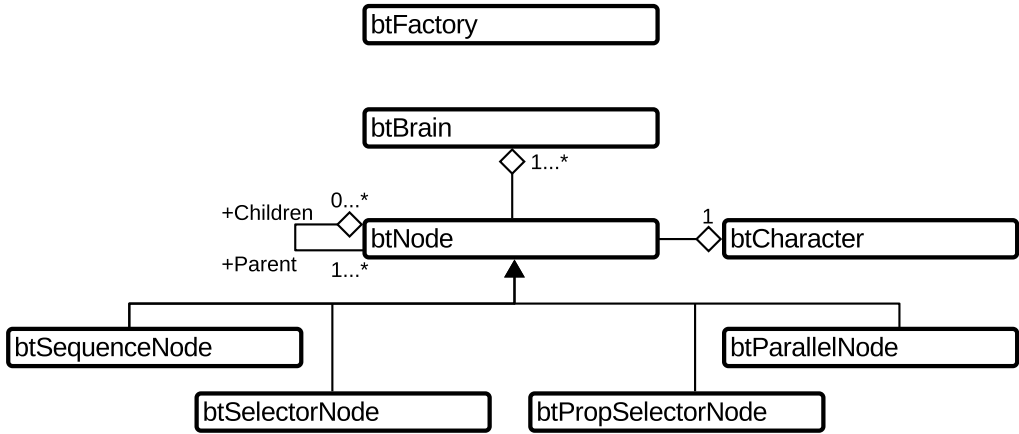


Figure 5.1.: Class diagram showing the overall structure of the SMARTS Library

for managing the execution of both behavior trees, with particular features for parallels. With this decision the SMARTS Library will run at fixed time/frame where previously it would run until the behavior tree had finished executing.

Figure 5.1 shows the over all design of the architecture of SMARTS Library. It shows that the `btNodeType` has been removed from the library, and the class `btNode` has taken its place, both as the internal structure of the tree but also as the class that every tree node has to inherit from. Furthermore the `btCharacter` class will contain the scheduler, which means that every character in a game will have to inherit from `btCharacter` to be able to utilize the scheduler and the perception system described later. Both the `btBrain` and the `btFactory` have not been re-designed, as this was not necessary.

5.2.2. `btNode`

The `btNode` class is basically a merge between `btNode` and `btNodeType`[16, p. 33], where the `btNodeType` is removed and its functionality relocated to `btNode`. This means that `btNode` is functioning as both the internal tree structure and as the super class for every node type in the behavior tree.

Outside of the two standard node types[16, p. 24] we have added two new node types to the library, which were previously found in the test bed application, which are:

Probability Selector: As a normal selector this node type selects as child for execution and only returns success if the child succeeds, else it will continue selecting children until there is none left and thereby return false. The difference is that this selector uses probabilities for selecting a child, and these probabilities are set in the SMARTS Designer.

Parallel: This node type was described[16, p. 9] but not implemented in the previous version of SMARTS Library. A parallel runs all of its children simulta-

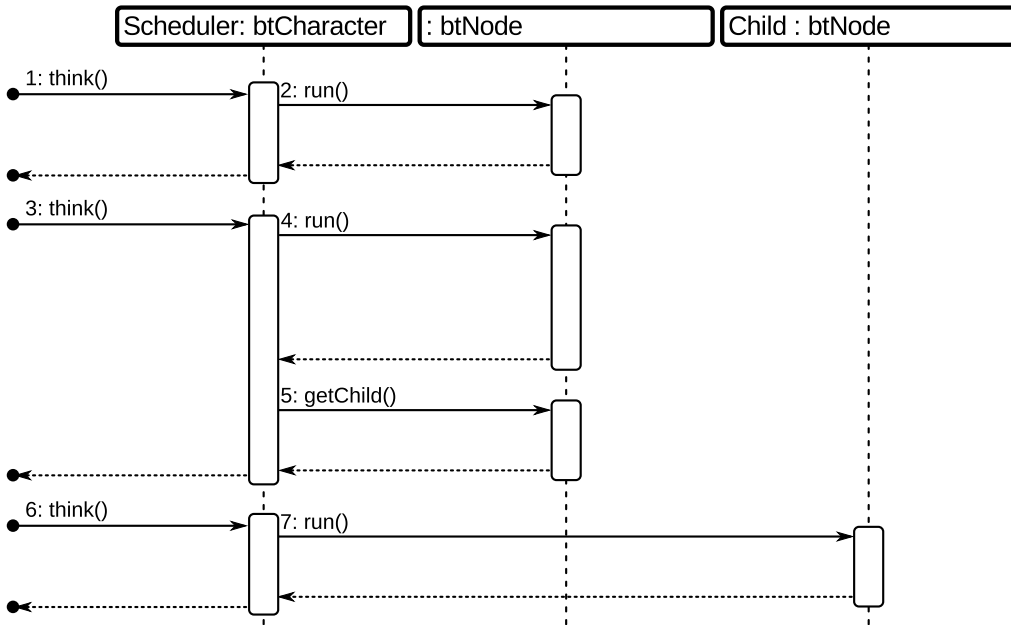


Figure 5.2.: Sequence diagram showing how the scheduler works

neously and finishes execution when the children have fulfilled certain termination conditions. It also insures cleanly termination of all its children.

5.2.3. btCharacter

The `btCharacter` is base class for every character in a game using the SMARTS Library[16, p. 24]. The only change to this class is that it now contains a scheduler for executing its behavior tree and a function that is used for making the `btCharacter` take a decision with the behavior tree. This function will then be called with a fixed time/frame interval from the game loop.

5.2.4. Scheduler

The new scheduler is the biggest change to both the `btCharacter` class specifically and the SMARTS Library in general. It is placed on the `btCharacter` and is called through a function called `think`.

Figure 5.2 shows an example of how the scheduler works in the `btCharacter`. If we follow the sequence numbers of the figure, the scheduler runs as follows:

1. The `btCharacter`'s `think` function is called by the game loop in a frame/time slice.
2. In the same frame/time slice the current `btNode`'s `run` function is called. The `btNode` does not finish execution but because of the frame/fixed time restriction, it has to return with a status message which indicates it is not

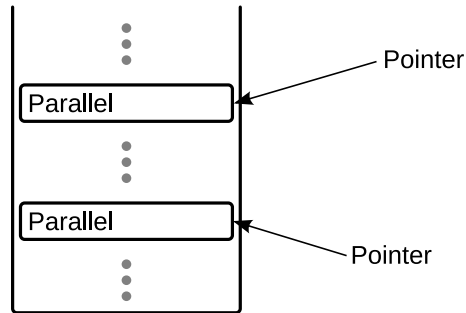


Figure 5.3.: Shows the scheduler structure when only one stack is used

done. The `btNode` has to return at a point, which can be resumed when `run` is called again. The `btCharacter` also returns, but does not return a status message, as this is not necessary outside the `btCharacter`.

3. In the next frame /time slice the `btCharacter`'s `think` function is called again.
4. During the same frame/time slice it also resumes the currently running `btNode`. The `btNode` decides that it wants to run a child and returns a status message to `btCharacter` indicating that it wants to run a child.
5. The `btCharacter` then retrieves the child and returns.
6. In the next frame /time slice the `btCharacter`'s `think` function is called once again.
7. The `btCharacter` then runs the child, and the child returns with a given status message.

There are different possibilities for creating the structure which the scheduler uses for executing the behavior tree.

Stack-based Scheduler

A stack-based scheduler would use a stack for its execution of the behavior tree. By doing it this way, the stack would contain the currently executing instances of `btNode`, where the top of the stack would be the currently running `btNode`. It is also possible to get the parent of the currently executing `btNode`, because it is just the `btNode` that is just beneath the top `btNode`.

To be able to support parallels with this kind of stack-based scheduler, the scheduler needs to point on each parallel in the stack, this is shown on figure 5.3. Furthermore, it has to point at the position of each of children's currently executing tree node of their subtrees. The reason for this is the scheduler has to run every parallel child simultaneously, which means it has to switch between children during the execution to give the illusion of running the children "simultaneously". If we are using only one stack there will have to be a great amount of data handling,

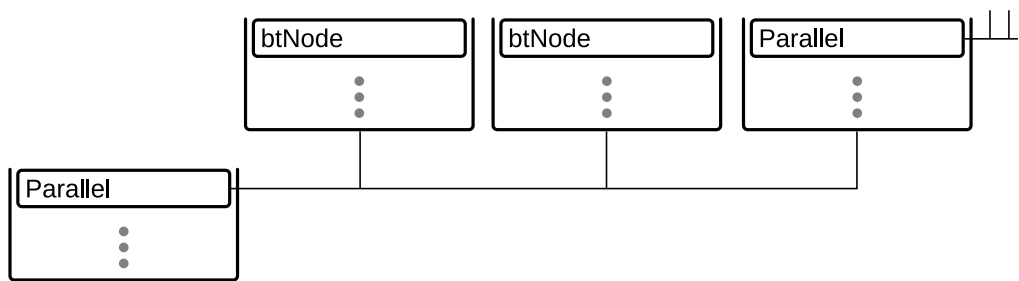


Figure 5.4.: Shows the scheduler structure when using branching stacks

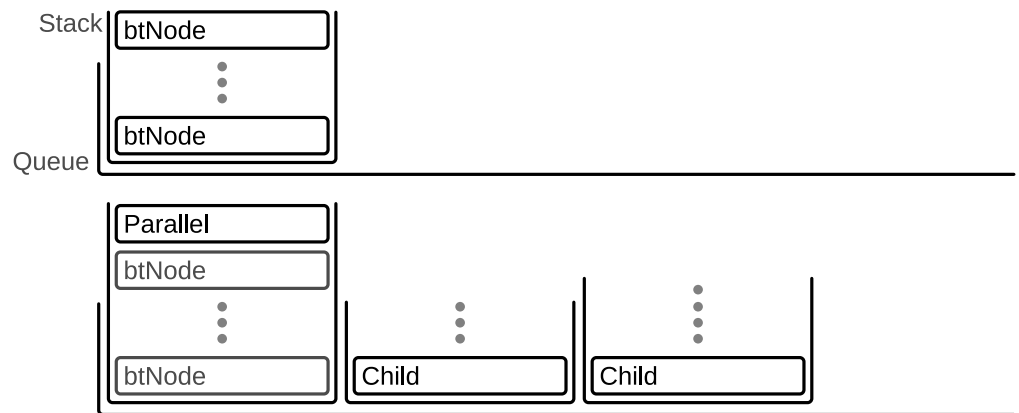


Figure 5.5.: Shows the scheduler structure when using queues and stacks

because it would necessary to pop the stack until a parallel is at the top and push the next parallel child’s subtree onto the stack before it’s execution.

Stack-based Scheduler with Branching

Another variant of the stack-based scheduler uses branching to handle the parallels, shown 5.4. Whenever a parallel executes it branches out such each child has its own execution stack, and these are then run for themselves, and when the stack returns it returns to the parent execution stack. If we are using branching stacks there will also be a great amount of data handling, but here it would not be necessary to pop a stack until a parallel has been reached. Instead it is just necessary to run the parallel’s children stacks. It is, however, necessary to keep a pointer to each stack, keep references to their parent parallels, and insure a fair execution policy for the stacks, such that it gives the illusion of running simultaneously.

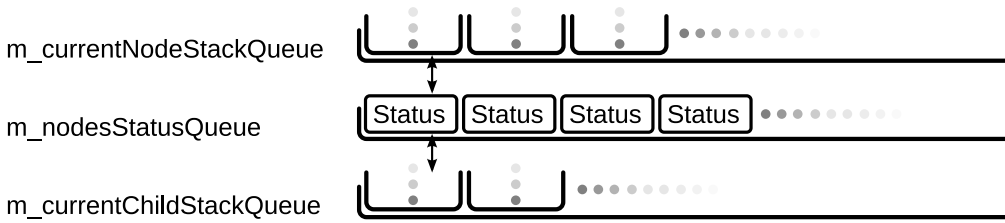


Figure 5.6.: Shows the three queues used in the scheduler

Queue- & Stack-based Scheduler

The scheduler could also combine queues and stacks for still having some of the advantages with a stack-based scheduler which uses branching but also by ensuring a fair execution of the stacks. If the scheduler is using queues it would contain a queue which holds the execution stacks. If there were no parallels in the behavior tree, the queue would only contain one stack which would be dequeued when it is time for execution and enqueued when the frame/time slice has expired.

When a parallel is encountered during execution it will remain at the top on its execution stack, and a execution stack for each of the parallel children will be added to the queue. The scheduler will then run through the queue and executing the stacks, dequeuing and enqueueing when necessary. When a stack then finishes execution it is removed from the queue, though this only applies for the stacks created from parallels. Figure 5.5 on the preceding page shows a queue which runs the normal stack, and a queue which has an parallel on the main stack, and all of its children on new stacks in the queue.

5.3. Implementation Details

In this section we will describe the scheduler which is implemented in the class `btCharacter` and which is used to execute the behavior tree of the `btCharacter`. In subsection 5.2.4 on page 40 a queue- & stack-based scheduler was described. This scheduler variant was chosen for implementation, because it is easier to support parallels than with a stack and branching stack. We first describe the queues used in the scheduler followed by how the execution is done by the scheduler. Then we describe how `btProbSelectors` and `btParallels` are handled inside the scheduler during execution, because these two node types require extra data structures for saving the execution states.

5.3.1. Queues

The scheduler uses three different queues for executing the behavior tree, these are shown in listing 5.1 on the next page. Before explaining what the purposes of the three queues are, it is important to state that each index in the queue `m_currentNodeStackQueue` implicitly states that the same index in the queue `m_nodesStatusQueue` and in the `m_currentChildStackQueue` queue holds infor-

mation used when running that particular execution stack. Therefore it is important that each of the three queues are dequeued and enqueued at the same time to keep the correspondence between the queues. Figure 5.6 on the preceding page illustrates this.

Listing 5.1: Queues used in the scheduler

```
1 QQueue<btNode::status> m_nodesStatusQueue;  
2 QQueue<QPair<QStack<btNode*>*, QStack<btNode*>*> > m_currentNodeStackQueue;  
3 QQueue<QStack<int> > m_currentChildStackQueue;
```

The `m_currentNodeStackQueue` holds the execution stacks but not as a `QStack` but as `QPair<QStack<btNode*>*, QStack<btNode*>*>`, where the first stack is the current execution stack while the second is the parent execution stack. The reason for the use `QPair` is described in Section 5.3.4 on page 47.

The `m_currentChildStackQueue` contains `QStack<int>` instances which hold the current child indices. These stacks have the same size as the corresponding stacks from `m_currentNodeStackQueue`, because they contain the current child index for each of the `btNodes` in the execution stacks.

Lastly `m_nodesStatusQueue` contains the `btNode::status` for the corresponding execution stacks.

5.3.2. Execution

We have described the three queues used in the scheduler, and we have come to the point where we must describe how the scheduler executes. The scheduler is part of the `btCharacter`'s function `think` and runs every time `think` is called. The scheduler runs in three stages:

1. Retrieve an execution stack and needed execution information
2. Restore the execution state and run the current `btNode`
3. Handle the returned `btNode::status` from the current `btNode`

Stage one

Listing 5.2 shows the first stage in `think` is to retrieve a execution stack and the top `btNode` from the stack. It also dequeues both `m_currentChildStackQueue` and `m_nodesStatusQueue` to be able to set the `nodeStatus` and child index for that particular execution stack. `currentParent` is initialized to `NULL` but the scheduler will set the `currentParent` if possible.

Listing 5.2: The code for retrieving an execution stack and the information needed for execution

```
1 ...  
2 currentChildParentStackPair = m_currentNodeStackQueue.dequeue();  
3 currentNodeStack = currentChildParentStackPair.first;  
4 currentChildStack = m_currentChildStackQueue.dequeue();  
5 nodeStatus = m_nodesStatusQueue.dequeue();  
6  
7 //retrieve the executing node  
8 btNode* currentNode = currentNodeStack->top();  
9  
10 //check if it s possible to set parent, and set it if possible
```

```

11 if(currentNodeStack->count() > 1)
12 {
13     currentParent = currentNodeStack->at(currentNodeStack->count() - 2);
14 }
15
16 //check if it s possible to set current child, and set it if possible
17 if(currentChildStack.count() > 0)
18 {
19     currentChildIndex = currentChildStack.pop();
20 }
21 ...

```

Stage two

In this stage the scheduler restores the current `btNode` execution state, by setting its current child index, its current child status, and its parent. The reason for setting the current child index and current child status is, that if in the last call of `think` one of the current `btNode` children finished execution, then this information would have to be passed up to the current `btNode`. When this is done the current `btNode` is run.

Listing 5.3 shows the code for this stage.

Listing 5.3: The code for restoring the execution state

```

1 ...
2 //restore the execution state of the current executing node
3 currentNode->setCurrentChildIndex(currentChildIndex);
4 currentNode->setCurrentChildStatus(nodeStatus);
5 currentNode->setParentNode(currentParent);
6
7 //run the node
8 nodeStatus = currentNode->run(this);
9 ...

```

Stage three

In the last stage the status from the current `btNode` is handled. There are three possible cases: `btNode::RunningChild`, `btNode::Running` and one case for both `btNode::Succeeded` and `btNode::Failed`.

When the returned status is a `btNode::RunningChild`, the scheduler pushes the child together with its index such that it is possible to return with same information when the child terminates. Listing 5.4 shows the code for this case. Furthermore a zero is pushed onto the `currentChildStack`, which is used to initialize the child's current child index in the next call of `think`. At the end both stacks are pushed onto their queues and an initial `btNode::None` is pushed on the status queue.

Listing 5.4: Code for the case `btNode::RunningChild`

```

1 ...
2 case btNode::RunningChild:
3     //when running child, push the child and child index onto the stacks
4     currentNodeStack->push(currentNode->currentChild());
5     currentChildStack.push(currentNode->currentChildIndex());
6     currentChildStack.push(0);
7
8     //enqueue the stacks and status
9     m_currentNodeStackQueue.enqueue(currentChildParentStackPair);
10    m_currentChildStackQueue.enqueue(currentChildStack);

```

```

11
12     m_nodesStatusQueue.enqueue(btNode::None);
13 ...

```

Listing 5.5 shows the case for whenever a `btNode` succeeds or fails. The code starts out with popping both `currentNodeStack` and `currentChildStack` stacks if the `currentNodeStack` stack size is bigger than one. If this is not the case, then the scheduler will push a zero onto the `currentChildStack` if it is empty and if the `currentNode` is the top node of the behavior tree it also resets the `nodeStatus`.

Furthermore if the `currentNodeStack` is not NULL then it is enqueued together with the `currentChildStack` and `nodeStatus`.

Listing 5.5: Code for the case `btNode::Succeeded` and `btNode::Failed`

```

1 ...
2 case btNode::Failed:
3 case btNode::Succeeded:
4 //when the node fails or succeeds
5 if(currentNodeStack->count() > 1)
6 {
7     //if there is more than one node in the stack, pop it
8     currentNodeStack->pop();
9     if(currentChildStack.count() > 0)
10     {
11         currentChildStack.pop();
12     }
13 }
14 else
15 {
16     //else if count == 0 then push a 0 on it
17     if(currentChildStack.count() == 0)
18     {
19         currentChildStack.push(0);
20     }
21
22     //if the node is the root, reset status
23     if(currentNode == m_behaviortree)
24         nodeStatus = btNode::None;
25 }
26 ...
27 if(currentNodeStack)
28 {
29     //if node stack is not deleted, enqueue it again
30     m_currentNodeStackQueue.enqueue(currentChildParentStackPair);
31     m_currentChildStackQueue.enqueue(currentChildStack);
32     m_nodesStatusQueue.enqueue(nodeStatus);
33 }
34 break;
35 ...

```

In the case of a `btNode::Running` status the `currentChildStack` pushes the current child index and both stacks are enqueued, as is the `nodeStatus`. Listing 5.6 displays the code for this case.

Listing 5.6: Code for the `btNode::Running` case

```

1 ...
2 case btNode::Running:
3 //enqueue stack and stuff
4 currentChildIndex = currentNode->currentChildIndex();
5 m_currentNodeStackQueue.enqueue(currentChildParentStackPair);
6 currentChildStack.push(currentChildIndex);
7 m_currentChildStackQueue.enqueue(currentChildStack);
8 m_nodesStatusQueue.enqueue(nodeStatus);
9 break;
10 ...

```

5.3.3. btProbSelectors

We have special cases for `btProbSelectors` because these need some extra handling in the scheduler. The special handling of this node type is done in stages one and three described in the previous section. The `btProbSelector` class uses a hash containing stacks to save its `probNodes` together with information regarding which child nodes have been visited.

In stage one we check if the `currentNode` is a `btProbSelectorNode` and if this is the case, we cast it. Next we check if we have run it before, and if so then we set the the information for that `btProbSelectorNode` by popping the stack which corresponds to the `currentNodeStack`. Listing 5.7 shows the code.

Listing 5.7: The special case code for a `btProbSelector` in stage one

```

1  ...
2  if (qobject_cast<btProbSelectorNode*>(currentNode))
3  {
4      //if the current node is a probselector, cast it
5      probSelector = qobject_cast<btProbSelectorNode*>(currentNode);
6
7      //and set the probnodes if we have run it before
8      if(m_visitedProbChildrenHash.contains(currentNodeStack))
9      {
10         probSelector->setVisitedProbNodes(m_visitedProbChildrenHash[currentNodeStack].pop());
11     }
12 }
13 ..

```

For stage three listing 5.8 shows how a `btProbSelector` is handled when it returns. For the case where the `btProbSelector` runs a child, we store the information regarding which children have been visited alongside the other information described in 5.3.2 on page 44.

If the `btProbSelector` succeeds or fails, and during the standard operations in stage three, the scheduler removes the current hash entry if the stack is empty.

Listing 5.8: The special case code for a `btProbSelector` in stage three

```

1  ...
2  case btNode::RunningChild:
3  ...
4      if(probSelector)
5      {
6          m_visitedProbChildrenHash[currentNodeStack].push(probSelector->visitedProbNodes());
7      }
8  ...
9  case btNode::Failed:
10 case btNode::Succeeded:
11 ...
12     if(qobject_cast<btProbSelectorNode*>(currentNode))
13     {
14         //if probselector then remove the probnodes for this stack
15         if(m_visitedProbChildrenHash[currentNodeStack].count() == 0)
16             m_visitedProbChildrenHash.remove(currentNodeStack);
17     }
18 ...

```

5.3.4. btParallels

The `btParallel` class also needs special handling in the scheduler because of the nature of it, where it runs every child in parallel. Again the special handling is done

in stage one and stage three, as with the `btProbSelector`. `btParallel` uses a multihash which uses the node stacks and entries and the list of `btNode::status` as values, for identifying and being able to stop the parallel execution when certain conditions has been met.

During stage one the scheduler first checks if the `btNode` is a `btParallel`. If this is case it is first cast and if the `btParallel` has not been run before a node stack and child stack is created for each child and pushed onto the queues together with a status of `btNode::None`. Furthermore a list of `btNode::status` is created and stored in the multihash. Next the scheduler sets the current list of `btNode::status` to the `btParallel` and it checks if it should terminate execution. Listing 5.9 shows the code for stage one.

Listing 5.9: Code for special casing of `btParallel` during stage one

```

1  ...
2  else if (qobject_cast<btParallelNode*>(currentNode))
3  {
4      //if the current node is a parallel, cast it
5      parallel = qobject_cast<btParallelNode*>(currentNode);
6
7      //if it is the first time, create execution stacks, child stacks etc. for each child and
          add them to the queue
8      if(nodeStatus == btNode::None)
9      {
10         QList<btNode::status*> childStatus = new QList<btNode::status*>();
11
12         for(int i = 0; i < parallel->childCount(); i++)
13         {
14             QStack<btNode*>* newStack = new QStack<btNode*>();
15             newStack->push(parallel);
16             newStack->push(parallel->child(i));
17             QPair<QStack<btNode*>*, QStack<btNode*>*> pair;
18             pair.first = newStack;
19             pair.second = currentNodeStack;
20             m_currentNodeStackQueue.enqueue(pair);
21
22             childStatus->append(btNode::None);
23             m_nodesStatusQueue.enqueue(btNode::None);
24
25             m_parallelNodeStatusHash.insert(newStack, childStatus);
26
27             QStack<int> childStack = QStack<int>();
28             childStack.push(0);
29             m_currentChildStackQueue.append(childStack);
30         }
31
32         m_parallelNodeStatusHash.insert(currentNodeStack, childStatus);
33         nodeStatus = btNode::Running;
34     }
35
36     //check if termination conditions is fulfilled, return btNode::Running if not
37     parallel->setRunningNodesStatus(m_parallelNodeStatusHash.value(currentNodeStack));
38     nodeStatus = parallel->conditionsFulfilled();
39 }
40 ...

```

In stage three the `btParallel` only has special handling in the case where it succeeds or fails, or one its children succeeds or fails. Listing 5.10 on the next page shows the code for stage three and lines 5 to 19 handles the case where a child of a `btParallel` succeeds or fails. If a child succeeds or fails the status list is updated with the child status, and the child's node stack and child stack is not enqueued again. Furthermore the entry in the `m_parallelNodeStatusHash` is removed and if there are no more entries in the `m_parallelNodeStatusHash` then

currentNodeStack is deleted. Because of this the scheduler needs to check if the currentNodeStack exists before trying to enqueue it during stage three.

Lines 21 through 32 handles the case where the currentNode first stops the execution of btParallel and also removes the entry in m_parallelNodeStatusHash if the currentNodeStack is not NULL. Finally it deletes the list of btNode::status.

Listing 5.10: Code for the special casing of btParallel during stage three

```

1  ...
2  case btNode::Failed:
3  case btNode::Succeeded:
4  ...
5      if(currentParent != NULL && qobject_cast<btParallelNode*>(currentNode))
6      {
7          //if the parent is a parallel, set the status for that parallel.
8          parallel = qobject_cast<btParallelNode*>(currentParent);
9          QList<btNode::status*> m_parallelNodeStatus = m_parallelNodeStatusHash.value(
10             currentNodeStack);
11             m_parallelNodeStatus->replace(parallel->childNodeIndex(currentNode), nodeStatus);
12             m_parallelNodeStatusHash.remove(currentNodeStack, m_parallelNodeStatus);
13
14             if(m_parallelNodeStatusHash.count(currentNodeStack) == 0)
15             {
16                 //remove the stack if there is no other node status in the hash
17                 delete currentNodeStack;
18                 currentNodeStack = NULL;
19             }
20
21             if(qobject_cast<btParallelNode*>(currentNode))
22             {
23                 //if parallel stop the children and remove the node status list from the hash
24                 stopParallelExecution(currentNode, currentChildParentStackPair.second);
25
26                 if(currentNodeStack)
27                 {
28                     QList<btNode::status*> status = m_parallelNodeStatusHash.value(currentNodeStack
29                        );
30                     m_parallelNodeStatusHash.remove(currentNodeStack, m_parallelNodeStatusHash.
31                        value(currentNodeStack));
32                     delete status;
33                 }
34             }
35         }
36     }
37 }

```

5.4. Conclusion

During the first implementation of SMARTS Library we encountered problems during runtime. We analyzed the problems, and we first discovered that there would be a potentially huge number of context switches when traversing the tree structure. Furthermore threading was a huge problem due to e.g parallels. A re-design of SMARTS Library was necessary to accommodate these problems and different scheduler options were purposed for solving these problems. A scheduler using queues and stacks was chosen for implementation as while it was more complex to implement, it had a number of advantages over the other options. The new scheduler uses queues, stack and hashes from the Qt framework and can run every parallels children using a form of simulated concurrency.

Perception

When creating behaviors for a game character, one problem which continuously surfaces is how this character perceives information in the world it inhabits. Daniel Sanchez-Crespo Dalmau[13] describes the structure of game AI systems as having four parts:

- A sensor or input system
- A working memory
- A reasoning/analysis core
- An action/output system

Where Behavior Trees as implemented by SMARTS Library supply the last three (the memory being the Character, the reasoning/analysis core being the structure of the Behavior Tree itself and the action/output system being the behaviors which affect changes in the world, tentatively named Actions), the first of these is not as well defined. However, since the Character is used to provide information for Conditions[10], which are a special type of leaf behavior node used for controlling access to other behaviors[16, p.10], a system can be devised which provides the character with a more varied view of the world.

The Conditions need a way of getting information about the world as seen through the eyes of the character whose behavior is being dictated by the current run through the behavior tree. Thus we arrive at the following proposal: Create a perception system which can provide this information with varying amounts of precision. This varying degree of precision is important when modelling characters who should be believable; their ability to not know everything about the world precisely makes them capable of performing behavior in a more believable manner than if you can simply see everything precisely - in short they will act less like the proverbial robot.

6.1. Inspiration

Previous work in the area of perception in computer game AI is wide spread and greatly different in style and complexity. To investigate the cause of the different complexities, this section presents the context of some of the perception systems. The following is based on information from an article by Tom Leonard on Gamasutra[19].

6.1.1. Pac-Man™

The simplest kind of perception in a game AI system is that exemplified by Namco's Pac-Man™: This type of system has exact knowledge of the entire game world, and is essentially all-knowing. This makes it easy to implement the perception system, since no calculations are required to limit the knowledge any character has of the world. It is suited to simplistic games or purely logic based games, in which the characters are expected to behave in a mechanical fashion.

6.1.2. Quake™

In id Software's successful first person shooter Quake™, the AI characters need to know where the player character is. They do this by first checking if the player is inside a certain distance from them; the limit of their perception. Then a line of sight check is performed, to ascertain whether the player is in fact visible to the AI character. This check is only performed if the player is inside the AI character's view cone (their field of view).

If the player is visible, the AI character knows the following:

- Where the player is and where he is looking
- The geometry of the surroundings
- Which weapons the AI character is using and which the player is using

Importantly, in Quake™ all this information is absolute: The AI either knows the information or it does not. This means that the calculations involved are minimal, and the processing time can be spent on drawing items on the screen.

This is of course important in a game which originally was doing real 3D calculations on a desktop-class processor, rather than what is prevalent today; i.e. doing graphics on a dedicated processing unit.

As such, the Quake™ AI characters always operate with precise information, but unlike Pac-Man™ this information is limited in scope - the AI does not have information on the player that it is not in fact able to sense (that is, it has no information on the player when the player is not in line of sight).

6.1.3. Thief: The Dark Project™

In the first person stealth based game Thief: The Dark Project™ the AI characters need to be able to sense their surroundings in a much more sophisticated way than those in Quake™.

The game is based around the player sneaking around, attempting to not be seen by guards. This requires the sensory system to be able to provide the AI characters with more granular information on the player. The basic concept is similar to that in Quake™, however multiple passes are performed on the sensing. The diagram in Figure 6.1 on the facing page shows an example of how the system employs multiple view cones to achieve varying levels of information certainty:

The five view cones are numbered in order of importance. This means that an object in position A will be evaluated through view cone number 3, whereas an

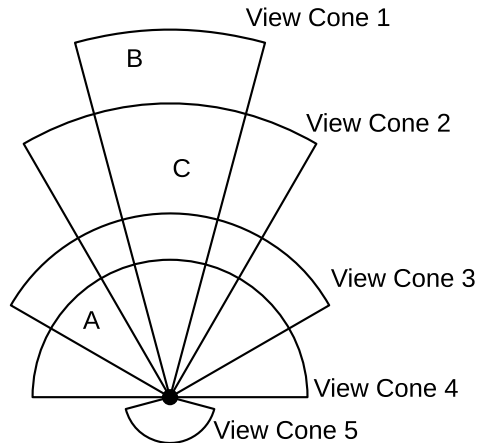


Figure 6.1.: The view cones of an AI character in Thief: The Dark Project™

object is evaluated with view cone 1 irregardless of whether an object is in position B or C.

This is a very high density information gathering system, which requires a not inconsiderable amount of computation when sensing the world. In other words: This type of system is only applicable in games where correct perception is highly important to the game-play.

6.1.4. Age of Empires™ and Warcraft™

The sensory system of the main AI character in a strategy game such as Age of Empires™ is fundamentally different to that of those in the previously mentioned games, in that the character has no physical representation in them. In stead it needs a varied collection of more esoteric information, as described by Dalmau[13]:

- What is the balance of power in each subarea of the map?
- How much of each type of resource do I have?
- What is the breakdown of unit types: infantry, cavalry, and so on?
- What is my status in terms of the technology tree?
- What is the geometry of the game world?

In other words, there are no simple line-of-sight tests to be done here, but rather it is a question of more general information about the state of the game world itself. In Age of Empires™ this information is contained in a memory base which does not decay over time, in other words the AI character has photographic memory.

Other strategy games employ fading memory, or varied memory states for different pieces of information: For example in Warcraft™ you can explore the level to gather information about the geometry of the game world, however once the reconnaissance unit leaves an area, the information enemy on units occupying the area is lost - that is it is immediately decayed - but the game world geometry is retained.

6.2. Design

This section describes the design of the general perception system implemented for use with SMARTS.

6.2.1. Setting Up the Criteria

After looking through four approaches to perception systems as found in the five games presented above we can describe a number of concepts which should be provided by a general perception system such as that which we propose the creation of, so as to allow for the creation of the widest possible array of game types:

Each item of perceived information, which shall be denominated a **perception atom**, should be available with variable precision. If a perception atom is known at 100% certainty, looking it up should not incur any performance penalties.

Any perception atom must be able to degrade over time, with a variable degree of entropy. This simulates **imperfect memory**. Based on the idea that some pieces of information should stay, while other memories should fade, the system should be able to support perfect memory at no penalty.

To simplify calculations, the system should be able to support three levels of precision:

- Perfect memory, which does not fade over time.
- An intermediate, Boolean level of precision, meaning in sort that it should be able to retain perfect memory, until an exact time when the entirety of the information is lost.
- Imperfect memory, which fades in precision over time, optionally fading entirely.

Positional information is that which is relevant to a particular position in the world. To find out whether a piece of information is available to a character, the character's position and orientation in space is combined with a geometric volume to create the **perception limit** as used in Quake™, and then any number of **view cones** can be applied.

The perception volume and all view cones have any number of values assigned, correlated to each perception atom, describing the precision at which each atom can be decided. This is done using a similar style of prioritisation as seen in Thief: The Dark Project™, with the addition that the system is usable with only the perception volume.

As a final note on this subsystem is the technicality that since level geometry is known, it should be possible to define a view volume larger than the level geometry, in which case the calculation to assess whether objects are inside the volume is not performed. The initial assessment is done on level load. This allows for a level of effectiveness similar to that seen in Pac-Man™.

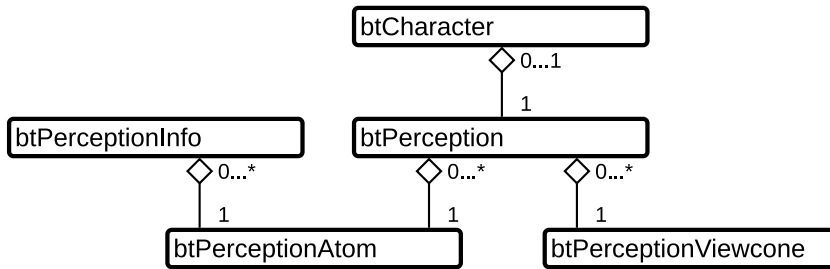


Figure 6.2.: A class diagram of the classes in the perception system

6.2.2. Library Design

The library as created should fit into the design of the existing SMARTS library. As such it would make sense to have all perception checks go through the class `btCharacter`. This, however, should very importantly not mean that the class would be bloated with large amounts of new functionality, but rather that the perception system should be accessed through the character.

The design is established in depth below, and is organised like so: The perception system which the `btCharacter` class contains an instance of is called `btPerception`, and this class in turn contains any number of references to perception atoms as described by the interface `btPerceptionAtom`. The perception atom class instances request their information from instances of `btPerceptionInfo`, which contain the actual information about the world. A class diagram describing this can be seen in Figure 6.2.

`btPerception` contains functions which allows the user of the system to request details on perception atoms through the character. This user can be either manual users by way of coders, or designers creating behavior trees who can check perception atoms through decorators and Conditions. The check is done either by name, or by pointer to the perception atom in question.

The `btPerception` class allows the perception atoms to request information on the character's position and orientation in the world (as provided by the game engine), and also contains the perception limit and view cone information. For reasons of easy calculation, the perception limit volume is described as a sphere centered on the character. The view cones are represented as pyramids, for similar reasons. Additionally, this makes each item easier to describe visually, which is something of which the relevance will become obvious later.

The information available to the full system is stored in `btPerceptionInfo` instances, and the value as perceived by each character is then stored in the instances of `btPerceptionAtom`. The storage per character only happens when the request for the information is made. Each `btPerceptionInfo` then informs the connected `btPerceptionAtom` instances of any changes in its value. It is then up to the `btPerceptionAtom` to decide when and how to update its internal knowledge next.

The perception limit volume and view cones are defined using a system like that seen in a 2D version in Figure 6.3 on the following page. As the sample data in

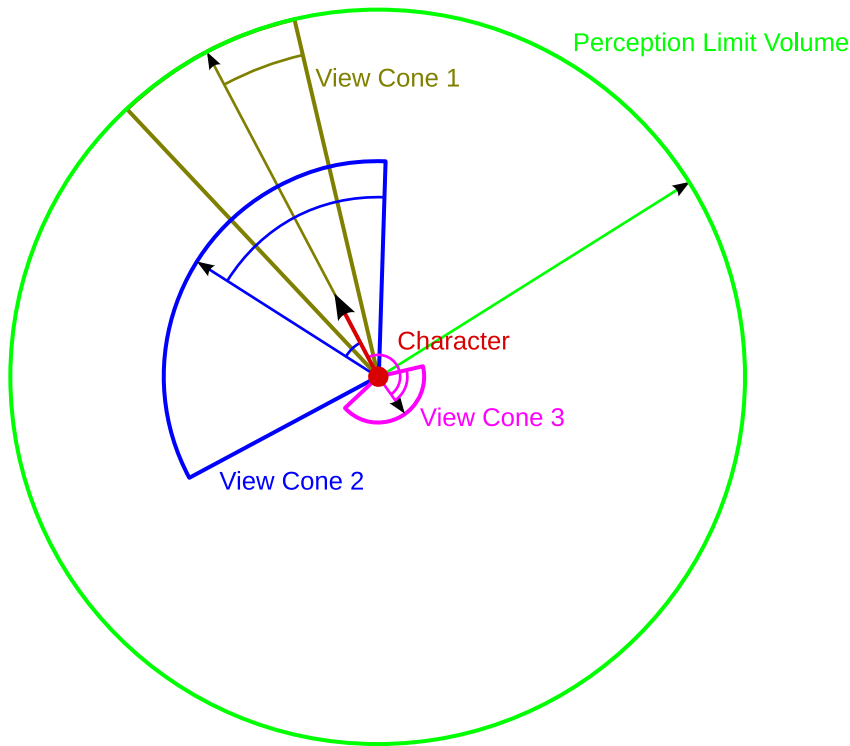


Figure 6.3.: A hypothetical set of perception limit data for an AI character in SMARTS. The character has three view cones: View cone 1 is oriented in the same direction as the player, its radius the same as the perception limit volume, and the only value changed from the defaults is the angle of extent. View cone 2 is rotated to the left, is a lot wider and has its own radius. View cone 3 provides the spidey-sense described in *Thief: The Dark Project™*, and is similarly value changed to view cone 2, though rotated 180 degrees from the player's orientation.

the figure indicates, these two pieces of information are represented as follows: The perception limit volume is represented simply by a radius, and each view cone is represented by a radius and two angles (one for extent and one for deviation from player orientation). To extend this into three dimensions, one only needs an additional angle to define the view cones' extent in the third dimension, and an extra angle to define its elevation.

6.3. Implementation Details

In this section we will describe the implementation of the perception system in SMARTS Library. We will start by describing the `btPerception` class, as this is the main container for the other perception elements. This is followed by a description of `btPerceptionViewcone` and lastly both of the classes `btPerceptionAtom` and

`btPerceptionInfo` are described together, as they are heavily interconnected.

6.3.1. `btPerception`

This class is used as a data container and bonds `btCharacter` and the other perception components together.

Listing 6.1: An excerpt of the header for the `btPerception` class

```

1 class btPerception : public QObject
2 {
3 ...
4 /**
5  * How precise the knowledge of an item found inside the perception limit is, if it is not
6  * inside a view cone. This defaults to 0, as the perception limit is normally used
7  * only as a potentiality check. If you set this above zero, a perception atom will be
8  * given this precision if it is found inside the perception limit, but not inside a view
9  * cone.
10  * 0 is no precision at all, 1.0 is full precision
11  */
12 qreal knowledgePrecision() const;
13 void setKnowledgePrecision(const qreal& newKnowledgePrecision);
14
15 /**
16  * The limit of the character's perception. If left at 0, the limit will be the entire
17  * level.
18  * Please note that this can potentially be very expensive, depending on level size.
19  */
20 qreal perceptionLimit() const;
21 void setPerceptionLimit(const qreal& newPerceptionLimit);
22
23 QList<btPerceptionViewcone> viewCones() const;
24 void setViewCones(const QList<btPerceptionViewcone> newViewCones);
25 void addViewCone(btPerceptionViewcone * viewcone);
26
27 QList<btPerceptionAtom*> perceptionAtoms() const;
28 btPerceptionAtom* perceptionAtom(const QString& name) const;
29 void addPerceptionAtom(btPerceptionAtom * perceptionAtom);
30 ...
31 };

```

Listing 6.1 shows an excerpt of the header file of the class `btPerception`. This class firstly contains information regarding the precision of the knowledge available, together with the perception limit for the given `btCharacter`. Besides these two values, `btPerception` also contains two lists where one contains instances of `btPerceptionViewcone` and the other contains `btPerceptionAtom` instances. By doing this, the perception is separated from `btCharacter`.

6.3.2. `btPerceptionViewcone`

The struct `btPerceptionViewcone` only contains public values, which are used for setting both the range, angle and radius of the view cone. Furthermore it is possible to set how precise the knowledge about information discovered inside the view cone's area is, and the position of the view cone in the list of instances of `btPerceptionViewcone` on `btPerception`.

Listing 6.2: The public variables on `btPerceptionViewcone`

```

1 struct btPerceptionViewcone
2 {
3 ...
4 /**

```

```
5      * The horizontal angle by which the view cone is offset from the character's
6      * orientation
7      * Note: This should be between -180 and 180
8      */
9      qreal offsetAngleHorizontal;
10     /**
11     * The vertical angle by which the view cone is offset from the character's orientation
12     * Note: This should be between -180 and 180
13     */
14     qreal offsetAngleVertical;
15     /**
16     * The horizontal angle of extent of the view cone
17     * Note: This angle is from the center to one side, not from side to side
18     * Note: This should not be larger than 180
19     */
20     qreal extentAngleHorizontal;
21     /**
22     * The vertical angle of extent of the view cone
23     * Note: This angle is from the center to top, not from top to bottom
24     * Note: This should not be larger than 180
25     */
26     qreal extentAngleVertical;
27     /**
28     * The radius of the view cone.
29     * If this is left at 0, the limit used will be the character's perception limit.
30     * Note: This should be no larger than the radius of the perception limit
31     */
32     qreal radius;
33     /**
34     * The precision of knowledge discovered in this view cone's area of perception
35     * 0 is no precision at all, 1.0 is full precision
36     */
37     qreal knowledgePrecision;
38     int position;
39 };
```

Listing 6.2 above shows an excerpt of the header file, which only contains public variables. The position of the `btPerceptionViewcones` in the list also specifies in which order these would be checked during execution.

6.3.3. `btPerceptionAtoms` and `btPerceptionInfo`

In a game world there exists information on different objects, which the AI wants know. Rather than having to search through all the objects in the game, the classes `btPerceptionAtoms` and `btPerceptionInfo` are introduced to supply this functionality in a way which allows the programmer to only have to query specific objects in the game world.

Listing 6.3: A pure virtual function and signals in `btPerceptionInfo`

```
1 class btPerceptionInfo : public QObject
2 {
3 ...
4     public:
5 ...
6         virtual QVariant getAdjustedValue(qreal precision) const = 0;
7
8     Q_SIGNALS:
9         void infoUpdated();
10        void positionUpdated(QVector3D);
11        void radiusUpdated(qreal);
12 ...
13 };
```


The class `btPerceptionInfo` is used for specifying information on a game object. Furthermore it is to be used as a super class and thereby the subclasses are customizable to fit the needs for that given game object. Listing 6.3 on the facing page shows the pure virtual `getAdjustedValue` which must be overridden in every subclass. It returns a value, adjusted according to the given precision. Furthermore there are three signals, where the `infoUpdate` is always connected, while the two others are optional. `infoUpdate` is emitted when the information has changed and the `btPerceptionInfo` subclass sees fit to emit it.

Listing 6.4: The `setPerceptionInfo` and `perceptionInfoUpdated` used for connecting to a `btPerceptionInfo` and getting its value

```

1  ...
2  void btPerceptionAtom::setPerceptionInfo(btPerceptionInfo* newPerceptionInfo)
3  {
4      disconnect(this, SLOT(perceptionInfoUpdate()));
5      d->perceptionInfo = newPerceptionInfo;
6      connect(d->perceptionInfo, SIGNAL(infoUpdated()), this, SLOT(perceptionInfoUpdated()));
7  }
8  ...
9  void btPerceptionAtom::perceptionInfoUpdated()
10 {
11     if(d->shouldUpdate)
12         d->knowledge = d->perceptionInfo->getAdjustedValue(d->precision);
13 }
14 ...

```

To be able to access the information in `btPerceptionInfo` it should be connected to a `btPerceptionAtom`. A `btPerceptionAtom` instance is added to a list on a given `btPerception` instance, where each `btPerceptionAtom` class points at a `btPerceptionInfo`, which is set using `setPerceptionInfo` shown in the listing above. This function first disconnects from a `btPerceptionInfo`, then sets the pointer and finally connects to the new instance.

`perceptionInfoUpdated` is the slot which `btPerceptionInfo`'s `infoUpdated` is connected to. When `infoUpdated` is emitted, `btPerceptionAtom` checks if it should update its information before updating its knowledge by through the `getAdjustedValue` function on `btPerceptionInfo`.

6.4. Conclusion

The SMARTS Library provides a way for creating game AI, but this was without any possibility of letting an AI character perceive the world through the SMARTS Library. Therefore we wanted to create a perception system in SMARTS Library which could provide that functionality.

We started out by looking at different games and how the AI perceived the game world. The Pac-Man™ AI had exact knowledge of the game world, while Quake™ used a perception limit together with a line of sight and exact information about the player, if the player is within the perception limit and visible, and the surrounding geometry. Thief: The Dark Project™ used several view cones for perceiving the world which in turn could be very expensive, while in strategy games as Age of Empires™ and Warcraft™ the AI does not use line of sight or view cones, but instead has a view on the state of world itself together with the world geometry.

After looking at how some games uses perception a perception system was designed. All perceived information in the world is a **perception atom** which has

to possibly fade over time, never fade or have a boolean level of precision, and is implemented as a `btPerceptionAtom` class. Furthermore the `btPerceptionAtom` points at the designated piece of info in the game world as a `btPerceptionInfo` which has to be reimplemented to customize it to a given game object and returning the correct value. Also a **perception limit** is used, which Quake™ also uses, and the ability of adding multiple **view cones** as done in Thief: The Dark Project™, which is implemented as a `btPerceptionViewcones` class.

It is also decided that the system should be as separated from `btCharacter` as possible, so as not to fill this class with a lot of unnecessary functionality. Therefore `btPerceptionAtom`, `btPerceptionInfo` and `btPerceptionViewcone` are contained in a `btPerception` class, which also contains functionality for defining the perception limit. Thus was created a functioning general perception system for SMARTS Library.

An important thing to note about the perception system described in this chapter is that it is not a drop-in system which is readily usable, but rather a system where as much boiler-plate code has been created, allowing for a smooth, consistent implementation of perception in a game using SMARTS. What this means is that you cannot simply add the perception system and use it in a game:

Like the behavior trees in SMARTS where it is left to the makers of games to build the actions themselves, one must implement the methods for gathering information from the world. What the perception system then does is make it easy to propagate this information in a consistent manner and use it in the behavior trees. This also means that a game programmer can access the information available to a character and see directly what an AI character would have available, and thus use this, presenting the world in an imperfect manner to the players of their games.

SMARTS in Gluon

In late 2009 a very simplistic proof of concept implementation of using SMARTS in the Gluon engine was created[16]. Based on this work it was decided that it was possible to perform this integration of SMARTS into Gluon. This chapter thus describes the work to create that integration.

One very important aspect of the Gluon engine (see also Appendix A on page 121) is the ability to distribute games without compiling anything. So, to enable SMARTS to be used properly in GluonEngine based games, expansion done per-game must be doable through scripts. As such, a way of providing a scripting system for extending SMARTS in Gluon Creator has to be created.

This chapter contains first a short analysis of how the existing functionality in Gluon is implemented, followed by a design of the method by which SMARTS is to be implemented into a similarly designed system. After this in Section 7.3 on page 63, certain interesting details of the implementation of this design are explained.

7.1. Analysis

As described in Section A.2 on page 123, GluonEngine uses a technique called Components when functionality is to be implemented. Since we will be building a system in which several discrete objects will be working together to provide functionality, we shall look at a similar system found in the engine already: The sound handling components. After this we go through a quick reminder of the SMARTS Library layout.

7.1.1. Sound in Gluon

The sound files available to a game are represented by the `SoundAsset` subclass of `Asset`. This class loads the sound file on request, and provides the raw data to Components in the game. The sound playing functionality itself is split into two components:

`SoundEmitter` represents a single sound in the game, and is attached to `GameObjects` which are making the sound. They are connected to a `SoundAsset`, and have an extent, a volume, and a fade type (which determines how the sound fades over distance; linear, logarithmic...).

`SoundListener` is applied to the `GameObject` which represents a player character in the game. One `SoundListener` can be active at any one time, and the sounds which can be heard at the position of the `GameObject` it is attached to are the ones which will be played through the speakers, at appropriately faded volume levels, according to the settings in the `SoundEmitter`.

What this means is that we end up with a system, which is connected like so: A `SoundAsset` can be referenced by any number of `SoundEmitter`. When the `SoundEmitter` is told to make a sound, its volume is set according to the position of the currently active `SoundListener`. The playback volume is then reset on each update, to account for changes in the game world.

7.1.2. The SMARTS Layout

As seen in Figure 5.1 on page 39, SMARTS is laid out in such a way that all the behavior trees for a game are contained in a `btBrain`. Any number of game characters, represented by the `btCharacter` class, can then use one of these behavior trees as the basis for their reasoning.

The perception system described in Chapter 6 on page 51 is then contained within the `btCharacter` class, and is made up of the `btPerception` class, which contains perception limit information, as well as information atoms as instances of `btPerceptionAtom`, which reference bits of information found scattered throughout the world, represented by the `btPerceptionInfo` instances.

7.2. Design

Based on the short analysis of the sound system in Gluon, and the layout of the SMARTS system itself, we can now describe a design for the Components and Assets needed to allow for SMARTS to be used fully in GluonEngine based games.

The most straight forward item of the design is the assets. This has not changed from the proof of concept implementation[16, p.50], and as such this new design will be structured in the same way as that: One asset, `BehaviorTree::Asset`, which contains a number of automatically created `BehaviorTree::Tree` instances, in accordance with the trees contained in the brain.

Similarly, the representation of the character is done through the component `BehaviorTree::Character`, which contains a reference to one of the assets representing a `BehaviorTree::Tree`.

The perception system is where the new design will diverge. Firstly the Character component above is amended with an instance of the `btPerception` class, which must then be filled out with information about its perception limit, view cones and related perception atoms. Only one of these is required (the perception limit), and everything else being optional is added through the following set of new components:

The `BehaviorTree::ViewCone` component represents a single view cone, and through a set of properties contains the information described in Section 6.2.2 on page 55 (such as extent and information precision of information in the cone). To represent multiple view cones for a single character, one simply adds multiple instances of this Component to the `GameObject` the `BehaviorTree::Character` is attached to. The order in which they are added to the Character defines the hierarchy when checking for information precision.

The `BehaviorTree::PerceptionAtom` component represents an instance of the `btPerceptionAtom` class and points to a `BehaviorTree::PerceptionInfo` com-

ponent connected to a `GameObject` somewhere in the scene. This is done to make the link between a character and specific pieces of perception information explicit.

7.2.1. Scriptability

Importantly, all of the above items contain only little logic of their own. Since their core functionality is dependent on the game in which they exist, this part cannot be generalized, and as described in Section A.2 on page 123 all the assets which make up a game must be of a nature by which they are usable without compiling. So, each of the items above implement functionality similar to that found in `GluonEngine::ScriptableComponet`.

7.3. Implementation Details

In this section we will describe how the SMARTS Library is connected to Gluon, and the most important implementation details are shown.

7.3.1. Registration with Gluon

We start out by describing some of the required macros for registering components and assets. The hierarchy and correlation between components and assets in Gluon has been described in “Where Game AI Meets Academic AI” [16, p. 21] and therefore we will not get into details. But to be able to register components and assets, there are a few macros needed which will be described before continuing with the implementation details. Not all of the classes use all macros.

REGISTER_OBJECTTYPE This macro registers the class for dynamic instantiation and creates a set standard functions which are needed if the class inherits from `GluonObject`, and can not be added in the conventional way.

GLUON_OBJECT This macro needs to be used in addition `Q_OBJECT` whenever a class inherits from `GluonObject` which all assets and components do.

Q_INTERFACES It is necessary to tell Qt which interfaces the class implements, as this is required when implementing plugins.

Q_DECLARE_METATYPE This macro has to be used twice, one for stack allocations and one for heap allocations, such that the class can be used in a property and as a type in Gluon.

Q_EXPORT_PLUGIN2 For exporting the class to the plugin, this macro is used.

Q_INVOKABLE Whenever it should be possible to call a function from `QMetaObject` or from `QScript`, this macro is necessary.

The uses of the macros described above will only be shown once as this would be the same code in most of the classes.

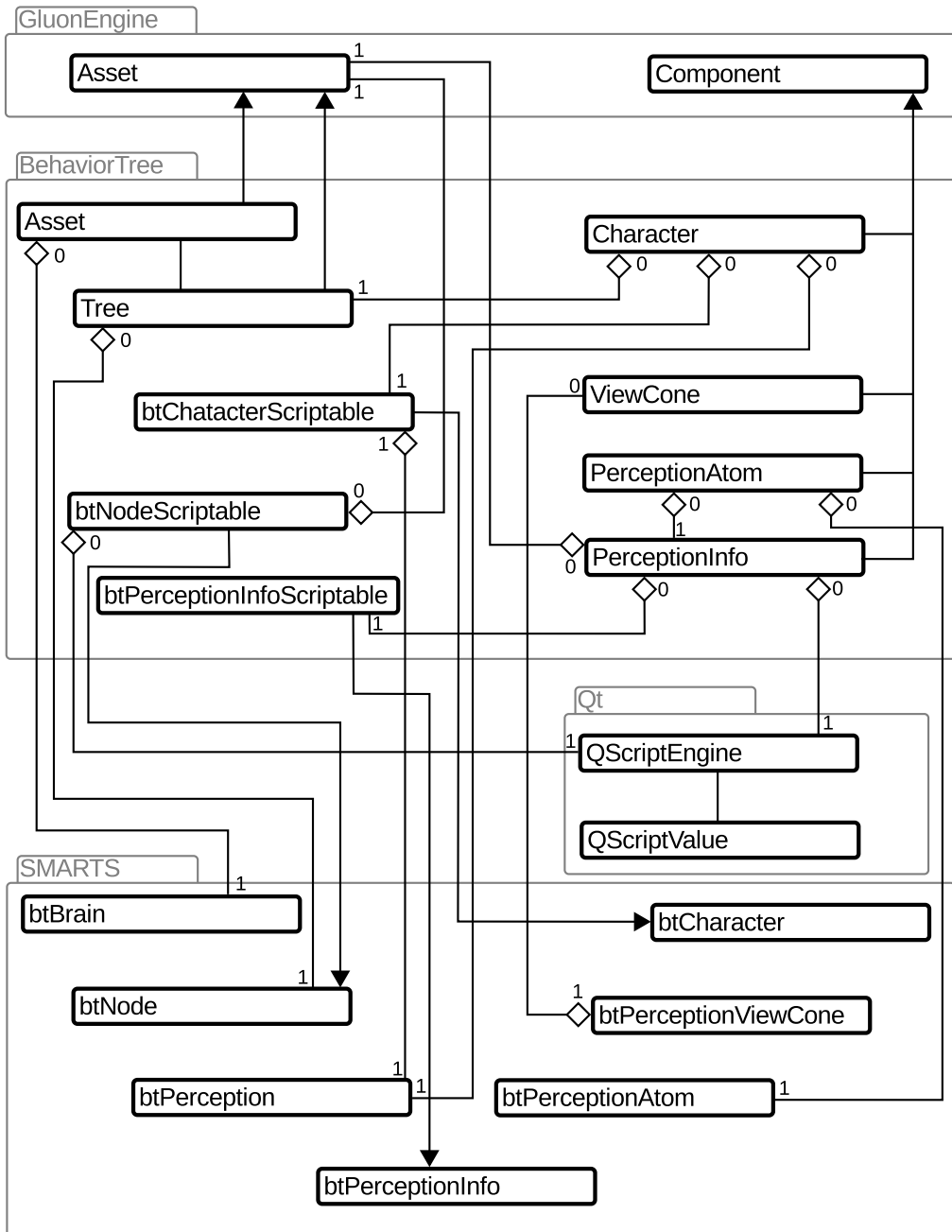


Figure 7.1.: The class diagram showing the relations of the classes in the Gluon SMARTS Library

7.3.2. Behavior Trees in Gluon

Behavior trees from SMARTS Library can be used in Gluon through the `Asset` class which inherits from `GluonEngine::Asset`, as shown in figure 7.1. This class also needs to be exposed to Gluon and thus it uses the macros from which were just presented.

Listing 7.1: Excerpt from the `Asset` header file

```

1  ...
2  class BTCOMPONENT_EXPORT Asset : public GluonEngine::Asset
3  {
4  ...
5      GLUON_OBJECT(BehaviorTree::Asset)
6      Q_INTERFACES(GluonEngine::Asset)
7  ...
8  Q_DECLARE_METATYPE(BehaviorTree::Asset)
9  Q_DECLARE_METATYPE(BehaviorTree::Asset*)
10 ...

```

The listing above shows the header file to illustrate how the macros are used, if they are used for a class. `GLUON_OBJECT` is used to register `Asset` as a `GluonObject` while also using the `Q_INTERFACES` for declaring the interface for this plugin. At the bottom the `Q_DECLARE_METATYPE` is used twice for being able to use `Asset` both as a `QVariant` type that can be used on the stack and on the heap.

Listing 7.2: The use of `REGISTER_OBJECTTYPE`

```

1  ...
2  REGISTER_OBJECTTYPE(BehaviorTree, Asset)
3  ..

```

Listing 7.2 shows the registration macro for creating some of the standard functions in a `GluonObject`.

Listing 7.3: `setFile` used for setting the xml created by the SMARTS Designer

```

1  ...
2  void Asset::setFile(const QUrl &newFile)
3  {
4  ...
5      const QObjectList& oldChildren = children();
6      QList<Tree*> newChildren;
7      for(int i = 0; i < newBrain->behaviorTreesCount(); ++i)
8      {
9          Tree* newTree = new Tree(this);
10         this->addChild(newTree);
11         newTree->setBehaviorTree(newBrain->getBehaviorTree(i));
12         newTree->setName(newTree->behaviorTree()->name());
13         newChildren.append(newTree);
14     }
15
16     // Run through all old children
17     foreach(QObject* oldChild, oldChildren)
18     {
19         Tree* theNewChild = NULL;
20         Tree* theOldChild = qobject_cast<Tree*>(oldChild);
21         // Find a tree with the same name in the new children
22         foreach(Tree* newChild, newChildren)
23         {
24             if(newChild->name() == theOldChild->name())
25                 theNewChild = newChild;
26         }
27         // Tell old child that new child is the tree that should be used
28         // If no new child could be found, inform the oldChild that it should be removed
29         emit theOldChild->treeChanged(theNewChild);

```

```
30     }
31     ...
32     GluonEngine::Asset::setFile(newFile);
33 }
34 ...
```

Listing 7.3 above shows `Asset::setFile`, which takes a file as argument. It starts by loading the file into a `btBrain` from SMARTS Library, but this is not shown in the listing. Afterwards it runs through all the behavior trees in the `btBrain` and creates new instances of `Tree` which is container for a single behavior tree. Next the function runs through both old and new instances of `Tree` to be able to set a behavior tree which already exist but is in another instance of `Tree` or for resetting and removing a nonexistent behavior tree. As the last thing the new file is set as the current file in the super class `GluonEngine::Asset`.

Listing 7.4: `supportedMimeTypes` used for specifying the supported types of `Asset`

```
1 ...
2 const QStringList Asset::supportedMimeTypes() const
3 {
4     QStringList list;
5     list.append("application/xml");
6     return list;
7 }
8 ...
```

It is necessary to specify which kind of files are supported by `Asset`, and this is done with the `supportedMimeTypes` function shown above. The function just returns a list of strings which specify the types, and in this case `Asset` only supports XML files.

Listing 7.5: The only place in SMARTS Library plugin where `Q_EXPORT_PLUGIN2` is used

```
1 ...
2 Q_EXPORT_PLUGIN2(gluon_plugin_asset_behaviortree, BehaviorTree::Asset)
3 ..
```

Listing 7.5 shows the macro `Q_EXPORT_PLUGIN2` being used in `Asset`, and this is the only place in the SMARTS Library plugin it is used.

When SMARTS Library is used separately the tree nodes are implemented in C++, but because Gluon Creator uses scripts to let the game programmers create the game logic, this has to be supported by the behavior tree components. To achieve this a subclass of `btNode` is used, called `btNodeScriptable`, together with marking a part of the functions in `btNode` with `Q_INVOKABLE` to let them be used in scripts.

Usually `btNodes` are instantiated during the parsing of the XML file, but the `btNodeScriptable` instances are created in Gluon Creator when a game starts.

Listing 7.6: The function used when instantiating a `btNodeScriptable` in Gluon Creator

```
1 ...
2 void btNodeScriptable::setScriptAsset(GluonEngine::Asset * asset)
3 {
4     d->script = asset;
5
6     d->script->load();
7 }
```



```

8   if (d->script->data()->hasText())
9   {
10      d->engine.evaluate(d->script->data()->text(), this->className());
11
12      ...
13
14      QScriptValue btnode = d->engine.newQObject(this);
15      d->engine.globalObject().setProperty("btnode", btnode);
16  ...
17      d->runFunc = d->engine.globalObject().property("run");
18  }
19 }
20 ...

```

The listing above shows `setScriptAsset` which is called during the instantiation in Gluon Creator. This function sets the scripts and loads them by using `QScriptEngine`. Furthermore `QScriptEngine` is also used for registering the class as a script object such that it is accessible from within the script. It uses `QScriptValue` to save the run function, which in turn is called from the overridden run function from the `btNode`.

Listing 7.7: `btNodeScriptable`'s run function

```

1  ...
2  btNode::status btNodeScriptable::run(btCharacter* self)
3  {
4      QScriptValue character = d->engine.newQObject(qobject_cast<btCharacterScriptable*>(self));
5      d->engine.globalObject().setProperty("character", character);
6
7      if(d->runFunc.isFunction())
8      {
9          d->runFunc.call();
10         if (d->engine.uncaughtException().isValid())
11         {
12             ...
13         }
14     }
15 }
16 ...

```

Listing 7.7 shows the run from `btNodeScriptable` which calls the function `runFunc` as saved in a `QScriptValue`. Furthermore it also registers the instance of `btCharacterScriptable` in the script, such that programmers would be able to access it during execution.

7.3.3. Behavior Tree Execution in Gluon

When trying to execute a behavior tree from the SMARTS Library, an instance of `btCharacter` is needed. The `Character` component is created for that purpose and is essentially a wrapper on top of a `btCharacter` which enables setting view cones and information on the `btPerception` instance which is created by the `btCharacter` instance. Furthermore the use of a scripting language in Gluon, it should also be possible to access a `btCharacter` instance in the scripts.

To be able to do this the `Character` points not to a `btCharacter` but instead on a subclass of `btCharacter` called `btCharacterScriptable`. This class adds `Q_INVOKABLE` on a number of virtual functions from `btCharacter`. By doing this the functions are accessible in a script. It also overrides some of the functions and redirects the calls to the `Character` component's `GameObject` as this class is used for querying and setting some of the values, which would be set on the `btCharacter` if SMARTS Library was used without Gluon.

The Character component is used by adding it to a Gluon GameObject.

Listing 7.8: A recursive function which is called when GluonEngine starts the game

```
1 ...
2 void Character::initScriptNodes(btNode* node)
3 {
4     for (int i = 0; i < node->childCount(); i++)
5         initScriptNodes(node->child(i));
6
7     if (node->type() == btNode::UnusableNodeType)
8     {
9         QList<GluonEngine::Asset*> assets = GluonEngine::Game::instance()->gameProject()->
10             findChildren<GluonEngine::Asset*>();
11
12         foreach(GluonEngine::Asset* asset, assets)
13         {
14             if (asset->metaObject()->className() == "ScriptAsset" && node->className() == asset
15                 ->name().left(asset->name().lastIndexOf(".")))
16             {
17                 btNodeScriptable * newNode = new btNodeScriptable();
18                 ...
19                 for(int i = 0; i < node->childCount(); i++)
20                 {
21                     newNode->appendChild(node->child(i));
22                     node->child(i)->setParentNode(newNode);
23                 }
24
25                 for(int i = 0; i < newNode->childCount(); i++)
26                 {
27                     node->removeChild(i);
28                 }
29
30                 btNode* parent = node->parentNode();
31                 parent->removeChild(node);
32                 parent->appendChild(newNode);
33                 newNode->setParentNode(parent);
34
35                 newNode->setScriptAsset(asset);
36
37                 delete node;
38             }
39         }
40     }
41 }
42 ...
```

When Gluon Creator starts the game calls the function `start` on the current scene's root GameObject which in turn calls the same function on each of the attached components and all its child GameObjects. The listing above shows `initScriptNodes` in `Character`, which is called from the component's `start` function. The function is recursive and checks if `btNode` could be scriptable. If this is the case it creates a `btNodeScriptable` instance and copies all of the information from the `btNode`, and then moving both parent and children to the `btNodeScriptable`. Then it sets the script and deletes the old node. This is done throughout the behavior tree.

Listing 7.9: The `Character::think` function

```
1 ...
2 bool Character::think()
3 {
4     ...
5     if (tree())
6     {
7         if (tree()->behaviorTree())
```

```

8      {
9          d->self->think();
10     ...
11     }
12     else
13     ...
14     }
15     else
16     ...
17 }
18 ...

```

Listing 7.9 on the preceding page shows the `think` which in turn calls the `think` function of the `btCharacterScriptable` which runs the scheduler.

7.3.4. Perception in Gluon

The view cone integration of the perception system in Gluon is done with the class `ViewCone` which is just a wrapper on top of `btPerceptionViewcone`. It enables Gluon Creator to set the relevant values on `btPerceptionViewcone`. The same goes for the class `PerceptionAtom` which wraps `btPerceptionAtom` but also points at a `PerceptionInfo` as the design specifies in Chapter 6 on page 51

Both `btPerceptionViewcone` and `PerceptionAtom` will look for the `Character` component, when they are added to a `GameObject` and if they can not find it, they will remove themselves from the `GameObject`. If they find it, they will add themselves to the `Character` component's instance of `btCharacterScriptable`

`btPerceptionInfo` specifies when a `btPerceptionAtom` should update its information from the attached `btPerceptionInfo`. Consequently it should be possible to trigger this from the scripting which GluonEngine uses. To allow for this a subclass of `btPerceptionInfo` is used. This class is called `btPerceptionInfoScriptable`. This class basically only overrides `getAdjustedValue` which calls a corresponding function on the `PerceptionInfo` that is pointing to it. It is not the class `btPerceptionInfoScriptable` which is pointing at the actual script but rather `PerceptionInfo` which contains the script and loads it, using `QScriptEngine` and `QScriptValue` as `btNodeScriptable`.

Listing 7.10: Calling the corresponding `getAdjustedValue` in the script.

```

1  ...
2  QVariant PerceptionInfo::getAdjustedValue(qreal precision)
3  {
4      if(d->getAdjustedValueFunc.isFunction())
5      {
6          QScriptValue returnVal = d->getAdjustedValueFunc.call(QScriptValue(), QScriptValueList
7              () << precision);
8          if (d->engine.uncaughtException().isValid())
9          {
10             debug(d->engine.uncaughtException().toString() + ": " + d->engine.
11                 uncaughtExceptionBacktrace().join(" "));
12             return QVariant();
13         }
14         else
15             return returnVal.toVariant();
16     }
17 }
18 ...

```

Listing 7.10 on the previous page shows the code for calling the script function `getAdjustedValue`, which first checks if it is a function or if it is set, then runs it with the correct parameter and finally returns the results if the call succeeds.

7.4. Conclusion

We wanted to integrate the SMARTS Library into Gluon together with the new perception system. To do this we use Gluon's plugin system to create plugins which function as wrappers on top of the SMARTS Library implementation. Furthermore, the three classes `btNodeScriptable`, `btCharacterScriptable` and `btPerceptionInfoScriptable` were created to allow programmers to use the scripting system in Gluon, and through this create scripts for executing functionality in SMARTS Library without having to create C++ classes and just use `GluonCreator`, and through that `GluonEngine`, for the implementation of behavior tree leaf nodes.

Part III

The Quantum Sea



The Quantum Sea is a simple game of smashing isotopes of various elements from the periodic table together and scoring many, many points in the process.

Introducing The Quantum Sea

Now that we in Part II have constructed the SMARTS engine and its perception system, and through Part III got it working inside GluonEngine, we can finally get to Part IV, wherein we will construct the game through which we in this and Part V will be evaluating the methods for behavior tree construction and evaluation presented in Part I.

Developed over the course of the new year holiday between 2009 and 2010, the concept which is presented under here is The Quantum Sea. The game concept itself is based on the graphical representations of particle collision events as created by the scientists working at particle colliders around the world. The visuals here have inspired a game in which the action of colliding particles has turned into a personally guided fight to create the most high-energy collisions and that way score points.

Appendix B on page 129 shows the design sheets created for the game. The design is explored in further depth over the course of this chapter, where the concept is fleshed out further, and finally implemented. We will not construct the full design for the game, but rather build the rules for doing so, and then create one example to show the procedure, and a prototype through which evaluation will be carried out.

8.1. Game Design

With the base game elements set down in the design sheets in mind, we can describe how the game progression works: Rather than being a simple progression through a linear set of levels, the design of The Quantum Sea suggests a progression system which is tree-like.

The reasoning behind this observation is the level design described in the design, which is based on experimentation. An old adage goes that the answer to one question will open up new questions. Thus, from one base experiment, it is likely that while one item was under investigation multiple new questions would arise. And so, the progression becomes a tree.

Representing this in the game in a way which makes it possible to select and read information about each experiment with as few controls as possible is done in the manner described on the second design sheet. The specific item where this is described can be seen in Figure 8.1 on the next page.

As each level is an experiment based on the collision of items, we can set down a series of rules for how these collisions are encouraged by the player of the game. Later, in Chapter 9 on page 77, this information will be used to create the characters in the game.

A character in the game has an amount of energy which can be applied to the particles to achieve two things: Increase the speed of the particle, and change the

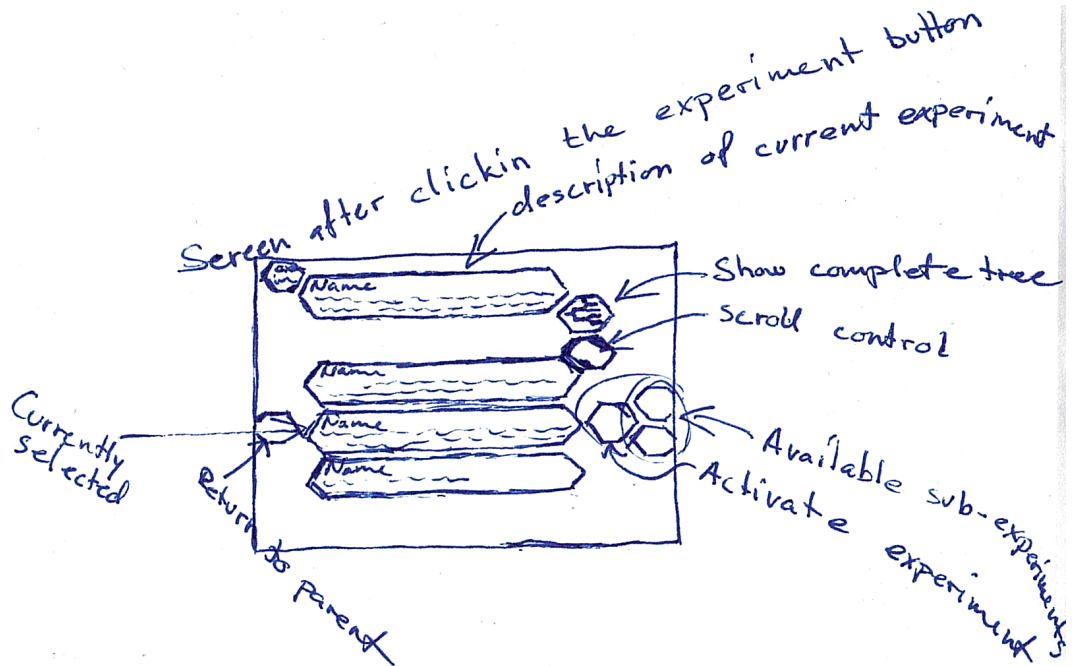


Figure 8.1.: A sketch showing the experiment selection screen

spin of the particle. The amount of energy a player has slowly regenerates over time, the particle's spin will return to the spin the particle started out with over time, and its speed will slowly reduce to zero.

8.2. A Small Level

As described in Section 8.1 levels in The Quantum Sea are all designed as experiments. Because of this, designing the level we will use for our testing is done in the same manner. To avoid an entirely trivial level setup such as the first experiment in the game, based on a collision of hydrogen against hydrogen, we create a hypothetical experiment some distance into the game progression. We further ignore the hints in the game design that there would be power-ups and further abilities (such as time adjustment and the like).

The level design seen below is the description which will be used in the game when presenting the level to the player. It includes both the basic premise for the level, as well as a short description of the objective, and a hint designed to indicate a small piece of help to the player, as opposed to being a strict instruction.

8.2.1. Level Design

Experiment SEA-42 - Carbon-14 vs. Nitrogen-14

Knowing from observation that Carbon-14 decays to Nitrogen-14, we wish to investigate what would result from a collision between the two.

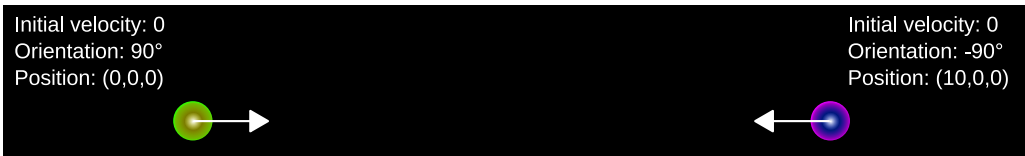


Figure 8.2.: The layout of the level

As such, the experimental setup consists of one Carbon-14 atom and one Nitrogen-14 atom in a vacuum and at rest.

Your objective is to investigate the impact of a collision between the two atoms, and further investigate the results of collisions between those results.

Hint: Speed is relative.

Multiplier levels

Energy remaining	100%	85%	70%	50%	Below 50%
Multiplier	5X	4X	3X	2X	1X

Level Layout

The layout of levels in The Quantum Sea is made up of particle positions and their initial velocities. As we have described this level as starting with the two particles Carbon-14 and Nitrogen-14 standing still, we define the initial layout of the level as can be seen in Figure 8.2.

One of the objectives for this level is thus to not simply smack the particles together at the first opportunity (that is, to simply increase the speed of the particle you are attached to at the beginning) as this will not yield the highest number of points.

The way in which specific details for the particles is defined is discussed in Chapter 9 on page 77, and the data for the various particles can then be found in Appendix C on page 135.

8.3. Conclusion

Over this chapter, an introduction to the game The Quantum Sea has been conducted, and various over-view elements of the game have been presented, including how the level design and game progression works. This base allows for specifics to be explored further in depth in the following chapters.

Particle Perception

The Quantum Sea has two distinct types of characters: Particles and Players. The first is the particles as used in the game, the collidable items. The second, the Player, is the abstract character we use to describe the player of the game. This is relevant to us, as we wish to attempt to describe a character able to play The Quantum Sea through use of behavior trees, and then evaluate this character using the methods described in Part I.

9.1. Particles

The various particles in The Quantum Sea, as mentioned in the previous chapter, function differently in a number of aspects:

- Particles give different points
- Particles yield different other particles when smashed
- Particles see the world differently

In particular the last of these items is important when focusing on SMARTS. While the other two make the game itself playable, the varying perception of the particles is more interesting to us. As such, in this chapter we will present the values for the first two, and go into more depth with the way in which the different particles view the world.

9.1.1. Defining the Particles

For the simple reason that The Quantum Sea is a game, and this project not being created under the Faculty of Particle Physics or even supposed to be edutainment, we do not aspire to create a game system which is true to real-world physics. However, to assist with the creation of game data, we still base parts of the logic on this. As such, the basic rules of the elements in the game are defined as follows:

- Smashing elements yield points, with a high speed and head-on collision yielding maximum points and slow speed and rear collision yielding no points
- Smashable elements:
 - Isotopes of elements found in the Periodic Table of Elements
 - Particles which make up the isotopes
 - Sub-particles which make up the particles (such as quarks)
- When isotopes smash, they yield various energies (x-rays, gamma rays and so on), sub-particles, particles and isotopes

- When particles smash, they yield various energies and sub-particles
- When sub-particles smash, they yield various energies
- Isotopes are provided with a fixed life span in the game to simulate half-life properties - if an isotope splits due to its half-life time limit being reached (decay), no points are given to the player

Varying classes of elements provide different information about the world, and as such the following does not list all the isotopes in the Periodic Table of Elements but rather defines what an isotope will yield, according to its various values.

The number of points yielded by any item in a collision event is equal to

$$1 + 2^{n*\phi*K} + E$$

where n is the weight of the item, ϕ is the normalized angle of impact deviation from 180 degrees divided by 180 (that is, a number between 0 and 1), and K is the amount of kinetic energy which is released by the impact. What this means, in essence, is that any rays created will always yield 1 point, as they are pure energy and have no weight, and that a high-speed collision will yield a higher number of points than a low-speed one, and similarly that a head-on collision will yield the most points and that a direct-from-the-rear collision will yield no points.

To create a suitable approximation of the K value in the points formula above, we define it as the velocity of a hypothetical single item resulting from a perfectly inelastic collision[14] between the two colliding items. As such, the velocity of this new item is defined as

$$v_f = \frac{m_1 v_1 + m_2 v_2}{m_1 + m_2}$$

where m_n is the mass of item n and v_n is the velocity of the item n . This formula functions in a manner such that it will release an amount of kinetic energy according to how much is left over from the two items in their natural state. The kinetic energy of an object in motion is defined as

$$E_k = \frac{1}{2}mv^2$$

With this information, we can now define the amount of kinetic energy K as follows:

$$K = E_{k_1} + E_{k_2} - v_f$$

The values v_f and K are further used to define the amount of new items created. Each item in the game field is created with a number of collision event levels defined, which describe which new items will be spawned at a collision event with a certain kinetic energy release K . They are spawned in a manner consistent with the velocity of the impact, meaning that a head-on collision will yield a circular spread of spawned items, and a collision directly from the rear will lead to no spread, spawning items all with the same velocity vector (straight ahead). Items are spawned in the order they are defined in the collision event level definition, and distributed evenly across the spread.

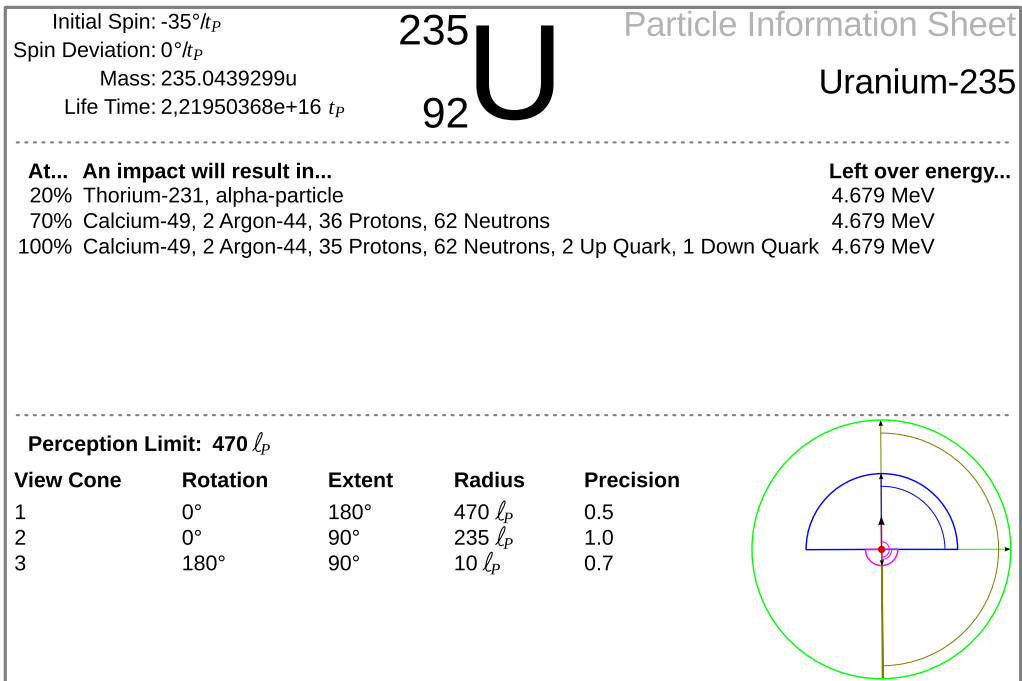


Figure 9.1.: Particle information sheet for the isotope Uranium-235

Two Sample Items

The following contains two examples of the definitions described above, and in the previous chapter. Note specifically how the total potential energy of each collision event level definition in the item is the same. This means that the energy contained in the system is stable, which is one of the basic assumptions in physics. Note also that the definitions are limited as a percentage rather than an absolute energy level. This is done to attempt to make it easier for the designer to create a larger spectrum of items.

We also define, at the same time, the perception values for the items.

To further ease the generation of this data the various isotopes, of which there is a very large number, can be generated automatically by following this procedure:

Initial spin The spin of the particle, multiplied by ten. This number is the number of degrees of rotation per planck time unit.

Mass Simply the mass in unified atomic mass units (u).

Life time Half-life in minutes, and that number then taken as represented in planck units. This is unlike what happens in real particle physics, but makes for a more interesting game, as highly radioactive items will decay at such a speed that it becomes difficult to manage¹.

¹Note in the example below that it seems a very long time, but note also that while Uranium-235 is considered highly radioactive, its half life of 703,800,000 years is nothing compared to for example

Decays to The item which the isotope will decay to is as defined by physics, and is always the first collision event level.

This of course only defines the particle itself, and not its various collision event levels. However, it does provide a starting point.

As the smashable items are essentially the same in respect to how the definitions look, we show one shasmable item, and one non-smashable item. The uranium isotope uranium-235 (or uranium II) can be seen in Figure 9.1 on the preceding page, and the gamma ray can be seen in the table below (since it is a ray there is not enough information to fill out a particle information sheet). Further items as used in the implemented game can be seen in Appendix C on page 135. All the values here are based on the Table of Nucleotides found at <http://atom.kaeri.re.kr/>.

Gamma Ray

Initial spin	Spin deviation	Mass	Life time
$0^\circ/t_P$	$10^\circ/t_P$	1u	$10t_P$

Note that a ray does not in fact have a mass. We use it here as a way of keeping point calculations sane.

9.2. Player

The player character in The Quantum Sea is a model of the human player of the game, through which we explore various possible angles of attack. What this means is that to allow us to build this character, we follow the methodology for building behavior trees described in Where Game AI Meets Academic AI[16, pp. 12], a method which is inspired by The Behavior-Oriented Design of Modular Agent Intelligence[6] and which functions as follows:

1. Specify the high level behavior of the character, i.e. what should it do.
2. Find possible activities and specify them as sequences of actions.
3. Use the sequences to identify all the low-level actions needed.
4. Identify conditions which need to be fulfilled to activate actions and goals, these provide the division into behaviors.
5. Identify and order goals that the character needs.
6. Select one of the behaviors to implement.

As such, the following is a run-through of the first five. The last item, implementation, is pushed to Chapter 10 on page 89, in which is described how the game is built using Gluon Creator and SMARTS Designer.

Rutherfordium-263, which has a half life of ten minutes, which then becomes an extremely short time in the game

9.2.1. High Level Behavior

The goal of the player is to attempt to smash two of the particles in the level together at the highest possible speed, and should usually try to smash the two heaviest particles together. This can be achieved using a number of different tactics, and as such the high level behavior is the description of these tactics. For reasons of simplicity we describe three tactics for the player: An aggressive tactic, an exploring tactic, and a cautious tactic.

Aggressive Directly attack the nearest item by setting optimal spin and speed to achieve a collision as soon as possible. Do not switch items until you have achieved a collision (at which time it is done automatically by the game)

Exploring Attempt to see larger parts of the level by spinning out of the way of other particles until the particle reaches high speed, and then attempt to hit the fastest other moving particle nearby from the front

Cautious Switch items every time you have adjusted speed and spin, applying between 5% and 10% of your initial energy to each of these. When energy is exhausted, wait until energy is fully recharged before resuming

9.2.2. Possible Activities

We now describe the activities indicated by these tactics:

Activities in the Aggressive Tactic :

- Find nearest and heaviest visible particle
- Attempt to hit the discovered particle

Activities in the Exploring Tactic :

- Avoid visible particles
- Increase speed continuously and slowly
- Adjust spin by some amount at an interval
- Upon reaching a certain speed, attempt to hit the heaviest visible particle

Activities in the Cautious Tactic :

- Adjust speed and spin lightly
- Switch to the nearest visible particle after one spin and speed adjustment
- Wait while energy replenishes when not enough energy is available to perform an activity

9.2.3. Low-level Actions

In the previous section, the sequences of actions were described superficially. In this section they are explored in further depth, describing each activity as sequences of actions.

The most common action for all three tactics is **Find nearest visible particle**, which can be split out into a number of possible actions:

- In case there are particles currently in your field of view, simply choose the heaviest particle
- If no particles are found like this, add a small amount of spin and wait a bit
- If this does not yield a result within a certain amount of time, add speed and wait a bit
- Repeat cycle until a particle is discovered

After this the **speed and spin adjustment** action is common as well, and can be split into the following actions:

- Apply energy to speed
- Apply energy to positive spin
- Apply energy to negative spin

This action is conceptually tightly bound to **the avoidance action**, but that action requires a different approach:

- Find required spin adjustment
- Apply energy to spin until spin adjustment is reached

As clearing the level of collidable items is the eventual goal of the game, attempting to **hit a particle of which you are aware** is often the immediate goal for the character:

- Find distance to particle, and particle's current velocity
- Find amount of energy required to adjust speed and spin to hit particle
- Apply energy to speed and spin to impact goal particle if amount of energy is available
- Otherwise shift to next potential target

9.2.4. Division Into Behaviors

We have now defined the actions and overall strategies, and now we want to define behaviors and behavior trees from these definitions. For each tactic the required behaviors are defined and reused if these are defined for one of the other tactics.

We start off by defining three basis behaviors which are used throughout the different activities:

Apply Speed Applying energy as speed to a particle

Apply Spin Applying energy as spin to a particle either as positive or negative spin

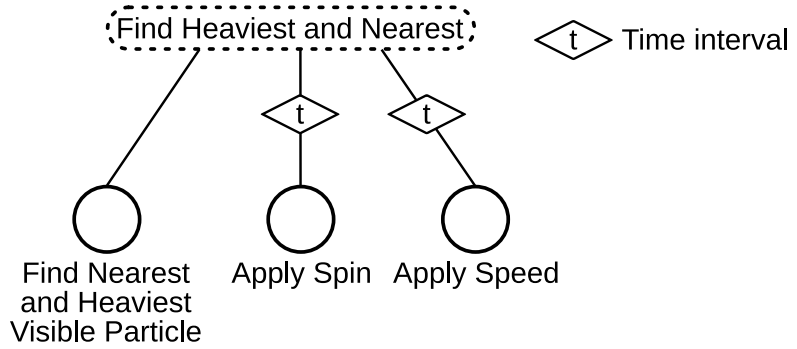


Figure 9.2.: The behavior for finding the nearest and heaviest visible particle

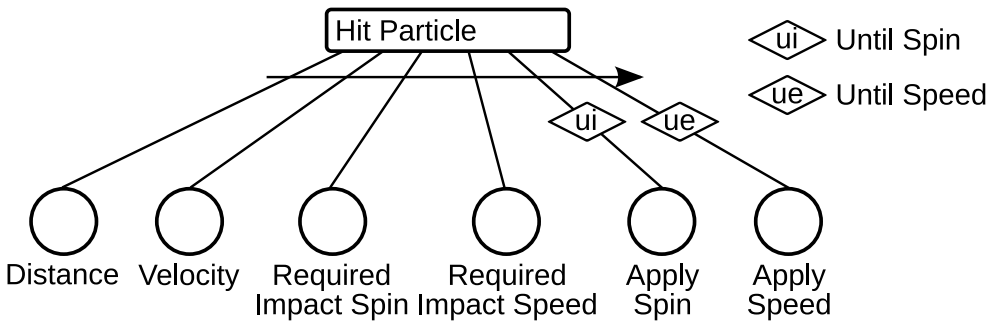


Figure 9.3.: The behavior for attempting to hit a particle

These three behaviors will act as the cornerstones of the rest of the behaviors. The next step is to define the rest of the behaviors together with any new smaller behaviors if needed.

We will start off by defining the behaviors for an **aggressive player**. Figure 9.2 shows the behavior for **Find visible, heaviest and, nearest particle**. It is a parallel with three different actions run in parallel and has three actions.

Nearest and Heaviest Visible Checks if there are any particles visible and if this is the case it selects the nearest and heaviest of these and terminates

Apply Spin Defined above. A decorator ensures that this action is only called at a given time interval

Apply Speed Defined above. A decorator ensures that this action is only called at a given time interval

If the **Nearest and Heaviest Visible** action terminates, the parallel also terminates.

Attempt to hit the discovered particle is shown on Figure 9.3 shows the behavior for this action.

The behavior has six different actions used for calculating and hitting the particle.

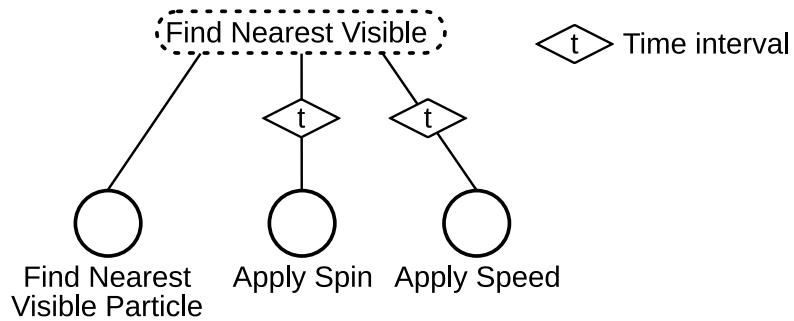


Figure 9.4.: The behavior for finding the nearest visible particle

Distance Calculates the distance to the particle

Velocity Finds the particle's velocity

Required Impact Spin Calculates the required spin for impact

Required Impact Speed Calculates the required speed for impact

Apply Speed Applying speed to the currently controlled particle. It has a decorator which prevents execution if there is not enough energy, and will stop execution if the targeted speed has been reached.

Apply Spin Applying spin to the currently controlled particle (either positive or negative spin). It has a decorator which prevents execution if there is not enough energy, and will stop execution if the targeted spin has been reached.

Now we define the different behaviors which a **exploring player** will use when player the game. Instead of finding the nearest, heaviest and visible particle, it should just find the nearest visible particle, so it can be avoided. This behavior is shown in Figure 9.4, is called **Find nearest visible particle** and consists of three actions:

Visible and Nearest Checks if there are any particles visible and if this is the case it selects the nearest particle and terminates

Apply Spin Defined above. A decorator insures that the a this action is only called at a given time interval

Apply Speed Defined above. A decorator insures that the a this action is only called at a given time interval

The **Avoid visible particles** behavior is shown in Figure 9.5 on the facing page, and consists of a previously defined behavior and two other behaviors:

Find nearest visible particle A behavior that is defined above

Required Avoid Spin Calculates the required spin for avoiding impact

Apply Spin Applying spin the current particle as positive or negative spin

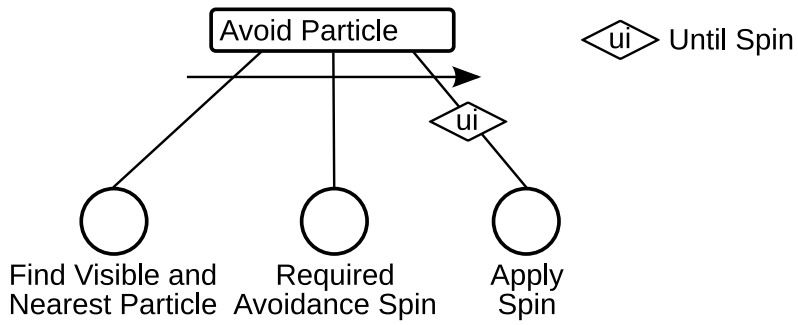


Figure 9.5.: The behavior for avoiding a particle

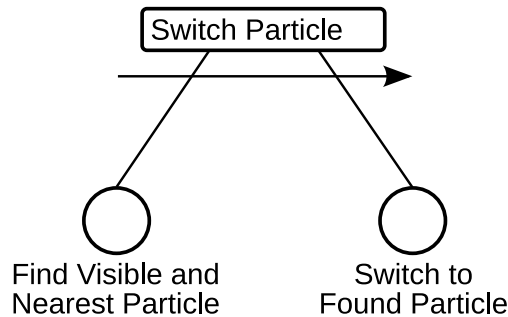


Figure 9.6.: The behavior for switching particle

The **Cautions tactic** only has one behavior which needs definition, and this is shown in Figure 9.6. The behavior contains the previous **Find nearest and visible particle** and one other behavior for switching particle:

Find visible, heaviest and, nearest particle A behavior which was defined under the **Aggressive tactic**

Switch Switches to the nearest particle

9.2.5. Goal Identification and Ordering

Now that the behaviors used by the three different tactics have been defined it is possible to create the behavior trees for each tactic.

Aggressive Tactic

The **Aggressive tactic** will always try to collide a particle to the nearest and heaviest particle, which renders the behavior tree in Figure 9.7 on the following page.

It shows that this tactic will immediately search for the nearest and heaviest visible particle and try to hit it. Because of this it will never switch to a new particle, but will continue repeating the steps on the next automatically selected particle because of the selector.

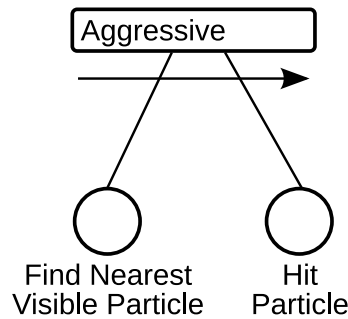


Figure 9.7.: The behavior tree for the **Aggressive tactic**

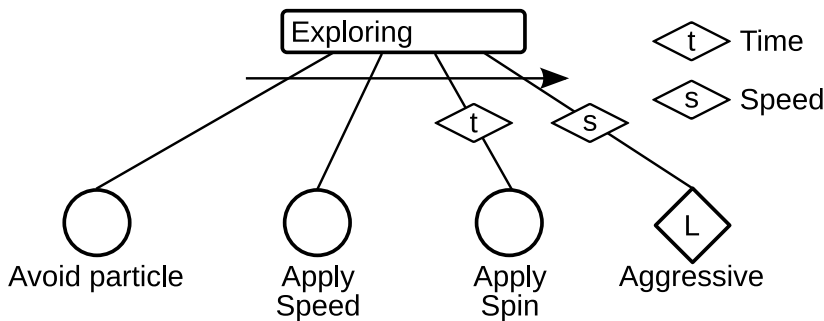


Figure 9.8.: The behavior tree for the **Exploration tactic**

Exploration Tactic

The **Exploration tactic** is a bit different as it will search through the surrounding area before attempting to make a collision.

Figure 9.8 shows the behavior tree for this tactic. The root is a selector where the tactic first avoids the nearest particle, then applies speed and if a given time interval has been reach also applies spin to the particle. Finally if the particle has reached a certain speed then the **Exploration tactic** runs the aggressive behavior tree for impacting two paricles.

Cautious Tactic

This tactic is hesitant in its reaction to the game world. Figure 9.9 on the facing page illustrates the behavior tree for this tactic.

This tactic uses a selector as the root node, and will always switch between two particles to start with. Afterwards it will run the exploration behavior tree, which will avoid any collisions and add speed and maybe spin to a particle. If the speed limit has not been reached, the **Cautious tactic** will make another switch and so forth. If at any point a controlled particle reaches a certain amount of speed it will try to collide this particle with another.

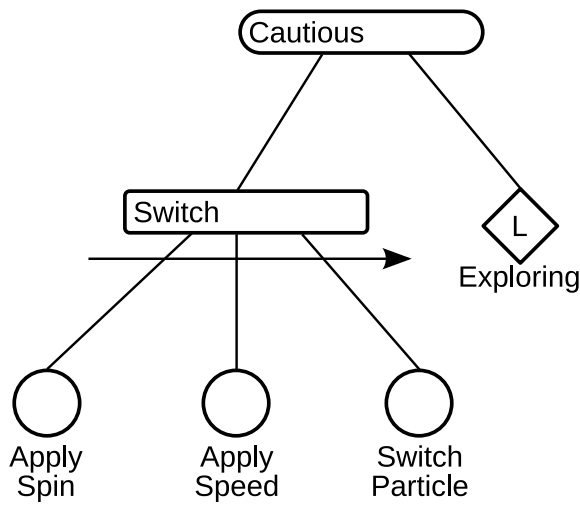


Figure 9.9.: The behavior tree for the **Cautious tactic**

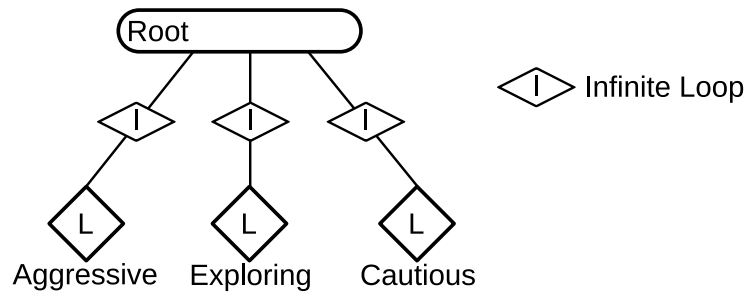


Figure 9.10.: The behavior tree for the player

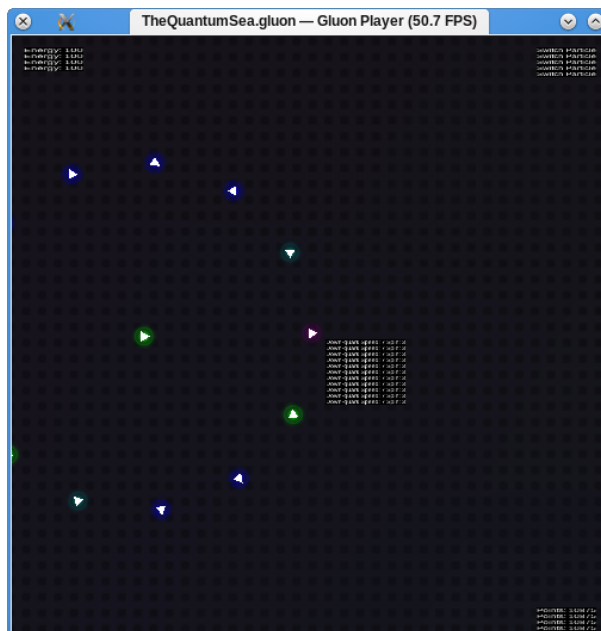
The Player Behavior Tree

Now we put all of the behavior trees together to form the behavior tree for the player. The root node is a selector and we have chosen that the player will stick to a selected tactic for the duration of the game.

Figure 9.10 shows this behavior tree, where each branch from the root node has a decorator. This decorator insures that the player will stick to the chosen tactic, by never letting the decorator return to the root, in stead continuing the execution of the sub behavior tree.

9.3. Conclusion

Through the preceding chapter has been created a system by which the particles in The Quantum Sea can be defined, and a number of them constructed. In it has also been designed the behavior trees representing a player of the game with three varying tactics. This all allows for the construction of the game itself.



Quantum Construction

Over the course of the two preceding chapters, we have described the various parts of the game design for The Quantum Sea. In the chapters before that we described how GluonEngine and SMARTS each work, and how they work together on a technical level. This chapter, then, is a description of how a game is constructed using these tools. We do not intend to show how one uses Gluon Creator in depth as that would be out of scope for this report, but rather focus on the non trivial parts of the game construction.

10.1. Level Building

One wish for level construction for The Quantum Sea is the ability for level designers to simply drop a particle into the scene, and then have the game pick up on these particles and react correctly to them when the level is started. That is: Drop in a number of particles, and mark one particle as the particle the player controls when the level starts.

10.1.1. The Anatomy of a Particle

With this in mind, the first item to be constructed is the self-contained particle. The particle contains the information as described by the particle information sheets described in Section 9.1.1 on page 77 and use the prefab system in Gluon to allow level designers to construct each particle, and then simply re-use them in the levels, and in the various other particles' impact levels.

Listing 10.1: The ParticleInfo script's properties

```

1 function start()
2 {
3     Component.particleName = "";
4     Component.collidable = true;
5     Component.initialSpin = 0;
6     Component.metaInfo().setPropertyRange("initialSpin", -360, 360);
7     Component.spinDeviation = 0;
8     Component.metaInfo().setPropertyRange("spinDeviation", -360, 360);
9     Component.mass = 0;
10    Component.lifeTime = 0;
11    Component.metaInfo().setPropertyRange("lifeTime", 0, quint.max());
12    Component.perceptionLimit = 0;
13    Component.metaInfo().setPropertyRange("perceptionLimit", 0, quint.max());
14    // ...
15 }
```

Conspicuous by their absense here are properties for the impact levels and view cones. This, of course, is because they are constructed using a separate script, where one is added per item. The view cones are simply the SMARTS components as described in Section 6.2.2 on page 55, while the impact levels are simple

information containers, and constructed as scripts which contains the following property definitions:

Listing 10.2: The ParticleImpactLevel script's properties

```
1 function start()
2 {
3     Component.maxEnergyLevel = 0;
4     Component.metaInfo().setPropertyRange("maxEnergyLevel", 0, 100);
5     Component.leftOverEnergy = 0;
6     Component.metaInfo().setPropertyRange("leftOverEnergy", 0, quint.max());
7     Component.spawnCounts = new Array();
8     Component.spawnItems = new Array();
9 }
```

The two first properties are straight forward, but the last two demand a little explanation. The reason we define them like so is that we cannot define arbitrary data structures in Gluon, and as such we define the items to be spawned from this impact level in two gos:

Firstly the number of items to be spawned, and secondly the items to spawn that many of. This does require some handling of the items when using them, but this is relatively trivial and shall not be explored further here.

With this in place we can now describe the structure of a particle prefab. Shortly: Prefabs are created as though you were constructing a normal `GameObject` in your Scene. When a `GameObject` is dragged into the Asset tree, it becomes a Prefab, and the `GameObject` in the Scene becomes an instance of that Prefab (see Appendix A on page 121 for further information). So with that in mind a particle prefab, exemplified by the Nitrogen-14 isotope, is structured as follows:

- Nitrogen-14
 - ParticleInfo
 - ParticleMotion
 - ParticleImpactLevel 1
 - ParticleImpactLevel 2
 - ViewCone 1
 - ViewCone 2
 - ViewCone 3

An important part of the workflow when dealing with Prefabs is the way in which you can change values, but not the basic structure. What this means is that we need to have the view cones and particle impact levels that we might need in the prefab. The only impact of this is on memory since disabling Components makes them have no impact on the performance of the game.

10.1.2. Putting Together Levels

The construction of a level is thus simply a matter of taking a number of these prefabs, and dragging then into the scene, and finally choosing which of the items will then be used as the initially selected particle, which is done by selecting it in the `PlayerManager` script.

This, then, would be the ideal world solution. However, as we have run into problems in creating this system (Gluon did not at time of writing have working prefabs) what had to be done was to create a system which would allow a particle to copy itself and then tell the children what they should transform into. This system, while convoluted and long-winded, with the transformation function for only the few particles needed in the prototype created here reaching 250 lines of code which looks mostly as in the listing below, has turned out to function for the most part acceptably.

Listing 10.3: A snippet from the `setParticleByName(name)` function

```

1 // ...
2 switch(name)
3 {
4     case "Nitrogen-14":
5         color = new QColor(255, 255, 0);
6         initialSpin = 10;
7         spinDeviation = 0;
8         mass = 14.0030740;
9         lifeTime = maxParticleLifetime;
10        impactLevels = new Array(
11            new Array(70, new Array(1, "Beryllium-10", 3, "Proton", 1, "Neutron"), 0),
12            new Array(100, new Array(1, "Beryllium-10", 2, "Proton", 2, "Up-quark", 1, "Down-
13                quark"), 0));
14        perceptionLimit = 25;
15        viewConeInfo = new Array(
16            new Array(1, 0, 180, 25, 0.5),
17            new Array(2, 0, 90, 20, 1.0),
18            new Array(3, 180, 90, 10, 0.7));
19        break;
20 // ...

```

Using this function the `PlayerManager` script can be used to allow for levels to be constructed in an almost identical manner to the intended, meaning that instead of adding the prefabs to a level the level designer copies the `GameObjects` representing particles directly, and sets their names to that of one of the items in the switch statement above. This is then handled by the `PlayerManager` which informs the particles to convert themselves to the particle by that name.

10.2. Behavior Trees

The behavior trees for which we described the design in the latter half of the previous chapter were constructed so far simply as conceptual constructs. This section is an explanation of how they are implemented for use in the game. However, rather than explain at length how each tree is built using SMARTS Designer, we instead believe it will be of higher value to describe how one works with the tool in a more general manner with examples provided by *The Quantum Sea*.

10.2.1. Working with SMARTS Designer

When constructing the trees themselves, SMARTS Designer[16] allows the game designer to build the tree using the same methods as when sketching them out on a blackboard or a piece of paper as we did in the previous steps in the method. While it only shows the tree using a standard tree view, it in essence shows the logic from left to right rather than from the top and down, and as such there is little real difference between the two ways of visualizing the trees.

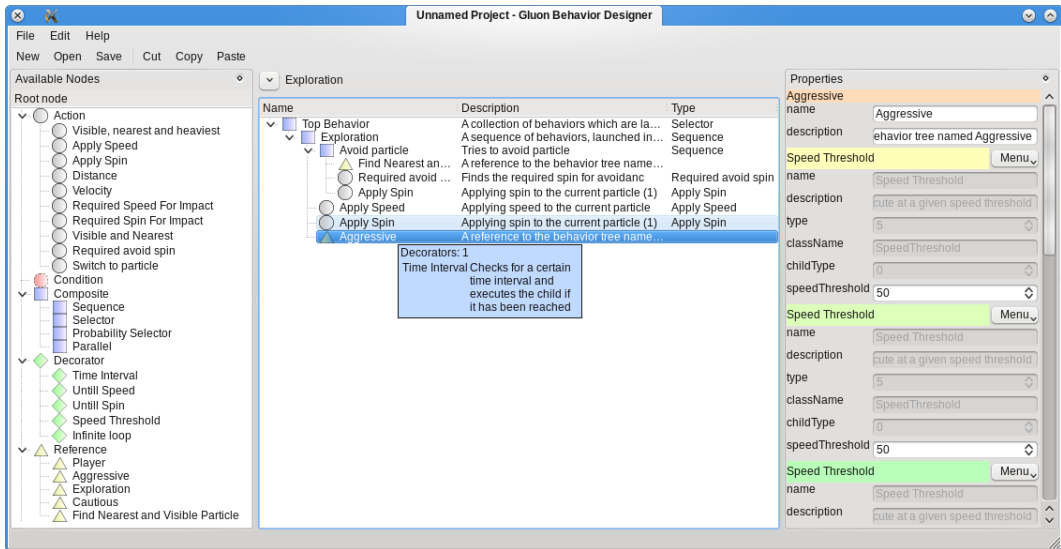


Figure 10.1.: SMARTS Designer working on the Exploration tree

Figure 10.1 shows the brain for The Quantum Sea as it looks in the program with the behavior tree representing the Exploration tactic selected. On the right side is a list of all the available components for your behavior trees, the middle shows the currently active tree (change which is the active by clicking on the button beside the tree's name), and the right side shows the various options for the currently selected node in the tree.

Decorators, being somewhat of a special case in behavior trees, are not shown directly in the tree, but rather they are shown in the tooltip when hovering over an item - in the case of the screenshot the Apply Spin node. This also means that when selecting an item in the tree, any decorators attached to the node will be shown in the properties panel along with the node they are attached to.

The following is a description of the work flow which is employed when constructing such a tree. Rather than simply assuming that the method has been followed and we have already got a design, we assume here that this is what would be done in steps 4 and 5, and that we as such show how the program supports the construction work flow in the method, once the initial analysis work has been done.

SMARTS Designer's base functionality is essentially drag'n'drop based: To build a behavior tree one simply drags the items from the tree view containing available nodes into the view containing the behavior tree itself. In the case of any behaviors which need values changed these can then be manipulated by selecting the behavior in question and then changing those values in the Properties view.

Adding Behaviors

Creating the many behaviors for a behavior tree is done by right-clicking in the Available Nodes pane on the appropriate category (action, composite or decorator) and then selecting New Node Type in the context menu, which will add a node

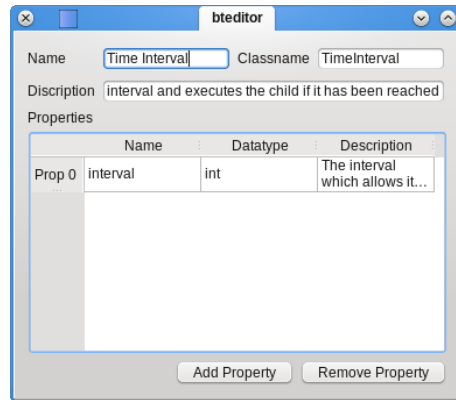


Figure 10.2.: Editing the Time Interval decorator's settings

named New Node Type at the end of the selected sub-tree. Then double-click on the newly added node to open up the node editing dialog.

In this dialog you can set the various parameters relevant to the node you wish to add, as seen in Figure 10.2. While name and description are relatively obvious, and simply human-readable information used in the editor, the Classname and Properties items are less obvious: Classname is the name of the class which represents this node type in the game code, and Properties are those properties which that class has available for controlling the way it functions.

As an example of this we have here the Time Interval decorator which will ensure that the behavior it is attached to is not run more often than at a certain interval, as defined by the integer property called interval. The class which represents this decorator in the game code is `TimeInterval` and in the next section we describe how these are then defined.

10.2.2. Creating the Actions

The final step in the method (from Section 9.2 on page 80), "Select one of the behaviors to implement." is, in fact, an iterative step. What this step covers is the creation of the logic in the game which is not already supported by SMARTS. In our case this means implementing functionality through the use of the scripting system described in Chapter 7 on page 61.

The scripts created for the SMARTS components are built in the same way any other script for Gluon is, except for the fact that in stead of having the `initialize`, `start`, `update`, `draw`, `stop` and `cleanup` functions, they have one single function: The `run` function.

Importantly, rather than having the Character passed to them through the run function's parameter this is done in a style similar to Gluon's other scripted objects, by setting it as a special object on the script's engine. This allows the script writer to use the object `Character` in their code. Similarly, the `Node` is available in the same way.

As can be seen in the following snippet, which is the entirety of the code for the `TimeInterval` node type, this has allowed for the creation of pleasant, readable

code:

Listing 10.4: The run function from the TimeInterval script

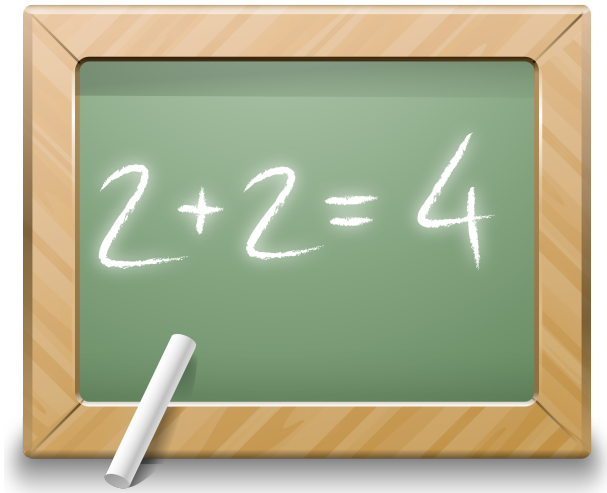
```
1 function run()
2 {
3   if(Character.time > Node.interval)
4   {
5     Character.useTimeEnergy = true;
6     Character.timeEnergy = Character.time;
7     Character.time = 0;
8     Node.runChild(0);
9     Node.status = 3;
10  }
11  else
12  {
13    Character.useTimeEnergy = false;
14    Character.time += Component.time;
15    Node.status = 2;
16  }
17 }
```

10.3. Conclusion

Through this chapter we have described parts of the implementation of The Quantum Sea relevant to SMARTS. What this means is that we have not gone into any serious depth with the game logic itself, which while not trivial is outside the scope of this project. Instead the implementation of the behavior trees designed in the previous chapter has been explained, and hints to the game implementation required for the understanding of the behavior trees have been provided. A screenshot of what the game looks like in the incarnation which existed when the prototype implementation was concluded can be seen in Figure 9.11 on page 88.

Part IV

Evaluation



A method is only a hypothesis without proof, and the methods proposed in this report are so far untested. We now alleviate that problem by performing evaluation of them.

Evaluation

Over the previous chapters we have laid the ground work for functionality and methodology for use in both defining, using and evaluating behavior trees and their applicability in various situations. While their use has been applied over the last couple of chapters, the evaluation method described in Chapter 4 on page 13 has yet to be evaluated past the proof of its mathematical correctness.

We now wish to conduct tests to evaluate the method on data which is both simple and at the same time reflects a real world situation. Towards this end we could conduct a data gathering exercise on the behavior trees designed for The Quantum Sea in Section 9.2 on page 80. These behavior trees are too general for testing and contains too much uncertainty, instead we are focusing on a single scenario where a player is trying to collide two particles.

11.1. Testing Procedure

We will perform three different tests to be able to evaluate our challenge and behavior diversity metrics against each other and those that were used in "Where Game AI Meets Academic AI"[16]. We will test the three different ways of optimizing behavior trees described in subsection 4.3.3 on page 31 where we will run an equal number of tests for each method and then compare the results and evaluate the methods to determine the best method for optimizing behavior trees.

We will use The Quantum Sea for testing the methods using a more focused set of behavior trees, where we will conduct three different tests for each method, first where k_1 and k_2 is equal, then where first where $k_1 = 0.75$ and $k_2 = 0.25$ and lastly first where $k_1 = 0.25$ and $k_2 = 0.75$. We perform the tests by running 10 calculations for each method using The Quantum Sea, with data from 1000 runs, and between each calculation the optimal weights are calculated with the given method and inserted into the behavior tree before the next 1000 runs.

The behavior trees for The Quantum Sea contains very special structures e.g infinite loops, parallels and decorators which could create very special results during the usage and collapsing of the behavior tree. Therefore we will test the methods on a constructed example ending up with a comparison of how the tests went and which methods are best suitable for optimizing behavior trees. Note that the global tests will not make any use of links, as these still are not accounted for in the calculations.

11.1.1. The Quantum Sea

We have designed and created new behavior trees for the collision scenario, as the player would have different strategies for trying to hit another particle. The creation of these behavior trees also were done using the method described in

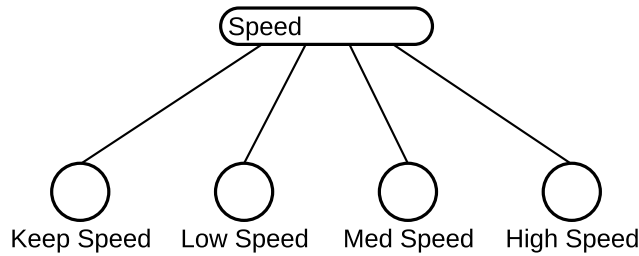


Figure 11.1.: The behavior tree for selecting the particle speed

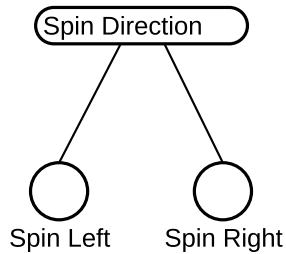


Figure 11.2.: The behavior tree for selecting the spin direction of the particle

section 9.2 on page 80. We start by describing some of the subtrees which are used throughout the bigger behavior trees.

Figure 11.1 shows the **Speed** behavior tree which selects the speed for the current particle, and has four different speeds that can be chosen, **Keep Speed**, **Low Speed**, **Med Speed** and, **High Speed**. The root node is a probability selector with a probability of 0.25 of selecting each speed.

Before applying spin the the particle it is necessary for selecting in which direction the spin should be applied. Figure 11.2 shows the behavior tree for selecting the spin direction, where the root again is a probability selector and each child has a probability of 0.5 of being selected.

Now that we have defined some of the often used behaviors we can define the strategies.

The **Direct** strategy is shown on figure 11.3 on the facing page, The root is a sequence such that the strategy first selects the speed for the particle and then runs a parallel for trying to hit the other particle. The parallel will calculate how to hit the other particle meanwhile there is a timeout before the parallel fails.

The behavior tree for trying to hit the other particle through an arc is shown on figure 11.4 on the next page. As before the root behavior is a sequence which first will select the spin direction. Then it will select the speed it wants to travel, and at last it will start a parallel for trying to hit the other particle. The parallel will calculate the required spin impact in a giving interval, apply the necessary spin and speed and a timeout will terminate the parallel, such that a new strategy can be chosen for execution.

Figure 11.5 on page 100 shows the last strategy for trying to hit a through a outgoing spiral. The root is a parallel which runs three children at the same time.

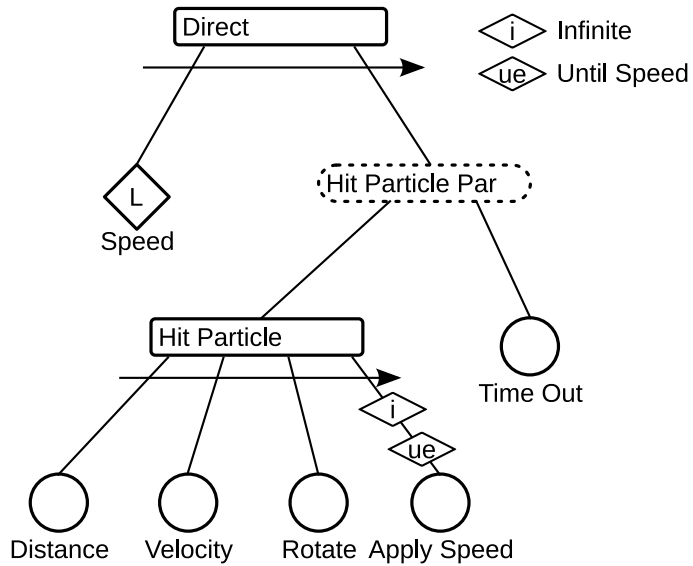


Figure 11.3.: The behavior tree for a direct approach to hit the particle

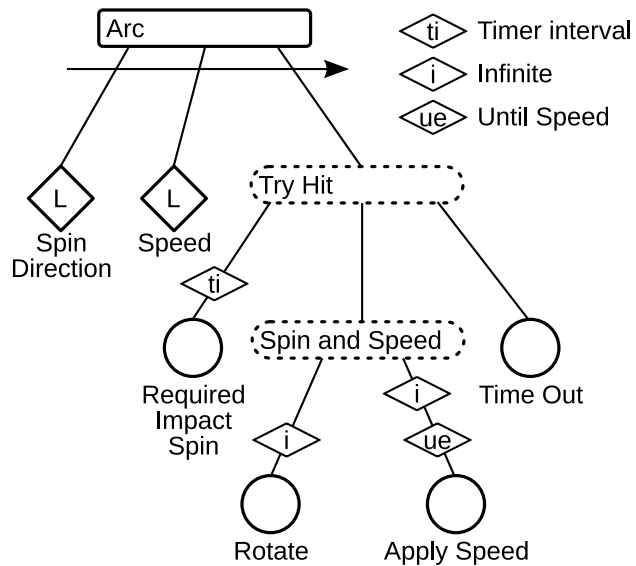


Figure 11.4.: The behavior tree for trying to hit the particle with an arc

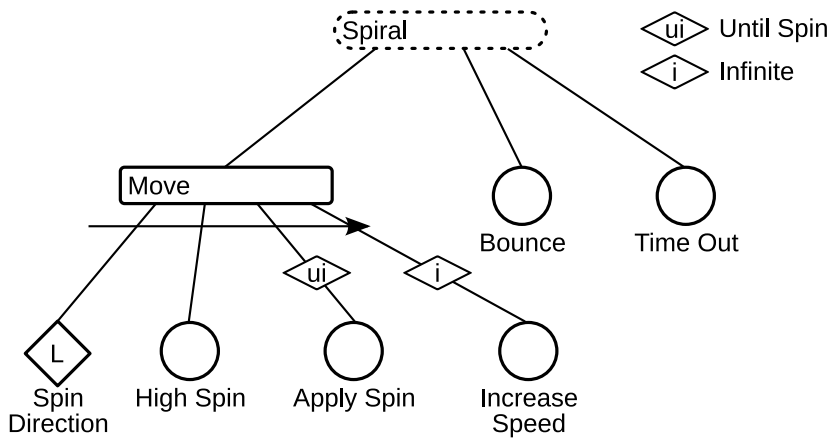


Figure 11.5.: The behavior tree for doing a spiral outwards attempting to hit any particle in the way

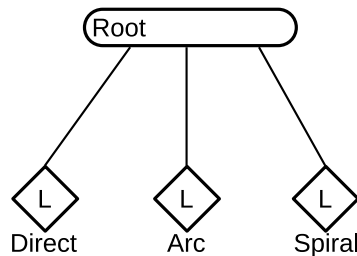


Figure 11.6.: The behavior tree for deciding which strategy to use for trying to hit a particle

The first is a sequence which chooses spin direction, sets the spin, applies the spin and lastly increases the speed slowly. The two other children are used for termination of the parallel checks if the particle is out of bounce of the level or if a certain time has passed before terminating the parallel.

Figure shows the root behavior tree for selecting a strategy, the **Root** node is probability selector and has three different strategies, **Direct**, **Arc** and, **Spiral**, where each strategy has probability of 0.33 of being selected.

Now that we have described the behavior trees used for testing, it is possible to use these trees for testing our methods.

11.1.2. Constructed Example

The constructed example usage the simple building blocks from section 2 on page 5 which means that parallels and decorators are not used in the behavior tree. Figure 11.7 on the facing page shows the behavior tree used for the constructed example.

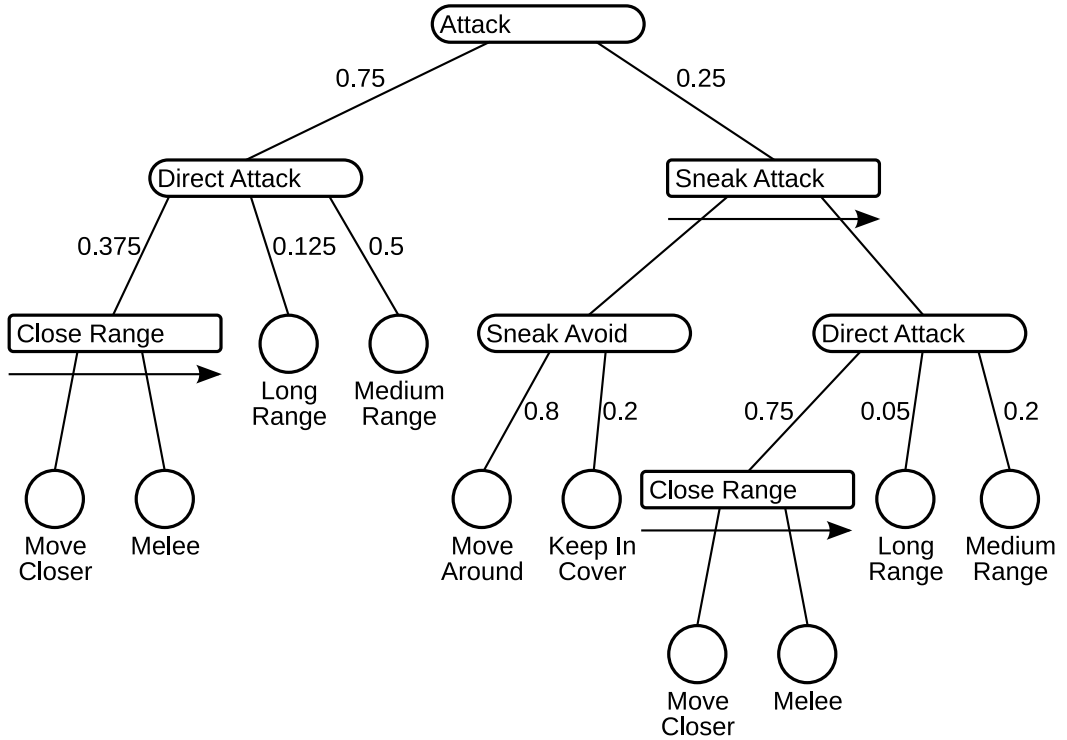


Figure 11.7.: The behavior tree of the constructed example

The behavior tree defines the attack possibilities of an AI in a First Person Shooter game. It has the possibility to either **Direct Attack** which selects the which kind of attack should be performed or to use **Sneak Attack**. During the **Sneak Attack** the AI first either moves around the opponent or stays put in the safety of cover, and then uses **Direct Attack** for selecting the attack type.

Every selector on the figure is a probability selector and the numbers on the paths indicates the success rate of each child node, and will be used during the use of the local and global method for calculating the new weights.

11.2. Evaluation of the Methods

With the test setup described, it is time to perform the evaluation of the three methods. This section contains that work.

11.2.1. Local Method

Tables D.2 and D.3 shown that when our optimization is run on simple actions it produces trivial solutions. It does this because these simple actions always succeeds, it was an expected result as we saw the same thing in [16]. This show a limitation of our method in that it does not support emergent behavior such as

Children	k_1	k_2	Local Method		Calculation 1	Success Rate 10	Calculation 10
			Start Weight	Success Rate 1			
Direct	0.5	0.5	0.33	0.108766	0.292461	0.139082	0.2867441
Arc			0.33	0.354559	0.528037	0.582345	0.586745
Spiral			0.33	0.0409731	0.179502	0.0270728	0.12651
Direct	0.75	0.25	0.33	0.192466	0.356178	0.167614	0.343552
Arc			0.33	0.37397	0.420521	0.447898	0.439249
Spiral			0.33	0.029734	0.223301	0.0267775	0.217199
Direct	0.25	0.75	0.33	0.203125	0.378015	0.142466	0.243032
Arc			0.33	0.313643	0.523616	0.573195	0.690409
Spiral			0.33	0.0337466	0.098369	0.0253378	0.0665591

Table 11.1.: A summation table of the results using the local method

going left more often than right results in covering particular areas of a game map. Obviously it does not matter how we weight challenge and diversity as the same is seen in tables D.5 and D.6.

What does provide useful results is when there is a selection between alternative strategies where these strategies have a distinctive goal that it can fail to achieve, and not simple actions as mentioned above. This can be seen in our selection of strategies in The Quantum Sea shown in table 11.1¹. Here the strategies are all trying to achieve the same goal, which result in more sensible data.

This is also the case when changing k_1 and k_2 . Changing these values such that diversity is weighted highest produces weights that are more even as we would expect and optimizing for challenge produces weights that are closer to the success rates of the behaviors as we expected.

It could be possible to add goal conditions to simple actions, such that their efficiency could be evaluated in context of the game. This could be done by simply replacing the action with a sequence with the action and the goal condition. In our game it could be a condition that return success if an action brought the player closer to the target.

The results using the local method on The Quantum Sea suggest that the local is sufficient for typical applications in behavior trees, such as selecting between alternative strategies, it does not need more than local data to evaluate the different strategies.

11.2.2. Global Method

The results for the global method supports what have been observed during the tests of the local method. All tables for probability selectors with only simple actions (e.g tables D.11, D.18 and, D.24) show that when the global method is run probability selectors which only have actions, it also produces trivial solutions and hence the same can be concluded as with the local method. This means that

¹For a more detailed table of results see appendix D.1.1

Children	k_1	k_2	Global Method		Calculation 1	Success Rate 10	Calculation 10
			Start Weight	Success Rate 1			
Direct	0.5	0.5	0.33	0.163842	0.125336	0.128269	0.0524793
Arc			0.33	0.376137	0.87068	0.555834	0.946352
Spiral			0.33	0.0261248	0.00398339	0.016129	0.00116912
Direct	0.75	0.25	0.33	0.168627	0.223966	0.170845	0.183037
Arc			0.33	0.334013	0.747159	0.481235	0.796028
Spiral			0.33	0.0277778	0.0288749	0.0239453	0.0209353
Direct	0.25	0.75	0.33	0.156398	0.0727333	0.160099	0.0710868
Arc			0.33	0.355994	0.926097	0.368503	0.928341
Spiral			0.33	0.0340368	0.00116986	0.0255987	0.000571787

Table 11.2.: A summation table of the results using the global method

measurement on probability selectors with only simple actions does not contribute with anything useful or emergent behavior.

Table 11.2.2² shows the a summation of the results of using the global method for calculating the new weights. The results are hard to interpret because of the nature of the global method, as this method is collapsing the behavior tree, calculating the new weights and then expands the behavior tree again.

This means that it is not possible to actually look at the results and conclude if these are valid or good results. If the methods defined for restoring the canonical behavior tree into the original behavior tree does not work as expected, the weights might not be correctly set in the behavior tree after the conversion.

For actually being able to conclude on the results they should have been shown for the canonical tree after the calculations, as these essential are calculations on a local probability selector. But this still leaves the problem of converting the canonical behavior tree into its original form.

11.2.3. Constructed Example

The calculations using the local methods show what it is expected as when the calculations were done on the data from The Quantum Sea, these results can be seen in appendix D.2 under the local subsection. It shows that when weighting either k_1 or k_2 higher it will produce weights towards diversity and challenge respectively. Furthermore because every probability selector actually selects a strategy to execute which means there no probability selector only with simple actions, which either succeeds or fails every run. Therefore the results of the nested probability selectors are more reasonable and follows the intuition of the weight optimization.

The global results actually resembles the local results but the reason for this could be, that the behavior tree is much simpler than those of The Quantum Sea, or it could simply be a coincidence. It also does not contain any special behaviors such as infinite loops and parallels, which could “obscure” the values such that they

²For a more detailed table of results see appendix D.1.2

does not make any sense to the naked eye. The uncertainty is the collapsing and expanding of the behavior trees, which calculates weights for the whole tree and not local. The results can be seen in appendix D.2 under the global subsection.

11.3. Comparison of the Methods

Now that we have performed tests on the each method it is now time to compare these methods together conclude on which method is best and for what usage.

The local method provides results expected while the results of the global method were not especial readable, only the results from the constructed example resembled that of the local method. The local method only optimizes all of the local probability selectors in the behavior tree, which means that any strategies present are only optimized locally. This can be used for optimize certain local high level probability selectors which selects a strategy, and should be applicable for the most typical uses of behavior trees.

The global method optimizes the whole behavior tree and collapses, calculates the new weights and expanding the behavior tree to its original form. If the method of expanding the behavior tree does not work entire correctly could provide sources of error on the weights, when trying to expand the behavior tree.

Because of the global optimization of the behavior tree, results in new weights which might not be as sensible for a designer look at and perceive if the new weights optimizes the behaviors as they should.

Instead of being forced to use either methods they could be used in conjunction with each other. Where the designer would select which probability selectors should use the global optimization and the rest would then use the local method. Of course any probability selectors which are nested under a probability selector which is marked for global optimization will not undergo any local optimization.

This means that it is possible for the designer to actually use the local method for any higher level of strategies which maybe are independent of other low level selectors, while the global method is used for lower level of probability selectors to optimize that particular subtree.

11.4. Evaluation of Behavior Tree Design Process

During the creation and testing of The Quantum Sea we have designed and created different behavior trees. We have been using the method described in "Where Game AI Meets Academic AI"[16, 10, pp. 12], and will now evaluate the method and how it is to use it for defining behavior trees.

We used the method as iterative development which made it feel very intuitive to use because it resembles the way we develop software. It gives us the possibility to revise any designed behavior trees and make any corrections necessary to them, without having to start from scratch. Furthermore the object oriented nature of both behavior trees makes it very easy to substitute behaviors and larger parts of the with having to actually change any lines of code. Of course this implies that there are no dependencies between behaviors that are substituted, but as the

method suggests you should group dependent behaviors together in e.g a sequence which would make it easier to replace that particular sequence with another behavior or composite.

Because of these two things it is very easy to create behavior trees and by following the method, either strict or as guidelines, makes it very easy to make changes to a behavior tree and you can put your main focus on the creation process rather than use a lot of energy and time on none essential matters.

Even though the method brings some benefits it is also essential to have the correct tools for creating behavior trees. The overall design process could be done with pen and paper but could be made functional and effective using a diagram drawing tool. Designing the behavior trees is one thing, but another is to create the behavior tree for usage in the game. The SMARTS Designer creates this connections between design and the game by creating an intermediate layer for library to parse and dynamically load the needed behaviors. This has helped to reduced a great amount of tedious work which should have been done by hand and in code, and also makes the SMARTS Library more flexible. The ideal solution would have been combining the diagram editor and the behavior tree editor into one application instead of being spread out between a number of applications.

Furthermore, by having a tool for the creation and design of behavior trees could "embed" the method more intuitively into the work process such that you would not have to learn the method or read the steps, but these would been done intuitively from the start through the workflow of the application.

11.5. Evaluation of Perception System

The perception system is used essentially as a method when trying to define what the perception of a given AI character feels and looks like, and it provides a set of building blocks which can be extended if necessary. At the moment there is no tool support for the perception system, and this makes the design and construction process for character perception cumbersome, as everything has to be done by hand in code and on paper, or by using some available diagram tool. This means that you can quickly loose the overview of how the designed perception system works, and it is hard to change and maintain.

If we see the perception system as method rather than a deployable piece of software, then it should be possible to define either a work flow or a concrete method for designing and creating such a perception system. With a method defined a tool could be created for designing and creating a perception system. While such a set of tools does exist for Gluon by way of the Components created for this purpose, it should of course be possible to use SMARTS outside of Gluon.

As such, one solution would be that the design process of the perception system could be integrated into the SMARTS Designer, such that it can be used in conjunction with characters there, and instead of defining the perception system and the behavior tree of the character separately, you could design them together creating a unified AI character in one tool.

On the technical side, the system does in fact function: Values gathered through the system are adjusted correctly. However, due to shortcomings in Gluon's scripting system (functions cannot be called between script objects, only from scripts to

C++ objects) the use of the system in The Quantum Sea became too cumbersome. Based on this, and the fact that it would be another level of variables, it was decided not to use the perception system in the evaluation of the measurements above, and simply leave the evaluation of the Perception System to a future project.

11.6. Conclusion

We have tested the local and global methods on two different test cases. All of the results of the local method provided new weights which were expected. The test of the global method on the constructed example returned new weights resembled the ones returned from the local method. For The Quantum Sea the global method returned weights which is not intuitively to spot if the new weights were as expected. This could be that the expanding of the behavior tree is not done correctly or the complexity of the behavior trees. Furthermore the two different methods should be usable together if necessary, but the local method should be sufficient for typical applications in behavior trees, such as selecting between alternative strategies.

We have used the method for designing behavior trees extensively during this project, and it is very intuitively to use. it is essential to have tools for this task, as it would become very cumbersome to maintain and do in hand and code. Because we have SMARTS Designer the SMARTS Library has become more flexible, but the design process and creation of behavior trees should have been in the same tool, e.g in the SMARTS Library.

A perception was also created and was essential used as a method when it was finished. Currently there exist no editor for the perception system, which makes it very time consuming and hard to use as everything has to be done by hand. One solution could be integrating the design and creation into SMARTS Library such that a unified game AI character could be created from one tool. Because of limitations of the Gluon the perception system was not used during the testing of the methods.

Part V

Concluding Matters



With all methods explained and all data collated and explored, conclusions must be drawn. As the word goes, all good things must come to a conclusion, and this is ours.

Conclusion

Different metrics have already been defined before for measurements of the entertainment value of an AI opponent[28]. These metrics, though, are heavily based on time and the coverage of a level during runtime. Therefore we wanted to define metrics which could be run on the behavior trees and just use the success rates of selectors' children for optimizing the behavior trees.

We defined a metric for behavior diversity by using the entropy[23] of the behavior trees by using the weights on probability selectors for calculating this metric. We also defined a challenge metric which is the formula for KL-divergence[24][18]. These two metrics added together then defines the interest metric, but the interest metric was instead used for defining the weight for a given child behavior in a probability selector.

Before using these new metrics for optimizing the weights in behavior trees, we wanted to be able to do some optimization on the behavior tree structure itself, which led to a series of optimization strategies. By using these optimizations it is possible to combine certain behaviors together without changing the overall behavior of the tree.

From the three new metrics and the structure optimization of the behavior trees it was possible to devise three different strategies for optimizing the weights in a behavior tree. The first was global which meant that the whole behavior tree is collapsed such that there are only one probability selector, the root, with a number of child behaviors. Then the calculations could be made and the behavior tree was expanded to its original form. The second method was a local method but where it was possible to concatenate selectors together, where it afterwards calculated the new weights and expanded the selectors again. The last method is a pure local method and only calculates the weights for each probability selector without any form for behavior tree structure optimization.

We have looked at the implementation of the SMARTS Library and analysed the problems which occurred during the first implementation of the library. We had a dual layer of behavior trees, which was unnecessary and just took up more memory than necessary, furthermore we had some problems using threads because of our testing bed was based on timing. We reimplemented the behavior tree structure such that it was a single layer, and removed any use of threads from the behavior trees. We instead changed the execution to use a scheduler instead.

Furthermore we have also added a perception system to the SMARTS Library which enables the AI to be able to sense the world and get information that corresponds to its view on the world.

We have also looked at how to incorporate SMARTS into Gluon, where we first looked at how Gluon works and how it supported components and plugins. The next was to integrate SMARTS into Gluon which was done using the components and plugin systems in Gluon and GluonEngine. We implemented new components

and plugins for the SMARTS Library such that the functionality of the library was accessible in Gluon Creator.

In the process of this integration, support was added directly to SMARTS Library for creating scriptable versions of some of the classes in the library. This allows other users to implement a scripted system akin to that found in Gluon, without having to actually make use of GluonEngine. This support was created for all the parts of the library where it makes sense, meaning any class which is otherwise intended to be reimplemented when using SMARTS Library in a game.

The game *The Quantum Sea* was implemented in a scaled-down version compared to the one described in the game design documents, as implementing the full game would have been entirely out of scope for this project. However, enough was implemented to allow us to perform evaluation of the perception system and behavior trees in general through use of the game.

The chapters pertaining to this describe both the thought processes which went into the game design itself, and that of the code design. This provides readers with the possibility to follow the concepts without having to study the code of the game in any serious depth.

Tests of the local and global methods has been performed on two different test cases, first *The Quantum Sea* and last on a constructed example. We performed three tests in each test case with different setups of the k_1 and k_2 , one for equal, one for more weight on diversity and one on weight on challenge. The tests of the local method on each test case provided results which meet the expectations of what the new weights should be. The global method resulted in new weights for the constructed example which resembles the pattern of the local method, while the global returned non sensible weights for the *The Quantum Sea*. The reason could be that the behavior trees used in *The Quantum Sea* are more complex and uses special behaviors such as parallels and infinite loops. The local method

The construction of behavior trees show that not only does the method described in *Where Game AI Meets Academic AI* [16, pp. 12] work, it shows that it is intuitive and is in fact what one implicitly does when constructing a behavior tree. One of the great advantages and essential parts of this method is to have an editor, and in this case it is the SMARTS Designer. The editor lacks some functionality which would increase the usefulness of it, and this feature is the possibility for designing the behavior tree directly in the tool.

The perception system is used as a method which means that doing the design and creation by hand is cumbersome and time consuming, because there are currently no tools for designing and creating perception in SMARTS. A good solution would be to integrate the design and creation process into SMARTS Designer such that a unified game AI character can be created with use of one tool. It has not been possible to test the perception system further than that values are correctly adjusted, because of limitations in Gluon. Due to this and the fact that it would be a further variable all tests have been performed without the perception system.

As we reach the end of the report, we believe that we have arrived at solutions to the various parts of the purpose of the project as defined in Section 1.1 on page 1: The behavior trees were formalized, methods for evaluating them were devised, the SMARTS system was revised and extended, the Gluon integration completed, a game prototype constructed and the various items needing evaluation

were evaluated.

We would like to finish by thanking the reader for following us this far, and invite you to continue by reading the next chapter, which contains various musings on the project itself as well as the various items in it. It is our hope that your experience with us has been as enjoyable as the creation of the project.

Discussion and Future Work

The following is a set of discussions and musings on various topics related to this report's subject matter, but which is of a looser nature, which means it does not fit into the flow of the report as such. We do, however, recommend that you still read it, if your interest has been peaked by the preceding chapters, on the basis that they suggest further research possibilities, as well as other items of relevance.

The first item up for discussion, but one so small that it does not warrant its own section, is that of prefab support in Gluon. During the creation of the test game used in this report, *The Quantum Sea*, the need for a prefab system was highlighted. As such, we highly recommend that this be made a focus for the next development cycle of the game engine.

13.1. The Quantum Sea

On the topic of *The Quantum Sea* itself, it was discovered that while the game itself does indeed function rather well (and has turned out to be decidedly more difficult to play well than anticipated), the various values for the particles need tweaking work, something which was anticipated but still begs mention. However, as the only test audience for the game has so far been the authors, it is suggested that effort be put into play-testing the game prototype with unprepared players, to evaluate the enjoyment level others will experience.

Further, it has been suggested that a generalized version of the implementation of the music player found in *The Quantum Sea* be translated into C++ and inserted into Gluon as an official component.

13.2. Metrics and Methods

The global method collapses and expands the behavior tree it works on, but this is a very inefficient way of doing the calculations. It could be possible to do the weight calculations without needing to collapse the tree. It could be possible to use hierarchical entropy for this purpose or customize the method for use on behavior trees.

Furthermore we are not sure if the collapsing and expanding of sequences is done correctly. These operations are based on tests and intuition, but we do not have any strongly defined method for doing this. This could prove to be an important matter if you want to use the global method as it is now. However as mentioned earlier this could prove to be unnecessary if a variant of hierarchical entropy is used instead.

13.3. Extensions to SMARTS Behavior Trees

One of the most basic concepts on which behavior trees are based is that it should be easy to author behaviors. One way we can do this is by limiting repetition in the design of behaviors. We already have one tool to save development time and memory usage with repeated behaviors: References. However references are limited in that they do not allow for reuse of general behavior structure patterns. Behavior patterns such as searching for an enemy and then attacking said enemy cannot be encoded generally as a pattern in the current behavior tree implementation.

We suggest an Object Oriented Programming implementation similar to the one described in [20]. This extension of OOP to Hierarchical State Machines transfers easily to behavior trees as behavior trees can be seen as HFSMs with an extra set of restrictions. This could be a huge part of what makes behavior trees a convenient choice for a game designer. This method has previously been employed by the authors of Halo, in their implementation of behavior trees[12].

13.4. Adjustments to SMARTS

The current version of SMARTS Library a behavior will continue executing until it is done. This means that the current implementation of a behavior tree could execute a behavior which has no relevance to the giving situation, because the world state has changed.

There are several ways for getting past this problem, where the first is to use interrupts such that behaviors can be interrupted during execution. Halo 2TM uses something similar to this but it is instead called impulses and are used as event triggers during the execution[12]. Another way is to check any conditions or decorators on behavior before executing it, which would catch any changes in the world state, and either prevent the behavior execute or continue execution.

Halo 2TM also uses a a prioritized list for execution of behaviors[12] which is basically a standard selector as described in section 2.1.1 on page 6, where the highest prioritized child has precedence and will be executed first if possible while having the ability of interrupting the other children if necessary. The only change necessary is to add conditions on each children of selector which defines if it is possible to run the child. This change could be in subtrees and even in the root of the tree, such that both the root and subtrees could “adapt” to the current world state. It would then be possible to set a flag, if the subtrees should have precedence over the root when checking the conditions such that the child selectors would be run first and lastly the root or the other way around.

While the current implementation of SMARTS Library uses a scheduler for running the behaviors it could be useful to add the possibility for the users to select if a character should use the scheduler or threads for executing parallels as this could prove useful for devices with multiple cores, while the other option could be used on the more powerful mobile phones. Another option could also be allowing the whole behavior tree to be run in a thread such that it runs without any uses of the scheduler. Giving those possibilities the designers could “scale” or chose the most appropriate execution method of the behavior trees.

13.4.1. Perception System

Through the use in The Quantum Sea of the new perception system, one item which has surfaced as a potential point for improvement is that currently the game programmer needs to know about the perception atoms. That is, the programmer needs to know to construct a perception atom before reading information about a piece of perception information.

We thus suggest that this can be abstracted away by adding a function on the perception system instance found on a Character, which takes a perceptionInfo as an argument and returns the information known by the player about that perceptionInfo. Behind the scenes, this would then keep track of perceptionAtom instances, thus reducing the boilerplate code required in the existing system. This method is essentially already employed in The Quantum Sea.

13.4.2. Measurements

While we have provided code for calculating the various measurements for use in this report, a vital piece of work is missing which is needed to support game designers, or specifically behavior designers, in their quest towards the perfect behavior tree: For the measurements to be useful to them, the measurements need to be available through the tools they already use, in our case SMARTS Designer.

We thus suggest that effort be put into the creation of this integration of the measurements calculator, which would finally close the loop and allow for the use of the tools all the way from idea through creation and refinement. However, the insertion of these values, and indeed any functionality they support, is non-trivial; they need to be both effective, but also non-intrusive to the use of the software. As such, this would potentially be an interesting target for a student project on the topic of usability and interaction with external customers.

- [1] H. Abelson, G.J. Sussman, and J. Sussman. Structure and interpretation of computer programs. Citeseer, 1996. 9
- [2] G. Alt. The suffering: A game AI case study. In Challenges in Game AI workshop, Nineteenth national conference on Artificial Intelligence, pages 134–138, 2004. 9
- [3] M.V. Aponte, G. Levieux, and S. Natkin. Scaling the Level of Difficulty in Single Player Video Games. Entertainment Computing-ICEC 2009, pages 24–35. 19
- [4] Tucker Balch. Hierarchic social entropy: An information theoretic measure of robot group diversity. Autonomous Robots, 8(3):209–238, 2000. 15
- [5] R. Berger and H.D. Burkhard. Modeling complex behavior of autonomous agents in dynamic environments. Proc. Concurrency, Specification and Programming (CS&P), 2006. 9
- [6] Joanna J. Bryson. The behavior-oriented design of modular agent intelligence. <http://www.cs.bath.ac.uk/~jjb/ftp/AgeS02.pdf>. 80
- [7] Bungie™. Building a better battle: Halo 3 ai objectives. <http://www.bungie.net/images/Inside/publications/presentations/betterbattle.zip>. 9
- [8] Bungie™. Evolving halo 3's behavior tree ai. <http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/gdc07.pdf>. 9
- [9] Alex J. Champanard. Understanding behavior trees. <http://aigamedev.com/open/articles/bt-overview/>. 1, 9, 10
- [10] Alex J. Champanard. Using conditions as the building blocks of a sensory system, 2007. <http://aigamedev.com/open/tutorials/conditions-sensory-system/>. 51
- [11] M. Cutumisu and D. Szafron. An Architecture for Game Behavior AI: Behavior Multi-Queues. 2009. 26
- [12] C.C. Cycle. Handling Complexity in the Halo 2 AI. 10, 26, 114
- [13] Daniel Sanchez-Crespo Dalmau. The structure of a game ai system, 2003. <http://www.peachpit.com/articles/printerfriendly.aspx?p=169492>. iii, 51, 53
- [14] Patrick et al. Inelastic collision, 04 2010. http://en.wikipedia.org/w/index.php?title=Inelastic_collision&oldid=3244446299. 78
- [15] David Harel. Statecharts: A visual formalism for complex systems, 1987. 10

- [16] Dan Leinir Turthra Jensen, Kim Jung Nissen, and Morten Justesen. Where game ai meets academic ai, 2009. i, iii, iv, 1, 2, 5, 11, 37, 38, 39, 40, 51, 61, 62, 63, 80, 91, 97, 101, 104, 110
- [17] Frank Karlitschek et al. Open collaboration services (v1.4), 03 2010. <http://www.freedesktop.org/wiki/Specifications/open-collaboration-services>. 121
- [18] S. Kullback. Information theory and statistics. Dover publications Mineola, MN, 1997. 20, 27, 109
- [19] Tom Leonard. Building an ai sensory system: Examining the design of thief: The dark project. 2009. 51
- [20] W. Mallouk and E. Clua. An Object-Oriented Approach for Hierarchical State Machines. In Proceedings of the SBGames conference in Computing, pages 8–10, 2006. 26, 114
- [21] M. Mateas and A. Stern. A behavior language for story-based believable agents. IEEE Intelligent Systems., pages 39–47, 2002. 10
- [22] M. Mateas and A. Stern. A Behavior Language: Joint action and behavioral idioms. Life-like Characters. Tools, Affective Functions and Applications, 194, 2004. 10
- [23] CE Shannon. A mathematical theory of communication. ACM SIGMOBILE Mobile Computing and Communications Review, 5(1):55, 2001. 15, 27, 109
- [24] J. Shlens. Notes on Kullback-Leibler Divergence and Likelihood Teory. System Neurobiology Laboratory, Salk Institute for Biological Studies, California, 2007. 20, 27, 109
- [25] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma. Difficulty scaling of game AI. In Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON 2004), pages 33–37, 2004. 19
- [26] Thomas Vander Wal. Explaining and showing broad and narrow folksonomies, 2 2005. http://personalinfocloud.com/2005/02/explaining_and_.html. 126
- [27] Koen Witters. dewitters game loop, 07 2009. <http://dev.koonsolo.com/7/dewitters-gameloop/>. 123
- [28] G. Yannakakis and J. Hallam. Capturing Player Enjoyment in Computer Games. Advanced Intelligent Paradigms in Computer Games, pages 175–201. iii, 13, 15, 19, 22, 23, 109
- [29] Teach yourself statistics. Statistics tutorial: Measures of variability, 6 2010. <http://stattrek.com/Lesson3/Variability.aspx>. 14

Part VI

Appendix

Introducing Gluon

One of the problems faced by game creators is that of publishing; that is, getting the game from the creator and into the hands of potential players. Gluon provides a way of doing this, based on the Open Collaboration Services (OCS)[17]. This distribution system is made possible because games created using Gluon's engine, cleverly named GluonEngine, do not have binaries. As such, there is nothing to compile for each of the different platforms.

In stead, GluonEngine based games are based around a simple description markup language named Gluon Definition Language (GDL), which is clear text and both human and machine readable, and all game logic is created using scripts. GluonEngine itself provides the computationally heavy functionality (such as graphics, asset management and so on) through a system of objects called Components. The following is a short introduction to how GluonEngine based games function. After this Gluon Creator is introduced, followed by a description of the Gluon distribution website. Lastly the collection of applications collectively known as Gluon Player is introduced.

Figure A.1 is a description of the process of creating games with Gluon. It shows how a game is first conceived by a game creator, who then through the KDE Development Platform based Gluon Creator creates the game and publishes it to the world through OCS and eventually to the players of games, who can be on any of a wide variety of operating systems.

A.1. The Gluon Packages

Before we start, the structure of Gluon should be introduced shortly. The diagram in Figure A.2 on the following page shows how the dependencies between the different packages in the Gluon software system are structured in a strict, hierarchical



Figure A.1.: The flow of Gluon based games, from conception to playing and back

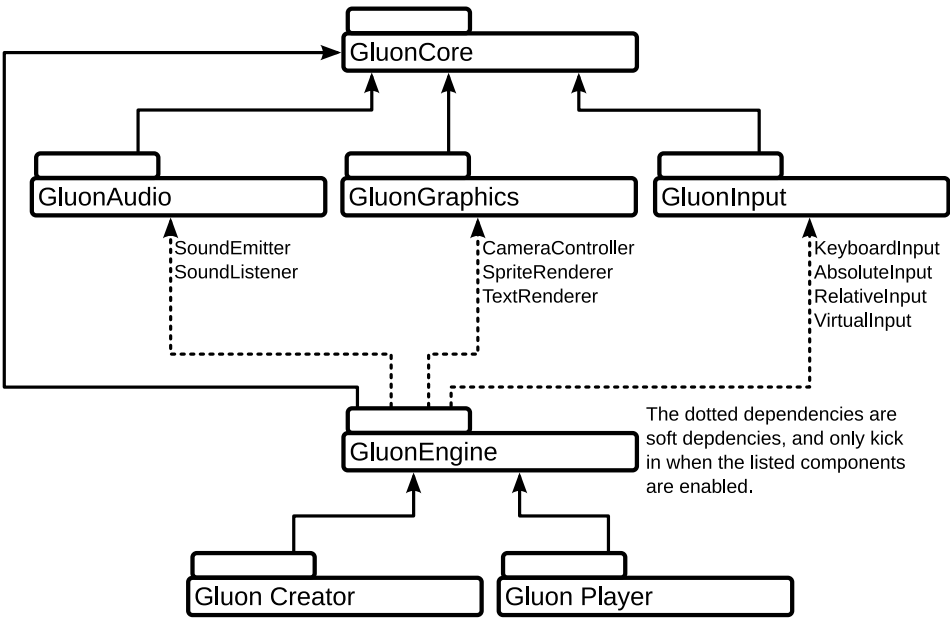


Figure A.2.: The hierarchy of the different packages in Gluon

manner.

At the top you find the **GluonCore** package, which contains common functionality for all of Gluon - this includes such things as the GDL handling system and the cross platform friendly singleton creation system used in the different libraries.

This is then inherited by the four main libraries: **GluonAudio**, which takes care of sound playback, **GluonGraphics**, which is the OpenGL based graphics rendering engine and **GluonInput**, which is the powerful game-centric input device handling system created in part by the authors of this report.

Underneath this you find **GluonEngine**, which is where Gluon becomes truly game centric. This is where the system described in the introduction sits, including the game loop (which takes care of updating and drawing the screen at appropriate intervals), and where the Component and Asset system sits. Since this is entirely modular, the engine does not have any hard requirements other than GluonCore, but rather has optional dependencies through numerous Component and Asset classes. Note that the Components shown are not the only ones available, but rather a few examples.

Finally the **Gluon Player** and **Gluon Creator** packages represent the Gluon Creator application, and the set of applications collectively known as Gluon Player. These depend only on GluonEngine, and all other contact with Gluon's packages is conducted through the engine's GameProject system.

A.2. Gluon Engine

From the game developer's perspective, the Gluon Engine is essentially a structured way of creating games, based on a hierarchy of types of objects:

- A **GameProject** contains a number of Assets.
- **Assets** are the only way of accessing the filesystem from inside the game, and provide handling of content files, such as sound files, graphics and so on. A special type of Asset is the Scene asset.
- The GameProject can contain any number of **Scenes**, which in turn contain any number of GameObjects.
- **GameObjects** represent the scene graph, and in themselves only exist as a position in the world. They contain any number of other GameObjects, which in turn contain any number of GameObjects, and so on. They furthermore contain any number of Components.
- **Components** are the way the game creator adds functionality to GameObjects. A Component can be for example a TextRenderer, which will render a text at the position in space represented by the GameObject it is attached to, or a KeyboardInput, which will react to the input of a key on the keyboard.

This is the persistence layer for games which the Gluon Engine provides. To make it a complete game engine, however, it must also have a game loop - the equivalent of an event loop in other applications. This is provided by the class `GluonEngine::Game`, which also takes care of keeping track of which scene is the current scene. The game loop functions as follows:

- Call `start()` on all GameObjects in the current scene
- Loop:
 - Call `update()` on all GameObjects at a specified interval
 - Call `draw()` on all GameObjects as often as possible
- When game is stopped, call `stop()` on all GameObjects

This game loop is heavily inspired by the "Constant Game Speed independent of Variable FPS" game loop described by Koen Witters[27]. There is a second game loop available in the class as well, which can optionally be used, which is a fixed frame rate game loop, where update and draw are called consecutively every time. This means that the game's update function will be run exactly as many times as the drawing function. While this seems like a good idea on the surface, the problem is that it does not degrade well with slow hardware. In a few situations, however, this simpler approach is appropriate - for example puzzle games where the update function's precision is less important than other types of games.

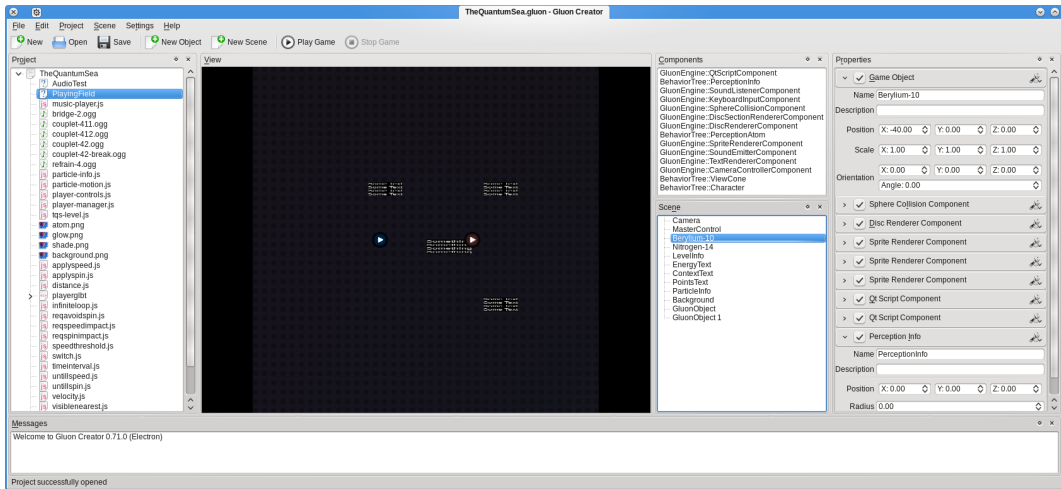


Figure A.3.: Gluon Creator, editing a scene in the game The Quantum Sea

A.3. Gluon Creator

While it is possible to create GameProjects for GluonEngine entirely by hand, using only a text editor to create the GDL files, this is not really convenient. Enter Gluon Creator, a tool designed with creative persons in mind. What this means is that rather than showing people code for things, as it happens when working with for example the 2D game engine LÖVE¹ or Microsoft's XNA Game Studio², Gluon Creator allows for the construction of games using a combination of drag and drop ordering of the GameObjects, property setting and direct manipulation.

This in turn allows people who are not familiar with programming create at least parts of the games. It is still required for for a programmer to create the final glue which binds the elements in the in the game together, that is the game logic. This, however, is thus reduced to a minimum, so that the work load is spread out more evenly, and the pressure on the programmer is reduced heavily.

The game logic is provided by a special component, the Script component, which as the name indicates uses scripts to manipulate and store information about the game world not stored implicitly by the GameObjects themselves. This includes arbitrary things such as health, what should happen when activating an input and other such game specific functionality.

As shown in Figure A.3 Gluon Creator is based around a central view of the level itself, with the project contents on one side and the contents of the current scene shown on the other. The Messages panel allows the game creator to view any errors that might be provided by any script, and the Property view provides game creators with a convenient, centralized way of manipulating any exposed settings in the Components attached to the currently selected GameObject - including those exposed from a script.

¹Getting Started with LÖVE: http://love2d.org/wiki/Getting_Started

²Getting Started with XNA: <http://creators.xna.com/en-US/education/gettingstarted>



Figure A.4.: Marketing material for the Gaming Freedom community website

A.4. Gaming Freedom

The website onto which Gluon Creator publishes the games created with it is called GamingFreedom.com. This website is based on the OCS based openDesktop.org framework provided by h i v e 01, a German company specialized in social networks, and which is heavily involved with open source projects. The company is furthermore owned by Frank Karlitschek, who is also the main author of the OCS specification.

Pictured in Figure A.4 is the current front page of the website. What is not visible here is that not only is Gaming Freedom a place where you can upload games and download them to play on your local device using Gluon Player, as described in the next section. It also doubles as a way to conduct mediated interaction between both the players of games and the creators of games - in short, a community website for Gluon's wide ranging community.

To assist in this community feel, the following is a selection of the functionality included on the website:

Avatars Simple as it seems, user avatars on community websites supply an important way of identifying friends: Humans are much faster at identifying shapes and colors than they are at identifying words, and as such a user avatar provides a straight forward way of doing this.

Achievements A relatively recent development in the world of gaming is that of achievements: The idea that in various games, there are other goals than

those set down by the normal flow of the game, basically providing an extra game within the game. This concept is known as achievements. Each game will have any number of achievements, and most games publish information on how to achieve each achievement in more or less vague ways, but always the number of achievements in the game is public. As such, the players of games are then able to show their achievements. Gaming Freedom itself provides, in addition to the in-game achievements, a way of gaining achievements for the community itself (such as publishing a number of highly rated games, posting a number of highly rated comments, performing a lot of tagging and others).

Comments To allow both the players and creators of games to speak with each other, every item on the website can be commented on: Games, players, creators, even other comments.

Concepts However simple it might be to create games with Gluon, it will still require skills that not everybody has. As such, Gaming Freedom provides a forum for people to write up and discuss ideas for games they might wish to create.

Friends One of the core concepts of community websites is that of friending people. However, rather than having a flat, Facebook-like structure of everybody simply being friends, Gaming Freedom will let you structure your friends by adding tags to them - same as any other piece of information, except you are the only tagger for your friends.

Player Profile The profile of any player of games on Gaming Freedom provides a wide range of information on that player: What games they have played and how much, what Achievements they have gained, what comments they have made. For creators of games it also includes which games they have published.

Ratings Some people may be less vocal than others, but still wish to take part in the community in other ways - or simply wish to show their support or dislike of something in a non-vocal manner, rather than saying again the same things others have already said in a comment. So, they are then allowed to rate on each item on the website as well.

Tags The term tagging in Web 2.0 covers two different concepts: Broad and narrow folksonomies. A narrow folksonomy is the type of tagging applied on anonymous websites, where the tagging is based on a word being connected to an item of information. A broad folksonomy is used on sites which require logging in to allow tagging, and connects the user, the word and the item of information, so that the applicability of a word to an item can be weighted on how many users have assigned that word to the item. Gaming Freedom, as a community website, employs broad folksonomies, except for the friends page mentioned above, which only employs narrow folksonomies[26].

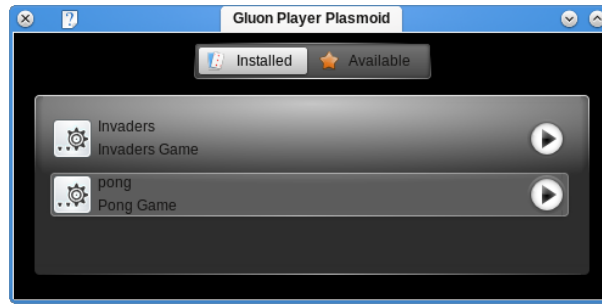


Figure A.5.: The Gluon Player Plasmoid in the version available at time of writing

A.5. Gluon Player

The last link in the distribution chain is Gluon Player. Firstly it functions as a device client for the Gaming Freedom website. That is, it enables the players of games to use all the functionality of the website, without loading up a web browser. This follows the concept of the Silk project, which wishes to "separate the web from the browser"³.

Further than this, it allows the players of games to take the games with them anywhere and everywhere. As indicated by the infographic in Figure A.1 on page 121, Gluon Player runs on a multitude of platforms, including hand-held devices such as Nokia's N900 and the Compal Jax10, as well as full blown desktop machines. This can be seen in real life in Figure A.5.

An important feature of Gluon Player is that it does not require you to be logged in to anything to play games. In fact it does not even require you to be logged in to download games new games to the device. This is possible due to the licensing of GluonEngine based games, which is based on the Free Culture movement⁴.

³The Project Silk website: <http://techbase.kde.org/Projects/Silk>

⁴Gluon Licensing Explained: <http://www.gamingfreedom.org>

Design for The Quantum Sea

The next four pages contain scaled-down versions of the design sheets created for the game The Quantum Sea over the course of a one month solid brain storming session during the new year holiday between 2009 and 2010. This design is the basis for the game created as a test bed in this report.

The Quantum Sea

perception is everything perception is key

Collide the world's smallest elements and score lots of points in the process

Right now, experimenting on the smallest things requires the largest tools. What if we could control the sub-atomic world directly? *The Quantum Sea* puts you inside that world, letting you induce your consciousness into individual particles in collision events as seen in particle accelerators like the Large Hadron Collider. *Score points* by smashing the smallest things we know and *watch the fireworks*.

Target Audience

Casual Gamers

primarily ages early 20s through late 30s

Unique Selling Points

- LHC is in the news and there are no casual games capitalising on this fact
- High-pace gameplay in a serene setting
- Point-grinding not unique, but very popular
- Old-school retro style game graphics coupled with high-tech user interface

Platform

Gloun

allows for simple distribution and global high-score systems (including achievements)



Game Mechanics

Game Progression

Levels are connected in a tree-like structure rather than a linear progression. This is done based on the concept in science that one answer often leads to many new questions.

Points System

Collisions between different particles give points (angle, speed of impact plus type of particles decide amount of points). Once a particle collides with another particle, both particles stop existing and either become one or more other particles, or in the case of non-composite particles become energy and simply stop existing as particles. Should the situation arise where there are no particles or only one particles left in play, the level ends.

Upon the level being finished, the amount of energy left for the player defines a multiplier. The multiplier levels are set per level. The total number of points for a level is defined by the amount of gathered points, plus this multiplier.

The point display shows the current multiplier and the current points, and how many points the player would have if the level ended at the current moment in time (for example showing "3 x 18248" and "54744"). This is done to allow for easy comparison with the current high score.

Perception

Each particle can only see part of the world (different types of particles have different views of the world - e.g. electrons have perfect local knowledge and no far knowledge, neutrons only know what is directly in front of them in a cone of a few degrees). This is presented to the player by means of altering the visible area of the game world when switching between each particle.

Trails

Trails show the past, and fade over time. This, combined with the time factor above, also ensures that it is not possible to have more than a certain amount of particles in the world at the same time.

Win/Lose Conditions

- Winning is done by gathering points on each level - using both local and global high-score
- Losing is not possible (though winning less is possible)

Time

Time limits are dictated by the amount of particle trails on the screen. Once the amount of trails in the world becomes too high (more than some percentage of the screen is taken up by trails) gameplay is stopped, the level ends, and in the case of a new high score, the score is saved to the local high score, and submission to global high score is offered.

Time Control

The player can control the passage of time, slowing it down further or stopping it completely. This costs the player energy - slowing it down costs an amount of energy relative to the player's level and how much time is slowed down, and stopping it completely costs a set amount relative to the player's level (duration of time being stopped is infinite).

Should the energy level ever drop to zero, time returns to its normal speed, and time cannot be controlled again until energy levels reach a certain percentage (this percentage depends on the player's level).

Interaction

The game is controlled by interacting directly with each particle in the world. Only one particle can be controlled at a time, and setting the two parameters speed and spin will cause others to change. Controlling the parameters is done

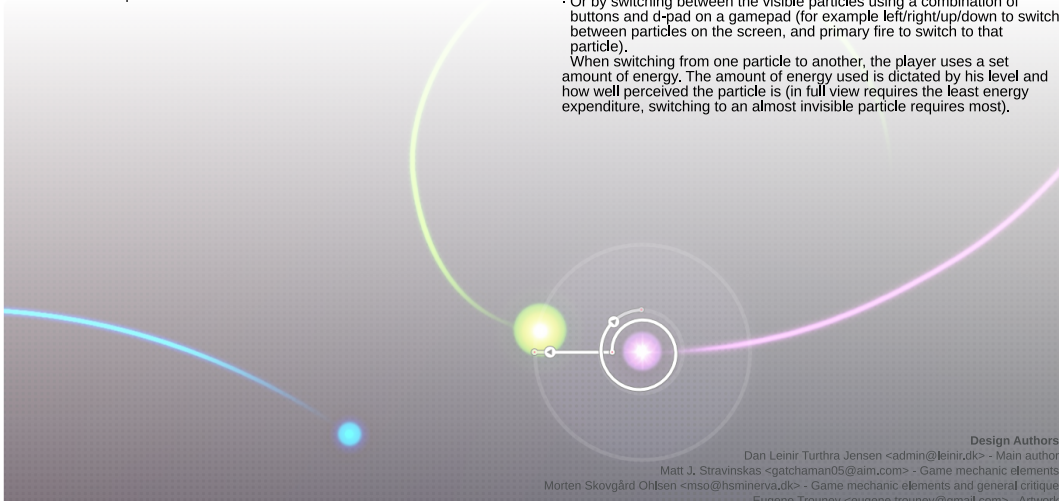
- Either through point and click/drag by using the mouse
- Or by direct interaction by using a gamepad, where one stick setting speed and spin (up/down, left/right)

Switching Particles

It is only possible to switch from controlling one particle to another if the new particle is perceived by the currently controlled particle. To switch, one of the following interaction methods is used:

- Either through point and click by using the mouse
- Or by switching between the visible particles using a combination of buttons and d-pad on a gamepad (for example left/right/up/down to switch between particles on the screen, and primary fire to switch to that particle).

When switching from one particle to another, the player uses a set amount of energy. The amount of energy used is dictated by his level and how well perceived the particle is (in full view requires the least energy expenditure, switching to an almost invisible particle requires most).



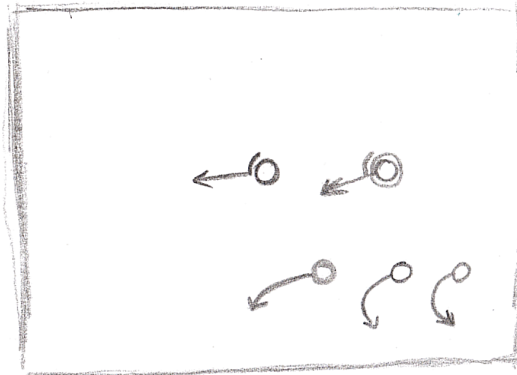
Design Authors

Dan Leinir Thurhra Jensen <admin@leinir.dk> - Main author

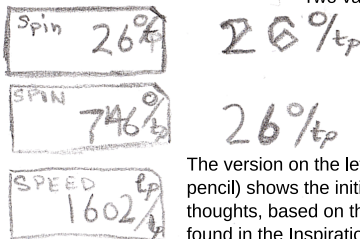
Matt J. Stravinskas <gatchaman05@aim.com> - Game mechanic elements

Morten Skovgård Ohlsen <mso@hsmiervia.dk> - Game mechanic elements and general critique

Eugene Trounev <eugene.trounev@gmail.com> - Artwork

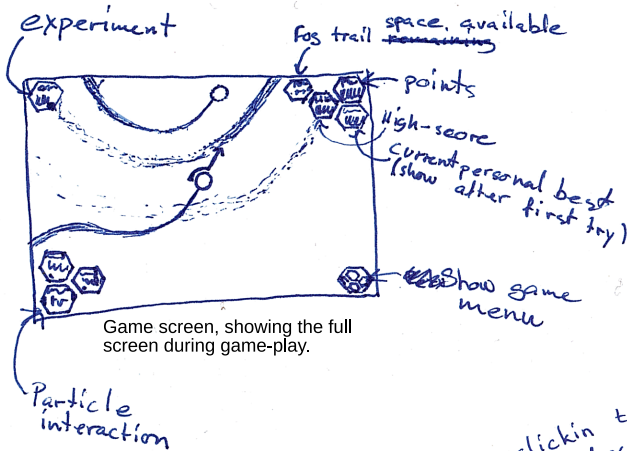
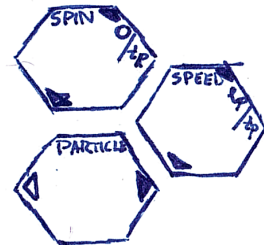


Variations on the theme of showing direction and spin directly around the currently selected particle.

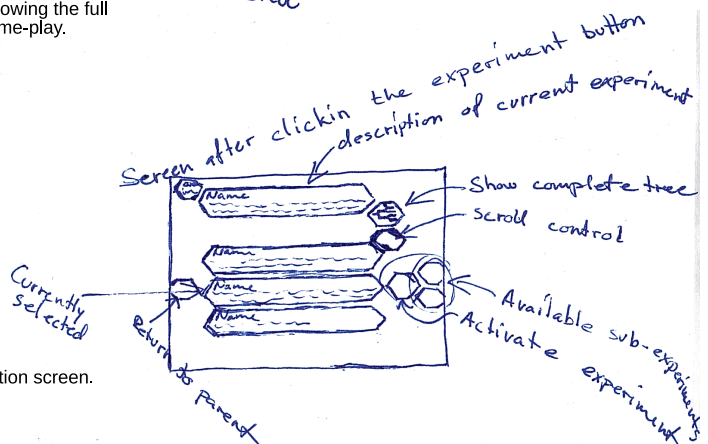


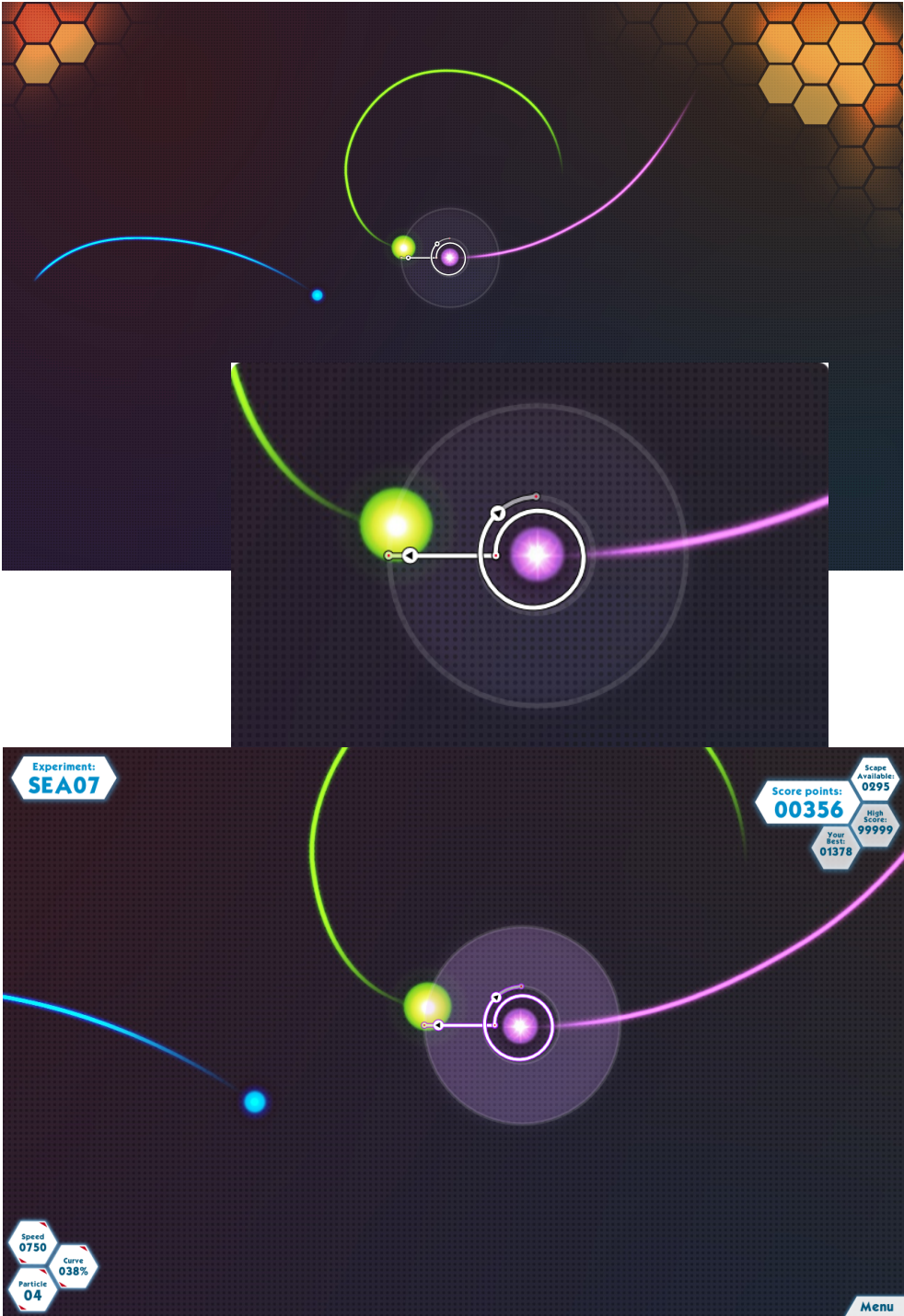
The version on the left (in pencil) shows the initial thoughts, based on the visuals found in the Inspiration section below.

The right side shows the chosen version, which rather than using the square objects uses a honey-comb layout. This allows for creating simple but consistent interaction elements: What is seen here are two types of scrolling - the two objects marked SPIN and SPEED are spin-boxes and will change their values by clicking on the arrows (up and down respectively), whereas the PARTICLE box will scroll through the particles. The white arrow shows that this is the active element.



Experiment Selection screen.





Visual Style

The game world is a 2D world, seen from above.

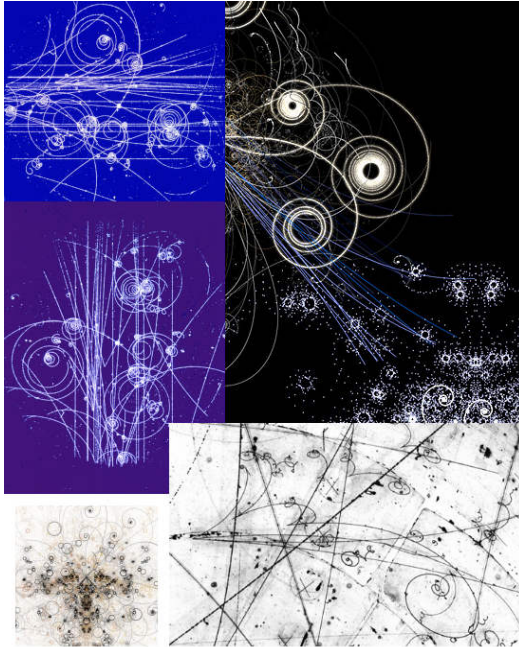
Inspiration

Game Visuals

Graphics from particle collisions, ghost trails, fog chamber trails such as <http://www.flickr.com/photos/nsccreative/3908863825/> and those on the right side of this sheet

User Interface

High information density AniméOS/MovieOS style graphics (Ghost in the Shell, Avatar, Matrix Revolution (Zion), Iron Man)



Audio Style

Minimalist effect sounds

Atmospheric, new-age style musical score (scour Magnatune for help)

Story

In the future, science has come so far that we can see even the smallest things, slowing our perception down so every detail is crystal clear. But, not everything is known yet. Particle collisions must still be carried out, to gain knowledge of the Universe. Now, however, they are not conducted in enormous particle accelerators. Instead it is done by manipulating the particles themselves through sheer power of mind, your perception of the world slowed down so each step can be seen in perfect detail. With one important difference: With your conscience existing inside the particles means that you can also only see the world from that particle's point of view.

Game Item Definitions

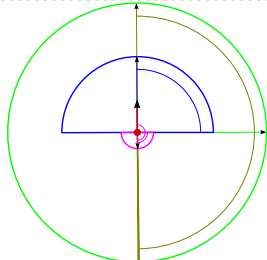
The following pages contain the non-collidable items and particle information sheets for the various items as used in the single level of The Quantum Sea created for this report.

C.1. Non-collidable Items

Particle	Initial spin	Spin deviation	Mass	Life time
Gamma Ray	$0^\circ/t_P$	$10^\circ/t_P$	1 u	$10 t_P$
Photon	$10^\circ/t_P$	$0^\circ/t_P$	1 u	$100 t_P$

C.2. Particle Definition Sheets

C.2.1. Isotopes

Initial Spin: $0^\circ/t_P$ Spin Deviation: $0^\circ/t_P$ Mass: 14.003241u Life Time: $1.80701e+11 t_P$					Particle Information Sheet Carbon-14	
<div> <div>14</div> <div>6</div> <div>C</div> </div>						
At... An impact will result in... 20% Nitrogen-14, electron, electron antineutrino 70% Helion-4, Helium-3, 2 protons, 7 neutrons 100% Helion-4, Helium-3, 2 protons, 6 neutrons, 1 up quark, 2 down quark					Left over energy... 0 MeV 0 MeV 0 MeV	
Perception Limit: $50 \ell_P$						
View Cone	Rotation	Extent	Radius	Precision		
1	0°	180°	$50 \ell_P$	0.5		
2	0°	90°	$25 \ell_P$	1.0		
3	180°	90°	$5 \ell_P$	0.7		

Initial Spin: $10^\circ/t_p$
Spin Deviation: $0^\circ/t_p$
Mass: 14.0030740u
Life Time: Non-decaying

14

N

7

Particle Information Sheet

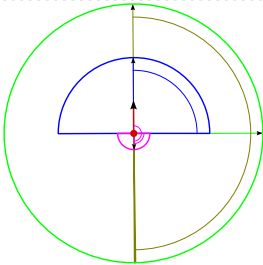
Nitrogen-14

At... An impact will result in...
70% Berylium-10, 3 protons, 1 neutron
100% Berylium-10, 2 protons, 1 neutron, 2 up quarks, 1 down quark

Left over energy...
0 MeV
0 MeV

Perception Limit: $25 \ell_p$

View Cone	Rotation	Extent	Radius	Precision
1	0°	180°	$25 \ell_p$	0.5
2	0°	90°	$20 \ell_p$	1.0
3	180°	90°	$10 \ell_p$	0.7



Initial Spin: $0^\circ/t_p$
Spin Deviation: $0^\circ/t_p$
Mass: 10.0135337u
Life Time: $4.76194e+13t_p$

10

Be

4

Particle Information Sheet

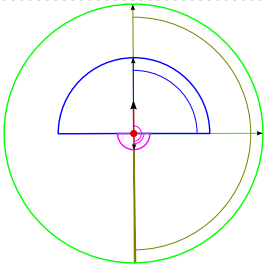
Berylium-10

At... An impact will result in...
20% Boron-10, electron, electron antineutrino
70% 3 Hydrogen-2, 1 protons, 7 neutrons
100% 3 Hydrogen-2, 7 neutrons, 2 up quarks, 1 down quark

Left over energy...
555.816 KeV
555.816 KeV
555.816 KeV

Perception Limit: $15 \ell_p$

View Cone	Rotation	Extent	Radius	Precision
1	0°	180°	$15 \ell_p$	0.5
2	0°	90°	$10 \ell_p$	1.0
3	180°	90°	$2 \ell_p$	0.7



136

Perceived Challenge

Initial Spin: $30^{\circ}t_P$
Spin Deviation: $0^{\circ}t_P$
Mass: 10.0129370u
Life Time: Non-decaying

10

B

5

Particle Information Sheet

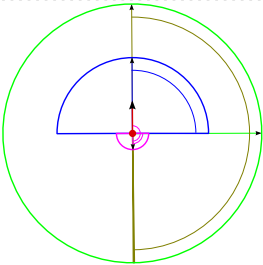
Boron-10

At... An impact will result in...
70% 5 Hydrogen-2
100% 4 Hydrogen-2, 1 proton, 1 up quark, 2 down quarks

Left over energy...
0 MeV
0 MeV

Perception Limit: $15 \ell_P$

View Cone	Rotation	Extent	Radius	Precision
1	0°	180°	$15 \ell_P$	0.5
2	0°	90°	$10 \ell_P$	1.0
3	180°	90°	$2 \ell_P$	07



Initial Spin: $10^{\circ}t_P$
Spin Deviation: $0^{\circ}t_P$
Mass: 2.0141018u
Life Time: Non-decaying

2

H

1

Particle Information Sheet

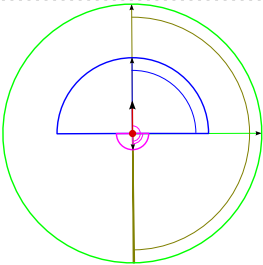
Hydrogen-2

At... An impact will result in...
70% 1 proton, 1 neutron
90% 1 proton, 1 up quark, 2 down quarks
100% 3 up quark, 3 down quarks

Left over energy...
1 MeV
1 MeV
1 MeV

Perception Limit: $100 \ell_P$

View Cone	Rotation	Extent	Radius	Precision
1	0°	180°	$100 \ell_P$	0.5
2	0°	90°	$50 \ell_P$	1.0
3	180°	90°	$10 \ell_P$	0.7



C.2.2. Particles

Initial Spin: $5^\circ t_p$
Spin Deviation: $5^\circ t_p$
Mass: 1.008664915u
Life Time: 14,761667 t_p

n

Particle Information Sheet

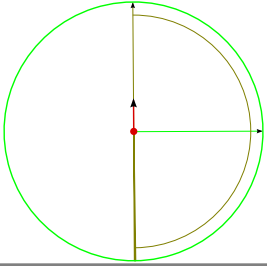
Neutron

At... An impact will result in...
90% 1 neutron
100% 1 up quark, 2 down quarks

Left over energy...
0 MeV
0 MeV

Perception Limit: 5 ℓ_p

View Cone	Rotation	Extent	Radius	Precision
1	0°	180°	5 ℓ_p	1.0



Initial Spin: $5^\circ t_p$
Spin Deviation: $5^\circ t_p$
Mass: 1.00727646677u
Life Time: Non-decaying

p

Particle Information Sheet

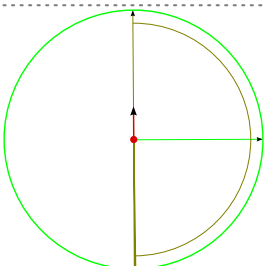
Proton

At... An impact will result in...
90% 1 proton
100% 2 up quark, 1 down quarks

Left over energy...
0 MeV
0 MeV

Perception Limit: 5 ℓ_p

View Cone	Rotation	Extent	Radius	Precision
1	0°	180°	5 ℓ_p	1.0



138

Perceived Challenge

Initial Spin: $5^{\circ}t_P$
Spin Deviation: $0^{\circ}t_P$
Mass: $5.4857990943(23)\times10^{-4}$
Life Time: Non-decaying

e^{-}

Particle Information Sheet
Electron

At... An impact will result in...
100% 1 photon

Left over energy...
1 MeV

Perception Limit: $1000\ell_P$

View Cone	Rotation	Extent	Radius	Precision
1	0°	180°	$100\ell_P$	0.1
2	0°	180°	$1000\ell_P$	0.5
3	0°	90°	$500\ell_P$	1.0

Initial Spin: $5^{\circ}t_P$
Spin Deviation: $0^{\circ}t_P$
Mass: $0.0000005u$
Life Time: Non-decaying

$\bar{\nu}_e$

Particle Information Sheet
Electron Antineutrino

At... An impact will result in...
100% 1 photon

Left over energy...
1.5 eV

Perception Limit: $1000\ell_P$

View Cone	Rotation	Extent	Radius	Precision
1	0°	23°	$1000\ell_P$	0.9
2	0°	45°	$500\ell_P$	0.4

C.2.3. Sub-particles

Initial Spin: $5^\circ t_P$
Spin Deviation: $0^\circ t_P$
Mass: $0.002u$
Life Time: $150t_P$

u

Particle Information Sheet

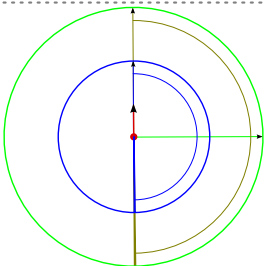
Up Quark

At... An impact will result in...
100% Gamma ray

Left over energy...
2 MeV

Perception Limit: $20 \ell_P$

View Cone	Rotation	Extent	Radius	Precision
1	0°	180°	$10 \ell_P$	0.1
2	0°	180°	$20 \ell_P$	0.9



Initial Spin: $5^\circ t_P$
Spin Deviation: $0^\circ t_P$
Mass: $0.006u$
Life Time: $150t_P$

d

Particle Information Sheet

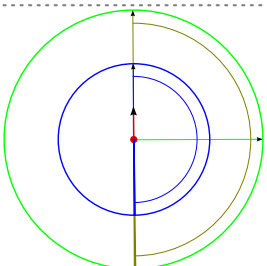
Down Quark

At... An impact will result in...
100% Gamma ray

Left over energy...
6 MeV

Perception Limit: $20 \ell_P$

View Cone	Rotation	Extent	Radius	Precision
1	0°	180°	$10 \ell_P$	0.1
2	0°	180°	$20 \ell_P$	0.9



Test Results

D.1. The Quantum Sea Results

D.1.1. Local Method

Equal

The results of the tests using $k_1 = 0.5$ and $k_2 = 0.5$.

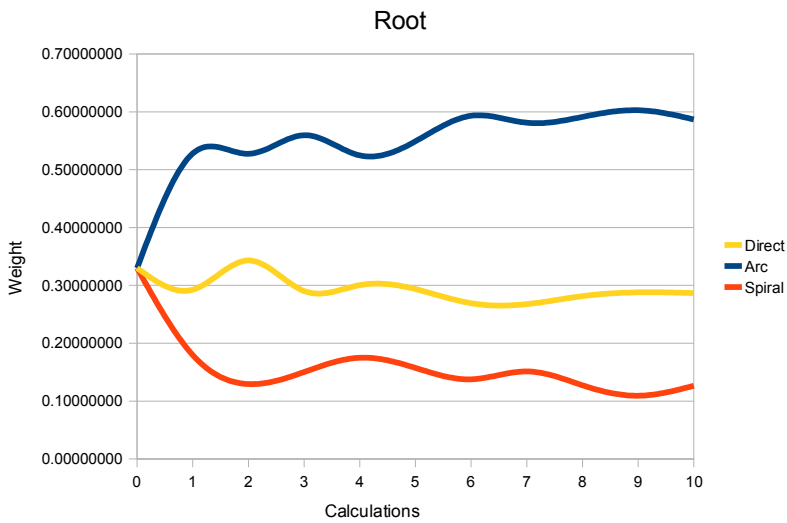


Figure D.1.: Graph over the weights on the Root children for equal k 's

Root					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Direct				0.33
	Arc				0.33
	Spiral				0.33
1	Direct	201	1848	0.108766	0.292461
	Arc	735	2073	0.354559	0.528037
	Spiral	64	562	0.0409731	0.179502
2	Direct	250	1394	0.17934	0.343101
	ArcRoot	724	1708	0.423888	0.527484
	SpiralRoot	26	1019	0.0255152	0.129415
3	Direct	169	1349	0.125278	0.290214
	Arc	802	1722	0.465738	0.559567
	Spiral	29	864	0.0335648	0.150219
4	Direct	186	1228	0.151466	0.30041
	Arc	771	1669	0.461953	0.524634
	Spiral	43	837	0.051374	0.174956
5	Direct	172	1276	0.134796	0.293616
	Arc	795	1687	0.471251	0.548993
	Spiral	33	852	0.0387324	0.15739
6	Direct	84	685	0.122628	0.269144
	Arc	897	1506	0.595618	0.593164
	Spiral	19	592	0.0320946	0.137691
7	Direct	86	693	0.124098	0.267607
	Arc	890	1521	0.585141	0.581092
	Spiral	24	605	0.0396694	0.151301
8	Direct	93	710	0.130986	0.281403
	Arc	890	1539	0.578298	0.591278
	Spiral	17	634	0.0268139	0.12732
9	Direct	98	761	0.128778	0.287941
	Arc	890	1577	0.564363	0.602784
	Spiral	12	647	0.0185471	0.109275
10	Direct	100	719	0.139082	0.2867441
	Arc	884	1518	0.582345	0.586745
	Spiral	16	591	0.0270728	0.12651

Table D.1.: The calculations for the Root node for equal $k's$

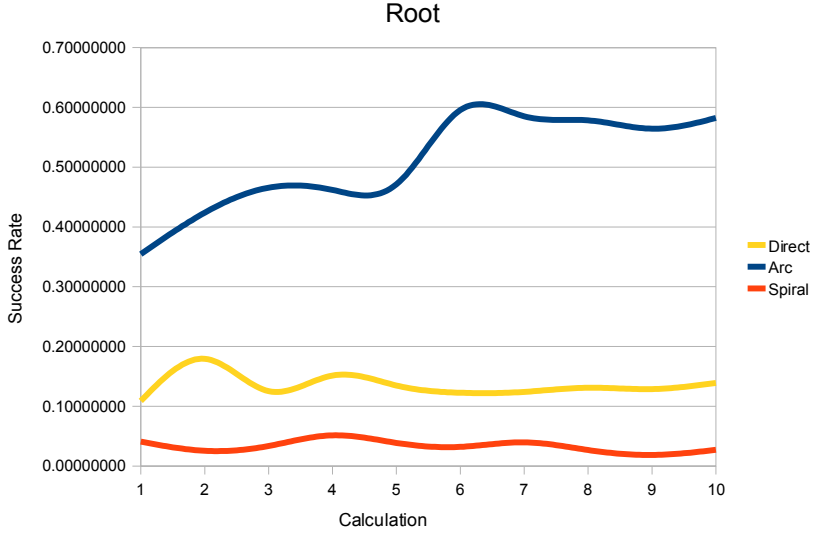


Figure D.2.: Graph over the success rates on the Root children for equal $k's$

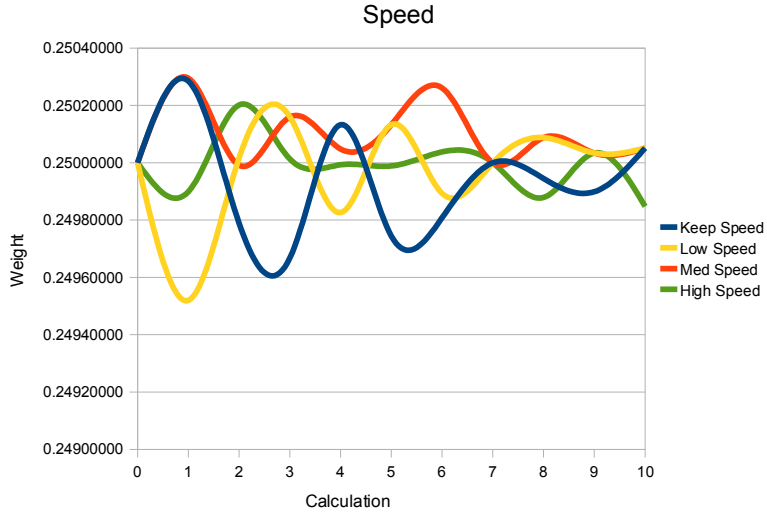
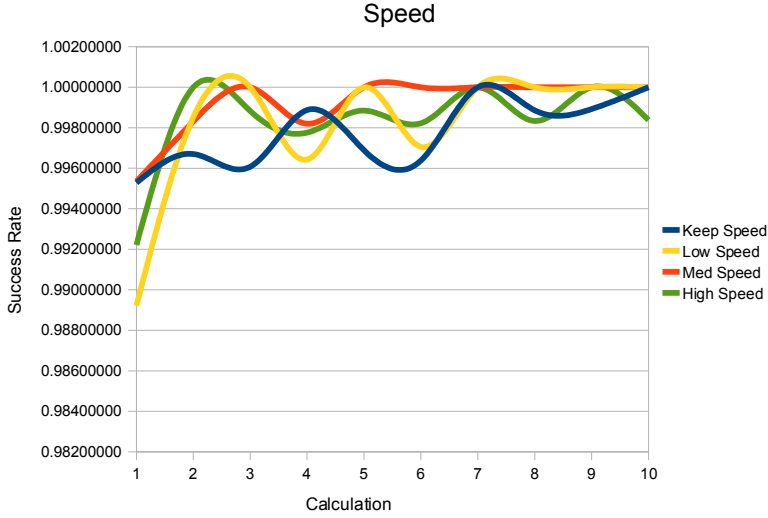


Figure D.3.: Graph over the weights on the Speed children for equal $k's$

Speed					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Keep Speed				0.25
	Med Speed				0.25
	Low Speed				0.25
	High Speed				0.25
1	Keep Speed	1264	1270	0.995276	0.250285
	Med Speed	857	861	0.995354	0.250295
	Low Speed	733	741	0.989204	0.24952
	High Speed	1019	1027	0.99221	0.249899
2	Keep Speed	907	910	0.996703	0.249789
	Med Speed	593	594	0.998316	0.249991
	Low Speed	676	677	0.998523	0.250017
	High Speed	918	918	1	0.250202
3	Keep Speed	1015	1019	0.996075	0.249668
	Med Speed	611	611	1	0.25016
	Low Speed	586	586	1	0.25016
	High Speed	852	853	0.998828	0.250013
4	Keep Speed	888	889	0.998875	0.250133
	Med Speed	553	554	0.998195	0.250048
	Low Speed	558	560	0.996429	0.249826
	High Speed	890	892	0.997758	0.249993
5	Keep Speed	962	965	0.996891	0.249744
	Med Speed	558	558	1	0.250133
	Low Speed	573	573	1	0.250133
	High Speed	865	866	0.998845	0.249989
6	Keep Speed	826	829	0.996381	0.249808
	Med Speed	460	460	1	0.250261
	Low Speed	339	340	0.997059	0.249893
	High Speed	561	562	0.998221	0.250038
7	Keep Speed	830	830	1	0.25
	Med Speed	471	471	1	0.25
	Low Speed	281	281	1	0.25
	High Speed	632	632	1	0.25
8	Keep Speed	877	878	0.998861	0.249945
	Med Speed	448	448	1	0.250088
	Low Speed	323	323	1	0.250088
	High Speed	599	600	0.998333	0.249879
9	Keep Speed	928	929	0.998924	0.249899
	Med Speed	468	468	1	0.250034
	Low Speed	341	341	1	0.250034
	High Speed	600	600	1	0.250034
10	Keep Speed	833	833	1	0.250051
	Med Speed	482	482	1	0.250051
	Low Speed	306	306	1	0.250051
	High Speed	615	616	0.998377	0.249848

Table D.2.: The calculations for the Speed node for equal $k's$

Figure D.4.: Graph over the success rates on the Speed children for equal $k's$

Spin					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Spin Left				0.5
	Spin Right				0.5
1	Spin Left	1722	1735	0.992507	0.499888
	Spin Right	1806	1818	0.993399	0.500112
2	Spin Left	1343	1346	0.997771	0.500284
	Spin Right	1329	1335	0.995506	0.4997161
3	Spin Left	1307	1311	0.996949	0.500015
	Spin Right	1259	1263	0.996833	0.499985
4	Spin Left	1240	1246	0.995185	0.500215
	Spin Right	1218	1226	0.993475	0.499785
5	Spin Left	1264	1269	0.99606	0.499707
	Spin Right	1244	1246	0.998395	0.500293
6	Spin Left	1009	1015	0.994089	0.499957
	Spin Right	1071	1077	0.994429	0.500043
7	Spin Left	1066	1069	0.997194	0.500369
	Spin Right	1038	1044	0.994253	0.499631
8	Spin Left	1077	1081	0.9963	0.500232
	Spin Right	1076	1082	0.994455	0.499768
9	Spin Left	1097	1101	0.996367	0.499769
	Spin Right	1113	1115	0.998206	0.500231
10	Spin Left	1071	1074	0.997207	0.500018
	Spin Right	1019	1022	0.997065	0.499982

Table D.3.: The calculations for the Spin node for equal $k's$

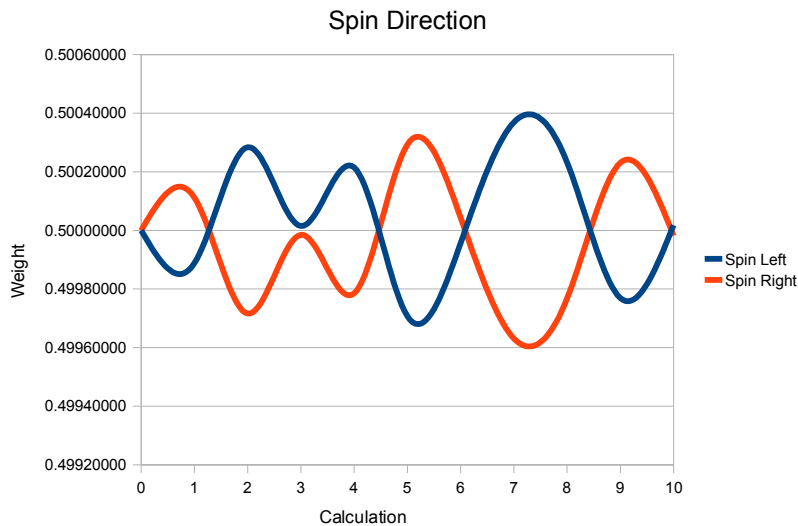


Figure D.5.: Graph over the weights on the Spin Direction children for equal $k's$

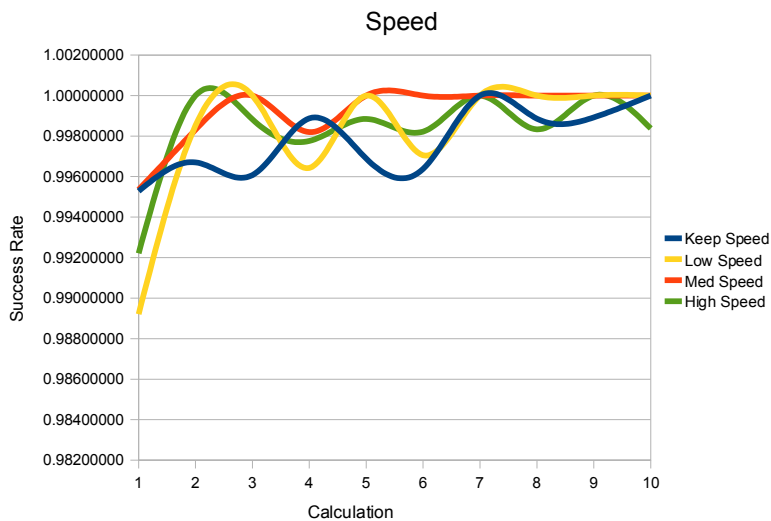


Figure D.6.: Graph over the success rates on the Spin Direction children for equal $k's$

Behavior Diversity

The results of the tests using $k_1 = 0.75$ and $k_2 = 0.25$.

Root					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Direct				0.33
	Arc				0.33
	Spiral				0.33
1	Direct	281	1460	0.192466	0.356178
	Arc	681	1821	0.37397	0.420521
	Spiral	38	1278	0.029734	0.223301
2	Direct	265	1511	0.175381	0.352825
	Arc	708	1748	0.405034	0.434947
	Spiral	27	1176	0.0229592	0.212228
3	Direct	252	1503	0.167665	0.347867
	Arc	718	1749	0.41052	0.435148
	Spiral	30	1182	0.0253807	0.216985
4	Direct	256	1528	0.167539	0.345522
	Arc	707	1771	0.399209	0.429286
	Spiral	37	1224	0.0302288	0.225191
5	Direct	258	1532	0.168407	0.347012
	Arc	707	1770	0.399435	0.430641
	Spiral	35	1233	0.0283861	0.222347
6	Direct	258	1592	0.16206	0.348327
	Arc	709	1859	0.381388	0.431429
	Spiral	33	1274	0.0259027	0.220244
7	Direct	266	1490	0.178523	0.349991
	Arc	701	1721	0.407321	0.430148
	Spiral	33	1187	0.0278012	0.219861
8	Direct	267	1517	0.176005	0.35098
	Arc	702	1757	0.399545	0.430816
	Spiral	31	1179	0.0262935	0.218204
9	Direct	261	1504	0.173537	0.349789
	Arc	708	1745	0.405731	0.432531
	Spiral	31	1191	0.0260285	0.217681
10	Direct	236	1408	0.167614	0.343552
	Arc	735	1641	0.447898	0.439249
	Spiral	29	1083	0.0267775	0.217199

Table D.4.: The calculations for the Root node for $k_1 = 0.75$ and $k_2 = 0.25$

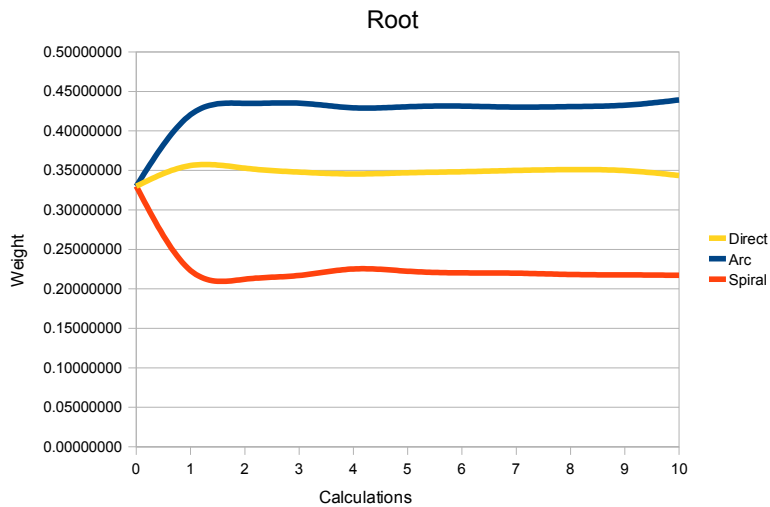


Figure D.7.: Graph over the weights on the Root children for $k_1 = 0.75$ and $k_2 = 0.25$

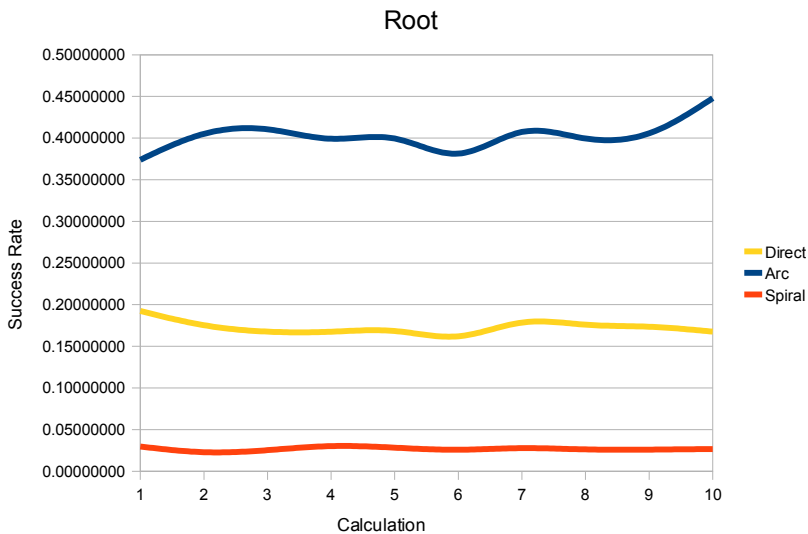


Figure D.8.: Graph over the success rates on the Root children for $k_1 = 0.75$ and $k_2 = 0.25$

Speed					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Keep Speed				0.25
	Med Speed				0.25
	Low Speed				0.25
	High Speed				0.25
1	Keep Speed	1087	1088	0.999081	0.249988
	Med Speed	538	538	1	0.250045
	Low Speed	644	644	1	0.250045
	High Speed	1007	1009	0.998018	0.249921
2	Keep Speed	1039	1041	0.998079	0.250011
	Med Speed	608	608	1	0.250131
	Low Speed	616	620	0.993548	0.249727
	High Speed	986	986	1	0.250131
3	Keep Speed	1007	1011	0.996044	0.249861
	Med Speed	582	582	1	0.250109
	Low Speed	650	650	1	0.250109
	High Speed	998	1001	0.997003	0.249921
4	Keep Speed	971	975	0.995897	0.249838
	Med Speed	646	646	1	0.250095
	Low Speed	660	660	1	0.250095
	High Speed	1012	1014	0.998028	0.249972
5	Keep Speed	979	981	0.997961	0.249965
	Med Speed	638	638	1	0.250093
	Low Speed	696	698	0.997135	0.249913
	High Speed	982	983	0.998983	0.250029
6	Keep Speed	1115	1118	0.997317	0.249942
	Med Speed	601	601	1	0.25011
	Low Speed	639	640	0.998437	0.250012
	High Speed	1086	1089	0.997245	0.249937
7	Keep Speed	964	968	0.995868	0.249886
	Med Speed	611	611	1	0.250145
	Low Speed	638	640	0.996875	0.24995
	High Speed	987	989	0.997978	0.250019
8	Keep Speed	1052	1053	0.99905	0.249955
	Med Speed	643	643	1	0.250015
	Low Speed	626	626	1	0.250015
	High Speed	951	951	1	0.250015
9	Keep Speed	999	1000	0.999	0.249976
	Med Speed	594	594	1	0.250038
	Low Speed	686	687	0.998544	0.249947
	High Speed	966	966	1	0.250038
10	Keep Speed	896	896	1	0.250074
	Med Speed	575	575	1	0.250074
	Low Speed	629	630	0.998413	0.249975
	High Speed	943	946	0.996829	0.249876

Table D.5.: The calculations for the Speed node for $k_1 = 0.75$ and $k_2 = 0.25$

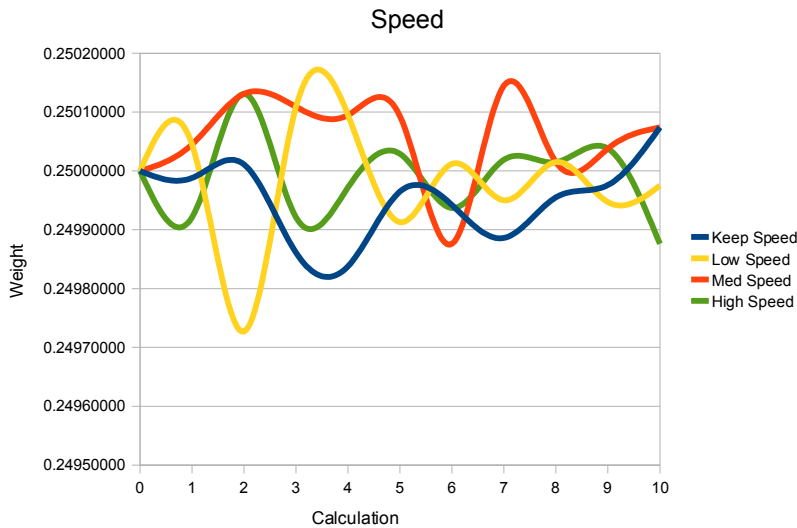


Figure D.9.: Graph over the weights on the Speed children for $k_1 = 0.75$ and $k_2 = 0.25$

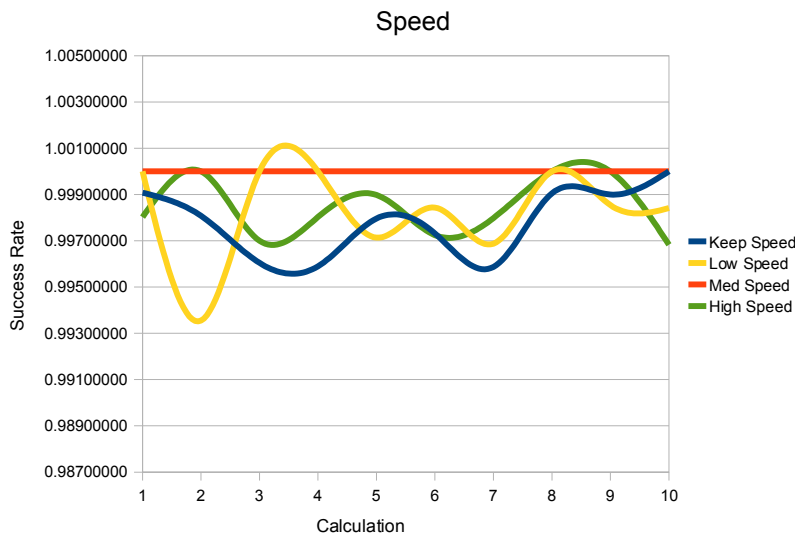
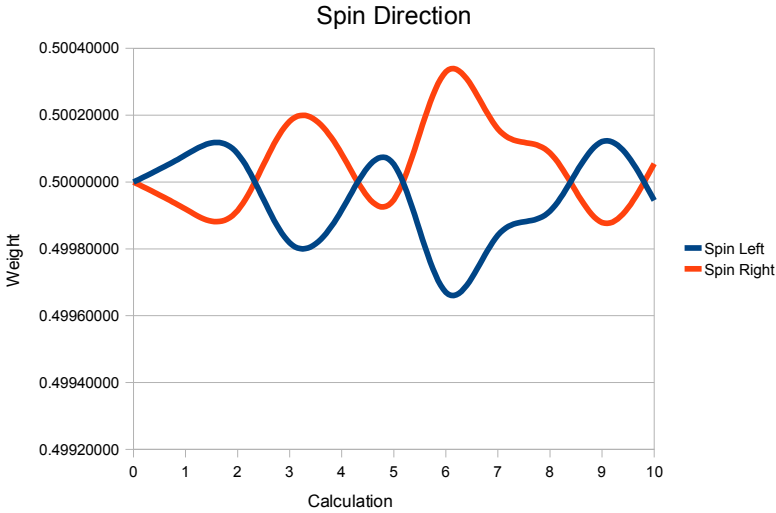


Figure D.10.: Graph over the success rates on the Speed children for $k_1 = 0.75$ and $k_2 = 0.25$

Spin					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Spin Left				0.5
	Spin Right				0.5
1	Spin Left	1509	1513	0.997356	0.500081
	Spin Right	1518	1524	0.996063	0.499919
2	Spin Left	1425	1430	0.996503	0.500085
	Spin Right	1435	1442	0.995146	0.499915
3	Spin Left	1408	1417	0.993649	0.499818
	Spin Right	1444	1449	0.996549	0.500182
4	Spin Left	1466	1472	0.995924	0.499916
	Spin Right	1455	1459	0.997258	0.500084
5	Spin Left	1521	1524	0.998031	0.500053
	Spin Right	1421	1425	0.997193	0.499947
6	Spin Left	1516	1525	0.994098	0.49967
	Spin Right	1553	1554	0.999356	0.50033
7	Spin Left	1352	1360	0.994118	0.499841
	Spin Right	1488	1493	0.996651	0.500159
8	Spin Left	1441	1443	0.998614	0.499913
	Spin Right	1436	1436	1	0.500087
9	Spin Left	1399	1402	0.99786	0.500121
	Spin Right	1469	1475	0.995932	0.499879
10	Spin Left	1307	1312	0.996189	0.499945
	Spin Right	1361	1365	0.99707	0.500055

Table D.6.: The calculations for the Spin node for $k_1 = 0.75$ and $k_2 = 0.25$ Figure D.11.: Graph over the weights on the Spin Direction children for $k_1 = 0.75$ and $k_2 = 0.25$

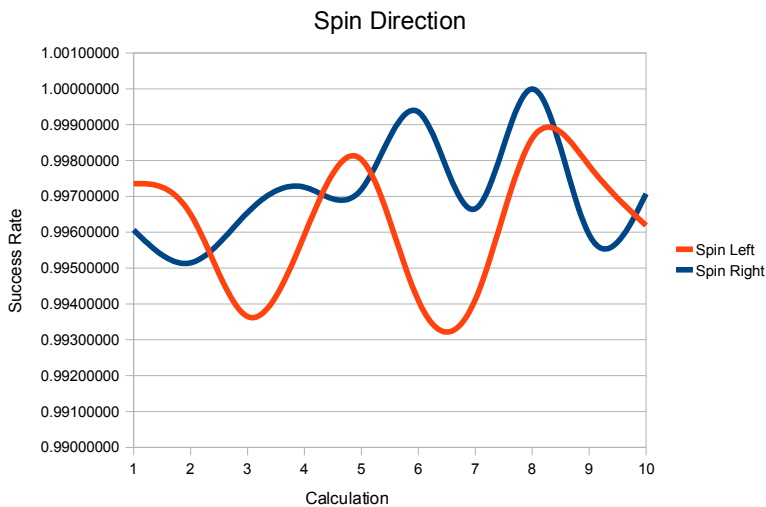


Figure D.12.: Graph over the success rates on the Spin Direction children for $k_1 = 0.75$ and $k_2 = 0.25$

Challenge

The results of the tests using $k_1 = 0.25$ and $k_2 = 0.75$.

Root					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Direct				0.33
	Arc				0.33
	Spiral				0.33
1	Direct	351	1728	0.203125	0.378015
	Arc	600	1913	0.313643	0.523616
	Spiral	49	1452	0.0337466	0.098369
2	Direct	153	1419	0.107822	0.222697
	Arc	825	1675	0.492537	0.695845
	Spiral	22	780	0.0282051	0.0814572
3	Direct	165	1054	0.156546	0.263876
	Arc	806	1579	0.51045	0.640298
	Spiral	29	715	0.0405594	0.0958266
4	Direct	185	1251	0.147882	0.272565
	Arc	776	1750	0.443429	0.621087
	Spiral	39	925	0.0421622	0.106347
5	Direct	198	1244	0.159164	0.285048
	Arc	773	1695	0.456047	0.627757
	Spiral	29	884	0.0328054	0.0871954
6	Direct	180	1187	0.151643	0.275749
	Arc	800	1664	0.480769	0.655163
	Spiral	20	835	0.0239521	0.0690883
7	Direct	106	724	0.146409	0.24734
	Arc	881	1526	0.577326	0.692125
	Spiral	13	580	0.0224138	0.0605348
8	Direct	100	685	0.145985	0.245685
	Arc	888	1516	0.585752	0.696516
	Spiral	12	566	0.0212014	0.0577989
9	Direct	130	756	0.171958	0.27319
	Arc	853	1540	0.553896	0.656856
	Spiral	17	608	0.0279605	0.0699533
10	Direct	104	730	0.142466	0.243032
	Arc	881	1537	0.573195	0.690409
	Spiral	15	592	0.0253378	0.0665591

Table D.7.: The calculations for the Root node for $k_1 = 0.25$ and $k_2 = 0.75$

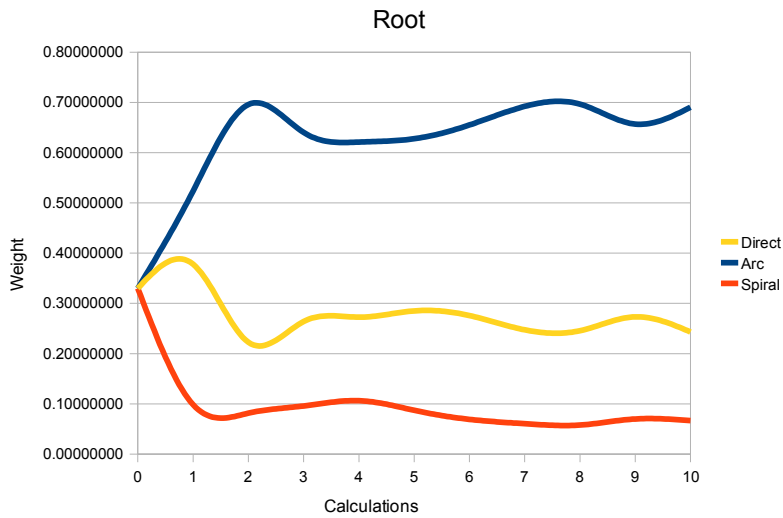


Figure D.13.: Graph over the weights on the Root children for $k_1 = 0.25$ and $k_2 = 0.75$

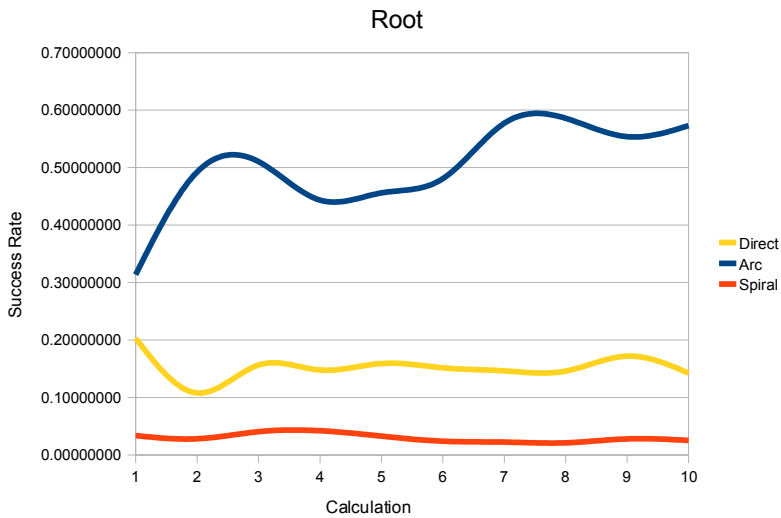


Figure D.14.: Graph over the success rates on the Root children for $k_1 = 0.25$ and $k_2 = 0.75$

Speed					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Keep Speed				0.25
	Med Speed				0.25
	Low Speed				0.25
	High Speed				0.25
1	Keep Speed	1131	1132	0.999117	0.250147
	Med Speed	653	653	1	0.250313
	Low Speed	756	759	0.996047	0.24957
	High Speed	1094	1096	0.998175	0.24997
2	Keep Speed	1011	1012	0.999012	0.24999
	Med Speed	591	591	1	0.250175
	Low Speed	628	629	0.99841	0.249877
	High Speed	861	862	0.99884	0.249958
3	Keep Speed	876	877	0.99886	0.249904
	Med Speed	523	523	1	0.250118
	Low Speed	503	503	1	0.250118
	High Speed	729	730	0.99863	0.249861
4	Keep Speed	1099	1100	0.999091	0.250015
	Med Speed	534	535	0.998131	0.249834
	Low Speed	510	510	1	0.250185
	High Speed	855	856	0.998832	0.249966
5	Keep Speed	1028	1029	0.999028	0.249919
	Med Speed	545	545	1	0.250101
	Low Speed	513	513	1	0.250101
	High Speed	847	848	0.998821	0.24988
6	Keep Speed	964	965	0.998964	0.250256
	Med Speed	539	539	1	0.250451
	Low Speed	477	480	0.99375	0.249276
	High Speed	863	865	0.997688	0.250017
7	Keep Speed	890	890	1	0.25
	Med Speed	498	498	1	0.25
	Low Speed	315	315	1	0.25
	High Speed	547	547	1	0.25
8	Keep Speed	819	821	0.997564	0.24973
	Med Speed	386	386	1	0.250187
	Low Speed	349	349	1	0.250187
	High Speed	643	6446	0.998447	0.249896
9	Keep Speed	911	911	1	0.25
	Med Speed	494	494	1	0.25
	Low Speed	307	307	1	0.25
	High Speed	583	583	1	0.25
10	Keep Speed	871	872	0.998853	0.249961
	Med Speed	434	434	1	0.250177
	Low Speed	381	382	0.997382	0.249685
	High Speed	578	578	1	0.250177

Table D.8.: The calculations for the Speed node for $k_1 = 0.25$ and $k_2 = 0.75$

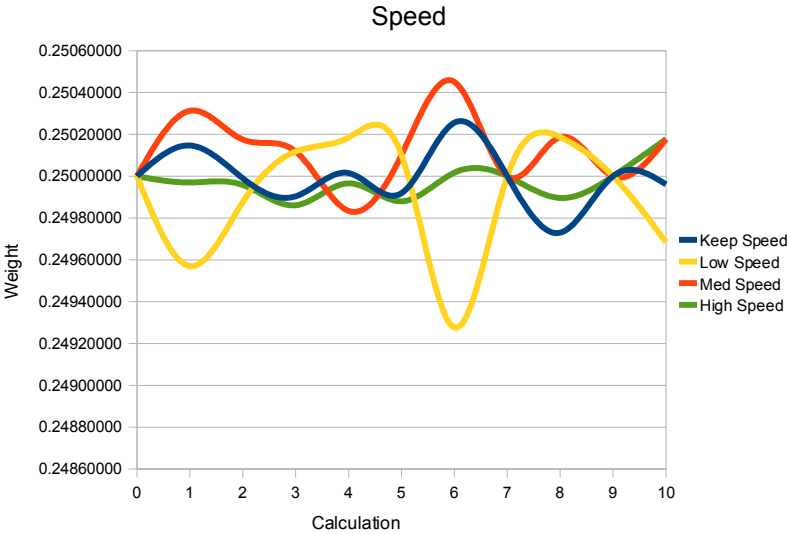


Figure D.15.: Graph over the weights on the Speed children for $k_1 = 0.25$ and $k_2 = 0.75$

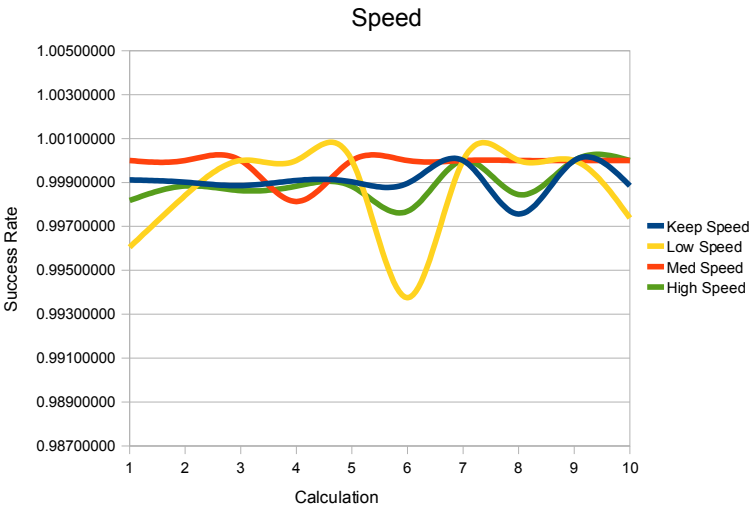


Figure D.16.: Graph over the success rates on the Speed children for $k_1 = 0.25$ and $k_2 = 0.75$

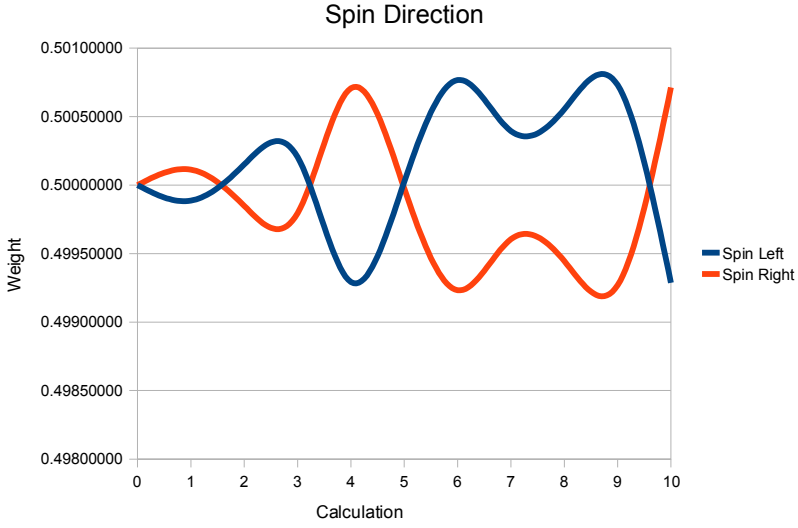


Figure D.17.: Graph over the weights on the Spin Direction children for $k_1 = 0.25$ and $k_2 = 0.75$

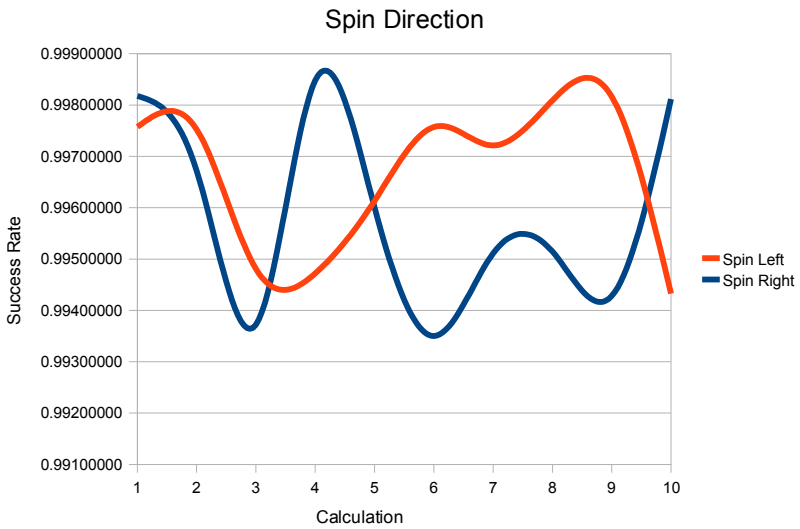


Figure D.18.: Graph over the success rates on the Spin Direction children for $k_1 = 0.25$ and $k_2 = 0.75$

Spin					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Spin Left				0.5
	Spin Right				0.5
1	Spin Left	1645	1649	0.997574	0.499887
	Spin Right	1643	1646	0.998177	0.500113
2	Spin Left	1210	1213	0.997527	0.500151
	Spin Right	1217	1221	0.996724	0.499849
3	Spin Left	1150	1156	0.99481	0.500205
	Spin Right	1108	1115	0.993722	0.499795
4	Spin Left	1319	1326	0.994721	0.499293
	Spin Right	1313	1315	0.998479	0.500707
5	Spin Left	1288	1293	0.996133	0.500023
	Spin Right	1248	1253	0.99601	0.499977
6	Spin Left	1233	1236	0.997573	0.500767
	Spin Right	1223	1231	0.993501	0.499233
7	Spin Left	1073	1076	0.997212	0.500395
	Spin Right	1018	1023	0.995112	0.499605
8	Spin Left	1045	1047	0.99809	0.500554
	Spin Right	1025	1030	0.995146	0.499446
9	Spin Left	1088	1090	0.998165	0.500731
	Spin Right	1043	1049	0.99428	0.499269
10	Spin Left	1052	1058	0.994329	0.499287
	Spin Right	1061	1063	0.998119	0.500713

Table D.9.: The calculations for the Spin node for $k_1 = 0.25$ and $k_2 = 0.75$

D.1.2. Global Method

Equal

The results of the tests using $k_1 = 0.5$ and $k_2 = 0.5$.

Root					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Direct				0.33
	Arc				0.33
	Spiral				0.33
1	Direct	261	1593	0.163842	0.125336
	Arc	703	1869	0.376137	0.87068
	Spiral	36	1378	0.0261248	0.00398339
2	Direct	162	1128	0.143617	0.079149
	Arc	815	1759	0.463331	0.918146
	Spiral	23	956	0.0240586	0.00270473
3	Direct	181	1227	0.147514	0.088658
	Arc	793	1810	0.438122	0.908202
	Spiral	26	1027	0.0253165	0.00314006
4	Direct	185	1307	0.141546	0.0862301
	Arc	783	1827	0.428571	0.909516
	Spiral	32	1056	0.030303	0.00425401
5	Direct	171	1249	0.13691	0.0768372
	Arc	805	1784	0.451233	0.920229
	Spiral	24	973	0.024666	0.00293413
6	Direct	148	1050	0.140952	0.0733083
	Arc	824	1716	0.480186	0.922607
	Spiral	28	856	0.0327103	0.00408442
7	Direct	101	886	0.113995	0.0479383
	Arc	884	1680	0.52619	0.950382
	Spiral	15	773	0.0194049	0.00168001
8	Direct	106	876	0.121005	0.0519728
	Arc	876	1663	0.526759	0.94577
	Spiral	18	758	0.0237467	0.00225696
9	Direct	92	868	0.105991	0.0410457
	Arc	895	1644	0.544404	0.957528
	Spiral	13	726	0.0179063	0.00142595
10	Direct	103	803	0.128269	0.0524793
	Arc	886	1594	0.555834	0.946352
	Spiral	11	682	0.016129	0.00116912

Table D.10.: The calculations for the Root node for equal $k's$

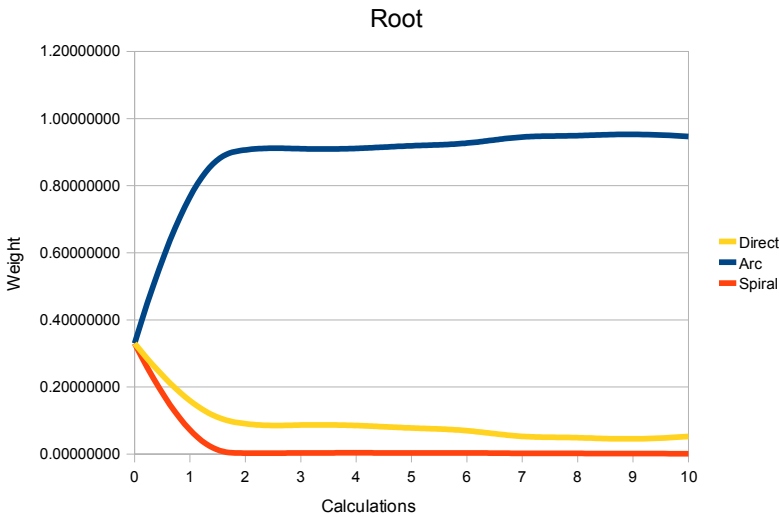


Figure D.19.: Graph over the weights on the Root children for equal $k's$

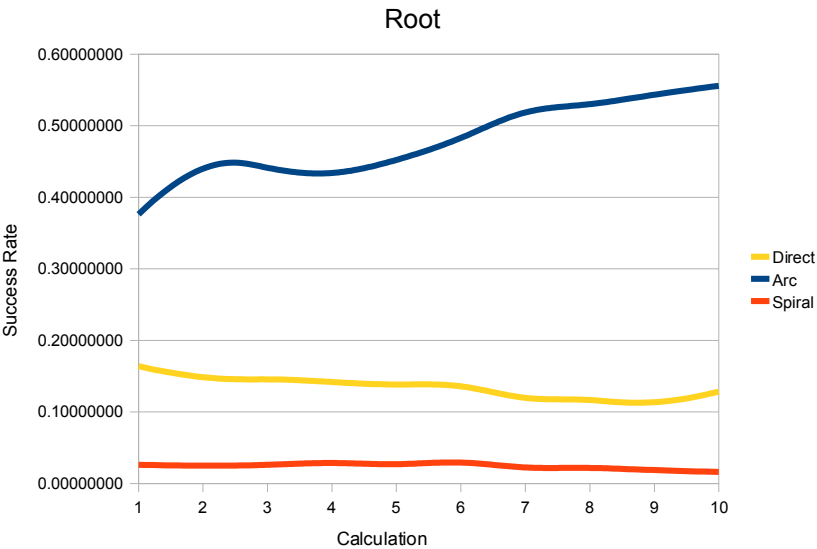


Figure D.20.: Graph over the success rates on the Root children for equal $k's$

Speed					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Keep Speed				0.25
	Med Speed				0.25
	Low Speed				0.25
	High Speed				0.25
1	Keep Speed	375	376	0.99734	0.250035
	Med Speed	300	302	0.993377	0.249538
	Low Speed	405	406	0.997537	0.250059
	High Speed	509	509	1	0.250368
2	Keep Speed	222	223	0.995516	0.249722
	Med Speed	218	219	0.995434	0.249711
	Low Speed	377	377	1	0.250283
	High Speed	309	309	1	0.250283
3	Keep Speed	250	251	0.996016	0.249716
	Med Speed	222	222	1	0.250215
	Low Speed	347	348	0.997126	0.249855
	High Speed	406	406	1	0.250215
4	Keep Speed	282	282	1	0.250248
	Med Speed	242	242	1	0.250248
	Low Speed	376	379	0.992084	0.249256
	High Speed	404	404	1	0.250248
5	Keep Speed	277	278	0.996403	0.249747
	Med Speed	230	230	1	0.250198
	Low Speed	367	368	0.997283	0.249857
	High Speed	373	373	1	0.250198
6	Keep Speed	235	236	0.995763	0.249708
	Med Speed	211	211	1	0.250239
	Low Speed	308	308	1	0.250239
	High Speed	294	295	0.99661	0.249814
7	Keep Speed	221	221	1	0.25
	Med Speed	145	145	1	0.25
	Low Speed	255	255	1	0.25
	High Speed	265	265	1	0.25
8	Keep Speed	183	183	1	0.25044
	Med Speed	200	202	0.990099	0.249197
	Low Speed	241	242	0.995868	0.249922
	High Speed	249	249	1	0.25044
9	Keep Speed	164	164	1	0.250504
	Med Speed	179	180	0.994444	0.249807
	Low Speed	283	286	0.98951	0.249186
	High Speed	238	238	1	0.250504
10	Keep Speed	144	144	1	0.25
	Med Speed	148	148	1	0.25
	Low Speed	262	262	1	0.25
	High Speed	249	249	1	0.25

Table D.11.: The calculations for the Speed node in the Direct strategy for equal $k's$

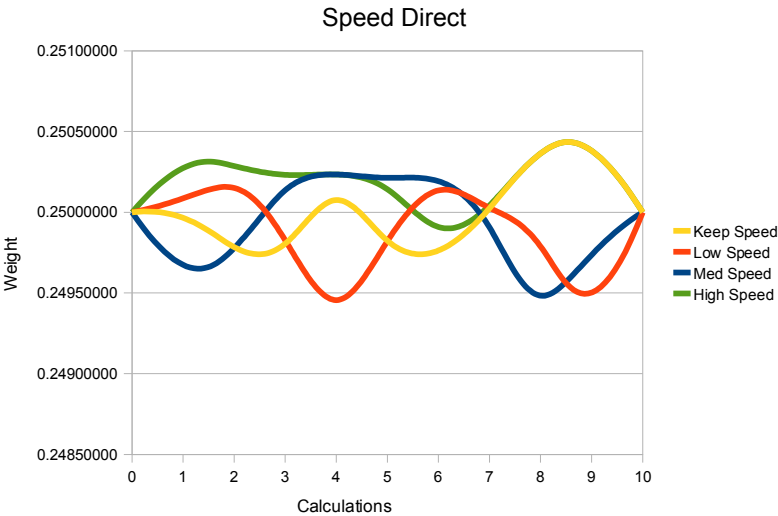


Figure D.21.: Graph over the weights on the Speed Direct children for equal $k's$

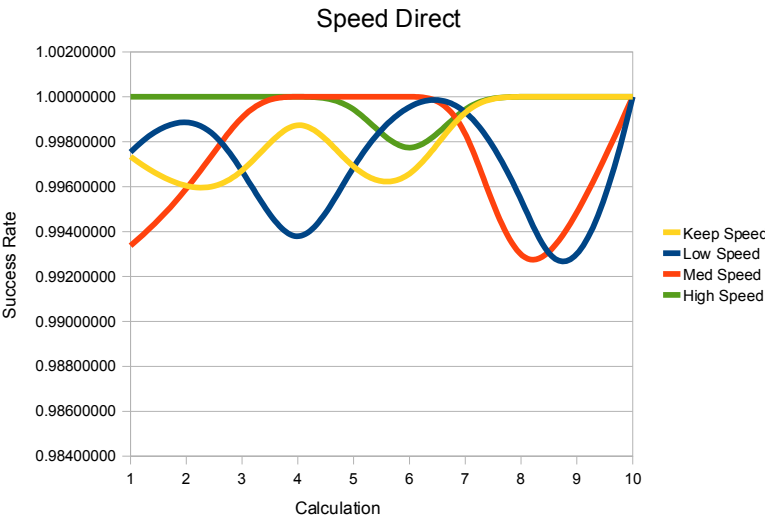


Figure D.22.: Graph over the success rates on the Speed Direct children for equal $k's$

Speed					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Keep Speed				0.25
	Med Speed				0.25
	Low Speed				0.25
	High Speed				0.25
1	Keep Speed	605	608	0.995066	0.250112
	Med Speed	217	218	0.995413	0.250156
	Low Speed	309	312	0.990385	0.249523
	High Speed	716	719	0.995828	0.250208
2	Keep Speed	642	643	0.998445	0.249854
	Med Speed	223	223	1	0.250049
	Low Speed	268	268	1	0.250049
	High Speed	609	609	1	0.250049
3	Keep Speed	683	684	0.998538	0.249863
	Med Speed	235	235	1	0.250046
	Low Speed	282	282	1	0.250046
	High Speed	602	602	1	0.250046
4	Keep Speed	717	717	1	0.25
	Med Speed	231	231	1	0.25
	Low Speed	286	286	1	0.25
	High Speed	590	590	1	0.25
5	Keep Speed	639	639	1	0.25
	Med Speed	236	236	1	0.25
	Low Speed	274	274	1	0.25
	High Speed	627	627	1	0.25
6	Keep Speed	621	622	0.998392	0.249849
	Med Speed	239	239	1	0.25005
	Low Speed	293	293	1	0.25005
	High Speed	550	550	1	0.25005
7	Keep Speed	618	618	1	0.250114
	Med Speed	205	205	1	0.250114
	Low Speed	274	275	0.996364	0.249659
	High Speed	575	575	1	0.250114
8	Keep Speed	590	590	1	0.25
	Med Speed	217	217	1	0.25
	Low Speed	287	287	1	0.25
	High Speed	562	562	1	0.25
9	Keep Speed	592	592	1	0.25
	Med Speed	213	213	1	0.25
	Low Speed	273	273	1	0.25
	High Speed	555	555	1	0.25
10	Keep Speed	539	541	0.996303	0.249653
	Med Speed	225	225	1	0.250116
	Low Speed	254	254	1	0.250116
	High Speed	564	564	1	0.250116

Table D.12.: The calculations for the Speed node in the Arc strategy for equal k 's

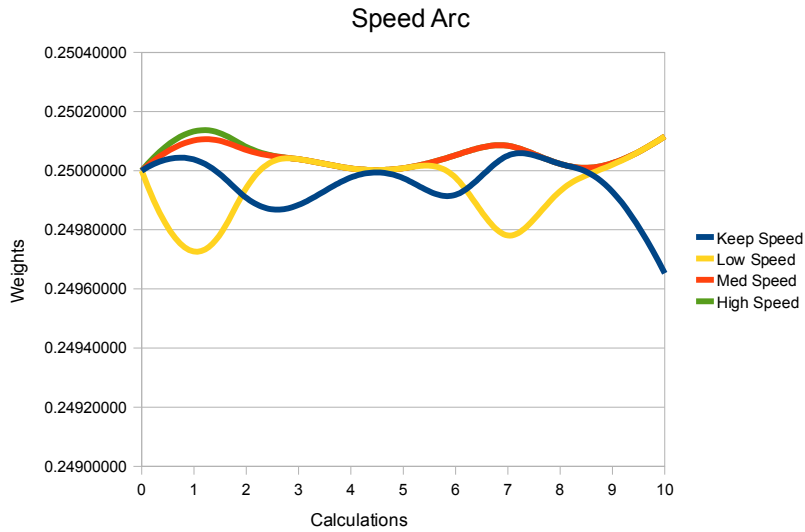


Figure D.23.: Graph over the weights on the Speed Arc children for equal $k's$

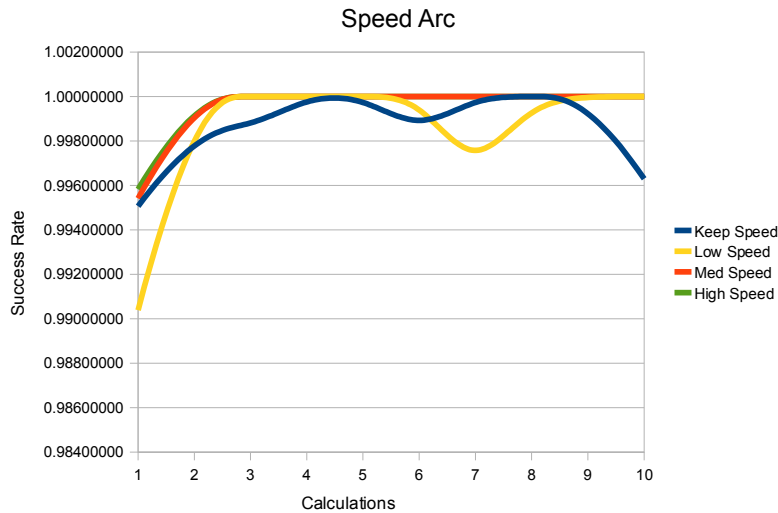
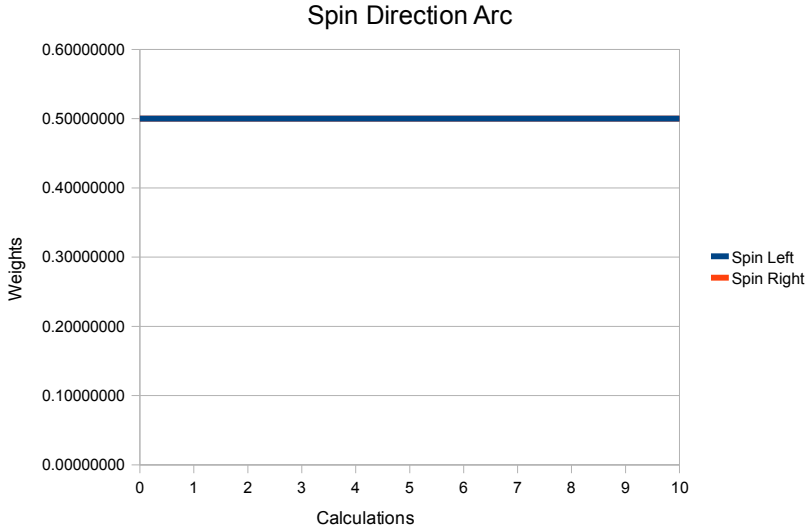


Figure D.24.: Graph over the success rates on the Speed Arc children for equal $k's$

Spin					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Spin Left				0.5
	Spin Right				0.5
1	Spin Left	917	927	0.989213	0.5
	Spin Right	940	942	0.997877	0.5
2	Spin Left	882	892	0.988789	0.5
	Spin Right	861	867	0.99308	0.5
3	Spin Left	896	902	0.993348	0.5
	Spin Right	907	908	0.998899	0.5
4	Spin Left	915	917	0.997819	0.5
	Spin Right	909	910	0.998901	0.5
5	Spin Left	910	914	0.995624	0.5
	Spin Right	866	870	0.995402	0.5
6	Spin Left	841	848	0.991745	0.5
	Spin Right	863	868	0.99424	0.5
7	Spin Left	835	841	0.992866	0.5
	Spin Right	838	839	0.998808	0.5
8	Spin Left	830	833	0.996399	0.5
	Spin Right	826	830	0.995181	0.5
9	Spin Left	816	821	0.99391	0.5
	Spin Right	817	823	0.99271	0.5
10	Spin Left	767	772	0.993523	0.5
	Spin Right	817	822	0.993917	0.5

Table D.13.: The calculations for the Spin node in the Arc strategy for equal $k's$ Figure D.25.: Graph over the weights on the Spin Direction Arc children for equal $k's$

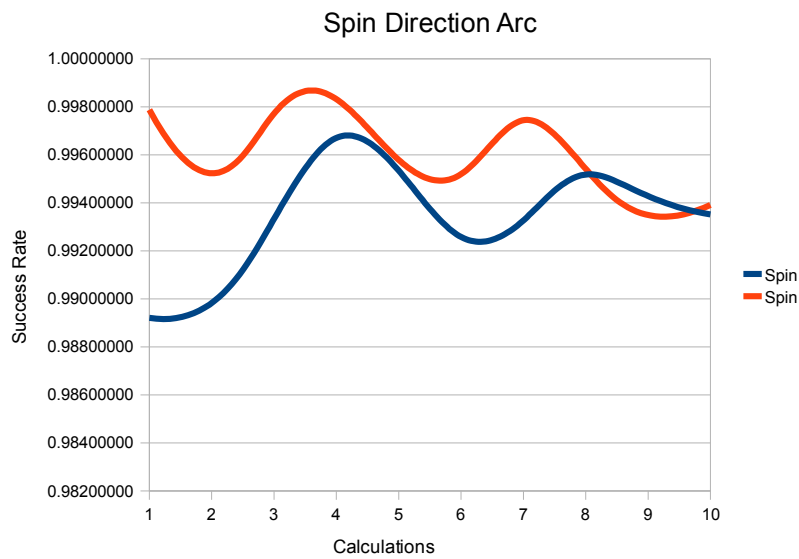


Figure D.26.: Graph over the success rates on the Spin Direction Arc children for equal $k's$

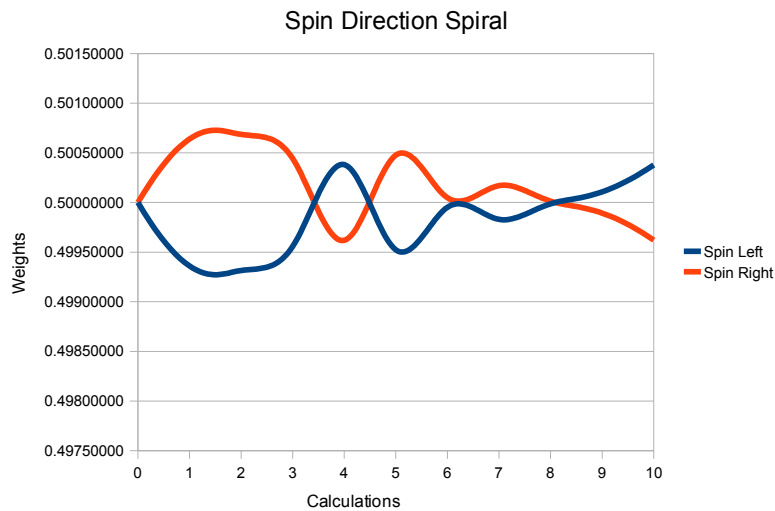
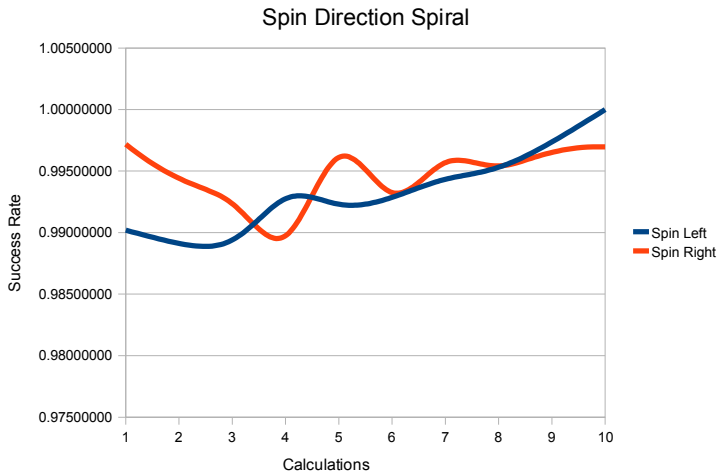


Figure D.27.: Graph over the weights on the Spin Direction Spiral children for equal $k's$

Spin					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Spin Left				0.5
	Spin Right				0.5
1	Spin Left	606	612	0.990196	0.499122
	Spin Right	707	709	0.997179	0.500878
2	Spin Left	451	456	0.989035	0.499409
	Spin Right	475	478	0.993724	0.500591
3	Spin Left	489	495	0.987879	0.499228
	Spin Right	497	500	0.994	0.500772
4	Spin Left	517	520	0.994231	0.501025
	Spin Right	497	504	0.986111	0.498975
5	Spin Left	479	483	0.991718	0.49896
	Spin Right	460	460	1	0.50104
6	Spin Left	409	412	0.992718	0.500274
	Spin Right	419	423	0.990544	0.499726
7	Spin Left	368	370	0.994595	0.499655
	Spin Right	375	376	0.99734	0.500345
8	Spin Left	389	391	0.994885	0.500086
	Spin Right	343	345	0.994203	0.499914
9	Spin Left	366	367	0.997275	0.500026
	Spin Right	340	341	0.997067	0.499974
10	Spin Left	338	338	1	0.500379
	Spin Right	329	330	0.99697	0.499621

Table D.14.: The calculations for the Spin node in the Spiral strategy for equal $k's$ Figure D.28.: Graph over the success rates on the Spin Direction Spiral children for equal $k's$

Behavior Diversity

The results of the tests using $k_1 = 0.75$ and $k_2 = 0.25$.

Root					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Direct				0.33
	Arc				0.33
	Spiral				0.33
1	Direct	301	1785	0.168627	0.223966
	Arc	655	1961	0.334013	0.747159
	Spiral	44	1584	0.0277778	0.0288749
2	Direct	95	881	0.107832	0.129009
	Arc	879	1679	0.523526	0.844117
	Spiral	26	772	0.0336788	0.0268737
3	Direct	117	875	0.133714	0.148933
	Arc	869	1649	0.526986	0.834132
	Spiral	14	754	0.0185676	0.016935
4	Direct	108	922	0.117137	0.138263
	Arc	877	1693	0.518015	0.843974
	Spiral	15	784	0.0191327	0.0177633
5	Direct	106	950	0.111579	0.134533
	Arc	873	1707	0.511424	0.843056
	Spiral	21	812	0.0258621	0.0224112
6	Direct	196	1206	0.162521	0.185803
	Arc	777	1752	0.443493	0.790016
	Spiral	27	996	0.0271084	0.0241809
7	Direct	198	1218	0.162562	0.186878
	Arc	776	1762	0.440409	0.789558
	Spiral	26	1002	0.0259481	0.0235636
8	Direct	190	1243	0.152856	0.179031
	Arc	790	1766	0.447339	0.801391
	Spiral	20	991	0.0201816	0.0195777
9	Direct	197	1217	0.161873	0.187725
	Arc	787	1786	0.440649	0.795563
	Spiral	16	986	0.0162272	0.0167118
10	Direct	184	1077	0.170845	0.183037
	Arc	795	1652	0.481235	0.796028
	Spiral	21	877	0.0239453	0.0209353

Table D.15.: The calculations for the Root node for $k_1 = 0.75$ and $k_2 = 0.25$

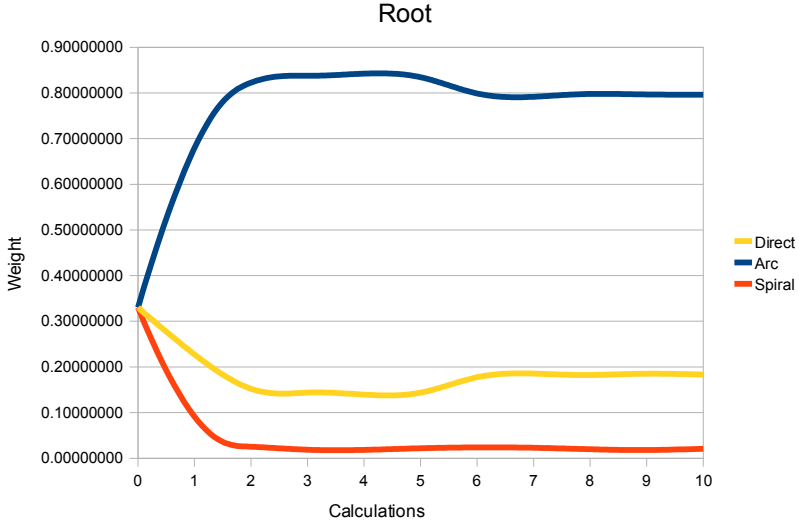


Figure D.29.: Graph over the weights on the Root children for $k_1 = 0.75$ and $k_2 = 0.25$

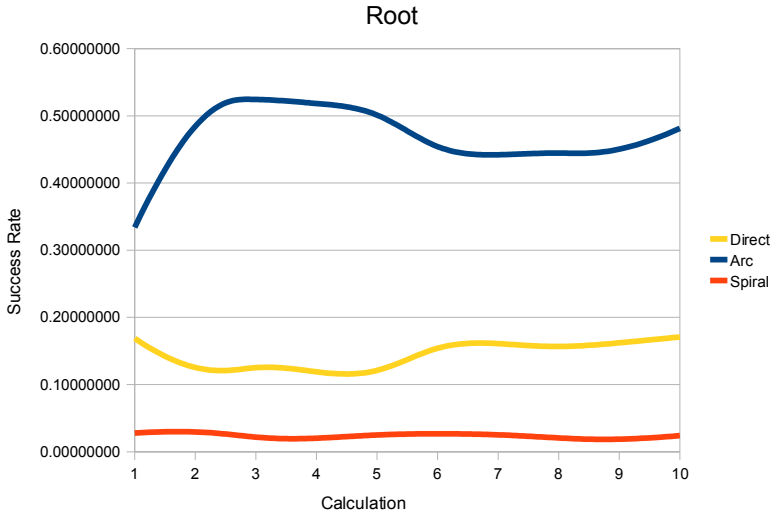


Figure D.30.: Graph over the success rates on the Root children for $k_1 = 0.75$ and $k_2 = 0.25$

Speed					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Keep Speed				0.25
	Med Speed				0.25
	Low Speed				0.25
	High Speed				0.25
1	Keep Speed	426	426	1	0.25006
	Med Speed	318	318	1	0.25006
	Low Speed	538	539	0.998145	0.249944
	High Speed	501	502	0.998008	0.249936
2	Keep Speed	226	226	1	0.250063
	Med Speed	197	197	1	0.250063
	Low Speed	209	209	1	0.250063
	High Speed	248	249	0.995984	0.249811
3	Keep Speed	206	206	1	0.250286
	Med Speed	175	176	0.994318	0.24993
	Low Speed	225	227	0.991189	0.249733
	High Speed	265	266	0.996241	0.250051
4	Keep Speed	220	222	0.990991	0.249697
	Med Speed	188	188	1	0.250263
	Low Speed	254	254	1	0.250263
	High Speed	256	258	0.992248	0.249777
5	Keep Speed	249	249	1	0.250057
	Med Speed	187	187	1	0.250057
	Low Speed	239	239	1	0.250057
	High Speed	274	275	0.996364	0.249829
6	Keep Speed	288	289	0.99654	0.250111
	Med Speed	221	223	0.991031	0.249765
	Low Speed	319	320	0.996875	0.250132
	High Speed	372	374	0.994652	0.249992
7	Keep Speed	314	314	1	0.250096
	Med Speed	263	263	1	0.250096
	Low Speed	314	314	1	0.250096
	High Speed	325	327	0.993884	0.249713
8	Keep Speed	328	329	0.99696	0.249908
	Med Speed	249	249	1	0.250098
	Low Speed	309	310	0.996774	0.249896
	High Speed	355	355	1	0.250098
9	Keep Speed	276	276	1	0.250042
	Med Speed	227	227	1	0.250042
	Low Speed	340	340	1	0.250042
	High Speed	373	374	0.997326	0.249875
10	Keep Speed	251	251	1	0.250104
	Med Speed	229	229	1	0.250104
	Low Speed	295	295	1	0.250104
	High Speed	300	302	0.993377	0.249689

Table D.16.: The calculations for the Speed node in the Direct strategy for $k_1 = 0.75$ and $k_2 = 0.25$

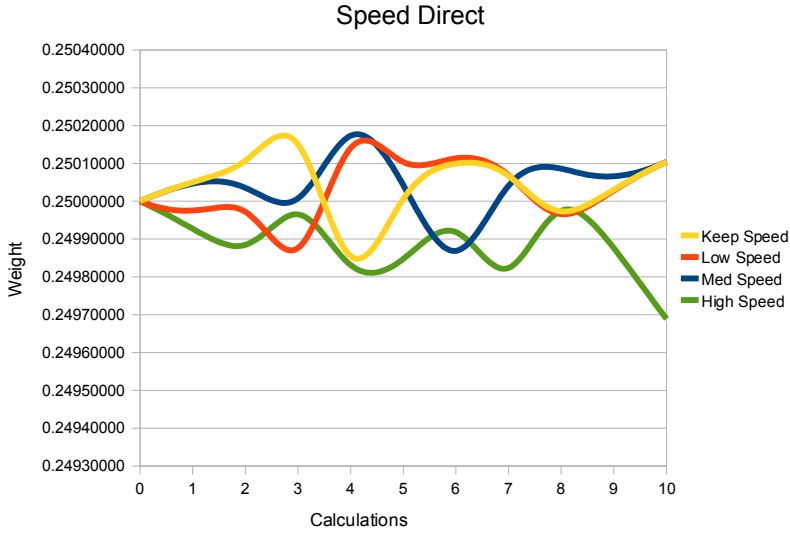


Figure D.31.: Graph over the weights on the Speed Direct children for $k_1 = 0.75$ and $k_2 = 0.25$

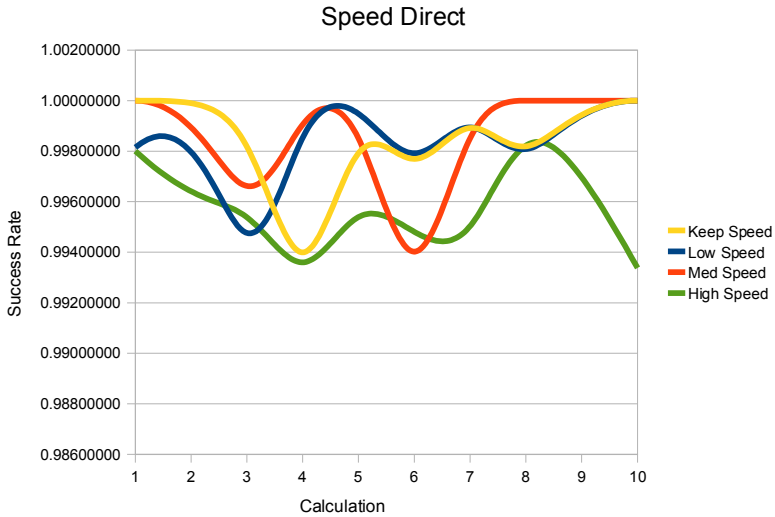


Figure D.32.: Graph over the success rates on the Speed Direct children for $k_1 = 0.75$ and $k_2 = 0.25$

Speed					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Keep Speed				0.25
	Med Speed				0.25
	Low Speed				0.25
	High Speed				0.25
1	Keep Speed	642	645	0.995349	0.250014
	Med Speed	260	261	0.996169	0.250066
	Low Speed	320	320	1	0.250306
	High Speed	719	727	0.988996	0.249614
2	Keep Speed	626	626	1	0.25
	Med Speed	243	243	1	0.25
	Low Speed	270	270	1	0.25
	High Speed	530	530	1	0.25
3	Keep Speed	603	603	1	0.25
	Med Speed	210	210	1	0.25
	Low Speed	284	284	1	0.25
	High Speed	545	545	1	0.25
4	Keep Speed	647	647	1	0.250058
	Med Speed	283	283	1	0.250058
	Low Speed	269	270	0.996296	0.249826
	High Speed	484	484	1	0.250058
5	Keep Speed	643	643	1	0.25
	Med Speed	268	268	1	0.25
	Low Speed	285	285	1	0.25
	High Speed	501	501	1	0.25
6	Keep Speed	613	613	1	0.25
	Med Speed	265	265	1	0.25
	Low Speed	318	318	1	0.25
	High Speed	541	541	1	0.25
7	Keep Speed	584	584	1	0.25
	Med Speed	296	296	1	0.25
	Low Speed	356	356	1	0.25
	High Speed	512	512	1	0.25
8	Keep Speed	619	619	1	0.250029
	Med Speed	290	290	1	0.250029
	Low Speed	307	307	1	0.250029
	High Speed	538	539	0.998145	0.249913
9	Keep Speed	614	616	0.996753	0.249876
	Med Speed	290	290	1	0.25008
	Low Speed	328	328	1	0.25008
	High Speed	541	542	0.998155	0.249964
10	Keep Speed	515	516	0.998062	0.249957
	Med Speed	297	297	1	0.250078
	Low Speed	326	327	0.996942	0.249887
	High Speed	505	505	1	0.250078

Table D.17.: The calculations for the Speed node in the Arc strategy for $k_1 = 0.75$ and $k_2 = 0.25$

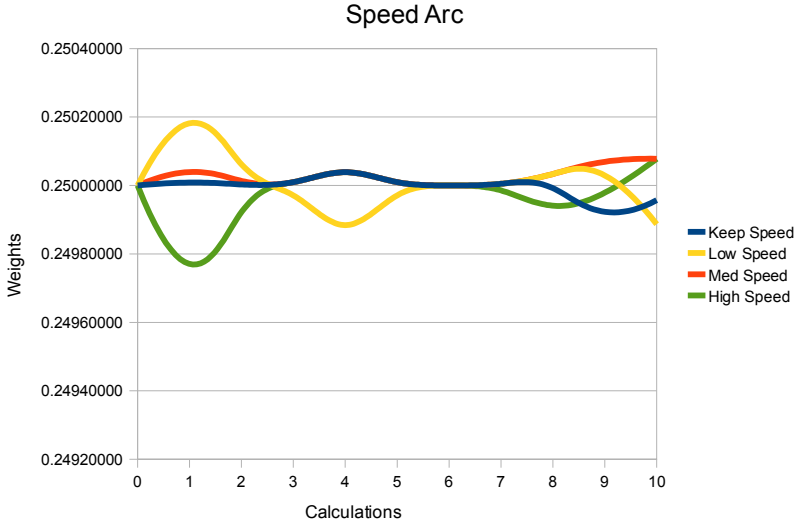


Figure D.33.: Graph over the weights on the Speed Arc children for $k_1 = 0.75$ and $k_2 = 0.25$

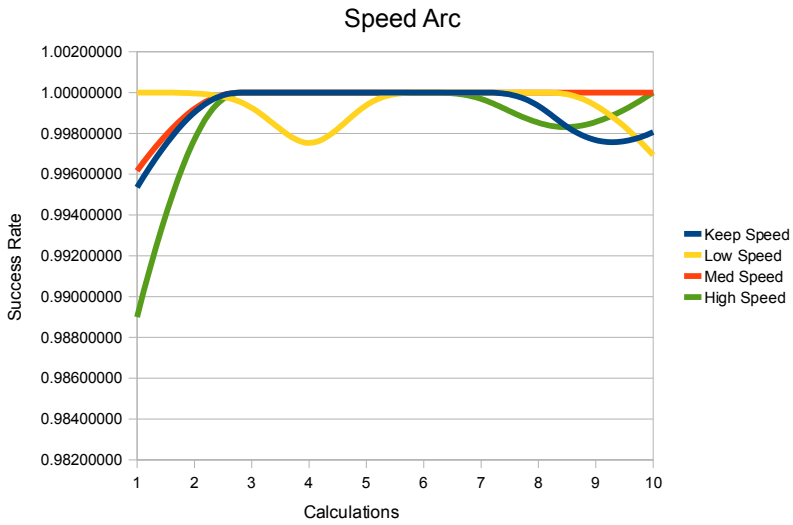


Figure D.34.: Graph over the success rates on the Speed Arc children for $k_1 = 0.75$ and $k_2 = 0.25$

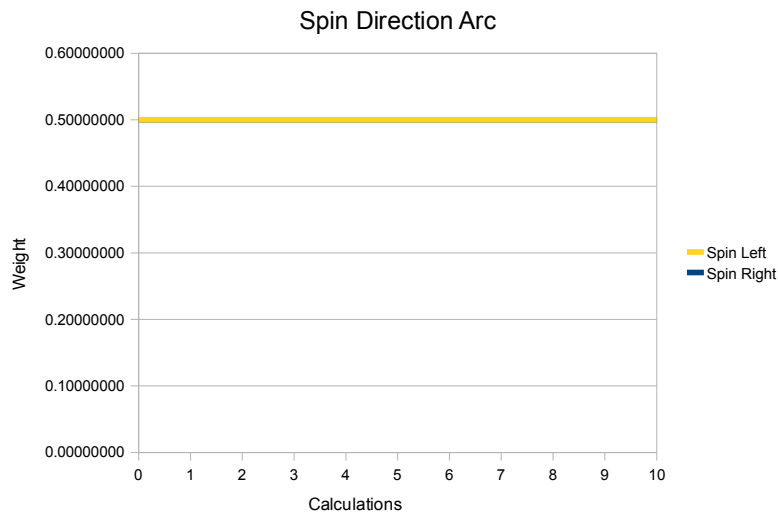


Figure D.35.: Graph over the weights on the Spin Direction Arc children for $k_1 = 0.75$ and $k_2 = 0.25$

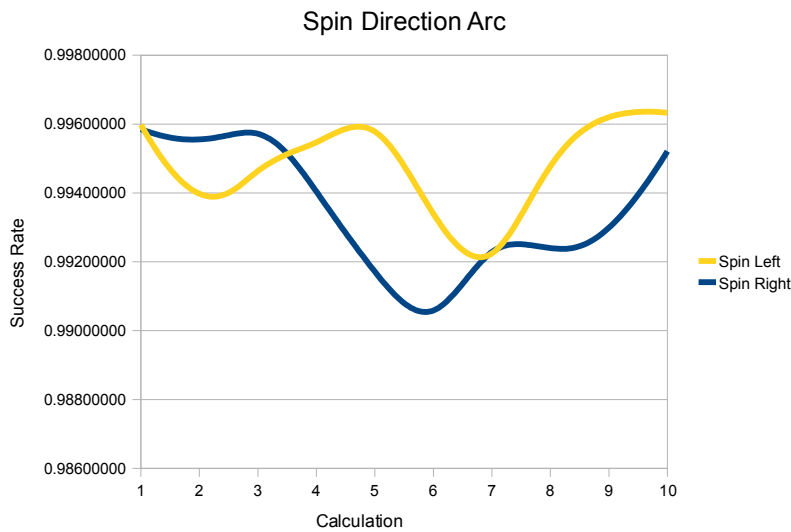


Figure D.36.: Graph over the success rates on the Spin Arc Spiral children for $k_1 = 0.75$ and $k_2 = 0.25$

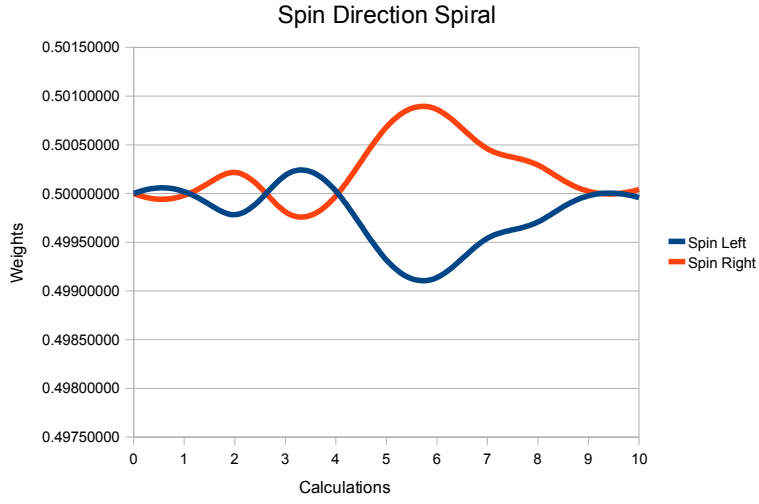


Figure D.37.: Graph over the weights on the Spin Direction Spiral children for $k_1 = 0.75$ and $k_2 = 0.25$

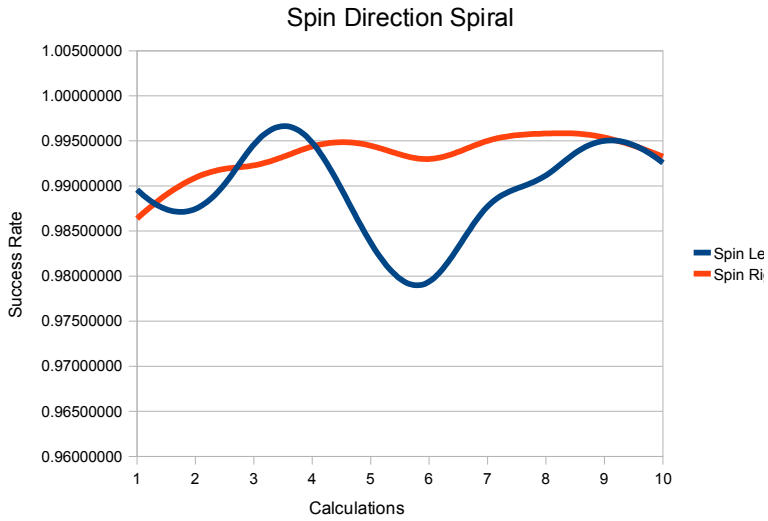


Figure D.38.: Graph over the success rates on the Spin Direction Spiral children for $k_1 = 0.75$ and $k_2 = 0.25$

Spin					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Spin Left				0.5
	Spin Right				0.5
1	Spin Left	992	996	0.995984	0.5
	Spin Right	961	965	0.995855	0.5
2	Spin Left	842	848	0.992925	0.5
	Spin Right	827	831	0.995187	0.5
3	Spin Left	794	798	0.994987	0.5
	Spin Right	848	851	0.996475	0.5
4	Spin Left	851	855	0.995322	0.5
	Spin Right	833	838	0.994033	0.5
5	Spin Left	868	871	0.996556	0.5
	Spin Right	829	836	0.991627	0.5
6	Spin Left	869	875	0.993143	0.5
	Spin Right	868	877	0.989738	0.5
7	Spin Left	904	912	0.991228	0.5
	Spin Right	844	850	0.992941	0.5
8	Spin Left	865	869	0.995397	0.5
	Spin Right	890	897	0.992196	0.5
9	Spin Left	855	858	0.996503	0.5
	Spin Right	921	928	0.992457	0.5
10	Spin Left	814	817	0.996328	0.5
	Spin Right	831	835	0.99521	0.5

Table D.18.: The calculations for the Spin node in the Arc strategy for $k_1 = 0.75$ and $k_2 = 0.25$

Spin					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Spin Left				0.5
	Spin Right				0.5
1	Spin Left	760	768	0.989583	0.5
	Spin Right	724	734	0.986376	0.5
2	Spin Left	355	361	0.98338	0.5
	Spin Right	387	390	0.992308	0.5
3	Spin Left	399	400	0.9975	0.5
	Spin Right	344	347	0.991354	0.5
4	Spin Left	365	366	0.997268	0.5
	Spin Right	397	399	0.994987	0.5
5	Spin Left	388	395	0.982278	0.5
	Spin Right	398	400	0.995	0.5
6	Spin Left	446	457	0.97593	0.5
	Spin Right	486	490	0.991837	0.5
7	Spin Left	510	515	0.990291	0.5
	Spin Right	451	453	0.995585	0.5
8	Spin Left	460	465	0.989247	0.5
	Spin Right	479	481	0.995842	0.5
9	Spin Left	444	445	0.997753	0.5
	Spin Right	494	496	0.995968	0.5
10	Spin Left	401	404	0.992574	0.5
	Spin Right	441	444	0.993243	0.5

Table D.19.: The calculations for the Spin node in the Spiral strategy for $k_1 = 0.75$ and $k_2 = 0.25$

Challenge

The results of the tests using $k_1 = 0.25$ and $k_2 = 0.75$.

Root					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Direct				0.33
	Arc				0.33
	Spiral				0.33
1	Direct	264	1688	0.156398	0.0727333
	Arc	686	1927	0.355994	0.926097
	Spiral	50	1469	0.0340368	0.00116986
2	Direct	199	1346	0.147845	0.0437964
	Arc	770	1801	0.42754	0.955541
	Spiral	31	990	0.0313131	0.000662867
3	Direct	139	1208	0.115066	0.0170102
	Arc	839	1637	0.512523	0.982586
	Spiral	22	739	0.02977	0.0004034
4	Direct	147	1190	0.123529	0.0211411
	Arc	822	1647	0.499089	0.978015
	Spiral	31	769	0.0403121	0.00084434
5	Direct	269	1609	0.167185	0.0766797
	Arc	714	1924	0.371102	0.923177
	Spiral	17	1217	0.0139688	0.000143209
6	Direct	250	1558	0.160462	0.0630921
	Arc	723	1849	0.391022	0.936483
	Spiral	27	1138	0.0237258	0.000424824
7	Direct	274	1594	0.171895	0.0815198
	Arc	697	1881	0.370548	0.917991
	Spiral	29	1201	0.0241465	0.00048867
8	Direct	264	1692	0.156028	0.0722255
	Arc	702	1970	0.356345	0.927097
	Spiral	34	1272	0.0267296	0.000677836
9	Direct	249	1656	0.150362	0.0645255
	Arc	710	1959	0.36243	0.934424
	Spiral	41	1245	0.0329317	0.00105062
10	Direct	260	1624	0.160099	0.0710868
	Arc	709	1924	0.368503	0.928341
	Spiral	31	1211	0.0255987	0.000571787

Table D.20.: The calculations for the Root node for $k_1 = 0.25$ and $k_2 = 0.75$

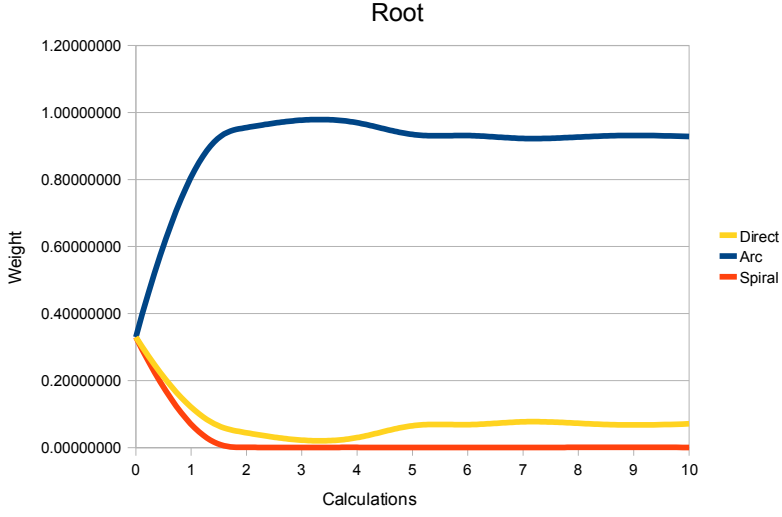


Figure D.39.: Graph over the weights on the Root children for $k_1 = 0.25$ and $k_2 = 0.75$

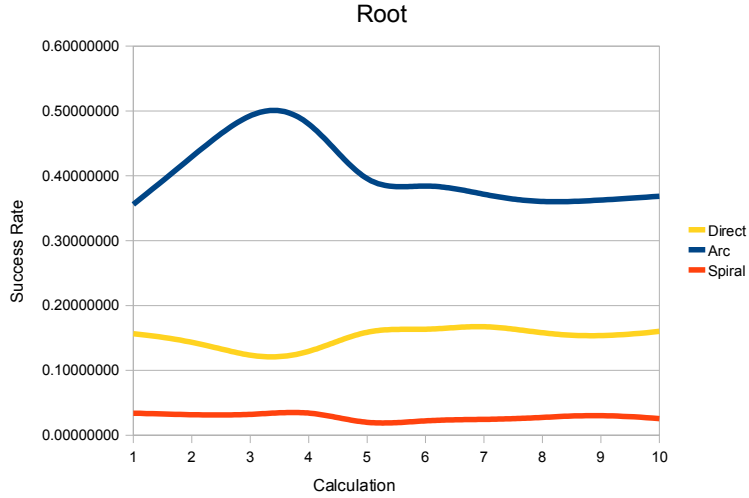


Figure D.40.: Graph over the success rates on the Root children for $k_1 = 0.25$ and $k_2 = 0.75$

Speed					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Keep Speed				0.25
	Med Speed				0.25
	Low Speed				0.25
	High Speed				0.25
1	Keep Speed	376	377	0.997347	0.249959
	Med Speed	334	335	0.997015	0.249896
	Low Speed	477	478	0.997908	0.250064
	High Speed	497	498	0.997992	0.25008
2	Keep Speed	339	339	1	0.25025
	Med Speed	281	281	1	0.25025
	Low Speed	351	351	1	0.25025
	High Speed	373	375	0.994667	0.249249
3	Keep Speed	282	282	1	0.250525
	Med Speed	266	269	0.988848	0.248426
	Low Speed	316	316	1	0.250525
	High Speed	341	341	1	0.250525
4	Keep Speed	296	296	1	0.250143
	Med Speed	270	270	1	0.250143
	Low Speed	295	295	1	0.250143
	High Speed	328	329	0.99696	0.249572
5	Keep Speed	381	381	1	0.250234
	Med Speed	346	347	0.997118	0.249693
	Low Speed	473	474	0.99789	0.249838
	High Speed	407	407	1	0.250234
6	Keep Speed	402	402	1	0.250228
	Med Speed	277	277	1	0.250228
	Low Speed	410	412	0.995146	0.249316
	High Speed	467	467	1	0.250228
7	Keep Speed	406	406	1	0.250213
	Med Speed	315	315	1	0.250213
	Low Speed	433	433	1	0.250213
	High Speed	438	440	0.995455	0.24936
8	Keep Speed	400	400	1	0.250232
	Med Speed	337	338	0.997041	0.249677
	Low Speed	450	450	1	0.250232
	High Speed	503	504	0.998016	0.24986
9	Keep Speed	367	368	0.997283	0.249618
	Med Speed	328	328	1	0.250127
	Low Speed	478	478	1	0.250127
	High Speed	482	482	1	0.250127
10	Keep Speed	364	365	0.99726	0.249713
	Med Speed	369	369	1	0.250228
	Low Speed	417	417	1	0.250228
	High Speed	472	473	0.997886	0.249831

Table D.21.: The calculations for the Speed node in the Direct strategy for $k_1 = 0.25$ and $k_2 = 0.75$

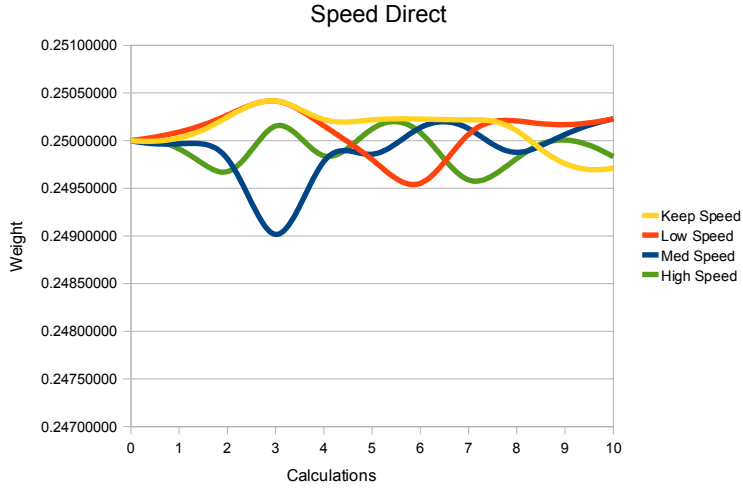


Figure D.41.: Graph over the weights on the Speed Direct children for $k_1 = 0.25$ and $k_2 = 0.75$

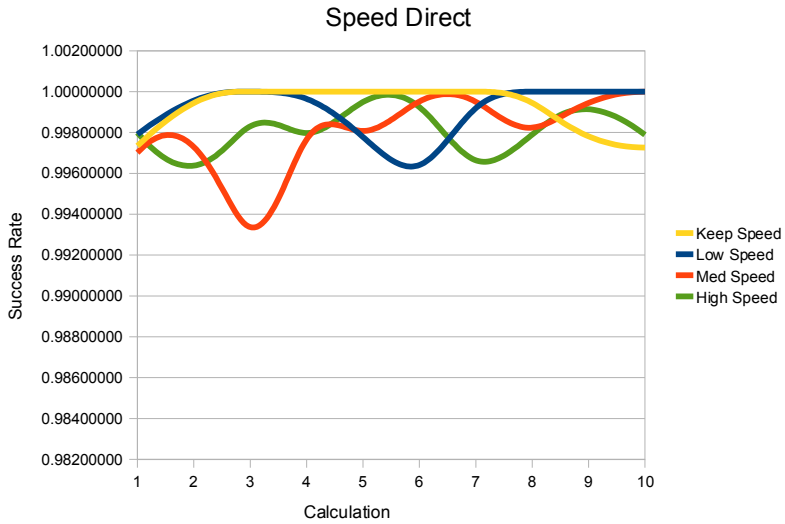


Figure D.42.: Graph over the success rates on the Speed Direct children for $k_1 = 0.25$ and $k_2 = 0.75$

Speed					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Keep Speed				0.25
	Med Speed				0.25
	Low Speed				0.25
	High Speed				0.25
1	Keep Speed	684	684	1	0.25036
	Med Speed	340	341	0.997067	0.249809
	Low Speed	341	342	0.997076	0.249811
	High Speed	551	552	0.998188	0.25002
2	Keep Speed	667	667	1	0.250164
	Med Speed	286	287	0.996516	0.249509
	Low Speed	317	317	1	0.250164
	High Speed	525	525	1	0.250164
3	Keep Speed	533	533	1	0.25
	Med Speed	328	328	1	0.25
	Low Speed	323	323	1	0.25
	High Speed	444	444	1	0.25
4	Keep Speed	527	527	1	0.250275
	Med Speed	299	299	1	0.250275
	Low Speed	340	342	0.994152	0.249176
	High Speed	467	467	1	0.250275
5	Keep Speed	682	682	1	0.25
	Med Speed	307	307	1	0.25
	Low Speed	316	316	1	0.25
	High Speed	602	602	1	0.25
6	Keep Speed	635	635	1	0.250153
	Med Speed	269	269	1	0.250153
	Low Speed	323	323	1	0.250153
	High Speed	612	614	0.996743	0.249541
7	Keep Speed	645	645	1	0.250221
	Med Speed	277	277	1	0.250221
	Low Speed	319	320	0.996875	0.249634
	High Speed	629	630	0.998413	0.249923
8	Keep Speed	703	703	1	0.25
	Med Speed	260	260	1	0.25
	Low Speed	302	302	1	0.25
	High Speed	691	691	1	0.25
9	Keep Speed	721	721	1	0.250075
	Med Speed	271	271	1	0.250075
	Low Speed	323	323	1	0.250075
	High Speed	626	627	0.998405	0.249776
10	Keep Speed	699	699	1	0.250074
	Med Speed	246	246	1	0.250074
	Low Speed	328	328	1	0.250074
	High Speed	637	638	0.998433	0.249779

Table D.22.: The calculations for the Speed node in the Arc strategy for $k_1 = 0.25$ and $k_2 = 0.75$

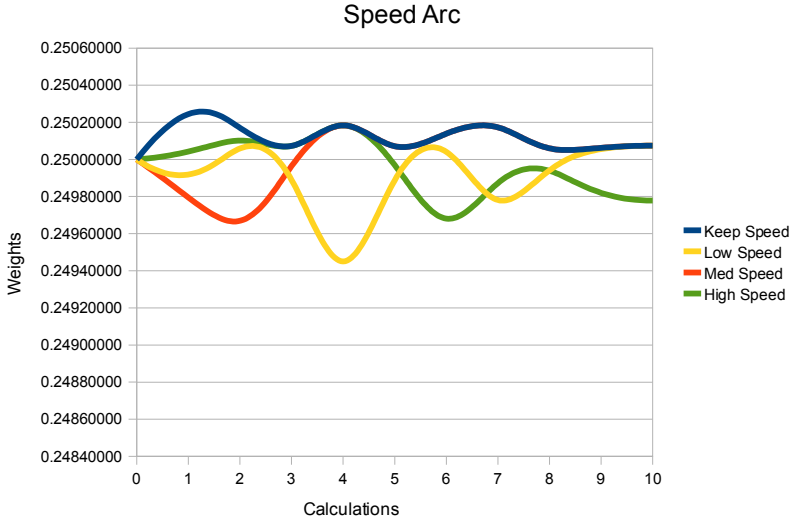


Figure D.43.: Graph over the weights on the Speed Arc children for $k_1 = 0.25$ and $k_2 = 0.75$

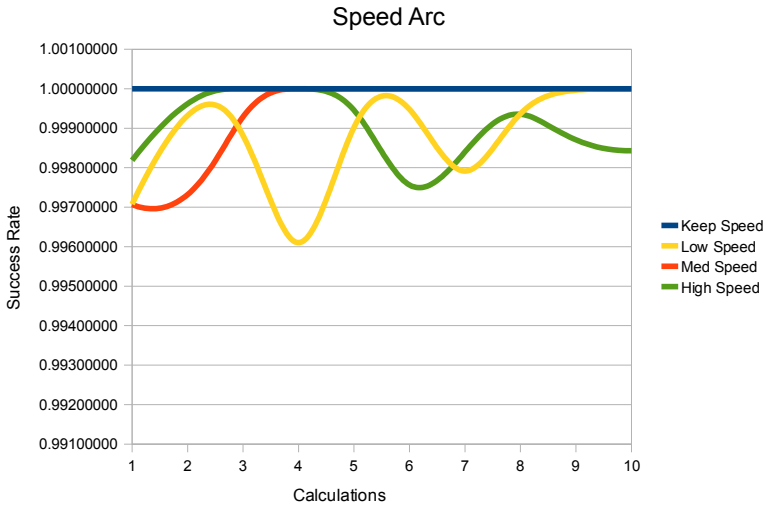


Figure D.44.: Graph over the success rates on the Speed Arc children for $k_1 = 0.25$ and $k_2 = 0.75$

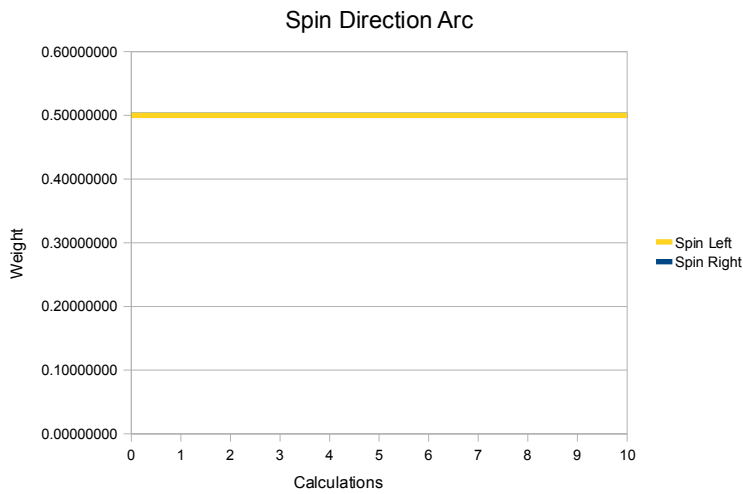


Figure D.45.: Graph over the weights on the Spin Direction Arc children for $k_1 = 0.25$ and $k_2 = 0.75$

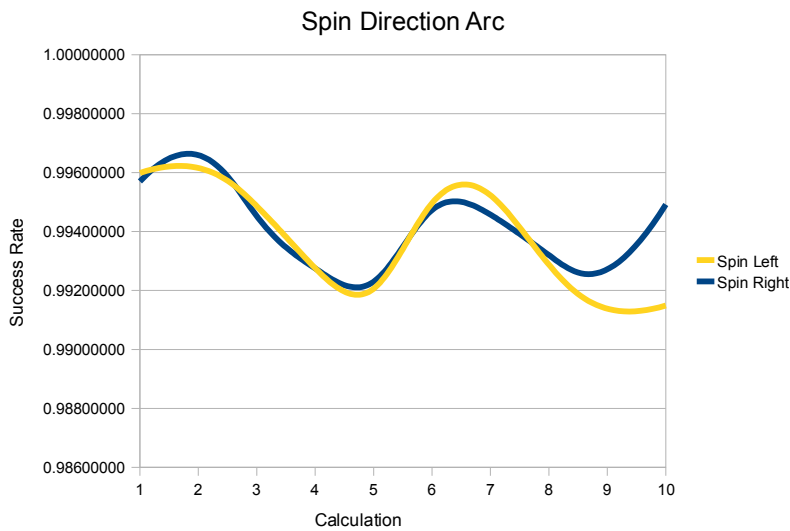


Figure D.46.: Graph over the success rates on the Spin Direction Arc children for $k_1 = 0.25$ and $k_2 = 0.75$

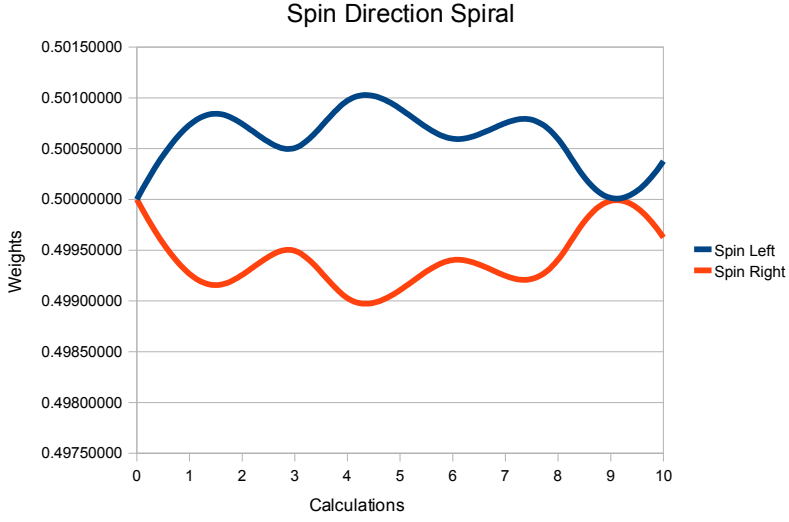


Figure D.47.: Graph over the weights on the Spin Direction Spiral children for $k_1 = 0.25$ and $k_2 = 0.75$

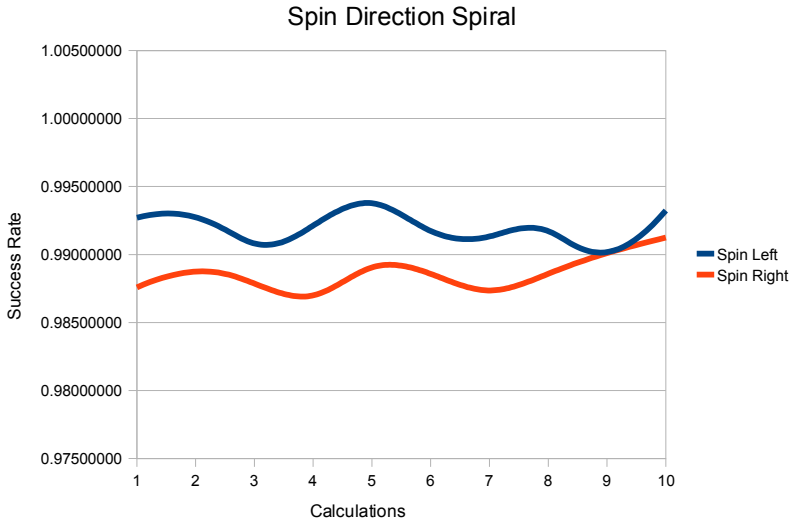


Figure D.48.: Graph over the success rates on the Spin Direction Spiral children for $k_1 = 0.25$ and $k_2 = 0.75$

Spin					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Spin Left				0.5
	Spin Right				0.5
1	Spin Left	993	997	0.995988	0.5
	Spin Right	926	930	0.995699	0.5
2	Spin Left	891	894	0.996644	0.5
	Spin Right	905	907	0.997795	0.5
3	Spin Left	799	803	0.995019	0.5
	Spin Right	829	834	0.994005	0.5
4	Spin Left	809	815	0.992638	0.5
	Spin Right	826	832	0.992788	0.5
5	Spin Left	990	999	0.990991	0.5
	Spin Right	917	925	0.991351	0.5
6	Spin Left	938	942	0.995754	0.5
	Spin Right	903	907	0.99559	0.5
7	Spin Left	940	944	0.995763	0.5
	Spin Right	932	937	0.994664	0.5
8	Spin Left	935	942	0.992569	0.5
	Spin Right	1021	1028	0.993191	0.5
9	Spin Left	979	988	0.990891	0.5
	Spin Right	963	971	0.991761	0.5
10	Spin Left	932	940	0.991489	0.5
	Spin Right	979	984	0.994919	0.5

Table D.23.: The calculations for the Spin node in the Arc strategy for $k_1 = 0.25$ and $k_2 = 0.75$

Spin					
Calculation	Children	Succeeds	Runs	Success Rate	Weight
0	Spin Left				0.5
	Spin Right				0.5
1	Spin Left	680	685	0.992701	0.500969
	Spin Right	716	725	0.987586	0.499031
2	Spin Left	484	487	0.99384	0.50083
	Spin Right	469	474	0.989451	0.49917
3	Spin Left	371	375	0.989333	0.500262
	Spin Right	328	332	0.987952	0.499738
4	Spin Left	379	382	0.992147	0.501159
	Spin Right	353	358	0.986034	0.498841
5	Spin Left	577	580	0.994828	0.500934
	Spin Right	587	593	0.989882	0.499066
6	Spin Left	561	566	0.991166	0.500463
	Spin Right	526	532	0.988722	0.499537
7	Spin Left	544	549	0.990893	0.500797
	Spin Right	593	601	0.986689	0.499203
8	Spin Left	586	590	0.99322	0.50086
	Spin Right	611	618	0.988673	0.49914
9	Spin Left	583	590	0.988136	0.499606
	Spin Right	607	613	0.990212	0.500394
10	Spin Left	587	591	0.993232	0.500376
	Spin Right	566	571	0.991243	0.499624

Table D.24.: The calculations for the Spin node in the Spiral strategy for $k_1 = 0.25$ and $k_2 = 0.75$

D.2. Constructed Example

D.2.1. Local Method

Selector	Local Method			Success Rate	Calculation
	Children	k_1	k_2		
Attack	Direct Attack	0.5	0.5	0.75	0.633975
	Sneak Attack			0.25	0.366025
Direct Attack	Close Range			0.375	0.366025
	Long Range			0.125	0.211325
	Medium Range			0.5	0.42265
Sneak Avoid	Move Around			0.8	0.666667
	Keep In Cover			0.2	0.333333
Direct Attack(Sneak Attack)	Close Range			0.75	0.568235
	Long Range			0.05	0.145497
	Medium Range			0.2	0.290994

Table D.25.: A summation table of the results using the local method with equal k s

Selector	Local Method			Success Rate	Calculation
	Children	k_1	k_2		
Attack	Direct Attack	0.75	0.25	0.75	0.568235
	Sneak Attack			0.25	0.431765
Direct Attack	Close Range			0.375	0.352808
	Long Range			0.125	0.268076
	Medium Range			0.5	0.379116
Sneak Avoid	Move Around			0.8	0.585786
	Keep In Cover			0.2	0.414214
Direct Attack(Sneak Attack)	Close Range			0.75	0.449087
	Long Range			0.05	0.228196
	Medium Range			0.2	0.322717

Table D.26.: A summation table of the results using the local method for $k_1 = 0.75$ and $k_2 = 0.25$

Selector	Local Method			Success Rate	Calculation
	Children	k_1	k_2		
Attack	Direct Attack	0.25	0.75	0.75	0.695076
	Sneak Attack			0.25	0.304924
Direct Attack	Close Range			0.375	0.373204
	Long Range			0.125	0.163721
	Medium Range			0.5	0.463074
Sneak Avoid	Move Around			0.8	0.738796
	Keep In Cover			0.2	0.261204
Direct Attack(Sneak Attack)	Close Range			0.75	0.665652
	Long Range			0.05	0.0873331
	Medium Range			0.2	0.247015

Table D.27.: A summation table of the results using the local method for $k_1 = 0.25$ and $k_2 = 0.75$

D.2.2. Global Method

Selector	Global Method			Success Rate	Calculation
	Children	k_1	k_2		
Attack	Direct Attack	0.5	0.5	0.75	0.584268
	Sneak Attack			0.25	0.415732
Direct Attack	Close Range			0.375	0.366025
	Long Range			0.125	0.211325
	Medium Range			0.5	0.42265
Sneak Avoid	Move Around			0.8	0.666667
	Keep In Cover			0.2	0.333333
Direct Attack(Sneak Attack)	Close Range			0.75	0.563508
	Long Range			0.05	0.145497
	Medium Range			0.2	0.290994

Table D.28.: A summation table of the results using the global method for equal k s

Selector	Global Method			Success Rate	Calculation
	Children	k_1	k_2		
Attack	Direct Attack	0.75	0.25	0.75	0.465964
	Sneak Attack			0.25	0.534036
Direct Attack	Close Range			0.375	0.352808
	Long Range			0.125	0.268076
	Medium Range			0.5	0.379116
Sneak Avoid	Move Around			0.8	0.585786
	Keep In Cover			0.2	0.414214
Direct Attack(Sneak Attack)	Close Range			0.75	0.449087
	Long Range			0.05	0.228196
	Medium Range			0.2	0.322717

Table D.29.: A summation table of the results using the global method for $k_1 = 0.75$ and $k_2 = 0.25$

Selector	Global Method			Success Rate	Calculation
	Children	k_1	k_2		
Attack	Direct Attack	0.25	0.75	0.75	0.678604
	Sneak Attack			0.25	0.321396
Direct Attack	Close Range			0.375	0.373204
	Long Range			0.125	0.163721
	Medium Range			0.5	0.463074
Sneak Avoid	Move Around			0.8	0.738796
	Keep In Cover			0.2	0.261204
Direct Attack(Sneak Attack)	Close Range			0.75	0.665652
	Long Range			0.05	0.0873331
	Medium Range			0.2	0.247015

Table D.30.: A summation table of the results using the global method for $k_1 = 0.25$ and $k_2 = 0.75$