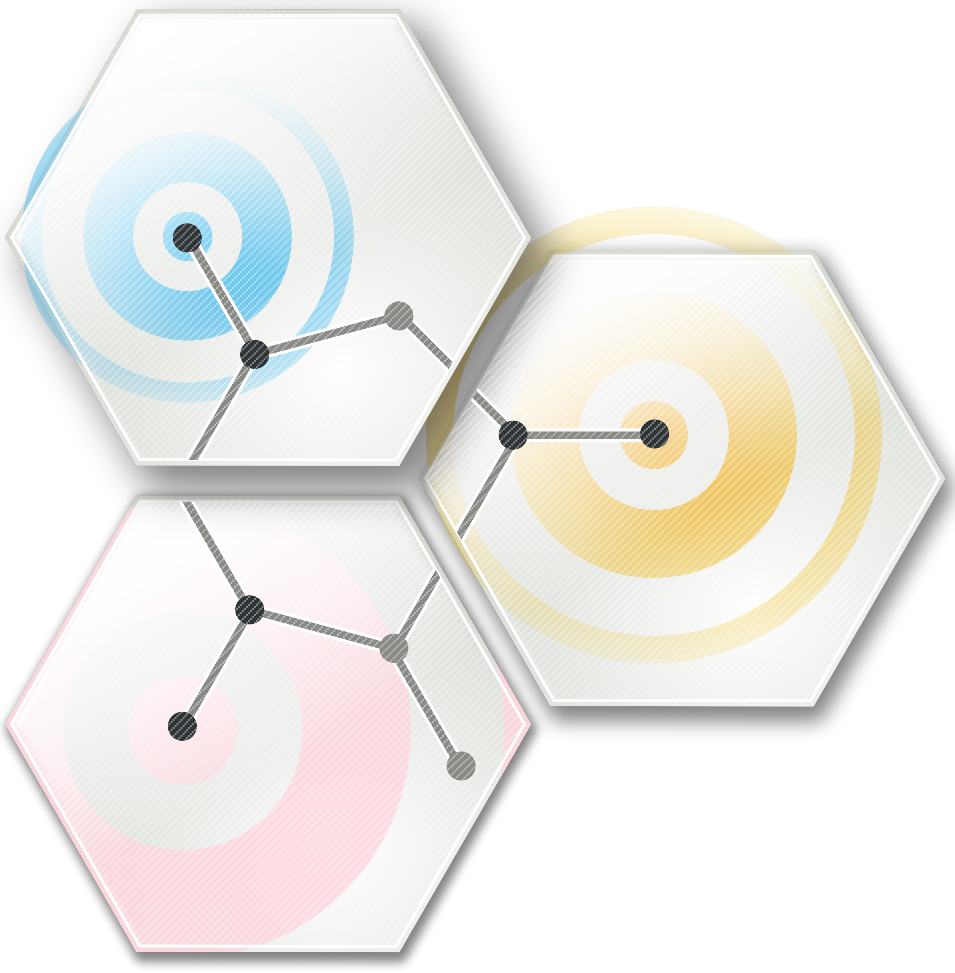
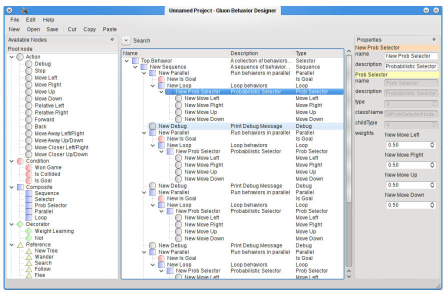
A project by: Morten Justesen Dan Leinir Turthra Jensen Kim Jung Nissen

Aalborg University, Department of Computer Science, Fall 2009

Where Game AI Meets Academic AI



An Investigation into Behavior Trees and Their Relation To Academic AI

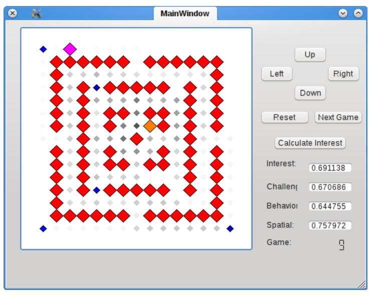

Unnamed Project - Gluon Behavior Designer
File Edit Help
New Open Save Cut Copy Paste
Available Nodes
Search
Properties

```
Root node
Action
    Debug
    Stop
    Move Left
    Move Right
    Move Up
    Move Down
    Relative Left
    Relative Right
    Forward
    Back
    Move Away Left/Right
    Move Away Up/Down
    Move Closer Left/Right
    Move Closer Up/Down
Condition
    Won Game
    Is Collided
    Is Goal
Composite
    Sequence
    Selector
    Prob Selector
    Parallel
    Loop
Decorator
    Weight Learning
    Not
Reference
    New Tree
    Wander
    Search
    Follow
    Flee
```

```
Name                          Description                  Type
Top Behavior                  A collection of behaviors... Selector
  New Sequence                A sequence of behavior...    Sequence
    New Parallel              Run behaviors in parallel    Parallel
      New Is Goal                                          Is Goal
      New Loop                Loop behaviors               Loop
        New Prob Selector     Probabilistic Selector       Prob Selector
          New Move Left                                    Move Left
          New Move Right                                   Move Right
          New Move Up                                      Move Up
          New Move Down                                    Move Down
    New Debug                 Print Debug Message          Debug
    New Parallel              Run behaviors in parallel    Parallel
      New Is Goal                                          Is Goal
      New Loop                Loop behaviors               Loop
        New Prob Selector     Probabilistic Selector       Prob Selector
          New Move Left                                    Move Left
          New Move Right                                   Move Right
          New Move Up                                      Move Up
          New Move Down                                    Move Down
    New Debug                 Print Debug Message          Debug
    New Parallel              Run behaviors in parallel    Parallel
      New Is Goal                                          Is Goal
      New Loop                Loop behaviors               Loop
        New Prob Selector     Probabilistic Selector       Prob Selector
          New Move Left                                    Move Left
          New Move Right                                   Move Right
          New Move Up                                      Move Up
          New Move Down                                    Move Down
    New Debug                 Print Debug Message          Debug
    New Parallel              Run behaviors in parallel    Parallel
      New Is Goal                                          Is Goal
      New Loop                Loop behaviors               Loop
        New Prob Selector     Probabilistic Selector       Prob Selector
          New Move Left                                    Move Left
```

```
New Prob Selector
name          New Prob Selector
description   Probabilistic Selector
Prob Selector
name          Prob Selector
description   Probabilistic Selector
type          2
className     btProbSelectorNode
childType     0
weights       New Move Left
              0.50
              New Move Right
              0.50
              New Move Up
              0.50
              New Move Down
              0.50
```

```
#ifndef _BTBRAIN_H_
#define _BTBRAIN_H_

#include <QObject>
#include <QList>
#include <QHash>
#include <QDomDocument>
#include <QDomNode>

#include "btlib_export.h"

class btNode;
class btNodeType;

class BT_LIB_EXPORT btBrain : public QObject
{
    Q_OBJECT

    Q_PROPERTY(QString name READ name WRITE setName)
    Q_PROPERTY(QString description READ description WRITE setDescription)

public:
    btBrain(QString data);
    ~btBrain();

    btNode* getBehaviorTree(int index);
    int behaviorTreesCount();

    QString name();
    void setName(QString name);
    QString description();
    void setDescription(QString description);

private:
    QHash<int, btNode*> m_behaviorTrees;

    QString m_name;
    QString m_description;

    void parseNodeTypes(QDomNode xNode);
    void parseBehaviorTrees(QDomNode xNode, btNode* node);
};

#endif
```



```
Interest:   0.691138
Challeng    0.670686
Behavioi    0.644755
Spatial:    0.757972
Game:       9
```

Featuring the creation of **SMARTS** a Behavior Tree system for Gluon

simple machine artificial reasoning tree structure

AALBORG UNIVERSITET

Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Ø
Telephone (+45) 9635 8080
http://www.cs.aau.dk

Title:
Where Game AI Meets Academic AI

Project theme:
An Investigation into Behavior Trees and Their Relation To Academic AI

Project period:
Fall Semester 2009,
September 2nd to December 18th

Project group:
d514a

Participants:
Morten Justesen
Dan Leinir Turthra Jensen
Kim Jung Nissen

Supervisor:
Zeng Yifeng

Print run:
5

Pages:
75

Appendices (number, type):
1 CD-ROM with source and binaries

Abstract:

An attempt at making the game AI technique behavior trees useful for academic purposes, and a view on which academic techniques are of a similar nature.
A behavior tree system is created, including a library and an editor, and is used to create a Pac-Man^{TM} alike test bed game and a preliminary implementation in the Gluon engine.
Testing based on work by Yannakakis and Hallam[21] shows that learning behavior trees have a positive effect on gameplay enjoyability.
Behavior trees are shown to be capable and powerful in use. Qt, the underlying framework, is found to be sufficiently advanced for use at this level.

Preface

This report and the SMARTS product was created by the project group d514a at the Aalborg University Computer Science department in the fall of 2009.

We would like to thank Nokia^{TM} and the KDE e.V. for gracefully sponsoring a developer sprint in Munich for the Gluon developers, and Nokia^{TM} for sponsoring their attendance to the Qt Developer Days 2009 conference in Munich in connection with this. We would further like to thank Aalborg University for sponsoring travel costs for the trip there. Finally we would like to thank Nokia^{TM} again, for sponsoring the Qt Essentials Certification exam which means that two of the group members are now certified Qt developers.

Figure 0.1.: Qt Essentials

When external sources are used, the reference for the source is written in square brackets e.g. [1], where 1 corresponds to an entry in the bibliography, which is located at the end of the report. In the case of specific page references, p. means page and pp. means pages.

Emphasising and **Bold** is used to highlight parts of text at times where a new heading would be too drastic a change in the text flow.

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 can further be found on the project website at `http://leinir.dk/gluon-bt/` and at the project page on gitorious at `http://gitorious.net/gluon-bt/`.

Kim Jung Nissen

Dan Leinir Turthra Jensen

Morten Justesen

CHAPTER
1

Introduction

When creating the personalities of the artificial people found in the world of a game, there are many different obstacles to overcome.

Depending on the game genre the game AI has to fulfill different roles and purposes, and thereby also different obstacles has to be overcome. Take for example first person shooter games which usually depend on a strong game AI to give the illusion of intelligence while being unpredictable. In an arcade shooter game in contrast the AI usually moves very predictable but instead there will be a massive number of enemies, which in turn is challenging to the player. Furthermore the game has to be fun to play which in turn means that the game AI should not be a perfect adversary at all time as this would ruin the game experience. This also depends on the genre of the game where in chess the AI can be perfect to be able to give the player a challenge.

There are many ways of designing these personalities, but according to Alex J. Champandard from AiGameDev.com[1] there are generally speaking three ways this is done:

HFSM First described by David Harell[12], Hierarchical Finite State Machines is a way of organising an FSM into logical groups of states, known as super-states, which are then connected by generalised transitions. While this is a good way to organise an FSM, it still inherits some of the problems: Reusing transitions is not trivial, and you have to manually edit the transitions.

Planners Planners basically perform searches to provide AI logic, meaning that they are goal oriented by default. However, they do not handle integration with procedural code: Since a plan is generated when the planner is run, and then ought to be able to be followed to the end, it cannot adapt to changes in the world. So in the end the planner has to be run far too often, thus changing the elegant solution into something closer to a brute force attack.

Scripting This is any type of normal programming you might have. The problem with this is obvious for our purposes - it requires knowledge of programming languages. Since this is something we specifically wish to avoid, this is not useable.

However, other than this there is one method which is becoming popular in game development circles known as Behavior Trees. This method sits somewhere between the three methods above, containing some aspects of all three. Chapter 2 on page 8 describes in further depth what defines a behavior tree from a technical point of view, so here shall simply be stated that they amongst other things contain a way of defining logic flow in a hierarchical manner, in other words a tree.

One of the other very important things when talking about behavior trees, one much less technical than those mentioned in Chapter 2, is that they are created using a design tool rather than through a text editor. In other words: It is something that you can use without needing to learn how to write code. While this may seem an odd thing to highlight, what it means is that you can give this tool to a Game Designer - the person or

[1] `http://aigamedev.com/insider/presentations/behavior-trees/`

persons on a game development team normally in charge of creating the flow in the game - who will then be able to create the personalities and test them out without needing to involve the programmers directly.

This is a very powerful thing to be able to do, because it removes one layer of communication in a process which otherwise very easily becomes a series of Chinese Whispers. While the programmers still need to implement the functionality in the game itself, the actual logic does not need to be translated from a description by the game designer into code, as this is done automatically by the tool.

One thing a game designer might want to do would be to create a number of actions, such as for example ”Take cover”, ”Run away” or ”Shoot at enemy”, which many different people might use, except they may not use them in the same situations. For example, where one gung-ho person may be more inclined to try and shoot at the enemy at all times, another more cowardly person might run away at the slightest sign of danger. In that case the only difference between the two persons would be the main behavior, and the way it chooses its children, while the three main behaviors which contain the majority of the logic are reused by both personalities.

1.1. Project Purpose

Through this project, we will create a behavior tree system consisting of the following items:

- Design tool for use by game designer
- Library for use by game developers
- Preliminary integration into an existing game development library

Furthermore we shall investigate methods of modifying the behavior tree in a computer-assisted manner, by using machine learning techniques and AI techniques designed for use in game development. Finally the impact of the system on the enjoyment of a concrete game scenario will be tested.

CHAPTER
2

Behavior Trees

In this chapter we will demonstrate how behavior trees work on the high level using one practical example. This is done to demonstrate how the different components work together, and what kind of strategies can be used when designing behavior using behavior trees.

First we will describe the individual components that make up a behavior tree and then put these into context using a small example.

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.

2.1. Building Blocks

There exist six different types of components for creating a behavior tree. This section will cover the usage and functionality of these components.

2.1.1. Composites

To choose between the different paths through the tree, logic structures are needed called Composites - named that way because they contain any number of sub-trees. Two base structures are created, Sequence and Selector, which are mutially exclusive. For convenience, a further structure called a Parallel is constructed, allowing multiple sub-trees to run at the same time.

Sequence

As the name indicates, a sequence is a sequence of behaviors executing in a specified order[7]. 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.

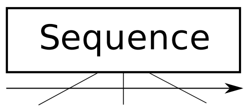


Figure 2.1.: Sequence node

A sequence will react differently depending on the termination statuses received from its children:

Child returns success: If a behavior returns success, then the sequence will continue and execute the next child.

Child fails: The sequence will return the termination status to the parent and end execution.

Selector

While sequences only execute behaviors in a given order, selectors are used for selecting a given behavior for execution[6]. 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 completely non-deterministic methods.

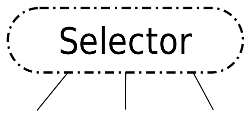


Figure 2.2.: Selector node

As a sequence, the selector has to handle the termination statuses from its children, though the reactions resembles those of a sequence there are some differences.

Child returns success: If a behavior returns success, then the selector also returns success to its parent.

Child fails: The selector will then select another behavior for execution.

Parallel

The components described so far, it is possible to create a behavior tree, but the execution of the tree will be very linear, and not supporting any form of concurrency[5]. This is where Parallels come into the picture.

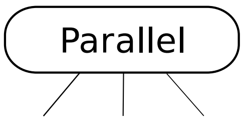


Figure 2.3.: Parallel node

Parallels are capable of running all of their children simultaneously, and are responsibly for all of their child behaviors, insuring they terminates cleanly when the parallel itself terminates. There are different ways of deciding when the parallel has to terminate, depending on in the context which it is used:

Failure policy: Specifies how many and optionally which behaviors have to fail, before the parallel itself fails.

Success policy: Almost same as above, but concerning how many behaviors have to executed successful before the parallel itself terminates successful.

Master behavior: Here the parallel tracks a specific behavior and returns the termination status of that given behavior.

Using parallels can be a daunting task if it is necessary to micro-manage them. It is recommended that a scheduler is used or implemented for this purpose. Furthermore using the scheduler's observers, such that the child behaviors are not run in every update, but instead wait for a notification before beginning execution. To help this further, it is ideal to design low-level behaviors which are either independent of each other or have the capabilities required for cooperating with each other.

There are three pieces of advice for using parallels:

- Group a single action together with multiple conditions. The action and the conditions will not conflict, because the action is the only behavior which changes the world state, while the conditions only query the information regarding the world state.
- Use completely different behaviors, such that they will not work on the same data, and thereby not conflict.
- If the behaviors are more complex, with multiple behavior trees, then consider patterns for keeping the behavior trees synchronized.

2.1.2. Leaves

As with any tree structure, Behavior Trees have leaves. In a Behavior Tree, however, Composites cannot be leaves, as they contain no actual logic in themselves other than that which runs sub-trees. As such, the Behavior and Reference nodes exist.

Behavior

Behaviors in a behavior tree can contain actions which the game AI has to perform or be subtree of the behavior tree. The actions are provided by the programmers, such that the AI designer can use them when constructing behavior trees. If a behavior does not change anything in the world it is called a condition, otherwise it is called an action.
Furthermore each behavior will end execution with a termination status, which could one of following:

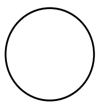


Figure 2.4.: Behavior node

Success: When the behavior successfully completed the execution.

Failure: Occurs when the behavior fails cleanly or any unexpected errors occur.

Reference

References are links to other behavior trees, and they make it possible to append one behavior tree on another.
This feature is that which makes behavior trees expandable and modular. It allows the AI designer to design subparts of a behavior, and later link these together. Because of this ability, behavior trees support reuse and easy interchanging of behaviors.

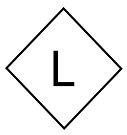


Figure 2.5.: Link node

2.1.3. Decorators

Composites and leaves are the corner stones when creating a behavior tree[9]. However at some point will arise the need 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: It can not be leaf node. Thereby a decorator can extend a behavior with extra functionality without the behavior having any knowledge about the decorator. Furthermore a decorator is not a branch, because it only has one child behavior.

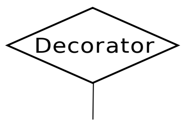


Figure 2.6.: Decorator node

A decorator can be almost anything, but some examples are:

Filters: Preventi behaviors being run on certain circumstances, such as limit the number of times a behavior can be run, or insures that a behavior only fires at certain intervals.

Managers & handlers: These can be used as handlers for the execution of the subtree or the behavior. Thereby an error handler if an error is returned, or as a blackboard which the children can query for information.

Control modifers: Can be used as a mask, hiding the real execution of a child behavior. E.g returning always success, no matter if the behavior fails or succeeds.

Meta operations: These are very useful when debugging as they can function as break points or logging mechanisms.

2.2. Example

This example demonstrate behaviors of a generic game enemy that perform tasks such as patrolling and attacking the player. We chose this example because it is simple, very

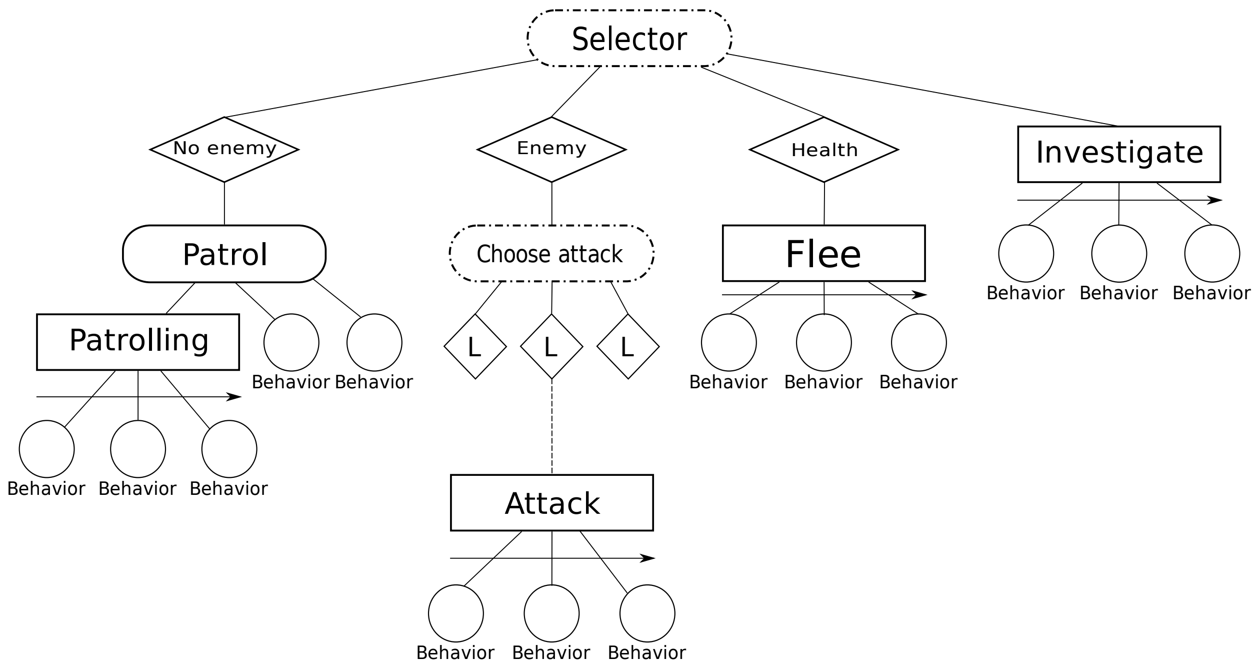


Figure 2.7.: Example of a behavior tree

general but still has all the core features of behavior trees. The example can be seen in Figure 2.7 on page 11

The top level behavior of this tree is a selector which can chose between a number of sub-behaviors to execute. The selector can chose between four behaviors: a patrol behavior, an attack behavior, a flee behavior and an investigate behavior.

There are several approaches to selecting which behavior to execute, these can be specified for each selector. In principle any form of decision making could be used to chose between the behaviors, but using only probability and priority based decision making is already very powerful and flexible.

2.2.1. Structure of the example

The first subtree is the Patrol subtree. The first component used in the Patrol subtree is a decorator node which here is used as a guard such that it does not execute this if there is a known enemy. The parallel is here used to add conditions which are to be monitored during the execution of a behavior, such as bailing out of the patrol behavior if an enemy is spotted or some other message is received. The Patrolling sequence performs the actions needed for performing a patrol, but can be interrupted by one of the parallel behaviors.

The second subtree is the Attack subtree. The first component is an inversed version of the one in the Patrol behavior, in that this behavior can only be selected when there is an enemy present. The choose attack selector works like the toplevel selector, but does not point to behaviors directly, instead it makes use of links. The use of links makes it easy to add or remove behaviors, and selecting a specific Attack could be where a developer would want this flexibility. A developer or another designer could create a set of different Attack behaviors such as sneak attacks or direct attacks and designers could then link to these behaviors whereever suitable. The Attack behavior shown is a sequence of behaviors or actions, but could be any subtree.

The Flee and Investigate behaviors are simpler behaviors, consisting only of sequences of actions these work like the first two with the exception that Investigate does not have a guard decorator, making it always available for execution. This is an important point, as it avoids the unfortunate case of a game character which stands still because it does not know what to do.

2.2.2. Control Flow

When execution starts, the top selector selects a sub-behavior to execute. Let us assume there is no enemy in range and health is full, this mean that it will execute neither Attack nor Flee as they are guarded with conditions on these variables. This only allows for Patrol or Investigate to be executed. Furthermore, if it is assumed that the Investigate behavior is selected first based on probabilities provided by the designer.

Now the Investigate behavior executes a sequence of behaviors in the designed order, these include actions such as code or other behaviors. When the investigate sequence is complete assuming no problems it returns success and the top level selector choose a new behavior to execute. We now assume that it selects the patrol behavior, which consists of a parallel execution of behaviors.

A behavior which performs the actions associated with patrolling and several condition monitoring behaviors are run in parallel. The Patrolling behavior is a sequence of actions and can be interrupted at any time by one of the other behaviors. If we assume that one of the conditions is whether there is an enemy and that it detect an enemy, this will return and cause the Patrol behavior to bail out.

2.3. Designing Behavior Trees

A technology such as behavior trees cannot exist in a vacuum: It needs to have a defined process for their construction and use. This is important as behavior trees need to be used by both the designers who define the behavior of a character and the programmers who implement the actions needed by the behavior.

Behavior Oriented Design (BOD)[4] suggests a method for creating Complete, Complex Agents (CCA). It is our belief that this method can be adapted and applied to the creation of behavior trees. We base this belief on the fact that a correspondence has been shown between BOD and behavior trees[14], and that it is similar to object decomposition in Object Oriented Design (OOD).

BOD uses action patterns and competences. These constructs are similar to sequences and priotized selectors in that they can easily replace each other as described by Chong-U Lim[14].

We now present the proposed development method to be used with behavior trees.

2.3.1. Development Method

The first step in the method is to create an initial decomposition of the overall behavior of a character or a class of characters as it can easily be reused.

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.

It is possible to start with an implementation of only one of the behaviors of a character, and it is possible to test and debug that code in isolation due to the modularity of behavior trees[8]. A behavior can be evaluated against the specification for that behavior and the specification can be revised based on the findings. This is repeated until all behaviors are implemented and perform as expected. All lists of behaviors and actions should be kept as documentation for both designers and developers.

We now suggest a series of best practices for building behavior trees. These are inspired by Bryson[4], Partington and Bryson[16] and by OOD in general.

Favor simplicity An action is generally better than a sequence of actions and a sequence of actions is usually better than a selection of actions or sequences. Start simple and elaborate as needed.

Reduce redundancy Reuse behaviors and actions as much as possible. If only part of an action or behavior can be reused, consider further decomposition.

Complex composites If a selector or sequence has more than five or six children, consider clustering some of these together in sub-selectors or sub-sequences. This reduces clutter and promotes reuse.

Too many decorators It can be tempting to explicitly control behavior through decorators, such as using them to guard certain behaviors. This is usually better handled as conditions in the behaviors themselves.

2.4. Summary

In this chapter behavior trees are described, together with an example and how to design behavior trees.

In Section 2.1 on page 8 the building blocks of behavior tree are described. These building blocks are used for constructing the behavior tree, and each building block has purpose where a sequence is a sequence of behaviors, selectors are used for decision making, decorators extend a behavior's functionality and parallels can run behaviors simultaneously.

In Section 2.2 on page 10 an example of a behavior tree is described together with the structure of the behavior tree. Furthermore the control flow for the execution the example behavior tree is described.

Lastly a procedure for designing behavior trees is described in Section 2.3 on the preceding page. The procedure specifies the first step as defining all high level behaviors of a given character. Next step is to group possible activities and specify them as sequences of actions. The third step is to use sequences for identifying all the needed low-level actions. The fourth step is to identify conditions which need to be fulfilled to activate actions and goals, providing the division into behaviors. In the fifth step identification and ordering of goals of the character is done. In the last step a behavior is selected and implemented.

CHAPTER
3

Method

This chapter focuses on the theoretical background of the project. Here is investigated how learning can be applied to the domain of behavior trees, and how it relates to the existing literature in the area of Machine Intelligence.

3.1. Behavior Tree Learning

What we are trying to model with behavior trees are unpredictable but reasonable behaviors. What is usually wanted when modelling with decision graphs and influence diagrams in particular is THE best choice, where we want a set of reasonable choices and a weighting of these so that the probability of choosing an action with a high weight is higher than actions with lower weights at runtime.

A selector in a behavior tree can select in what order it wants to try behaviors in many different ways. The way we are most interested in is to select a behavior depending on their weight, such that a behavior with a high weight is more likely to be chosen for execution first, compared to a behavior with a lower weight. The traditional way to control this is by manually assigning weights to behaviors, and using this to affect overall behavior.

But what we want to do is to learn these weights based on data from playing the game, and even doing it as an online process which can adjust the weights as the game is played.

The measure we use to figure out which behaviors we want to try is the succes and failure rates of a behavior. Every time an behavior is selected it is recorded whether or not this behavior succeeded or not. The relation between the success and failure rates is then used to calculate the weight of that behavior.

The weight should be seen as how likely a behavior is to succeed if selected, so we obviously want to select behaviors that are likely to succeed. We do not, however, want this selection to be deterministic, such that in a given situation this behavior is chosen; it should only be more likely to be chosen.

The intuitive approach to this problem and also an efficient one in terms of time and memory is to divide the number of successful runs with the total number of runs and then normalize the values such that the calculated weights add up to one.

3.2. Adapting Behavior Tree

There are two types of adapting behavior trees: First you can adapt the structure, secondly you can adapt its behavior internally. This section concentrates on the second of

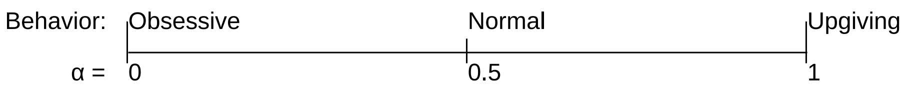


Figure 3.1.: The α value scale

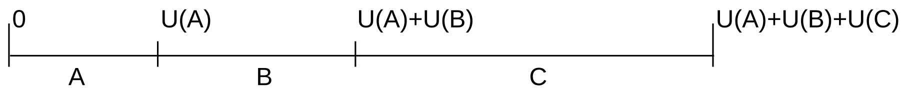


Figure 3.2.: The probability scale of a hypothetical composite containing three sub-trees (A, B and C)

the two. What that means is that we will devise a method by which we can adapt the way a probability based selector or sequence chooses between its children.

To allow for changing the behavior of one of the composites, a value is added signifying the type of behavior required. A floating point value $0 \leq \alpha \leq 1$ describes this (see Figure 3.1 on the facing page).

The likelihood of selecting one sub-tree on the composite can be describes as a number from 0 to the combined utility of all the possible sub-trees of the composite. An example distribution can be seen in Figure 3.2 in which a hypothetical composite has three sub-trees A, B and C, and the utility of selecting each of them is described as $U(A)$, $U(B)$ and $U(C)$ respectively.

What this means is that when you wish to select a sub-tree from a set, you find the value of $U(n)$ for each of the sub-trees, and choose a number r between 0 and the added values of all the values of $U(n)$ found in that manner ($U(A) + U(B) + U(C)$). r is then checked against the scale to see which sub-tree range it falls into. This method is equivalent to roulette wheel selection[1, pp. 118-120].

The value of $U(n)$ is found through

$$U(n) = (1 - \alpha) * o(n) + \alpha * u(n) \tag{3.1}$$

which consists of the weighting of obsession and upgivingness of the composite node in relation to the node n through α as described above. The two extremes are defined by the formulas

$$o(n) = n.success + 1 \tag{3.2}$$

which describes obsessive behavior according to the amount of times the node has been visited and returned a success, and

$$u(n) = \frac{1}{1 + n.failure} \tag{3.3}$$

describes upgiving behavior according to the number of times a node has been visited and returned failure.

To allow this to work over time, without total memory, incomplete memory is applied. This is emulated by adding a fading system. A value $0 \leq \lambda \leq 1$ is used to define the amount by which the previous time step's utility function is remembered (1 for complete memory, 0 for no memory). This leads to the adapted utility function

$$U(n,t) = (1 - \alpha) * o(n) + \alpha * u(n) + U(n, t - 1) * \lambda \tag{3.4}$$

finally describing how a game character views the likelihood that the subtree n will succeed if they choose it, with a weighted memory of what has happened in the past.

3.3. Existing Work

In this section we will look at existing learning algorithms for game AI, and we will find a suitable algorithm which matches our own.

3.3.1. Learning in Decision Trees

Learning in decision trees uses ID3 but there exist optimized versions of the algorithm e.g of C4.5 and C5, and also incremental versions e.g ID4. ID3 uses an example set, which could either be generated at runtime and used for online learning or data which is collected during execution and used for offline learning. The example set consists of variables considered and the actions possible for the AI, where one entry in the set is the state of the variables and what action was taken.

The algorithm calculates the entropy of the set and afterwards the gain for each variable, and uses the gain to generate the new decision tree. The variable with the highest gain is used for the first decision and so forth[15].

We will not use decision tree learning for comparison because the algorithms are doing structural learning, which our own does not. We only change the order of execution of the child nodes. Further more the decision tree learning algorithms are using entropy and gain instead of utility in correlation with probability which our algorithm use, as described in Section 3.2 on page 14. Therefore the algorithms found in decision tree learning is not comparable with our own.

3.3.2. Reinforcement Learning

Another technique for learning in games is reinforcement learning. In this technique the algorithm looks at the world as a state machine where each state holds relevant information regarding the world, and at any given time the algorithm is in a given state. In each state there is a table of possible actions and the current Q value for that action. When ever an action has been taken, the algorithm updates the Q value for the current action taken while it takes the highest Q value in the new state into account[15].

We have decided not to use reinforcement learning even though our own algorithm from Section 3.2 resembles that of reinforcement learning. The reason for this is that the state space of a game can be of arbitrary size and could therefore be incredibly memory intensive, even though there exists hierarchical reinforcement learning variants. Furthermore the algorithm from reinforcement learning uses the old Q value together with the highest Q value from the new state when calculating the new Q value. Our algorithm just uses the old value for calculating the new value, and is also used together with probability in the selection of the next action.

3.3.3. Learning in Bayesian Networks

A bayesian network[3] can be learned from a set of data either through parameter learning or constructing a model from the data[13]. In parameter learning the bayesian network learns parameters from the data set. If the data set is complete it is just a question of counting, while an incomplete data set will require an algorithm for learning the parameters e.g expectation-maximization.

When doing structural learning on bayesian networks there are two ways of approaching this problem either by constraint-base learning or score-base learning. In constraint-base learning the PC algorithm can be used. The algorithm starts of with a complete graph and removes links such that conditional independence is kept. Afterwards 4 rules are used for setting the direction of the remaning links. Score-base learning uses a scoring function for determining the usefulness of a bayesian network. By using score-based learning, a bayesian network is generated, the score of the bayesian network is calculated, until a terminate condition has been met, then the bayesian network and score is saved. Lastly the bayesian network with the highest score is selected.

Our personality based learning algorithm, Section 3.2 on page 14, adapts runtime with through parameter learning. While it is run every time new information regarding the success or failure of child nodes is available. Parameter learning in bayesian networks could have been useful, but because parameter learning in bayesian network is very expensive and is more suitable for doing learning offline, this option is discarded. We also discard structural learning in bayesian networks, because first of all it is very expensive and is also more suitable for offline learning. Second our own algorithm does not use structural learning but instead parameter learning and therefore not comparable.

3.3.4. Learning in Behavior Trees

Behavior trees can be seen as hierarchical dynamic scripts[20] where instead of explicitly generating scripts based on weights on the rules, these scripts are implicit in the form of sequences and selectors. In dynamic scripting each character class has a rulebase with a set of rules, where each rule has a weight and defines a piece of script. On each encounter of the human player rules are extracted from the rulebase and used to form the AI script for the given encounter. After the encounter each used rule is evaluated to figure out if it has contributed to success and fail, and the weight of rule is then increased if it contributed to success, otherwise is is decreased[20].

After running a behavior in a behavior tree, we update the weights of the selected behaviors based on the $U(n,t)$ function defined in Section 3.2 on page 14 such that successful behaviors are rewarded and unsuccessful behaviors are penalized. Note that while this seems straight forward, the reasoning that this is the game character's view on the successfulness of it makes it somewhat more complicated: The weights increases when the character perceives a higher likelihood that taking some action will succeed.

Dynamic scripting uses integer values as weights and uses roulette wheel selection[1, pp. 118-120] to select which actions to include in the finished script. We use the same method except that we use it on-the-fly (on-line), selecting actions at a finer level.

It is important that if a behavior is penalized unjustified, it does not make it impossible to select the behavior: As less fit behaviors are selected more often, they will be punished such that the levels will be more even again. This in turn causes the space in the total fitness taken up by a behavior which is initially chosen often but is unsuccessful to be chosen less. It does not, however, mean that this particular behavior ever reaches a fitness of zero (unless the fitness function allows for this, though this should be discouraged). Because of this, the behavior is never invalidated entirely, and as such it is always possible to choose it, though it becomes more unlikely.

For the purposes of testing, we construct behavior trees with three types of selectors: The first will execute behaviors from a prioritized list. The second selector selects children in random order, meaning that each behavior has an equal likelihood of being executed. The last type will use the algorithm from Section 3.2 on page 14 to define the fitness of each child, and thus exhibits changing behavior over time.

We are not going to compare our algorithm in perspective of already existing learning algorithms for computer game AI, but instead focus on how the different behavior trees perform in contrast to entertainment value of the game as defined by Yannakakis and Hallam[21].

3.4. Summary

During this chapter was investigated the methods of learning used in Decision Trees and Bayesian Networks, as well as how Reinforcement Learning works, after which is looked at how learning could be performed in the context of the Behavior Tree structure presented in Chapter 2. Further is arrived at a method for gauging the utility of a sub-tree in a Composite node in a Behavior Tree, and the $U(n, t)$ function is constructed.

CHAPTER

4

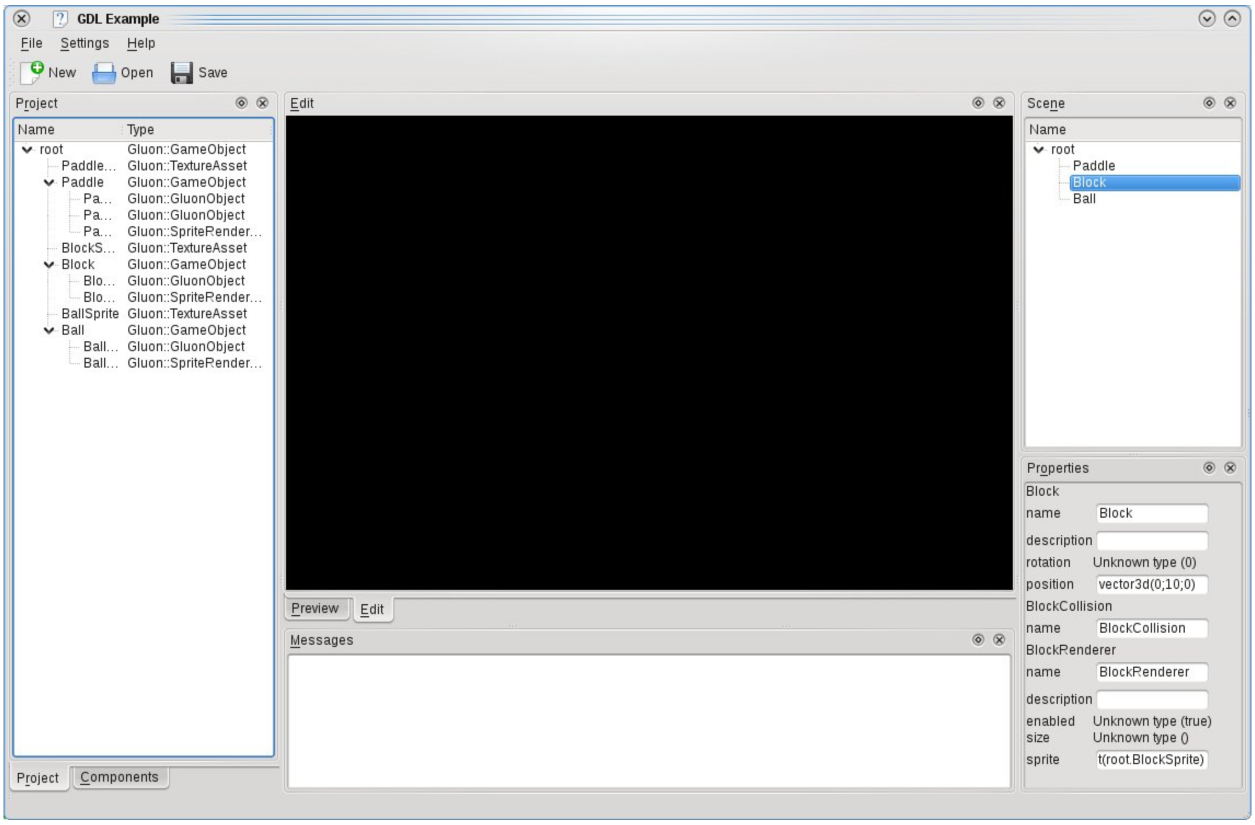
Analysis

Before beginning the design of the system itself, we first need to look at just what it is supposed to be doing. To do this, we take a look at the ecosystem it has to fit into, namely Gluon. Gluon is made up of two main parts:

- The Gluon library itself
- Gluon Creator, the editor inside which games build on Gluon are created

4.1. Gluon Creator

In the following we first look at Gluon Creator, as pictured in Figure 4.1, describing the workflow of game creation using it. The relevance of this to the creation of our behavior tree system is that, as mentioned in the introduction, they are not supposed to be created in code, but rather using some tool. As such, the workflow for creating and using behavior trees should be similar to how the rest of Gluon's subsystems (such as input, graphics and physics) work.



Name	Type
root	Gluon::GameObject
Paddle...	Gluon::TextureAsset
Paddle	Gluon::GameObject
Pa...	Gluon::GluonObject
Pa...	Gluon::GluonObject
Pa...	Gluon::SpriteRender...
BlockS...	Gluon::TextureAsset
Block	Gluon::GameObject
Blo...	Gluon::GluonObject
Blo...	Gluon::SpriteRender...
BallSprite	Gluon::TextureAsset
Ball	Gluon::GameObject
Ball...	Gluon::GluonObject
Ball...	Gluon::SpriteRender...

Name
root
Paddle
Block
Ball

Block	
name	Block
description	
rotation	Unknown type (0)
position	vector3d(0;10;0)
BlockCollision	
name	BlockCollision
BlockRenderer	
name	BlockRenderer
description	
enabled	Unknown type (true)
size	Unknown type ()
sprite	t(root.BlockSprite)

Figure 4.1.: Gluon Creator

4.1.1. GameObjects and Components

To understand how Creator works, first it is important to describe some of the basic concepts used. Namely GameObjects, Components [11].

A **GameObject** represents a position in the game world, including rotation and scale. Each GameObject has either one or no parents, and any number of children, which leads to a tree structure of GameObjects, which then represent the items found in the game world.

Note that a GameObject has no inherent logic, except for that which allows for the chaining of its transformations; that is, the position, rotation and scale of a GameObject is relative to its parent, except in the case of one which has no parent, in which case it is relative to the identity (that is, the center of the world, and no rotation and scale).

Each GameObject furthermore has any number of **Components**, which is where all game logic and other functionality sits, including such things as audio playback, sprite rendering, physics handling etc. Components are small, self-contained pieces of code, which operate on the GameObject they are attached to. They are either one of the generic, predefined Components which ship with Gluon, or they are created by a game programmer for use in a specific game.

Finally, each Component can reference any number of **Assets**, which is the way Gluon reads game content - sounds, graphic files, text files and so on. Assets themselves can contain sub-assets, which are created at run-time from the contents of the file they represent. This allows for such things as handling for example a tar file by creating sub-assets according to what files are found inside it.

4.1.2. Workflows

Now that the GameObjects and components are described, the workflows in Creator can be described. The construction of a game using Gluon can be described as the action of putting together sets of GameObjects and Components, each set representing one level in the game. As such, the workflows under scrutiny here are those which allow you to put the levels together.

The **Properties** panel allows the user to interact with all relevant aspects of an object which is selected. The right side of Figure 4.1 on the previous page shows how the object named "Block" is selected and its properties shown in the panel below. The name of the GameObject is shown at the top, and the two components "BlockCollision" and "Block-Renderer" are then shown underneath in that order, with all their properties shown in a list.

Drag and Drop interaction is used heavily throughout the interface. Adding a new Component to an existing GameObject, for example, is done by dragging it from the Components pane on the left side in the screenshot and onto the GameObject it should be added to in the Scene pane on the right. Similarly, reorganisation in the Scene and Project panes is done by dragging the elements around in the tree view.

4.2. The Gluon Library

Next we look at the Gluon library, to find out how the library enables the workflows presented in Gluon Creator. In addition to the GameObjects and Components, some amount of convenience functionality is needed. At the core of this is the Gluon Project concept, which is also used by Creator as its persistence layer.

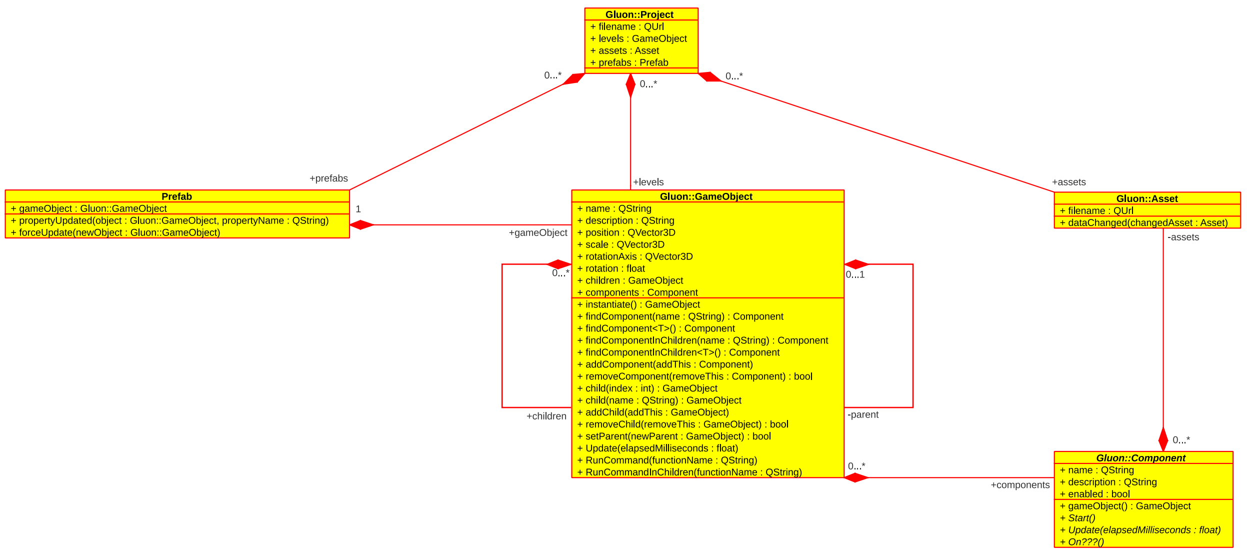

Gluon::Project
+ filename : QUrl
+ levels : GameObject
+ assets : Asset
+ prefabs : Prefab
0...*
0...*
0...*
+prefabs
+levels
+assets
Prefab
+ gameObject : Gluon::GameObject
+ propertyUpdated(object : Gluon::GameObject, propertyName : QString)
+ forceUpdate(newObject : Gluon::GameObject)
1
+gameObject
Gluon::GameObject
+ name : QString
+ description : QString
+ position : QVector3D
+ scale : QVector3D
+ rotationAxis : QVector3D
+ rotation : float
+ children : GameObject
+ components : Component
+ instantiate() : GameObject
+ findComponent(name : QString) : Component
+ findComponent<T>() : Component
+ findComponentInChildren(name : QString) : Component
+ findComponentInChildren<T>() : Component
+ addComponent(addThis : Component)
+ removeComponent(removeThis : Component) : bool
+ child(index : int) : GameObject
+ child(name : QString) : GameObject
+ addChild(addThis : GameObject)
+ removeChild(removeThis : GameObject) : bool
+ setParent(newParent : GameObject) : bool
+ Update(elapsedMilliseconds : float)
+ RunCommand(functionName : QString)
+ RunCommandInChildren(functionName : QString)
0...*
+children
0...1
-parent
0...*
+components
Gluon::Asset
+ filename : QUrl
+ dataChanged(changedAsset : Asset)
-assets
0...*
Gluon::Component
+ name : QString
+ description : QString
+ enabled : bool
+ gameObject() : GameObject
+ Start()
+ Update(elapsedMilliseconds : float)
+ On???()

Figure 4.2.: The Gluon::Project class hierarchy

A **Gluon Project** represents all the assets (that is, arbitrary data such as sounds, images and other items used by a game), all levels and their GameObjects and their Components, as well as a number of convenient helpers known as Prefabs, which are not relevant to our investigation but are described in the Gluon documentation[1].

In the library these projects are represented by the Gluon::Project class. The class diagram in Figure 4.2 on page 21 shows this class and the four other main classes in Gluon:

Gluon::GameObject is the core component of any game in Gluon, and contains a number of functions to rapidly find both Components and GameObjects in the hierachy, as well as the data structures required for keeping track of the parent/child relationship between GameObjects, as well as their Components.

Gluon::Asset is a representation of a lump of data, an instance of Gluon::Asset helps Gluon keep track of whether a file is changed on disk, and informs all the Components which contain a reference to that Asset of any changes, allowing them to react to those

Gluon::Component is an abstract class, which describes the formal layout required for any Component

Gluon::Prefab describes the convenient class which itself contains a GameObject/Component hierarchy. The prefab can be instantiated, but as an instance of a Prefab retains its link with the Prefab, any changes in the Prefab are propagated into its instances. Furthermore, any change in an instance can, in turn, be forced back into the Prefab, which then in turn propagates this into its instances

Gluon::Project represents the game project itself, and all data associated with it, including which level is the first

[1] `http://gluon.tuxfamily.org/wiki/`

4.3. Summary

Through the preceding chapter was investigated how Gluon Creator and the Gluon library are structured, and a number of key points were discovered that will be used in the design of the SMARTS system.

CHAPTER

5

Design

Here is described the design of the SMARTS Library and SMARTS Designer system, and the process of arriving at that design.

5.1. SMARTS Library

The SMARTS Library contains all the functionality for using and running behavior trees and will be included in the Gluon library. To be able to provide this functionality, the library provides both internal classes and public classes with different purposes.

Figure 5.1 illustrates which classes SMARTS Library contains, and what relation they have to each other. Each class has its own purpose and is as follows:

btBrain: The btBrain is a data container which contains all the behavior trees created by the btFactory.

btFactory: This class is a singleton which has the capabilities for creating an instance of btNode and btNodeType.

btNodeType: This is the base class for behavior tree components which are described in Section 2.1 on page 8. Every user customized behavior tree component has to

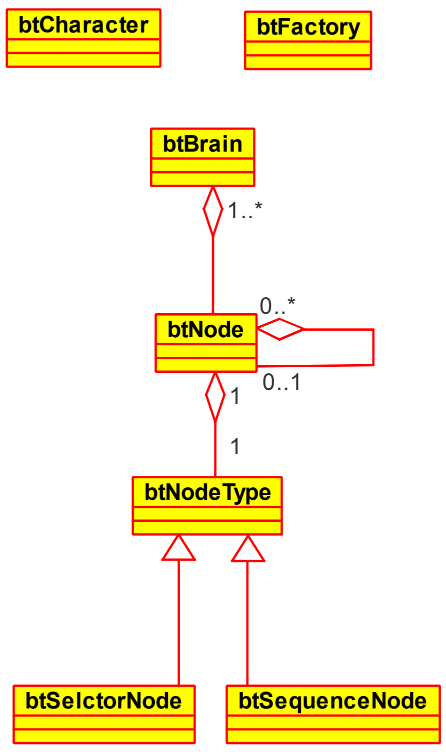


Figure 5.1.: SMARTS Library class diagram

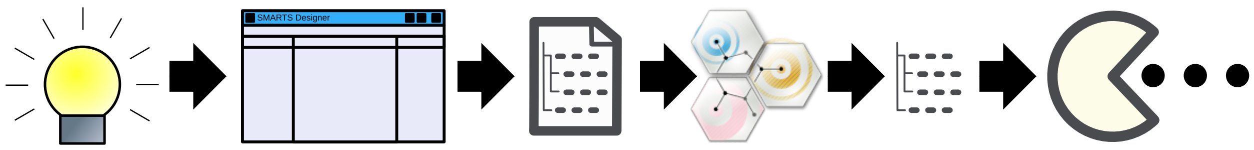


Figure 5.2.: The Workflow

inherit from this class. It contains all the basic functionality for a behavior tree node, which is used in the SMARTS Library for execution of the AI.

btNode: This is an internal representation of nodes in the behavior tree. It is linked together with a parent and any giving number of decorators or children. Furthermore it contains a pointer to the corresponding btNodeType, which handles the AI logic for a given behavior tree node.

btCharacter: The base class for all characters created in the game. It is used for querying information regarding the world state during the execution of the behavior tree.

btSelectorNode and btSequenceNode: These are standard selector and sequence nodes which are accessible to the end user. These classes inherit from the btNodeType class. Both classes runs its children in a specific order: btSelectorNode uses a prioritized list, and btSequenceNode is an ordered sequence of behaviors.

Both the btBrain and btFactory will have public methods but will do more work behind the scene, The end user will not have any knowledge concerning this work. Because of btNodeType the behavior tree components can be extended as the end user sees fit, and provides a base for any new behavior tree components the end user creates. btNode is only editable by the developers and the end user will not have any knowledge about this class.

The Link Between Gluon and SMARTS Library

Using SMARTS Library with Gluon is done through the creation of a set of Gluon plugins, which wrap the behavior trees in a manner which allows users of Gluon to interact with them as they would any other part of Gluon itself. Three plugins are created:

Brain Asset: This asset will load the brain file, which is then passed to SMARTS Library for loading.

Tree Asset: This asset is instantiated by the Brain asset. One instance is created per behavior tree in the Brain file.

Character Component: This component takes care of when to think. It further allows for setting the data corresponding to a single character, as used by SMARTS Library for distinguishing between each in-game character and their personalities.

5.2. SMARTS Designer

One of the most important requirements that has to be met in order to make behavior trees a viable technology for both programmers and designers to use, is the availability of good tool support. We have attempted to make our own tool for creating behavior

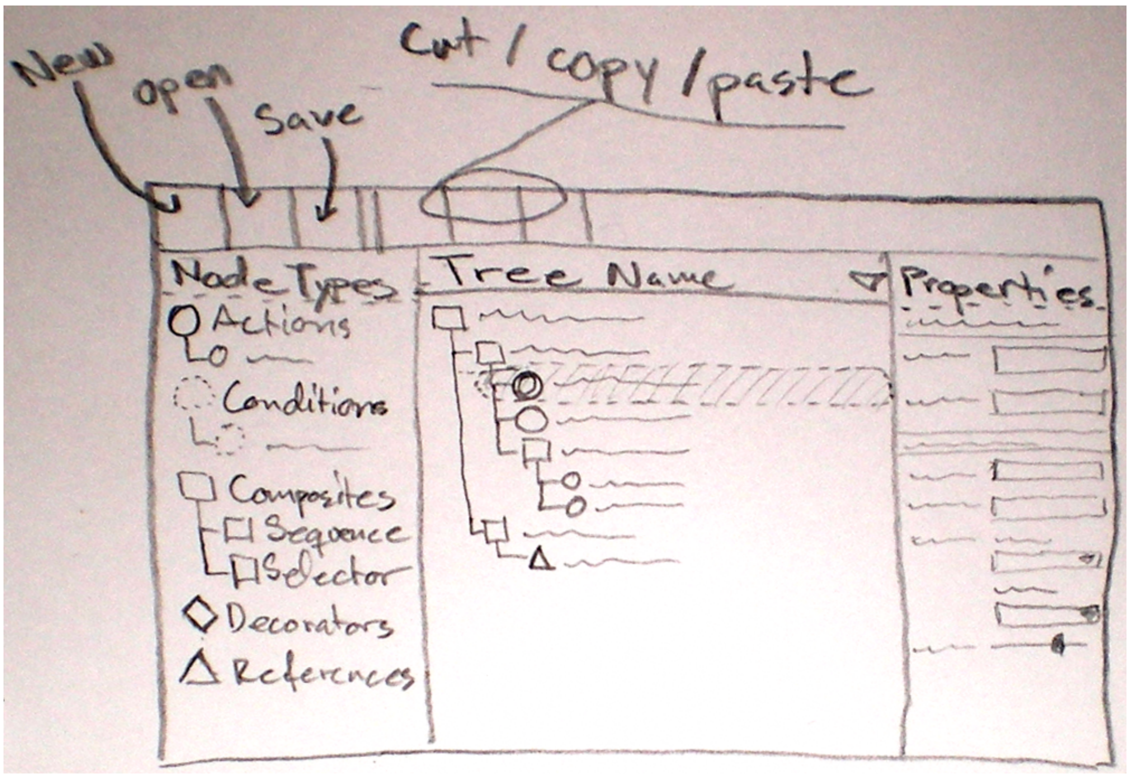


Figure 5.3.: A Sketch of the Behavior Tree Editor

trees, that can be used with our game framework. This tool is to bridge the gap between the low level details of the game engine and the high level of designing behaviors.

The idea behind the designer is that the programmers can create behavior, composite and decorator nodes as C++ classes, and the designer can then use these in a drag and drop fashion to create complete behavior trees. The designer does not need to know the implementation details in order to use the nodes, the only thing need is a high level understanding of a given action given in the description and the name of the action.

Hide Complexity The editor should not require the designer to know about the implementation of the nodes.

Direct Manipulation The designer should be able to manipulate the behavior tree directly through drag and drop.

Powerful It must be possible to extend the available items in the editor easily, thus allowing the game programmer to expose custom behaviors and their properties directly in the editor to the designer

The process goes like this: the designer needs a set of actions, the programmer implements an action, e.g. running. This action is turned into a node which can be used in SMARTS Designer. The designer creates the tree and the tree is saved to a Behavior Tree file. This file is then loaded through SMARTS Library by the game, the classes are instantiated and the behavior is executed. This process is illustrated in Figure 5.2 on the facing page.

5.2.1. User Interface

Pictured in Figure 5.3 on the previous page, the interface consists of three views: A nodes view which shows the available nodes, a properties view which shows the details of the currently selected node and the actual behavior tree view that allow the designer to build a behavior tree. The interface work by drag and drop, a node from the nodes view can be dragged to the behavior tree view to create a new instance of the node in the tree.

Information on the node types in the node view can be changed through the node editor, which can be accessed by double clicking the node in the node view. Information on any instantiated nodes in the behavior tree view can be edited through the properties view, which shows the properties of any selected node.

5.2.2. Motivation and Inspiration

The inspiration for the SMARTS Designer comes from Alex J. Champandards example implementation demonstrated on `http://aigamedev.com` and Brainiac Designer[1], a tool created by Daniel Kollmann.

Alex's implementation is very simple and not in any way complete, but it does demonstrate that an editor can be made using a simple tree-view widget without need for sophisticated graphical views.

Daniel's Designer is in many ways more advanced than what we are attempting to make, it has graphical view and support for plugins such as the ability to export to native code. The problem with the Brainiac Designer is that it is built specifically to work with Project Hoshimi and C#, and even though it does have the opportunity to function with others, we deem that it would be benefitial to work with the subject from scratch, to not have to limit ourselves.

We want to create a more general tool for building behavior trees, and thus we will use model-view extensively in the editor such that if we later wanted to create a more sophisticated graphical view we could replace the current tree widget without deep changes to the rest of the code.

5.2.3. Persistence Layer

The SMARTS Designer saves a file, as described in section 5.2 on page 24, which the SMARTS Library uses when loading the behavior trees into the library. This file is the persistence layer between the designer and the library, and the design of this file is described in this section.

Very early on tt was decided that the data saved from SMARTS Designer should be formatted as XML.

When the SMARTS Designer saves the file, the structure is as follows:

The root element of the XML file is a `project` element which only has one attribute called `name` which contains the name of the project. `project` only has the two child elements `nodetypes` and `behaviortrees`. The `nodetypes` element contains all the node types in the behavior tree, while the `behaviortrees` element contains all the behavior trees specified in SMARTS Designer. The structure of these three XML elements are shown in listing 5.1.

Listing 5.1: The root element of the XML file with the two child elements

[1]`http://www.codeplex.com/brainiac`

```
...
<project name="behavior tree">
    <nodetypes>
    ...
    </nodetypes>
    <behaviortrees>
    ...
    </behaviortrees>
</project>
```

Each node type under the `nodetypes` element is a `nodetype` element which has following four attributes:

`name` Which is the name of the node type

`description` The node type description

`classname` Name of the corresponding C++ class

`category` Which is only used in the SMARTS Designer.

If a `nodetype` element has any child elements, then these elements specifies the user defined properties from the SMARTS Designer. These elements are `property` elements and they contain three attributes:

`name` The name of the node type

`description` Description of the property

`datatype` Specifies the data type of the property.

Listing 5.2 illustrates this.

Listing 5.2: The structure of the `nodetype` element

```
...
<nodetype name="node" description="node description" classname="nodeClass" category="composite">
    <property name="newProperty0" description="property description8" datatype="QString" />
    ...
</nodetype>
...
```

Listing 5.3 on the next page illustrates the structure of `behaviortree` elements. These elements are child elements of a `behaviortrees` element and contains two attributes which can be seen on line 2:

`name` Specifies the name of the behavior tree

`uid` Defines the id of the behavior tree, as used by reference nodes

Each child element of `behaviortree` is a `behaviornode` element and corresponds to a `nodetype` element, but as a tree node in the behavior tree. Each `behaviornode` has the following attributes:

`name` The name of the node

`description` Node description

`nodetype` Corresponds to the `classname` of a `nodetype`

When the `nodetype` is surrounded by brackets it indicates that a standard `btNodeType` subclass is used, e.g `btSelectorNode` which is described in Section 5.1 on page 23. Furthermore a decorators has its own XML element wich is `decorator`. These elements must be a child element of a `behaviornode` element and has the same attributes as the `behaviornode` element. The `decorator` and `behaviornode` elements can be seen on lines 3, 4 and 5.

A `behaviornode` element can have `property` elements, `decorator` elements and other `behaviornode` elements as child elements, while `decorator` elements only can have `property` elements as child elements. The `property` element specifies a user defined property and is the same property defined under the corresponding `nodetype` element. A `property` element has two attributes, and can also be seen on line 8:

`name` The name of the property

`value` The value of the property

These two attributes are used for finding and setting the value of the giving property in the both the SMARTS Library and SMARTS Designer.

Listing 5.3: The structure of the `behaviortree` element

```
 1 ...
 2 <behaviortree name="New Tree" description="a behavior tree" uid="0">
 3     <behaviornode name="Top Behavior" description="A collection of behaviors which are launched
           in order, until one succeeds (only fails if all fails)" nodetype="[selector]" >
 4         <behaviornode name="New node" description="node description" nodetype="theNode" >
 5             <decorator name="New Decorator Node" description="decorator description" nodetype="
                   decoratorNode" >
 6                 ...
 7             </decorator>
 8             <property name="newProperty0" value="hello world" />
 9             ...
10         </behaviornode>
11         ...
12     </behaviornode>
13     ...
14 </behaviortree>
15 ...
```

5.3. SMARTS Test-bed

Pac-Man^{TM}, as seen in Figure 5.4 on the next page, was originally released in Japan on the 22nd of May, 1980 as an arcade game. It features a world made up of a grid of corridors, in which pellets are found that the Pac-Man character, controlled by the player, has to pick up. To prevent him from doing this, four opponents in the form of the ghosts Shadow, Speedy, Bashful and Pokey are inserted into the game. They attempt to catch Pac-Man on his way through the level by following a set of pre-defined, deterministic behaviors. Pac-Mac^{TM} has the opportunity to pick up a number of power-ups called power pellets, which make the ghosts reverse direction and in the earlier levels allow him to eat the ghosts.

For these reasons we have selected a variation of Pac-Mac^{TM} as our test-bed for behavior trees. As others have used Pac-Mac^{TM} before to test different forms of behavioral algorithms and methods it makes it easier to compare results and build on existing knowledge so we do not have to build everything from scratch.

Furthermore, arguably the choice of Pac-Man^{TM} has been made for us since the test method we wish to employ, created by Yannakakis and Hallam[21], is performed on a

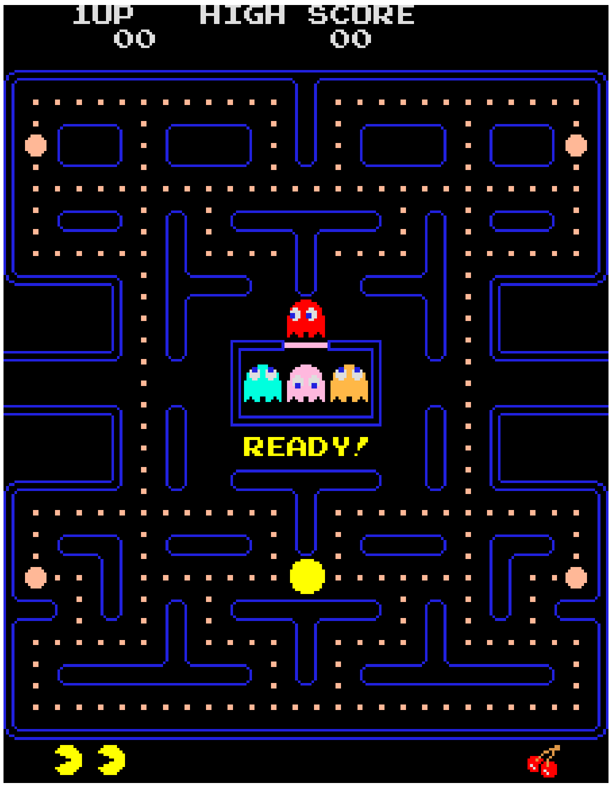


Figure 5.4.: A screenshot of the original Namco version of Pac-Man™

variation of that game in the original material, and as such to be able to compare results, we will need to use as much of the same material.

Our game differs from Pac-Mac™ in that the goal is not to eat pellets but to capture a set of points scattered around the map. Our version also ignores the ability of Pac-Mac™ to eat the ghosts. This is a reasonable limitation for our purposes as others we want to compare to do the same[21]. Our game is inspired not only by Pac-Mac™, but also by the game NoEsc[10]. This game has similar gameplay and even uses behavior trees, but is much too complicated to do any reasonable reasoning about.

5.3.1. Game Rules

To evaluate the performance we need clear game rules:

1. The player's goal for each level is to capture a set of points on the map.
2. Upon the player completing the objectives of a level, the game continues to the next level. This is equal to resetting the level, but does not count as a player death.
3. The map has guards, these guards do not want the player to succeed so they will try to capture the player before he wins.
4. The player cannot kill or otherwise harm the guards.
5. If the player touches any of the guards he dies and the level is reset.

5.3.2. The Problem

We want to find out whether using behavior trees leads to easier development and better behaving agents. As behavior trees in general are based on the idea that they should make it easier for the game designer to define the behaviors of the computer-controlled in-game characters to the benefit of the players of that game's enjoyment of the game, it seems obvious to look at whether machine learning can assist in achieving this goal.

If it turns out to be advantageous to use this in a game scenario, it would mean that it would further assist the designer, giving them more opportunities while requiring less time, thus augmenting the existing power that behavior trees have in that area when using a good tool to create them, such as the one suggested in Section 5.2 on page 24. Also by using the metrics from Yannakakis and Hallam[21], we can measure if using behavior trees as a tool makes it easier to create interesting behaviors.

5.3.3. Use of Decorators

The implementation of learning in behavior trees uses components of behavior trees to organize it. We use learning decorators that can be applied to any compatible composite in a tree. This design allows the behavior designer to decide if e.g. a selector should use learning, and indeed what kind of learning it should employ. This allows for any kind of probability adjustment to be employed.

5.3.4. Design Challenges

A number of challenges arise when designing the system, specifically pertaining to the running of behavior trees at the same time as game logic and drawing runs elsewhere. This section shortly touches on the main issue of this, namely threading.

The idea behind the Parallel composite is that its children should run simultaneously, and as such the first option which springs to mind is to use threading for this. This brings with it a number of issues, but the most problematic is the return value problem when working across thread borders.

We need some way for asynchronous calls to be able to return whether or not they succeeded. This problem occurs due to the threading involved in Parallels. The reason is that calling a function on an object which exists in a different thread is not safe.

The solution employed here is to store the return value somewhere outside the function, such that it can be used in the behavior tree. The problem then boils down to where to store these values, optimally in a transparent manner: How do we do it in a general way such that it will work both when executing asynchronously and synchronously?

One issue is that signals/slot connections cannot return a value (the functions must be void) thus there is no way of directly knowing whether or not an action succeeded. Possible storage positions for the value are:

- Save it on the Enemy class (thus storing the value per-character)
- Save it in the behavior tree (which would cause problems when several characters use the same tree for calculations)
- Some other place (a singleton designed for that purpose, potentially problematic across thread borders)
- More advanced use of signal/slot or method invokation, maybe integrated into the behavior tree structure

We select the first of these options, storing the values on a per-character basis. The reasoning is that while the last option would be more powerful, it is unknown territory and the risk deemed too high for this project. The first option is a known risk, and as such is simpler to work with.

5.4. Summary

The preceding chapter describes the design of SMARTS Designer and SMARTS Library, as based upon the results in Chapter 4. It furthermore describes the design of the persistence layer used to allow the editor and library to interact with each other. Lastly, it describes the technical design of the test bed, and the adaptation of the methods in the article upon which it is based to use with behavior trees.

CHAPTER
6

Implementation

The following chapter describes the implementation of each part of the project product. Firstly is described the library and the editor, highlighting interesting aspects of their code. Secondly the test case which was devised, an implementation of Pac-Man in which each character is controlled by a behavior tree. Finally the implementation of behavior trees as Gluon plugins is described.

6.1. SMARTS Library

The SMARTS Library is the implementation of the behavior trees and is described in this section.

Before starting to describe the implementation more in detail, it is important to mention and describe the two shared classes, `btNode` and `btNodeType`, and the shared header file, btglobal.h. These classes and header file are shared with the SMARTS De-

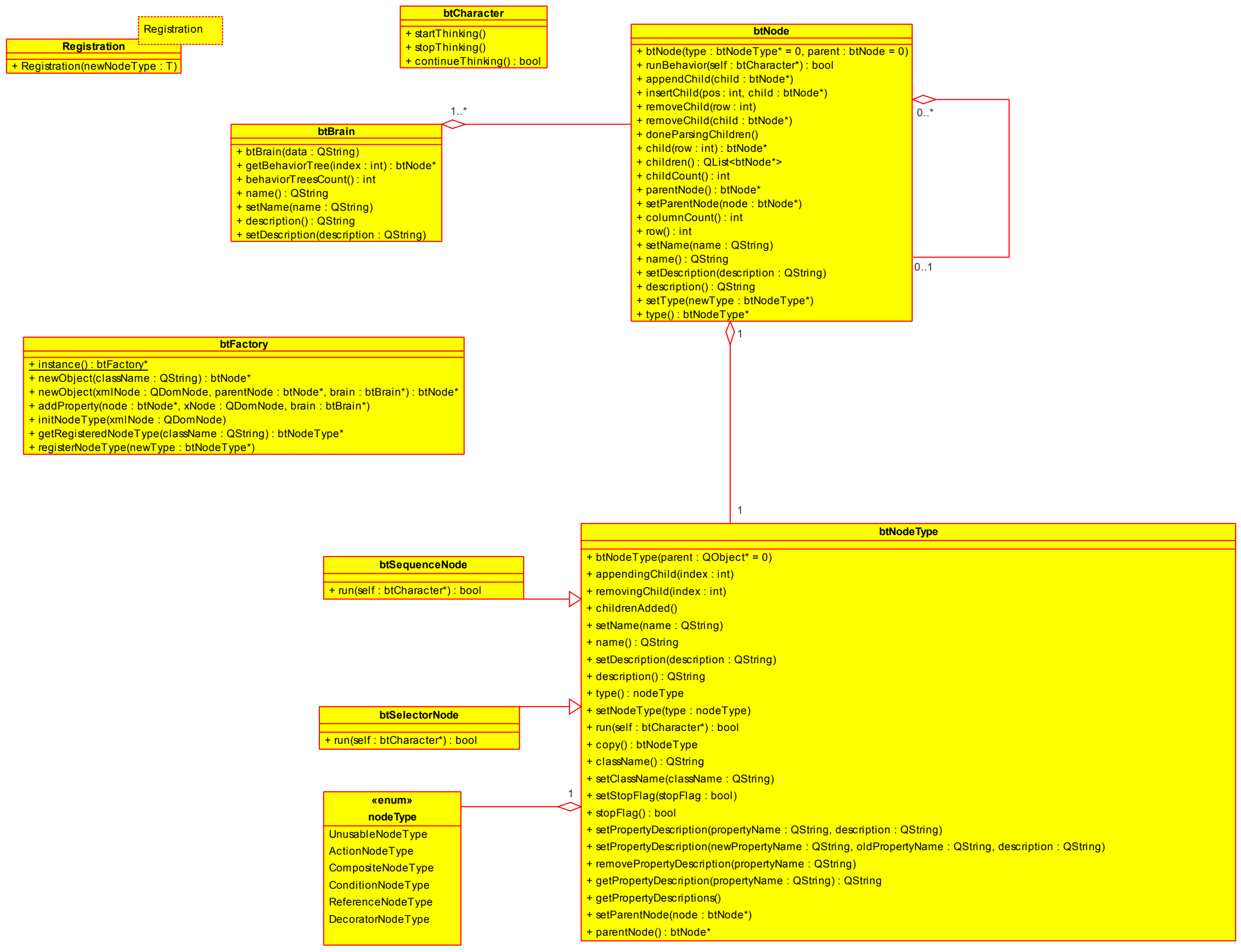

Registration
Registration
+ Registration(newNodeType : T)
btCharacter
+ startThinking()
+ stopThinking()
+ continueThinking() : bool
btNode
+ btNode(type : btNodeType* = 0, parent : btNode = 0)
+ runBehavior(self : btCharacter*) : bool
+ appendChild(child : btNode*)
+ insertChild(pos : int, child : btNode*)
+ removeChild(row : int)
+ removeChild(child : btNode*)
+ doneParsingChildren()
+ child(row : int) : btNode*
+ children() : QList<btNode*>
+ childCount() : int
+ parentNode() : btNode*
+ setParentNode(node : btNode*)
+ columnCount() : int
+ row() : int
+ setName(name : QString)
+ name() : QString
+ setDescription(description : QString)
+ description() : QString
+ setType(newType : btNodeType*)
+ type() : btNodeType*
1..*
0..*
0..1
btBrain
+ btBrain(data : QString)
+ getBehaviorTree(index : int) : btNode*
+ behaviorTreesCount() : int
+ name() : QString
+ setName(name : QString)
+ description() : QString
+ setDescription(description : QString)
btFactory
+ instance() : btFactory*
+ newObject(className : QString) : btNode*
+ newObject(xmlNode : QDomNode, parentNode : btNode*, brain : btBrain*) : btNode*
+ addProperty(node : btNode*, xNode : QDomNode, brain : btBrain*)
+ initNodeType(xmlNode : QDomNode)
+ getRegisteredNodeType(className : QString) : btNodeType*
+ registerNodeType(newType : btNodeType*)
1
1
btSequenceNode
+ run(self : btCharacter*) : bool
btSelectorNode
+ run(self : btCharacter*) : bool
«enum»
nodeType
UnusableNodeType
ActionNodeType
CompositeNodeType
ConditionNodeType
ReferenceNodeType
DecoratorNodeType
1
btNodeType
+ btNodeType(parent : QObject* = 0)
+ appendingChild(index : int)
+ removingChild(index : int)
+ childrenAdded()
+ setName(name : QString)
+ name() : QString
+ setDescription(description : QString)
+ description() : QString
+ type() : nodeType
+ setNodeType(type : nodeType)
+ run(self : btCharacter*) : bool
+ copy() : btNodeType
+ className() : QString
+ setClassName(className : QString)
+ setStopFlag(stopFlag : bool)
+ stopFlag() : bool
+ setPropertyDescription(propertyName : QString, description : QString)
+ setPropertyDescription(newPropertyName : QString, oldPropertyName : QString, description : QString)
+ removePropertyDescription(propertyName : QString)
+ getPropertyDescription(propertyName : QString) : QString
+ getPropertyDescriptions()
+ setParentNode(node : btNode*)
+ parentNode() : btNode*

Figure 6.1.: SMARTS Library class diagram

signer. Then `btBrain` and `btFactory` are described and also what the their roles and capabilities are in the SMARTS Library. Lastly the `btCharacter` is described together with the purpose of this class.

6.1.1. The `btNode` Class

The `btNode` class is used for the internal structure of a behavior tree, including the root nodes, and therefore not intended to be used for inheritance or be altered during implementation and execution of a behavior tree. Because the `btNode` is used for the internal structure, it also contains a list of child nodes in the tree. All the functions in `btNode` are virtual, but they should not be overridden. Each `btNode` has one and only one `btNodeType` attached which contains the functionality of that given node type. This can also be seen on Figure 6.1 on the preceding page.

The functione `runBehavior(btCharacter* self)` is shown below:

Listing 6.1: `btNode`'s runBehavior

```
bool btNode::runBehavior(btCharacter* self)
{
    if(m_type)
    {
        if(!m_type->run(self))
        {
            return false;
        }
    }

    return true;
}
```

Since it is `btNodeType` which contains the current functionality the `runBehavior` will return the same as the `run` on `btNodeType`. The function has one parameter which is an instance of `btCharacter`. The purpose of this class is described in Section 6.1.5 on page 36.

6.1.2. The`btNodeType` Class

The `btNodeType` is the base class for any kind of node types in the behavior tree, e.g selectors, sequences, decorators, etc. This implies that if a new node type is needed, the new class has to inherit from `btNodeType`, and as a minimum, the `run` function has to be overridden with the needed functionality. All the other functions of `btNodeType` are virtual, which gives the possibility to override the functions and modifying the functionalities they provide. Furthermore `btNodeType` also contains any user defined properties which are added through the SMARTS Designer. Here the shared header file comes into play, because it contains an internal datastructure which is used for certain types of properties. This is described in Section 6.2.3 on page 41.

Listing 6.2: `btNodeType` functions used in SMARTS Library

```
...
virtual void appendingChild(int index){};
virtual void removingChild(int index){};
virtual void childrenAdded(){};
...
virtual bool run(btCharacter *self);
...
virtual QString className() const;
...
btNode* parentNode();
...
```

The listing above shows the important functions of `btNodeType`. `appendingChild`, `removingChild` and `childrenAdded` are used when constructing the behavior trees. `appendingChild` and `removingChild` are called every time a new child has been added or removed from the `btNodeType`'s `btNode`. When overriding these two functions it is possible for the `btNodeType` to perform certain task when ever such events happen. The same goes for `childrenAdded` which is called when there are no more child nodes going to be added to the `btNodeType`'s `btNode`.

The `run` function runs the functionality of the `btNodeType` and is called from the `btNodeType`'s `btNode` as shown in Listing 6.1 on the preceding page.

During the construction of the behavior tree, the `className` is used for finding the correct `btNodeType` subclass for instantiation. The classname is defined in the SMARTS Designer.

The last function is `parentNode` which is very important when creating the functionality of the `btNodeType` subclass. Since `btNodeType` does not contain the list of child nodes, which in turn is placed on `btNode`, it is necessary for the `btNodeType` being able to access this if the `btNodeType` has to do any operations on any child nodes.

There are already two standard `btNodeType` subclasses available in the library which are `btSelectorNode` and `btSequenceNode` which correspond to the two composites selector and sequence, described in Section 2.1 on page 8.

Each subclass of `btNodeType` fall under one of four different categories defined by the enum `nodeType` and they are:

`CompositeNodeType` Such as selectors, sequences and parallels.

`DecoratorNodeType` These nodes are just decorators.

`ActionNodeType` Actions which are the leaf nodes of a behavior tree.

`ReferenceNodeType` References to other behavior trees.

At the moment these categories at primarily used in the SMARTS Library, which is described in Section 6.2 on page 37, as way of distinguishing between the nodes types when creating the data xml file.

6.1.3. The `btBrain` Class

The class `btBrain` is a container for the behavior trees and is also used for instantiating the behavior trees. The name can be misleading as the `btBrain` is not the brain of a character but merely used as a container for the behavior trees, while providing access to them, thus closer resembling a hive mind than a brain.

Listing 6.3: Functions for using the `btBrain`

```
...
public:
    btBrain(QString data);
...
    btNode* getBehaviorTree(int index);
    int behaviorTreesCount();
...
private:
...
    void parseNodeTypes(QDomNode xNode);
    void parseBehaviorTrees(QDomNode xNode, btNode* node, int nodeIndex);
...
```

Listing 6.3 on the preceding page shows three public functions used for constructing, accessing and running the behavior trees, and two private functions which are used internally for instantiating the behavior trees.

When an instance of `btBrain` is created the constructor takes the data as a QString parameter. This QString contains all the data for the instantiation of the behavior trees and consists of xml created through the SMARTS Designer. The `btBrain` constructor parses the data and starts to instantiate the behavior trees.

Algorithm 1 `btBrain::btBrain(QString data)`

```
QDomDocument xmlDocument ← load the QString data into a QDomDocument
QDomNode nodeTypes ← find the root node of the node types in the xmlDocument
parseNodeTypes(nodeTypes)
QDomNode behaviorTrees ← find the root node of the behavior trees in the xmlDocument
for all Behavior trees in behaviorTrees do
    Create root node
    Append root node to the list of behavior trees
parseBehaviorTrees(behaviorTrees, NULL, 0)
```

Algorithm 1 shows the procedure when the `btBrain` is instantiated. During the instantiation the `parseNodeTypes` is used for parsing and append all information to each node type. In the next step the `btBrain` first instantiates the root nodes of the behavior trees, and then uses `parseBehaviorTrees` to parse and instantiate the behavior trees defined in the QString data, building the tree depth first. Both `parseNodeTypes` and `parseBehaviorTrees` use the `btFactory` during this process, where `parseBehaviorTrees` is a recursive function.

After the instantiation of `btBrain`, it is possible to get the number of behavior trees with `behaviorTreeCount`. With `getBehaviorTree` it is possible to retrieve a pointer to the root node of a behavior tree. Since all tree nodes are instances of `btNode` it is possible to start the execution of the tree just by calling the `runBehavior` on the `btNode`.

6.1.4. The `btFactory` Class

`btFactory` is, as the name indicates, a factory. It is used for instantiating `btNode`s and `btNodeType`s during the initialization of the behavior trees.

Listing 6.4: Used `btBrain` functions during the the initialization of behavior trees

```
...
public:
    static btFactory* instance();
    btNode* newObject(QString className);
    btNode* newObject(QDomNode xmlNode, btNode* parentNode, btBrain* brain);
    btNode* createRootNode(QDomNode xmlNode, btBrain* brain);
    void addProperty(btNode* node, QDomNode xNode, btBrain* brain);
    void initNodeType(QDomNode xmlNode);

    btNodeType* getRegisteredNodeType(QString className);
    void registerNodeType(btNodeType* newType);
...
```

The `btFactory` is a singleton class which is accessed through the `instance` function. During the parsing of the different node types, the `initNodeType` function is called to add the user defined properties to the current node type, which in turn are defined in the xmlNode parameter.

When starting to parse a behavior tree, a root node needs to be created. The root node is always a sequence node, as described in Section 2.1 on page 8, and for this purpose the `btBrain` calls `btFactory`'s `createRootNode`. When the rest of the behavior tree is initialized both `newObject` and `addProperty` are used.

There are two different `newObject` functions, where the `newObject(QString className)` is used for creating a new `btNode` with a `btNodeType` corresponding to the className parameter. But if this function is used, it is up to the programmer to correctly attach the returned `btNode` to a behavior tree. On the other hand `newObject(QDomNode xmlNode, btNode* parentNode, btBrain* brain)` not only creates a `btNode` with a `btNodeType` from the xmlNode parameter, but attaches the `btNode` as child node on the parentNode parameter.

If a `btNodeType` has property which needs to be assigned, the function `addProperty` is used. This function assigns the property on the `btNodeType` of the node parameter. Furthermore the xNode parameter is the current xml node, that holds the information concerning the property in question.

In functions where an instance of `btBrain` is a parameter, the `btBrain` instance is used for instantiating reference nodes.

The `btFactory` instantiates the different node types at runtime. For the `btFactory` to be able to do this any kind of subclasses of `btNodetype` has to be registered with the `btFactory`.

Listing 6.5: Macro and template for registering node types

```cpp
template<class T>
class BT_LIB_EXPORT Registration
{

public:
    Registration(T* newNodeType)
    {
        btFactory::instance()->registerNodeType(newNodeType);
    }
};

#define REGISTER_NODETYPE(NEWNODETYPE) \
Registration<NEWNODETYPE> NEWNODETYPE ## _registration_(new NEWNODETYPE());
```

Listing 6.5 shows a template class and a macro, which are used for this purpose. The template class `Registration` is used in conjunction with the macro `REGISTER_NODETYPE`. The macro has to placed in the cpp file of a `btNodeType` subclass, with the class name as aurgment. The macro is then expanded by the preprocessor and a new instance of `Registration` is placed on stack, while the constructor registers the node type into the `btFactory`.

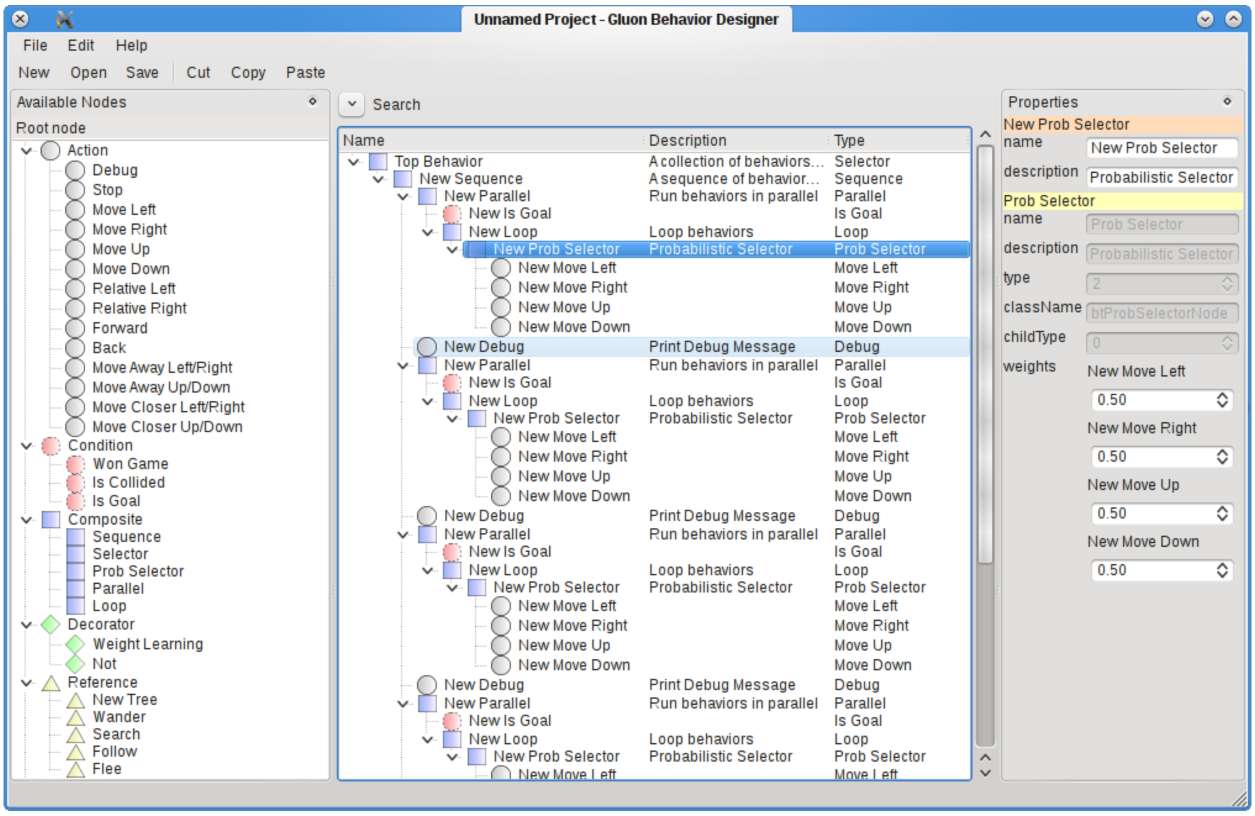
Listing 6.6: Example use of the `REGISTER_NODETYPE` macro

```cpp
...
REGISTER_NODETYPE(decoratorNode)
...
```

Listing 6.6 shows an example of this, where a class `decoratorNode` is registered in the `btFactory`.

6.1.5. The `btCharacter` Class

The `btCharacter` class is used as a base class for any implementation of a character, and therefore the implementations must inherit from this class. The reason for this is that a behavior tree is only used for decision making, and will query the `btCharacter`



Name	Description	Type
Top Behavior	A collection of behaviors…	Selector
New Sequence	A sequence of behavior…	Sequence
New Parallel	Run behaviors in parallel	Parallel
New Is Goal		Is Goal
New Loop	Loop behaviors	Loop
New Prob Selector	Probabilistic Selector	Prob Selector
New Move Left		Move Left
New Move Right		Move Right
New Move Up		Move Up
New Move Down		Move Down
New Debug	Print Debug Message	Debug
New Parallel	Run behaviors in parallel	Parallel
New Is Goal		Is Goal
New Loop	Loop behaviors	Loop
New Prob Selector	Probabilistic Selector	Prob Selector
New Move Left		Move Left
New Move Right		Move Right
New Move Up		Move Up
New Move Down		Move Down
New Debug	Print Debug Message	Debug
New Parallel	Run behaviors in parallel	Parallel
New Is Goal		Is Goal
New Loop	Loop behaviors	Loop
New Prob Selector	Probabilistic Selector	Prob Selector
New Move Left		Move Left
New Move Right		Move Right
New Move Up		Move Up
New Move Down		Move Down
New Debug	Print Debug Message	Debug
New Parallel	Run behaviors in parallel	Parallel
New Is Goal		Is Goal
New Loop	Loop behaviors	Loop
New Prob Selector	Probabilistic Selector	Prob Selector
New Move Left		Move Left

New Prob Selector
name: New Prob Selector
description: Probabilistic Selector
Prob Selector
name: Prob Selector
description: Probabilistic Selector
type: 2
className: btProbSelectorNode
childType: 0
weights:
New Move Left: 0.50
New Move Right: 0.50
New Move Up: 0.50
New Move Down: 0.50

Figure 6.2.: A screenshot of SMARTS Designer editing a brain

class for any information concerning the world state. That is the reason that an instance of `btCharacter` is sent down through the nodes using `run` and `runBehavior` in `btNodeType` and `btNode` respectively.

The `btCharacter` contains three functions which are used for stopping the behavior tree if necessary. `startThinking` sets a bool that defines that the `btCharacter` can run its behavior tree, where `stopThinking` is the opposite. `continueThinking` is used by the `btNode` to check if it should continue execution.

6.2. SMARTS Designer

SMARTS Designer (pictured in Figure 6.2) makes heavy use of Qt's Model/View system, Interview, for the two main views, and the property editing system uses the introspection system provided by QMetaObject to work with the properties on QObject instances in a generic manner.

6.2.1. Models

To allow the model/view system to work, a number of data models were created, specifically the ones defined by the `btNodesTypesModel` and `btTreeModel` classes. As teaching model/view programming is outside the scope of this report, these models shall only be described superficially in so much as they allow for the understanding of their use in the views.

Importantly, however, the Brain concept is caried over, as is NodeType. Two specific types are created when a new brain is created: Selector and Sequence. These two types are vital to the behavior tree working, as they provide the basic logic flow. The hidden root node is a Sequence with a single child, a Selector. This is done for convenience, but also allows better internal control. Forcing the two composite types allows for easier internal consistency, and as such it is partly for technical reasons as well.

Listing 6.7 shows the enumeration used to distinguish between different node types in the editor. Note that while this is also set in the library, it is only used for debugging purposes there, and has no functional reason behind existing. In the editor, however, it is used to distinguish between which node types are available where, for sorting in the tree-views and in general for graphical representation.

Listing 6.7: The btNodeType::nodeType enumeration

```
btNodeType::nodeType
{
        UnusableNodeType = 0,
        ActionNodeType,
        CompositeNodeType,
        ConditionNodeType,
        ReferenceNodeType,
        DecoratorNodeType
    };
}
```

btNodesTypesModel

The `btNodesTypesModel` class is based on QAbstractItemModel, which is the most advanced of the model types found in Qt. What this means is that it is an arbitrary level nested model, or a tree. Listing 6.8 shows the header file of the class. A short description of each part of the model follows.

Listing 6.8: The btNodeTypesModel class header

```
class btNodeTypesModel : public QAbstractItemModel
{
Q_OBJECT
public:
    btNodeTypesModel(btBrain *brain, QObject *parent = 0);
    ~btNodeTypesModel();

    QMimeData* mimeData(const QModelIndexList &indexes) const;
    QStringList mimeTypes() const;
    Qt::DropActions supportedDropActions() const;

    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const;
    Qt::ItemFlags flags(const QModelIndex &index) const;
    QVariant headerData(int section, Qt::Orientation orientation, int role = Qt::DisplayRole)
        const;

    QModelIndex index(int row, int column, const QModelIndex &parent) const;
    QModelIndex parent(const QModelIndex &child) const;

    int rowCount(const QModelIndex &index) const;
    int columnCount(const QModelIndex &index) const;

    bool removeRows(int row, int count, const QModelIndex &index = QModelIndex() );
    bool insertRows(int row, int count, const QModelIndex &parent);

    btNodeTypesModelNode* nodeFromIndex(const QModelIndex &index) const;

public Q_SLOTS:
    void newBehaviorTreeTypeAdded(btEditorNodeType* newType);

private:
    btNodeTypesModelNode* rootNode;

33     btNodeTypesModelNode *nodeAction;
34     btNodeTypesModelNode *nodeCondition;
35     btNodeTypesModelNode *nodeComposite;
36     btNodeTypesModelNode *nodeDecorator;
37     btNodeTypesModelNode *nodeReference;
38     btBrain *m_brain;
39 };
```

Lines 8, 9 and 10 are used by the Qt QAbstractItemView classes to allow drag'n'drop interaction with the model. Lines 12 through 20 are similarly used by the views for gathering information on what to show when and where. Lines 22 and 23 are used for editing data in the model. Line 25 is a convenience function allowing for simple conversion of the QModelIndex instances used by the model into btNodeTypesModelNode which is the actual data structure found behind the model itself. Finally, line 28 is the slot connected to by the editor, and which is called whenever a new node type is added. This slot processes the data and adds a corresponding item to the tree.

The private section contains the actual data of the model. `rootNode` contains a pointer to the hidden root of the model. This means that while the view seems to have multiple roots (see the left side of Figure 6.2 on page 37) it in fact has only one root, except that this root is not exposed in the view.

Lines 33 through 37 contain five variables used in the `newBehaviorTreeTypeAdded` slot. They are created when the model is instantiated, and then used to create the model itself. The reason they are then stored and not simply reused is for speed - they are essentially a cache. See Listing 6.9 for a short rundown of the function.

Listing 6.9: The btNodeTypesModel::newBehaviorTreeTypeAdded slot

```
void btNodeTypesModel::newBehaviorTreeTypeAdded(btEditorNodeType* newType)
{
    btNodeTypesModelNode *node;
    QModelIndex parent;
    switch(newType->type())
    {
        case btNodeType::ActionNodeType:
            node = new btNodeTypesModelNode(newType, nodeAction);
            parent = createIndex(nodeAction->row(), 0, node);
            break;
/* [a number of similar cases for each other node type] */
        default:
            break;
    }
    node->setName(newType->name());
    emit dataChanged(parent, parent);
}
```

btTreeModel

The tree model contains a model which describes a complete behavior tree. The model model outline is similar to the one above, there are vital differences. Listing 6.10 has some of the contents removed which is similar to the `btNodeTypesModel` class.

Listing 6.10: The btTreeModel class header

```
class btTreeModel : public QAbstractItemModel
{
    Q_OBJECT
    Q_PROPERTY(QString name READ name WRITE setName)
    Q_PROPERTY(QString description READ description WRITE setDescription)

...

public Q_SLOTS:
    void updateTreeModel();

Q_SIGNALS:
    void addRemoveBTNode();

...
};
```

While much of the outline is similar, a few differences are apparent: The slot in the other class is not present (as it is not interesting) while a new signal and a new slot are added. The `updateTreeModel` simply forces the dataChanged signal to be emitted when the tree model is changed. This ensures data sanity in the attached view when the model is changed. The `addRemoveBTNode` signal is emitted whenever a node is changed, and subscribed to by the property view. This again ensures that the property view is able to gracefully handle changes in the model.

6.2.2. The Available Nodes and Editor Views

The implementation of these two views is extremely simple as it is really a matter of using a QTreeView to show the contents of the models. There are, however, a few signals caught in the btEditor class which handles some of the interaction. These are relatively trivial and shall not be explained here. In stead, Listing 6.11 shows the function called when the current behavior tree is changed.

Listing 6.11: The showBehaviorTree(btTreeModel*) function

```
void bteditor::showBehaviorTree(btTreeModel* showThis)
{
  disconnect(this, SLOT(editorSelectionChanged(QItemSelection,QItemSelection)));
  this->btEditor->setModel(showThis);
  this->btEditor->setSelectionModel(new QItemSelectionModel(showThis));
  connect(
     this->btEditor->selectionModel(), SIGNAL(selectionChanged(QItemSelection,QItemSelection)),
     this, SLOT(editorSelectionChanged(QItemSelection,QItemSelection))
     );
  this->currentBTNameLabel->setText(showThis->name());

  if(m_currentBehaviorTree)
  {
    disconnect(m_currentBehaviorTree, SIGNAL(addRemoveBTNode()), propertyWidget, SLOT(
        dragDropUpdate()));
    disconnect(propertyWidget, SIGNAL(treeModelUpdate()), m_currentBehaviorTree, SLOT(
        updateTreeModel()));
    disconnect(m_currentBehaviorTree, SIGNAL(dataChanged(const QModelIndex&, const QModelIndex&))
        , this, SLOT(updateView(const QModelIndex&, const QModelIndex&)));
  }
  m_currentBehaviorTree = showThis; // keep track of behaviortree
  connect(m_currentBehaviorTree, SIGNAL(addRemoveBTNode()), propertyWidget, SLOT(dragDropUpdate()
      ));
  connect(propertyWidget, SIGNAL(treeModelUpdate()), m_currentBehaviorTree, SLOT(updateTreeModel
      ()));
  connect(m_currentBehaviorTree, SIGNAL(dataChanged(const QModelIndex&, const QModelIndex&)),
      this, SLOT(updateView(const QModelIndex&, const QModelIndex&)));
}
```

The function disconnects from any existing signals (to avoid connections to potentially non-existant listeners), and then sets model and selection model for the view. Importantly it then connects the `selectionChanged` signal on the selection model to the approprite slot on the class. This slot simply gets the selected node out of the editor's selection model and passes it on to the Properties View, which then handles everything from there. The second half of the function connects signals between the model, the Properties View and the editor window, to make sure that everything is kept up to date when any parts of the data changes.

6.2.3. Properties View

The final part of the editor which shall be presented is the Properties View, as seen on the right side of the screenshot in Figure 6.2 on page 37. It consists of a set of classes, to which the entrypoint is `btPropertyWidget`, the outline of which can be seen in Listing 6.12.

Listing 6.12: The Property Widget Class

```
class btPropertyWidget : public QWidget
{
    Q_OBJECT

    public:
        btPropertyWidget(QObject * parent = 0);
        ~btPropertyWidget();

        btEditorNode * node() const;
        void setNode(btEditorNode * theNode);

    public Q_SLOTS:
        void dragDropUpdate();
        void updateTreeModel();

    Q_SIGNALS:
        void treeModelUpdate();

    private:
        btEditorNode * m_node;
        ColorGen * colorgen;

        QWidget * createComponentPropertyView();

        void setupPropertyView();
        void appendToPropertyView(QGridLayout *layout, qint32 & row, QObject * object, QString
            name, QString description, bool enabled, QVariant options = 0);
        void appendObjectToPropertyView(QGridLayout * layout, qint32 &row, btEditorNode * node,
            bool enabled);
        void appendComponentToPropertyView(QGridLayout *layout, qint32 &row, btEditorNodeType *
            node, bool enabled);
        void appendMetaObjectToPropertyView(QGridLayout * layout, qint32 &row, QObject * object,
            bool enabled);
        QString getPropertyDescription(QObject *object, QString propertyName);
};
```

The widget is designed in such a manner that it is used by creating the widget once, and then setting the node on it using the `setNode(btEditorNode*)` function. This function sets the `m_node` variable and launches `setupPropertyView()`, which in turn performs the actions seen in Algorithm 2. Note how the node passed is a pointer to a `btEditorNode` instance. This class is used internally the editor for persistence reasons (it includes functionality not required in the library), but it implements the `btNode` class, which is found in the shared header files the library also uses. The shared class contains the vital aspects of the behavior tree structure (the parent/children relationship and the main properties), so that it is the same in both editor and library.

Algorithm 2 `btPropertyWidget::setupPropertyView()`

```
Append self to property view
Append decorators to property view
Append the node's node type to property view
Wrap everything in a set of layouts and adjust margins
```

Appending an object to the property view happens using one of the `append*(...)` functions seen in the listing. These functions follow a similar pattern, simply adding a header and then adding it's meta object to the view. Algorithm 3 shows simplified outline of `appendMetaObjectToPropertyView(...)`.

Algorithm 3 `btPropertyWidget::appendMetaObjectToPropertyView(...)`

```
QMetaObject metaobject ← the passed object's metaobject
for all MetaProperty property in metaobject.properties do
  if property.name $\neq$ "objectName" AND property.name $\neq$ "stopFlag" then
    appendToPropertyView(property.name, property.description)
for all MetaProperty property in object.dynamicProperties do
  if property.name $\neq$ "objectName" AND property.name $\neq$ "stopFlag" then
    appendToPropertyView(property.name, property.description)
```

It can be seen how this calls the `appendToPropertyView(...)` function. This function very simply appends a row to the view, with a label for the name and description to the left, and to the right an instance of `btPropertyViewItem`, which is the class which takes care of informing objects of changes to the property they represent, as well as show an appropriate editor widget for the type of contents in that property. Listing 6.13 shows the header of the `btPropertyWidgetItem` class.

Listing 6.13: The btPropertyWidgetItem class header

```
class btPropertyWidgetItem : public QWidget
{
Q_OBJECT;

public:
    btPropertyWidgetItem(QObject * parent = 0, Qt::WindowFlags f = 0);
    ~btPropertyWidgetItem();

    void setEditObject(QObject * editThis);
    void setEditProperty(QString propertyName, bool readOnly);

Q_SIGNALS:
    void sendUpdateTreeModel();


private Q_SLOTS:
    void propertyChanged(QVariant value);
    void propertyChanged(int value);
    void propertyChanged(double value);
    void propertyChanged(QString value);

    void QVariantListItemRemoved(QListWidgetItem * item, int index);
    void QVariantListItemAdded(QListWidgetItem * item);
    void QVariantListItemChanged(QListWidgetItem * item, int index);

private:
    void setupPropertyWidget(bool readOnly);

    QObject * editedObject;
    QString propertyName;

    QWidget *createLineEdit (QVariant value, bool enabled);
    QWidget *createSpinBox (QVariant value, bool enabled);
    QWidget *createDoubleSpinBox (QVariant value, bool enabled);
    QWidget *createList(QVariant value, bool enabled);
    QWidget *createChildProbabilitiesList(QString propertyName ,bool enabled);

    QWidget * editWidget;

    const QString getPropertyType(QString propertyName);
};
```

The main functionality of these can be found in first the `setEditProperty(QString,bool)` function, which creates the editor widgets using one of the `create*(QVariant,bool)`

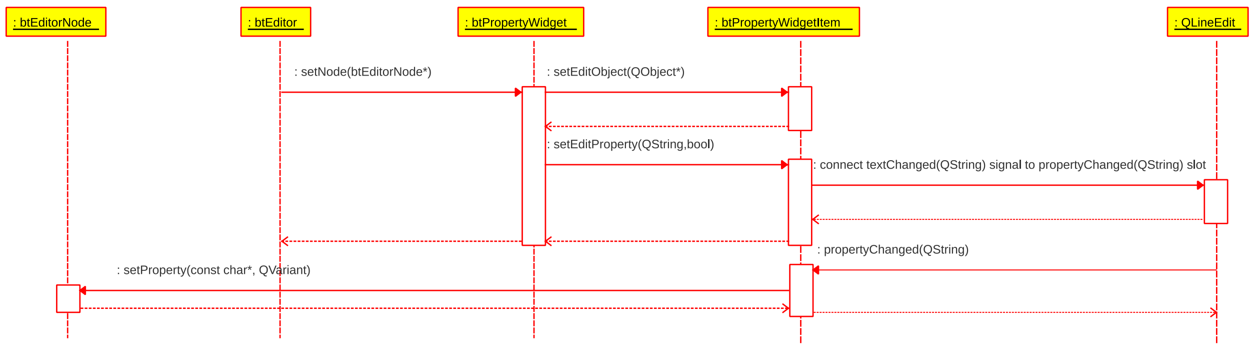

: btEditorNode
: btEditor
: btPropertyWidget
: btPropertyWidgetItem
: QLineEdit
: setNode(btEditorNode*)
: setEditObject(QObject*)
: setEditProperty(QString,bool)
: connect textChanged(QString) signal to propertyChanged(QString) slot
: propertyChanged(QString)
: setProperty(const char*, QVariant)

Figure 6.3.: Sequence diagram describing how the creation and signalling functions in the `btPropertyWidget` system

functions as well as sets up the correct signal/slot associations, and secondly in the set of slots, which take care of handling the actual changes in the editor widget's data. Each function is trivial, but the interaction is shown in the sequence diagram in Figure 6.3, in which a simple string property is edited.

6.3. SMARTS Test-Bed

This section describes the implementation of the SMARTS Test-Bed from section 5.3 on page 28. It first describes how the game has been implemented followed by how the we implemented the parallels from section 2.1 on page 8. Last, it is described how we implemented the calculations and containers defined by Yannakakis and Hallam[21].

6.3.1. The `Game` Class

The `Game` class represent the entire game and takes care of all the map, all the items on the, guards and players. It contain the map, represented by an two dimensional array of `GameItems` which are the general form of item in the game. The `Game` class is built on top of the `QGraphicsScene` class and thus handles the whole game as a scene.

When the `Game` is instantiated it creates the level together with loading the behavior tree from the XML file using a `btBrain`. Furthermore the `Game` instantiates a `Player` together with a number of `Guard` instances, where the `Player` equals Pac-Man and the `Guard` equals a ghost. Both these classes inherits from `Agent` which is the base class for any moving `GameItems`. `Game` also creates an instance of a `ScenarioSet` class which keeps track of all the test data collect during the current game. A `ScenarioSet` exist for every game.

Furthermore every `GameItem` that is capable of moving is also wrapped into two other classes, where `Guard` and `Player` is put inside a `Enemy` and the `Enemy` is put into a `Runner`. The reason for this is a threading problem would occur with the current design of the SMARTS Test-Bed, where a moveable `GameItem` would not be able to send signals or receive any signals from other classes.

Whenever a game ends two functions are used for resetting the game. `Game::reset()` requests every `Runner` instance in the game to stop their execution, this occurs every time the `Player` lost a game. When every `Runner` has stopped, the `Game` resets the game for a new run, by placing every `Player` and `Guard` instances to the initial position. Also the state of each `Player` and `Guard` is reset but the behavior tree will keep any

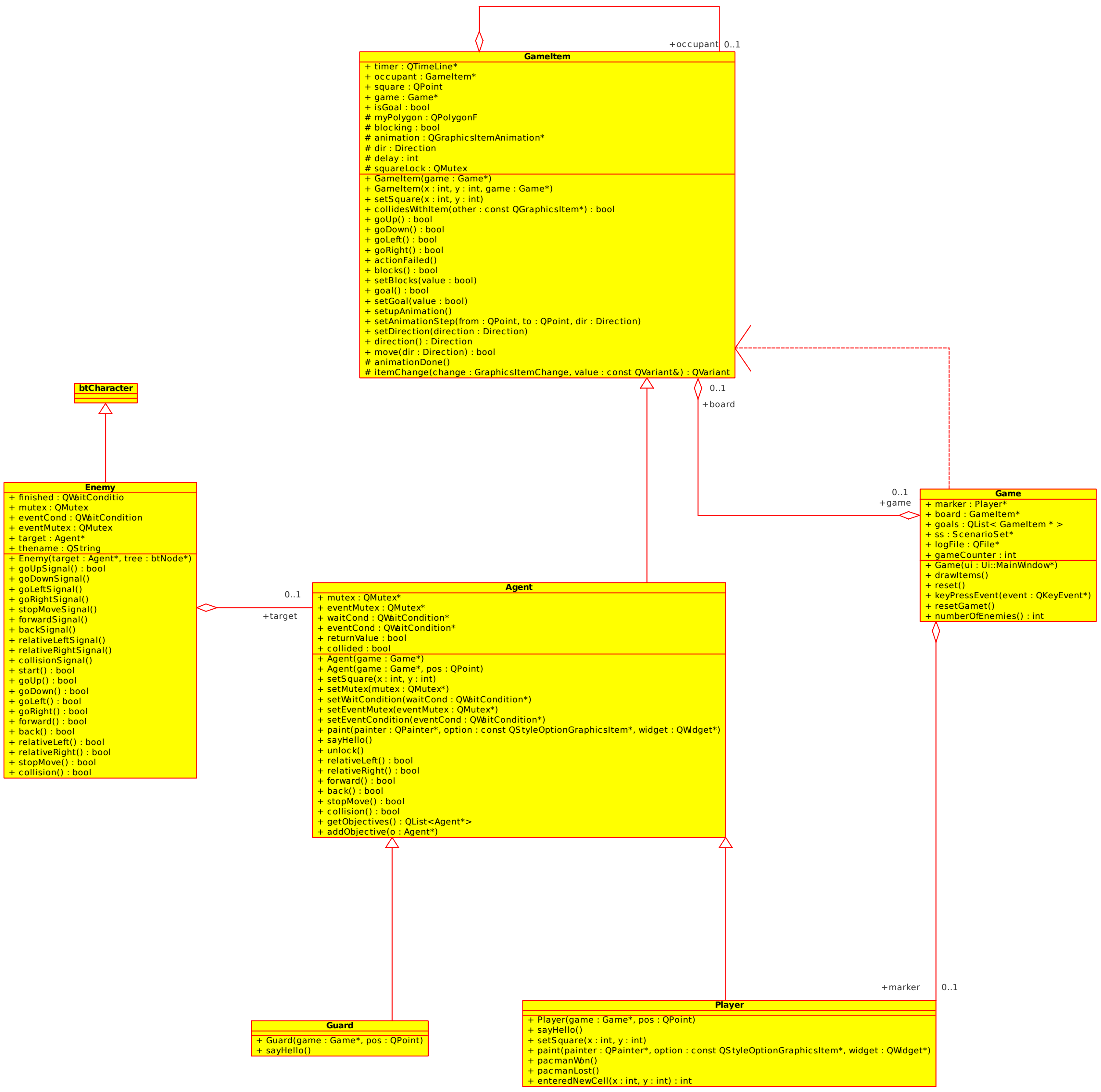


Figure 6.4.: Implemented Test-Bed structure

information learned when a new game starts. Note that the behavior trees lose their information if the SMARTS Test-Bed is closed.

6.3.2. The `GameItem` Class

The base class for creating items in the game is the `GameItem` inherits from `QGraphicsPolygonIt`
This is done for simplicity as the item can be defined as a simple polygon without having a explicit paint routine. Also the inheritance from `QGraphicsPolygonItem` means that it can be added to `Game` class directly and be drawn on the scene.

`GameItem` contains all the functionality for moving around on the level, which means that this class handles the animations between squares and the position of the `GameItem` in level together with collision detection on the level. Furthermore the `GameItem` also contains if the current instance is solid, which means that it blocks any movement through it in the level.

Listing 6.14 shows the `goUp` of `GameItem`. The function first checks if it is possible to move upwards, if this is the case the `GameItem` starts the animation for moving upwards and returns true when the animation is finished. If the check fails `goUp` emits `actionFailed` and returns false. There exist functions for left, right and down but these are similar to the `goUp`. These functions are only used when the `GameItem` is movable.

Listing 6.14: `goUp` of `GameItem`

```
bool GameItem::goUp()
{
    //qDebug("going up");
    if(move(Up)){
        setAnimationStep(square,QPoint(square.x(),square.y() + 1), Up);
        return true;
    }
    qDebug("action up failed");
    emit actionFailed();
    return false;
}
```

If the `GameItem` is stationary it is possible to set if the `GameItem` is a goal or is blocking, by setting a bool through `setBlocks` and `setGoal`

6.3.3. Interfacing With Behavior Trees

Since the behavor trees controlled by SMARTS Library are separate entities to the games which use them to make decisions, a method must be created by which they are allowed to interact with the game and vice versa. This section describes the implementation of the code which allows for this interaction.

The `Runner` Class

This class is used for running `Enemy` together with its behavior tree in a separate thread than in the GUI thread. This allows for the behavior tree to be paused at defined points in its execution, which is important as the behavior tree should be able to start an action, pause and wait until the action is complete and then resume.

When the `Runner` is started it will run the code in listing 6.15

Listing 6.15: Run function of `Runner`

```
void Runner::run()
{
    qDebug("runner");
```

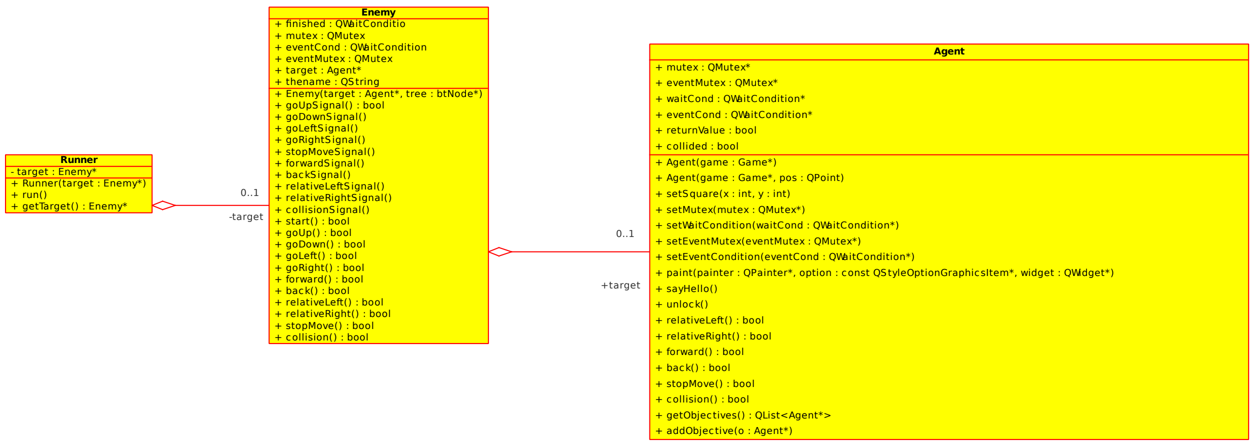


Figure 6.5.: Running the behavior trees

```
    qsrand(QDateTime::currentDateTime().toTime_t() - reinterpret_cast<quint64>(this));
    this->target->start();
}
```

By calling start on `Runner`'s target, which is an instance of `Enemy`, the execution of the behavior tree starts. When the behavior tree finishes execution and `run` returns, the `Runner` sends a signal to the `Game`, described in section 6.3.1 on page 43, which in turn starts the `Runner` again.

The `Enemy` Class

The `Enemy` class provides the link between the behavior tree and the game. This class came out of technical issues, we needed the behavior tree to have a character to work on such that it could control said character in the game. This have the issue that we wanted to use our `Player` and `Guard` classes but could not, due to a technical problem regarding multiple inheritance: It is not possible to create a class which inherits from two classes which are both QObject subclasses themselves.

Thus this class was created, of which an instance exists in each of the `Player` and `Guard` classes.

Another issue is that one thread cannot call a method in another thread which uses timers. It would have to use signals and slots to do this, or the QMetaObject method invocation system, both of which are advanced techniques. Consequently, this class functions as that glue layer.

The`Agent` Class

The `Agent` class functions as glue between `Enemy` and `GameItem` instances. It contains most actions that can be performed by any agent in the game, and takes care of both unlocking the threading system, as well as the animations providing the continuous motion of any of the characters in the game.

The `Guard` and `Player` Classes To allow a difference between the computer controlled guards and the potentially human controlled player, two separate subclasses are created for these. This allows for e.g. difference in win and lose conditions.

The Player class handles a collision with an other item differently than a guard does, and it "eats" pellets, while the guard does not. The only thing additional thing the guard does in the current implementation is to set a different Z-index, this is used to differentiate between the items.

6.3.4. Behaviors

The behaviors used in the test bed are very simple constructs, in that they generally only call a function on a btCharacter subclass, in this case the Enemy instance to which the behavior tree is attached. Below are described shortly the two main implementation cases of simple actions and conditions.

Simple Action

Simple actions are in principle any piece of code one wishes to run. Some of the classes in the game which represent a simple action are goUpNode, moveCloserLeftRight, stopMove, relativeRight and a good few more. Listing 6.16 shows the code for one such.

Listing 6.16: The implementation of the goDownNode class

```
#include "go_down.h"
#include "enemy.h"

#include <QDebug>
#include <QThread>

REGISTER_NODETYPE(goDownNode)

goDownNode::goDownNode()
{
}

bool goDownNode::run(btCharacter *self)
{
    qDebug() << "down " << qobject_cast<Enemy*>(self)->name();
    return qobject_cast<Enemy*>(self)->goDown();
}

#include "go_down.moc"
```

As visible here, the only thing the node does is to call the appropriate function on the Enemy instance.

Condition

A condition is equivalent to a c++ const operation: it performs a check, but makes no changes to the underlying data structures. In our case what this means is that the conditions call functions on the Enemy instance which fulfill this criterion, without necessarily being const operations. Listing 6.17 shows the run(btCharacter*) function of the isCollided condition node type.

Listing 6.17: The run function on isCollided

```
bool isCollided::run(btCharacter *self)
{
    while(self->continueThinking())
    {
        qobject_cast<Enemy*>(self)->eventMutex.lock();

        qobject_cast<Enemy*>(self)->eventCond.wait(&qobject_cast<Enemy*>(self)->eventMutex,
            30000);
```

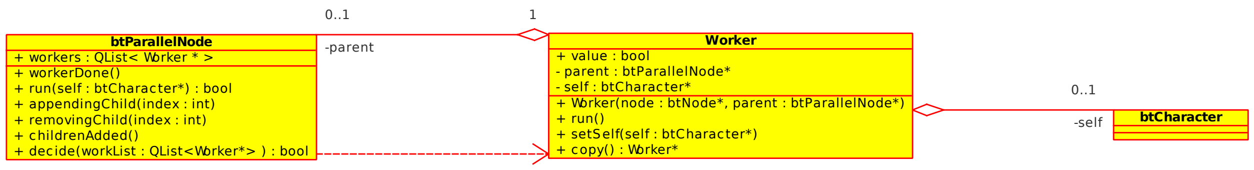

btParallelNode
+ workers : QList< Worker * >
+ workerDone()
+ run(self : btCharacter*) : bool
+ appendingChild(index : int)
+ removingChild(index : int)
+ childrenAdded()
+ decide(workList : QList<Worker*>) : bool
0..1
1
-parent
Worker
+ value : bool
- parent : btParallelNode*
- self : btCharacter*
+ Worker(node : btNode*, parent : btParallelNode*)
+ run()
+ setSelf(self : btCharacter*)
+ copy() : Worker*
0..1
-self
btCharacter

Figure 6.6.: Parallel Execution

```
        if(qobject_cast<Enemy*>(self)->collision() == true)
        {
            qobject_cast<Enemy*>(self)->eventMutex.unlock();
            return true;
        }

        qobject_cast<Enemy*>(self)->eventMutex.unlock();
    }
    return false;
}
```

6.3.5. Parallel Execution

One of the trickier in terms of implementation is the parallel execution of behaviors.

The `Worker` Class

To run behaviors we use threads, these threads are instances of the `Worker` class and are independent from the structure of the behavior tree. The implementation is created such that the use of `Worker` threads are transparent to both the parent of the behavior and the sub trees run in each `Worker`. This allows any sub tree to be run in parallel.

When a new `Worker` is instantiated the behavior subtree, which has to be executed, is passed to the `Worker`. Then the `Worker` instance is started and begins to run the behavior subtree. The main issue with running behaviors as threads in parallels is termination, because the termination of threads has too be done, at a point during execution where it is safe to terminate. To achieve this each `btNodeType` has a `stopFlag`.

An example of this is, if a parallel has a `Worker`, which contains a condition. When the condition is met, the behavior subtrees `Worker` will cycle through the other `Worker` instances of the parallel set their flags, such that they stop their execution. The same happens if the `btCharacter` is asked to stop thinking, but instead of only the parallel's `Worker` instances stops execution, the whole behavior tree will stop execution, by preventing the `btNode` running its `btNodeType`.

6.3.6. Measuring Performance

For measuring performance of the behavior trees we use the metrics defined by Yannakakis and Hallam[21]. The evaluation is implemented such that we have a information container foreach game played, which is the `Scenario` class. Whenever the game resets a `Scenario` instances is created and used for the new game, while the old `Scenario` instance is saved. The `Scenario` class gather all the informations described in 5.3 on page 28, which are used for calculating the metrics.

For storing each `Scenario` instance the class `ScenarioSet` is used. Furthermore the `ScenarioSet` class is capable of calculating the metrics, e.g the behavioral diversity, from the already collected `Scenario` instances together with support for changing the

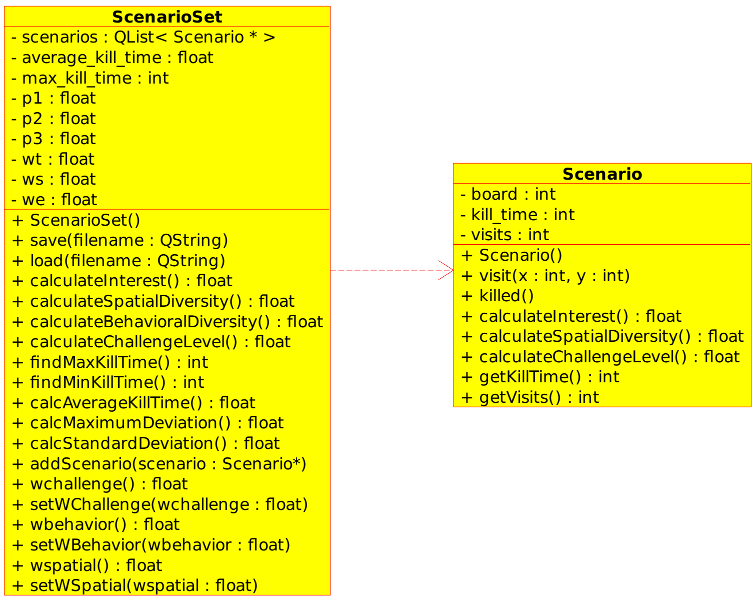

ScenarioSet
- scenarios : QList< Scenario * >
- average_kill_time : float
- max_kill_time : int
- p1 : float
- p2 : float
- p3 : float
- wt : float
- ws : float
- we : float
+ ScenarioSet()
+ save(filename : QString)
+ load(filename : QString)
+ calculateInterest() : float
+ calculateSpatialDiversity() : float
+ calculateBehavioralDiversity() : float
+ calculateChallengeLevel() : float
+ findMaxKillTime() : int
+ findMinKillTime() : int
+ calcAverageKillTime() : float
+ calcMaximumDeviation() : float
+ calcStandardDeviation() : float
+ addScenario(scenario : Scenario*)
+ wchallenge() : float
+ setWChallenge(wchallenge : float)
+ wbehavior() : float
+ setWBehavior(wbehavior : float)
+ wspatial() : float
+ setWSpatial(wspatial : float)
Scenario
- board : int
- kill_time : int
- visits : int
+ Scenario()
+ visit(x : int, y : int)
+ killed()
+ calculateInterest() : float
+ calculateSpatialDiversity() : float
+ calculateChallengeLevel() : float
+ getKillTime() : int
+ getVisits() : int

Figure 6.7.: Measuring behavior tree performance

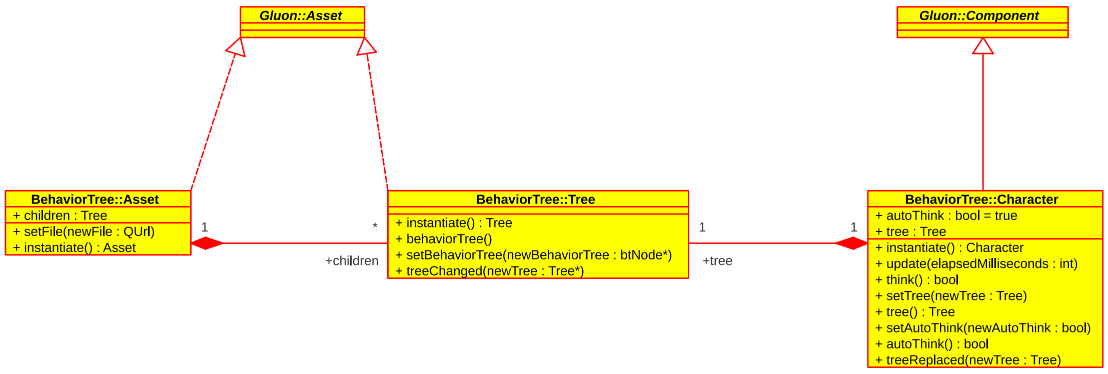

Gluon::Asset
Gluon::Component
BehaviorTree::Asset
+ children : Tree
+ setFile(newFile : QUrl)
+ instantiate() : Asset
1
*
+children
BehaviorTree::Tree
+ instantiate() : Tree
+ behaviorTree()
+ setBehaviorTree(newBehaviorTree : btNode*)
+ treeChanged(newTree : Tree*)
1
1
+tree
BehaviorTree::Character
+ autoThink : bool = true
+ tree : Tree
+ instantiate() : Character
+ update(elapsedMilliseconds : int)
+ think() : bool
+ setTree(newTree : Tree)
+ tree() : Tree
+ setAutoThink(newAutoThink : bool)
+ autoThink() : bool
+ treeReplaced(newTree : Tree)

Figure 6.8.: The class diagram for the Gluon plugins

weights of the metrics if necessary. Whenever a game ends, the calculations are outputted to a file, such that they can be used for plotting graphs for being able to make any conclusions on the test results.

A instance of the `ScenarioSet` is placed on the `Game` class, where the `Game` insures that a new `Scenario` instance is created a new game. while storing the old `Scenario` instances into the `ScenarioSet`. It is also the task of `Game` class to output the data and calculated results into a file.

6.4. Gluon Plugins

The use of a Brain in Gluon is done through a combination of Gluon Component and Asset plugins. Specifically, one Component and two Assets, all three found inside the BehaviorTree namespace. The following section describes the three in detail. Gluon plugins make heavy use of the many extensions found in the Qt framework. Specifically properties and the signal/slot mechanism are both used throughout. The class diagram in Figure 6.8 shows the classes and their inheritance from Gluon. Note that the Gluon classes are not filled out here, as the full class hierarchy for Gluon can be seen in Figure

4.2 on page 21 and the repetition is not relevant.

6.4.1. The `BehaviorTree::Tree` Class

The `BehaviorTree::Tree` class represents a single behavior tree, in reality a very thin wrapper class around the `btNode` class (see Section 5.1). The following listing shows the outline of the class by way of the contents of the header file seen in Listing 6.18.

Listing 6.18: The BehaviorTree::Tree class header

```
class BTCOMPONENT_EXPORT Tree : public Gluon::Asset
{
    Q_OBJECT
    Q_INTERFACES(Gluon::Asset)

    public:
        friend class BehaviorTree::Asset;

        Tree(QObject * parent = 0);
        ~Tree();

        virtual Tree * instantiate();

        void setBehaviorTree(btNode* newBehaviorTree);
        btNode* behaviorTree() const;

    Q_SIGNALS:
        void treeChanged(Tree* newTree);

    private:
        TreePrivate* d;
};
```

The class extends the `Gluon::Asset` class, which Qt is further informed of by use of the `Q_INTERFACES` macro. This ensures that Gluon is able to identify objects created from the plugin correctly. The `instantiate()` function is required for all plugins to return the proper type of object in the factory inside Gluon. The TreePrivate pointer named d at the end is an implementation of the pimpl design pattern.

The three most important parts of the implementation of this class are the functions `behaviorTree()` and `setBehaviorTree(btNode*)` and the `treeChanged(Tree*)` signal. They allow the `BehaviorTree::Asset` and `BehaviorTree::Character` classes to interact with eachother in an abstract manner. This shall be explained further in the next sections.

6.4.2. The `BehaviorTree::Asset` Class

The `BehaviorTree::Asset` class represents the Brain file in Gluon. It will load the file from disk on Gluon's request and automatically create a set of sub-assets, which are instances of the `Tree` class above. Listing 6.19 shows the contents of the header file.

Listing 6.19: The BehaviorTree::Asset class header

```
class BTCOMPONENT_EXPORT Asset : public Gluon::Asset
{
    Q_OBJECT
    Q_INTERFACES(Gluon::Asset)

    public:
        Asset(QObject *parent = 0);
        ~Asset();

        virtual Asset* instantiate();

        virtual void setFile(const QUrl &newFile);

    private:
        AssetPrivate* d;
};
```

The outline of this class is very similar to that of `BehaviorTree::Tree`, but has nothing on top of `Gluon::Asset` except for reimplementing the setFile function. What this function does is what makes it possible for the class to represent a Brain: It loads the brain and automatically creates a number of sub-assets of type `BehaviorTree::Tree`, and sets their behaviorTree accordingly. The `setFile(const QUrl&)` function can be seen in pseudocode form in Algorithm 4.

Algorithm 4 `BehaviorTree::Asset::setFile(const QUrl&)`

```
brainFile ← contents of file represented by const QUrl&
newBrain ← an instance of btBrain created from the contents of brainFile
oldChildren ← the list of old children of this Asset
newChildren ← an empty list
for all btTree tree in newBrain do
    newTree ← new instance of Tree
    add newTree to children of this Asset
    newTree.setBehaviorTree(tree)
    newChildren.append(newTree)
for all children in oldChildren do
    newChild ← an object in newChildren with the same name as oldChildren, or NULL
    emit treeChanged(newChild) signal on oldChild
```

This function is called by Gluon when either of the following events occur: 1) The project is loaded 2) The file on disk changes. Specificall 2) is important here as this is where the second loop becomes important. What this does is that it informs every `BehaviorTree::Character` instance which uses the children what the new object it should use is called.

6.4.3. The `BehaviorTree::Character` Class

The `BehaviorTree::Character` class (the header of which is seen in Listing 6.20) represents one in-game character. This character contains information on the world as viewed by that character. This is where the alpha value is found, allowing the same `Tree` to be used with different `Characters`, and still give different results.

Listing 6.20: The BehaviorTree::Character class header

```
class BTCOMPONENT_EXPORT Character : public Gluon::Component
{
    Q_OBJECT
    Q_INTERFACES(Gluon::Component)
    Q_PROPERTY(bool autoThink READ autoThink WRITE setAutoThink)
    Q_PROPERTY(BehaviorTree::Tree* tree READ tree WRITE setTree)
    Q_PROPERTY(double personality READ personality WRITE setPersonality)

    public:
        Character(QObject * parent = 0);
        Character(const Character &other, QObject * parent = 0);
        ~Character();

        virtual Character* instantiate();
        virtual void update(int elapsedMilliseconds);

        bool think();

        void setTree(Tree* newAsset);
        Tree* tree() const;

        void setAutoThink(bool newAutoThink);
        bool autoThink() const;

        void setPersonality(double newPersonality);
        double personality() const;

    private Q_SLOTS:
        void treeReplaced(Tree* newTree);

    private:
        QSharedDataPointer<CharacterPrivate> d;
};
```

Two important parts here are the `Q_PROPERTY` statements - these allow Gluon Creator to do introspection on the class and expose the values to the user in the property view (see lower right of Figure 4.1 on page 19).

`setTree(Tree*)` will assign the passed instance of `BehaviorTree::Tree` to the `tree` member variable, and connect the `treeChanged(Tree*)` signal on that `Tree` to the `treeReplaced(Tree*)` slot. What this achieves is that as soon as the Brain file changes, and the `Asset` plugin notices this, every `Character` which is using any of the `Tree` sub-assets is notified of the change and allowed to act accordingly.

Similarly the `setPersonality(double)` and `personality()` functions set the personality property's value, which is equivalent to the alpha value in our utility function as described in Section 3.2.

The `update(int)` function is called by Gluon on each update. If `autoThink` is true, this will cause the class to call the `think()` function on each update (practically it will not do this in case the character is already thinking, but this is a technicality and thus not explained further here). The `think()` function can of course also be called directly - this should be done at appropriate times during gameplay, and since this timing is not deterministic, `autoThink` is then turned off and `think()` called manually.

6.4.4. Using The Plugins

To actually use the plugins, a small tool was created, which did nothing but read a Gluon game project (in simplified form for clarity, meaning no external Scene files) and run it. As such, what you see in Listing 6.21 only contains the bare minimum for testing out the behavior tree plugins: A single `BehaviorTree::Asset` named "test-brain", a single `GameObject` representing the scene, containing one `GameObject` named "Some Game Character" representing the in-game character, with one `BehaviorTree::Character` component attached pointing to a `BehaviorTree::Tree` inside test-brain named "New Tree". When run, this GameProject will think once each update.

Listing 6.21: Contents of test.gdl

```
{ GameProject(BTTest)
    description string(A test project for the behavior tree plugins)
    entryPoint Gluon::GameObject(Scene 1)
    { GluonObject(Assets)
        { BehaviorTree::Asset(test-brain)
            file file(test-brain.xml)
        }
    }
    { GameObject(Scene 1)
        { GameObject(Some Game Character)
```

```
            { BehaviorTree::Character(Behavior)
                autoThink bool(true)
                tree BehaviorTree::Tree(Assets.test-brain.New Tree)
            }
        }
    }
}
```

CHAPTER
7

Test

This chapter contains a description of the tests performed using the test bed which was designed and implemented, based on the theories of Yannakakis and Hallam[21], and described in Sections 5.3 on page 28 and 6.3 on page 43. The chapter describes first the scenario, and then the algorithms being tested. Finally it analyses the results found during the test, and discusses them.

7.1. Scenario

In Figure 7.1 you see the level used by the test bed, representing the scenario described in the Game Rules part of Section 5.3 on page 28:

- The white area is space in which movement can happen.
- Fields with a red diamond in are impassable.
- The small blue diamonds represent the goal objectives.
- The player start position is the middle of the lowest row, under the entry through the lowermost wall.
- The guards start in each their corner of the middle square.

The tests are all based on the same rules, the player attempt to capture a the blue diamonds around on the map. If successful it is counted as a win and the game proceeds to the next level. If the player dies the game is over and results are measured.

7.1.1. Limitations

Due to time constraints in the project, the test has not been conducted using human trial participants. However, we are of the opinion that this is not necessary in this case. The reason is that the test method itself has already been deminstrated as functioning

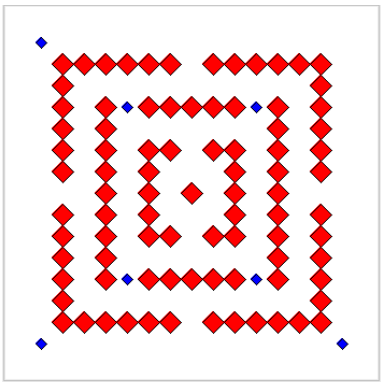


Figure 7.1.: The level used in the scenario

correctly when compared with results from human subject trials. As such, the results we gather here are directly comparable with those found in the tests conducted by Yannakakis and Hallam[21].

The Pac-Man^{TM} character implemented for this test is relatively simple. Further effort could be put into making it more human-like in its behavior, but because we have no structural learning implemented we have decided to create this simpler version and postpone the more advanced version until a later point in time.

Once structural learning is implemented, it is suggested that the test be re-conducted using the learned behavior tree Pac-Man, to asses the quality of the learning based on the results found in this test and the tests in the original paper.

7.1.2. Behavior Trees

To create behaviors similar to that found in Pac-Man^{TM}, a number of behavior trees were created to emulate the behaviors of the guards in the original game, as deciphered by Jamey Pittman[17].

We have for the test recreated the behavior of three of the classic ghosts in Pac-Man^{TM}. There are subtle differences between our implementation and the actual implementation in the game, such as using non-determinism for choosing movement direction instead of being purely deterministic.

The ghosts are implemented as individual behavior trees. We have only implemented three of the four ghosts as the only difference between two of them is the is the target they follow.

The three ghost behaviors are:

Blinky This ghost follows the player directly, always moving closer to the player if it can.

Pinky Is move of an ambusher and will not target the player directly but a point a few steps in front of the player.

Clyde This ghost is more reserved and tries to keep its distance to the player, but still not get to far away.

All the ghosts follow a **Scatter** then **Chase** cycle where the ghost will alternate between chasing the player and scattering to the corners of the map. In the actual Pac-Man^{TM} game the scatter behavior is achieved by setting the ghosts to chase a point outside the map. In our implementation they are more likely to go in the desired direction.

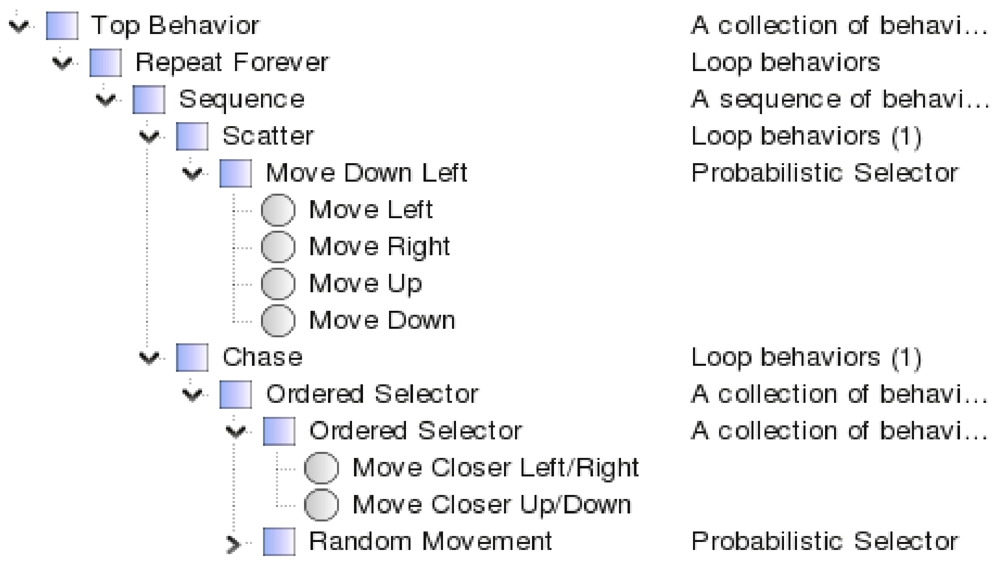
The behaviors of the ghosts are built by combining simple actions such as move a step closer to the player, move up, move forward, move back etc.

All the ghosts have an endless loop at the top that keep running the behavior until the game ends. In this loop is a sequence of behaviors the first one is the **Scatter** behavior and the second is the **Chase** behavior.

They also have essentially the same scatter behavior except the weights are different, e.g. the Blinky scatter behavior is a probabilistic selector with four actions attached. It can chose a move left or a move down action with higher probability than the other move because they have higher weight causing the ghost to move to the bottom-left corner.

Blinky Behavior

The chase behavior of the **Blinky** ghost consist of a prioritized selector with two sub-behaviors. The first one is a probabilistic selector choosing between whether it should



```
Top Behavior                                  A collection of behavi...
  Repeat Forever                              Loop behaviors
    Sequence                                  A sequence of behavi...
      Scatter                                 Loop behaviors (1)
        Move Down Left                        Probabilistic Selector
          Move Left
          Move Right
          Move Up
          Move Down
      Chase                                   Loop behaviors (1)
        Ordered Selector                      A collection of behavi...
          Ordered Selector                    A collection of behavi...
            Move Closer Left/Right
            Move Closer Up/Down
          Random Movement                     Probabilistic Selector
```

Figure 7.2.: Blinky behavior in SMARTS Designer

move closer vertically or horizontally. If one of them fails it tries the other one. If this first subbehavior fails the next selector simply chooses randomly direction to go. This whole chase behavior is repeated 15 times before it shifts to the scatter behavior as the whole behavior is inside a loop.

Clyde Behavior

The **Clyde** ghost has the most complex of the behaviors as it tries to both get close to the player, but also keep its distance. The chase behavior of Clyde consist of a selection between two behaviors: One to move away and another one to move closer.

The move away behavior is a sequence starting with a condition on whether it is close to the player, if it is it probabilistically selects to whether to move away or move some random direction with higher weight on moving away. If the condition had failed the sequence breaks and it tries the move closer behavior.

The move closer behavior work the same except its condition check whether it is not close and moves closer instead of further away. This second condition is the same as the first except with an inverting decorator.

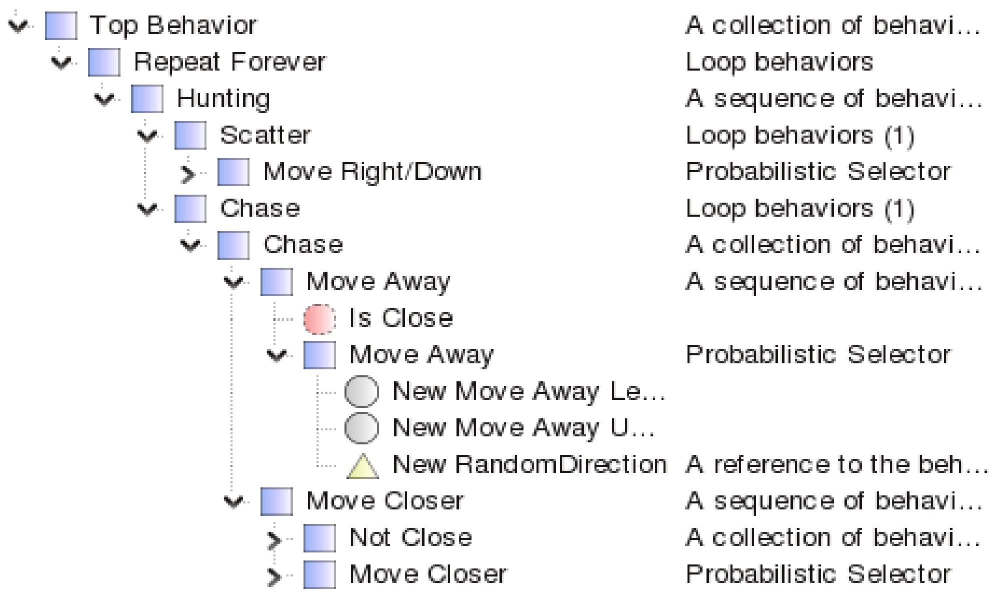
Pinky Behavior

The **Pinky** behavior is identical to the Blinky behavior except it uses an other move closer action that will target a location on the map a few steps ahead of the player in the players movement direction instead of targeting directly. This leads to an ambush behavior where the ghost moves ahead of the player.

7.2. Conducting the Test

The test is conducted by running each ghost independently against the Pac-Man$^{\text{TM}}$, which tries to reach each of the goal objectives with a probability of moving in a random direction.

Blinky will be tested as follow:



```
Top Behavior                                  A collection of behavi...
  Repeat Forever                              Loop behaviors
    Hunting                                   A sequence of behavi...
      Scatter                                 Loop behaviors (1)
        Move Right/Down                       Probabilistic Selector
      Chase                                   Loop behaviors (1)
        Chase                                 A collection of behavi...
          Move Away                           A sequence of behavi...
            Is Close
            Move Away                         Probabilistic Selector
              New Move Away Le...
              New Move Away U...
              New RandomDirection             A reference to the beh...
          Move Closer                         A sequence of behavi...
            Not Close                         A collection of behavi...
            Move Closer                       Probabilistic Selector
```

Figure 7.3.: Clyde behavior in SMARTS Designer

- With no learning and with scatter
- With no learning and no scatter
- With learning and with scatter

To be able to detect any changes of **Blinky**'s behavior, because **Blinky** will always chase Pac-Man^{TM}.

Clyde will be tested as follows:

- With no learning and with scatter
- With learning and with scatter

Clyde will always keep a little distance to Pac-Man^{TM}, and therefore **Clyde** is not tested with no learning and no scatter.

Pinky will be tested as follows:

- With no learning and with scatter]

Because **Pinky** resembles **Blinky**, **Pink** is only test with the default behavior, because we believe that testing will yield almost identical results.

Furthermore, a purely random ghost will also be tested to gather data for use in a comparison with ghosts that have objectives.

Each setup of the ghosts is run in one session and with approximately 80 games to collect enough data for comparison and evaluation. Furthermore the weights for each diversity is set to 0.33.

7.3. Results

Table 7.3 on the following page shows converging results from the tests, and the results will be elaborated for each ghost and in comparison of the different ghosts.

Character	**Learning**	**Scatter**	**Interest**	**Spat. Div.**	**Behav. Div.**	**Challenge**
Blinky	no	yes	$0,671808$	$0,896868$	$0,24228$	$0,876275$
	no	no	$0,695875$	$0,885844$	$0,391025$	$0,810756$
	yes	yes	$0,65127$	$0,804475$	$0,317629$	$0,831707$
Clyde	no	yes	$0,686703$	$0,860894$	$0,349942$	$0,849273$
	yes	yes	$0,663075$	$0,763156$	$0,433837$	$0,792233$
Pinky	no	yes	$0,605758$	$0,744861$	$0,335173$	$0,73724$
Pure random	n/a	n/a	$0,679769$	$0,807552$	$0,407001$	$0,824755$

Table 7.1.: The converged diversities, challenge, and interest for each test

If we starting at looking at the tests for **Blinky** we can see that **Blinky** has the highest challenge and spatial diversity from all the tests at almost 0.9. On the other hand the behavioral diversity is the lowest, which indicates that the default **Blinky** is the most predictable version of the ghosts with a mid range interest level. From the graph in Figure A.2 on page 67 it is clear to see that both challenge and spatial diversity have converged while both the interest and behavior diversity keeps getting smaller for every game.

If we remove the **Scatter** from **Blinky** then both interest and behavioral diversity increases, spatial diversity is almost the same, but challenge decreases. The reason for this is that **Blinky** becomes more and more random, thereby increasing the behavioral diversity and interest but does not kill that often, which decreases the challenge. The graph in Figure A.4 on page 68 shows that the only metric that has not converged completely is the behavioral diversity.

Adding both **Scatter** and learning to **Blinky** causes both the behavioral diversity and challenge to converge between the two previous results. Both the spatial diversity and interest decreases with this type of **Blinky**. The graph of the learning **Blinky**, in Figure A.6 on page 69, shows that behavioral diversity has almost converged while the others have completely converged.

From the three versions of **Blinky** it can be seen that the default version is the most challenging and predictable, and will be suitable for very challenging games, where it does not mater how behaviorally diverse it has to be. A **Blinky** without scatter is the least challenging but is the most behaviorally diverse and interesting, and suitable for games where behavioral diversity and interest have the highest priority. The learning **Blinky** is placed in the middle regarding behavioral diversity and challenge, but it is the least interesting and spatially diverse. But if you are looking for something in between the two other versions in terms of behavioral diversity and challenge then a learning **Blinky** would be the choice.

The default version of **Clyde**, even though the behavior is more "reserved" has almost the same metric values as a default version of **Blinky**. The only major difference is that **Clyde** is more behaviorally diverse than **Blinky**. Furthermore the default version of **Clyde** has metrics that all are more or less converged, as can be seen on graph in Figure A.8 on page 70.

With the learning version of **Clyde** we see the same tendency as with the learning **Blinky**, where behavioral diversity is higher, so much so that it is in fact the highest in all our tests, and the rest of the metrics are smaller than the default **Clyde**.

As with **Blinky**, a default version of **CLyde** is suitable for the most challenging games. The learning **Clyde** would be more suitable for games that require high behavioral diversity, because of the high behavioral diversity the learning **Clyde** has converged to.

We only tested the default version of **Pinky** because of the resemblance with **Blinky**, but the tests show that **Pinky** is more behaviorally diverse because of the ambushing behavior. The rest of the metrics are smaller than **Blinky**. Because of this we should conduct the same tests as with **Blinky** to achieve a clearer picture of the differences. From the graph in Figure A.12 on page 72 we can see that all metrics has converged, except for behavioral diversity which is still declining. It is assumed that if more games had been run, the behavioral diversity would have converged towards the same metric as for a learning **Blinky**.

The random ghost places itself in the mid range with all metrics bar behavioral diversity, which is the second highest of the tests. We believe the reason for this is emergent from the randomness in the ghost and consequently highly unpredictable leading to it being more inclined to achieve metrics in mid range compared to the others. Furthermore the graph in Figure A.14 on page 73 shows that all of the metrics have more or less converged. It is only the behavioral diversity which makes small deviations but that is also almost converged.

The initial idea was to use diversity as the measure for the behavior of a behavior tree: The more diverse behavior it would produce, the better the tree. But even though we are using interest as the measure for evaluating behavior tree there might be better measures that more accurately capture the differences in the behavior of behavior trees.

When studying the graphs depicting the behavior of the behavior trees it is evident that the measures of behavioral diversity and challenge are highly correlated. They seem to cancel each other out as when one goes up the other goes down (the sum of the two values is roughly equal to 1.1 at all times). This suggests that using these measures together might not be as useful as intended. This is also a reason why we want other and better measures for evaluating behavior.

The method also does not differentiate between two different routes in the game covering the same amount of unique locations in the same number of steps. It only captures if each individual run of the game is spatial diverse, not if it is diverse over several games played. As a complete measure the method uses the average spatial value over N games played, alleviate this and capture more spatial differences, one could simply calculate the spatial diversity by using the spatial information from several games instead of only averaging over individual games. Combining spatial diversity with behavioral diversity goes some way to alleviate this problem because this differentiates paths of different length, and could be useful maybe in correlation with other testing techniques.

7.4. Summary

This chapter has described the scenario in which testing on the behavior tree based test bed is conducted, as well describing shortly the method by which the test is conducted. Finally the results are presented, and an analysis of them is performed.

CHAPTER
8

Conclusion

The following chapter is divided into two parts: Discussion and conclusion. The conclusion describes shortly what has been created through the course of the project, and the discussion takes up where the project left off, looking at causes for potential problems, as well as items for further study.

8.1. Conclusion

Through the course of this project was created a set of four concrete products:

SMARTS Library A library for handling and running behavior trees

SMARTS Designer A tool for creating behavior trees without any programming required

Pac-Man A game, which allows for testing behavior trees in a real game situation

Gluon Plugins A preliminary implementation of an integration of SMARTS Library with a game engine

Furthermore was conducted a test which showed that a learning version of the Pac-Man^{TM} ghost Blinky placed itself in the mid range of all the tests, and even though challenge, spatial diversity, and interest is lower than the default Blinky, it is in turn more behaviorally diverse. Furthermore the behavioral diversity converged only for the learning Blinky, which means that the learning version learns a diverse behavior. The tests showed that the metrics of a default Clyde resembles those of a default Blinky, but that it is more behaviorally diverse. For the learning Clyde we got the same results as for the default Blinky and learning Blinky. From this we can say that learning ghosts versus the default ones, in terms of Clyde and Blinky, the learning ghosts are more behaviorally diverse but have smaller values for the rest of the metrics.

In terms of Pinky the test showed the metrics had lower values, except for behavioral diversity. We assumed that Blinky and Pinky would provide almost the same results, but instead we should perform the same kind of tests for Pinky as for Blinky. Lastly the random ghost places its self in the mid range with all metrics bar the behavioral diversity, which is the second highest of the tests. The reason for this is that the random ghost is pure random, and therefore would place itself in the mid range of the metrics compared to the others.

8.2. Discussion

During the course of the project, much effort was put things which were not directly related to the field of machine intelligence, but which we believe is still important aspects of the creation of AI systems for games.

Specifically we have put a lot of weight on the design of the API being such that it is pleasant to work with for anybody implementing the system, and for those maintaining it in the future. This work included direct interaction with the team behind Gluon, to the extent of taking out a full month to work actively on that project. Specifically we worked on the GluonCL library, which is a cross platform input handling library designed for high performance applications, such as games, and the deeper systems of libgluon, the library handling the Gluon object hierarchy described in Section 4.1.1 on page 20.

What this has meant for the project is that the all round structure of the library code is such that it fits together with Gluon in a manner which would not have been possible without this collaboration, and the ready availability of the code to the authors' perusal. It has been of vital importance that the cooperation has gone as well as it has, and indeed still is. It is the hope of the authors that Nokia^{TM}, who sponsored the meeting in Munich, will assist further in the continued development of both Gluon and SMARTS.

A related problem has been the introduction of all three members of the team to an entirely new programming framework, namely Qt. While two had touched on it previously, one through participation in the KDE community, and one through using PyQT, the Python bindings for the framework, actually using it actively for C++ development was a new experience for all three members. However, everybody has taken it in and used the framework, doing their best to suck up knowledge as much as possible.

As a result of this, the project has worked as a platform for learning about multiple subjects, including an alternative OOP framework than those commonly taught at Aalborg University, namely those of Java and C#/.NET, furthering the knowledge of the project members in this area. As it turns out, Qt shows itself as being very powerful, and while certainly not without its faults, it seems to the authors somewhat odd that this framework is not mentioned in any of the courses on the topic. That discussion, however, is not for this report.

It turned out that creating the the graphics and structure of the game was not too difficult as the game is quite simple. But integrating behavior trees in a game in a general way, such that one behavior tree implementation can be used across many games, is somewhat of a challenge. The reason for this is that we run each AI it its own thread, which in turn gave some complications for running the behavior trees. However, this was necessary because we wanted to reuse the behavior trees without having to create an instance for each AI.

Furthermore parallels had to be thread safe and work properly, meaning each sub behavior tree had to be run in a separate thread. This resulted in complications of stopping a parallel when a certain condition has been meet. Together with complications for stopping the behavior trees from executing, proved a source of many frustrations but after heavy testing, this structure worked in the SMARTS Test-Bed.

8.2.1. Future Work

During the course of the project, several items worth further study and work have risen to the surface as particularly interesting. This section describes each of these. Three sections of of further work are suggested for the project. The first and most obvious is structural learning in behavior trees. Less obvious but none the less an important item, is the creation of a perception system. Finally is described the need for further criteria for the test method. Furthermore, the Gluon plugins should be completed, since they are as mentioned in the conclusion only a preliminary implementation.

Structural Learning

The fitness function described by Spronck et al.[20] is based on context specific information, in that case the health of the agents, the health of the team, survival capabilities and the damage done by the agents. As behavior trees should work very generally, we want to define the fitness on arbitrary conditions. We want to base our fitness on whether an arbitrary behavior have succeeded; that is run until some condition is met. It is important that these behaviors have a goal and that achieving this goal can be either success or failure.

The simplest solution is be to weight all successes and all fails the same such that we add one to successes if a behavior succeeds and add one to failures if it fails, which is what we have done in this report. But this does not capture the difference between trying a (computationally) complex behavior to reach a goal and failing, and failing the same goal using a simple behavior. It should be more expensive to fail the complex behavior and should be punished harder. The idea would be to find some measure by which this can be evaluated programmatically.

One way to capture this is to manually assign reward and penalty to behaviors. But we do not want the designer to have to assign arbitrary values to their designed behaviors, so we would want an other measure. Because we have behaviors in a tree structure we can actually say a lot about the complexity of a behavior, and use this knowledge to calculate rewards and penalties.

We propose a method to do structural learning in behavior trees, which is able to emulate the behavior of a player by learning the behavior from information gathered during a game. Throughout a game the following is gathered:

Actions Move to certain area, move away from opponent, move towards opponent, etc.

Game State Is the enemy close by?, far away?, is the health low, etc.

These are processed into sequences of actions combined with how often a sequence occur. This is like using n-grams (described shortly below), but we count sequences per game state (this could be considered part of the action $< moveaway, enemyclose >$).

These sequences are turned into sequences in a behavior tree, the ones with the least support are pruned using a technique from association rules mining[19]. These sequences are now added to a selector and assigned weights based on their support (how many times a sequence is seen). This is clustered by game state context and conditions are added on the game state conditions are added to the behavior tree.

If two or more sequences under the same selector or share $< k >$ actions at the same place in the ordering and if they are approximately equally likely, they are combined into a partial ordering, using an ordered sequence where the shared actions are modelled using a probabilistic ordered sequence.

N-Gram algorithm N-Gram is an algorithm that is used for action prediction and uses a window size for prediction. Depending on the window size, the algorithm looks at a sequence of actions and creates probabilities based on the sequences. These probabilities are then used for predicting the next action[15].

The algorithm depends heavily on the window size when trying to reach a good performance level. Therefore there exists a hierarchical version, which has several N-Gram algorithms working in parallel, each with increasingly large window size. E.g a hierarchical 3-Gram has N-Gram algorithm with window size 1, a N-Gram with window size 2, and a N-Gram with window size 3. When predicting an action it is first looked up in the 3-Gram and if this does not have enough confidence, then the 2-Gram will be queried and so forth.

Perception System

A perception system is that which allows for a character in a game to have incomplete knowledge of the game world. This concept is at the core of interacting agents in games: If an agent knows everything about the world at all times, then it has no reason to interact with other agents. If it does *not* have complete knowledge of the world, it will have to interact with others to gain knowledge.

As such, the perception system is created to not only allow for removing information from a character, which models the fact that humans only know that which they have experienced, but also to assist in the encouragement to create games in which agents interact with each other. The intuition for this encouragement is that it will create more believeable scenarios in games when agents do not instantly know everything about the game world.

The ability to forget information by introducing the concepts of short and long term memory is a part of perception as well – very few people have photographic memory, and as such most agents in a game should simulate forgetfulness as well, in order to be able to increase the level of believeability.

Finally, humans do not have instant reaction times. Reflexes are not instant, it takes time until someone realises what is going on around them when new information arrives. As such, any perception system should allow for non-instant reactions to changes in the environment. That is, when for example a character in an FPS is shot at, and potentially hit, he may not know instantly from where the sound of the gunshot came, it may well take time for this information to gather.

So, the suggested future work is an implementation of such a system into the Character in SMARTS Library (and exposure of it in SMARTS Designer), to allow for behaviors to take into account information found in the world in a manner which may or may not be complete. It is suggested that this work be based at least in part on the work of Frank Puig Placeres[18]. Arguably, the beginnings of such a system already exists, in the form of the `btCharacter` class, which is the behavior tree's per-character data store. If the information stored on there has more logic applied, it becomes a perception system in stead of simply a data store.

Parallels

The parallel composite created for use with the test bed in this project has only one termination policy, while the description in Section 2.1 on page 8 includes descriptions of multiple. It would therefore be advantageous, and indeed obvious, to implement these. The termination policy for a parallel should be exposed in the properties view, such that we have a property which is the policy of the parallel. This will allow the designer to change it on the fly.

Test Criteria

The test method employed in Chapter 7 uses three criteria for assessing the game being tested. These three criteria are proven by the authors of the method as showing correct results in the specific scenario type they and consequently we use. However, the action-adventure video game Ōkami[1] might cause a different set of criteria to be appropriate compared to the predator/prey type Pac-Man™used here. Consequently, an interesting point for investigation would be to see which weights and criteria would make sense in

[1] `http://en.wikipedia.org/wiki/Okami`

other types of scenarios. This is directly stated by Yannakakis and Hallam[21], who say that

> ”Other successful quantitative metrics for the appropriate level of challenge, the opponents’ diversity and the opponents’ spatial diversity may be designed and more qualitative criteria may be inserted in the interest formula”[21].

Other Tests

Our goal is not to evaluate the diversity of a group of agents, but to evaluate how diverse the behavior of a single agent is over several games. We can us the same method by assuming that each individual run of an agent is equivalent to a separate agent. Measuring the diversity of a single agent reduces to running it several times and using the method proposed in Hierarchical Social Entropy[2] on the resulting data.

A measure of behavioral difference, Hierarchical Social Entropy by Balch[2] suggest an evaluation chamber for evaluating robots, in which the robots are exposed to different situations and their responses are recorded. This idea for robots though is soon abandoned. Instead the robots are evaluated based on differences in their policies or programs. Now, this works fine if the robots are deterministic, but a behavior tree can have arbitrary conditions which must be met for a behavior to run, and there is no way of knowing ahead of time exactly how likely it is for a condition to be true.

We revisit the idea of an evaluation chamber for unpredictable behaviors for use in games, and recording the actions and reactions in a simulated environment might be the best approach. The intuition is that as more games are played the probabilities of a condition can be approximated by calculating the ratio between the number times a condition has been evaluated and the number of times in which it has been true. It might also be possible to define this difference solely on the structure and weights of a behavior tree.

If we could provide a distance function[2] it could be used in correlation with the metrics from Yannakakis and Hallam[21]. By providing a measure such that if the manifested behavioral diversity of a behavior tree A (written $D_m(A)$) is bigger than that of a behavior tree B, that is $D_m(A) > D_m(B)$, then the latent behavioral diversity of A (written $D_l(A)$) is bigger than that of behavior tree B. It might not be possible except for the most simple of cases, but it might again be possible to give a qualified guess on which behavior tree produces the most diverse behavior.

[1] Thomas Bäck. "Evolutionary Algorithms in Theory and Practice". "Oxford Univ. Press", 1996. 15, 17

[2] Tucker Balch. Hierarchic social entropy: An information theoretic measure of robot group diversity. Autonomous Robots, 8(3):209–238, 2000. 64

[3] Irad Ben-Gal. Encyclopedia of Statistics in Quality and Reliability, chapter "Bayesian Networks". John Wiley & Sons, 2007. 16

[4] Joanna J. Bryson. The behavior-oriented design of modular agent intelligence. `http://www.cs.bath.ac.uk/~jjb/ftp/AgeS02.pdf`. 12, 13

[5] Alex J. Champandard. Enabling concurrency in your behavior hierarchy. `http://aigamedev.com/open/articles/parallel/`. 9

[6] Alex J. Champandard. The flexibility of selectors for hierarchical logic. `http://aigamedev.com/open/articles/selector/`. 8

[7] Alex J. Champandard. The power of sequences for hierarchical behaviors. `http://aigamedev.com/open/articles/sequence/`. 8

[8] Alex J. Champandard. Understanding behavior trees. `http://aigamedev.com/open/articles/bt-overview/`. 13

[9] Alex J. Champandard. Using decorators to improve behaviors. `http://aigamedev.com/open/articles/decorator/`. 10

[10] Anders Ejlersen, Anders Tankred Holm, Kim Jung Nissen, Mads Bøgeskov, and Rasmus Kristensen. Noesc. Technical report, Aalborg University, June 2009. 29

[11] Dan Jensen et al. Gameobject hierarchy, 2009. `http://gluon.tuxfamily.org/wiki/index.php?title=GameObject_Hierarchy`. 20

[12] David Harel and Communicated A. Pnueli. Statecharts: A visual formalism for complex systems. In Science of Computer Programming 8, 1987. 6

[13] Finn V. Jensen and Thomas D. Nielsen. Bayesian Networks and Decision Graphs. Springer Verlag, 2007. 16

[14] Chong-U Lim. An a.i. player for defcon: An evolutionary approach using behaviortrees. `www.doc.ic.ac.uk/teaching/distinguished-projects/2009/c.lim.pdf`. 12

[15] Ian Millington. Artificial Intelligence for Games. Morgan Kaufmann Publishers, 2006. 16, 62

[16] Samuel J. Partington and Joanna J. Bryson. The behavior oriented design of an unreal tournamentcharacter. `www.cs.bath.ac.uk/~jjb/ftp/iva05sam.pdf`. 13

[17] Jamey Pittman. "the pac-man dossier, v. 1.0.19 ", February 2009. `http://home.comcast.net/~jpittman2/pacman/pacmandossier.html`. 55

[18] Frank Puig Placeres. Generic perception system. In AI Game Programming Wisdom 4, 2004. 63

[19] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. ”Database System Concepts, Fifth Edition”. ”McGraw-Hill”, May 2005. 62

[20] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. Adaptive game ai with dynamic scripting. Machine Learning, 63(3):217–248, 2006. 17, 61

[21] Georgios N. Yannakakis and John Hallam. Towards optimizing entertainment in computer games. Applied Artificial Intelligence, 21(10):933–971, 2007. 2, 17, 28, 29, 30, 43, 48, 54, 55, 64

Selected Test Data

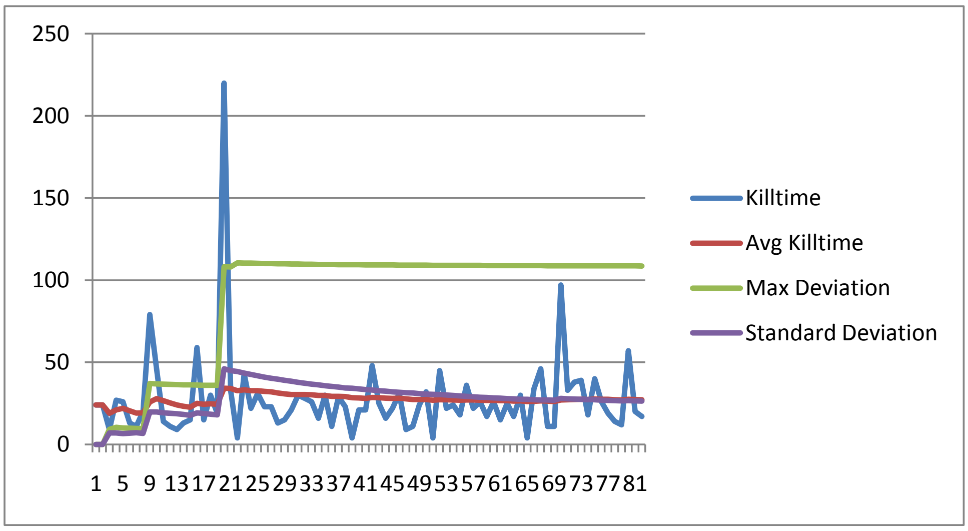


Figure A.1.: Deviations and kill times for Blink with scatter and no learning

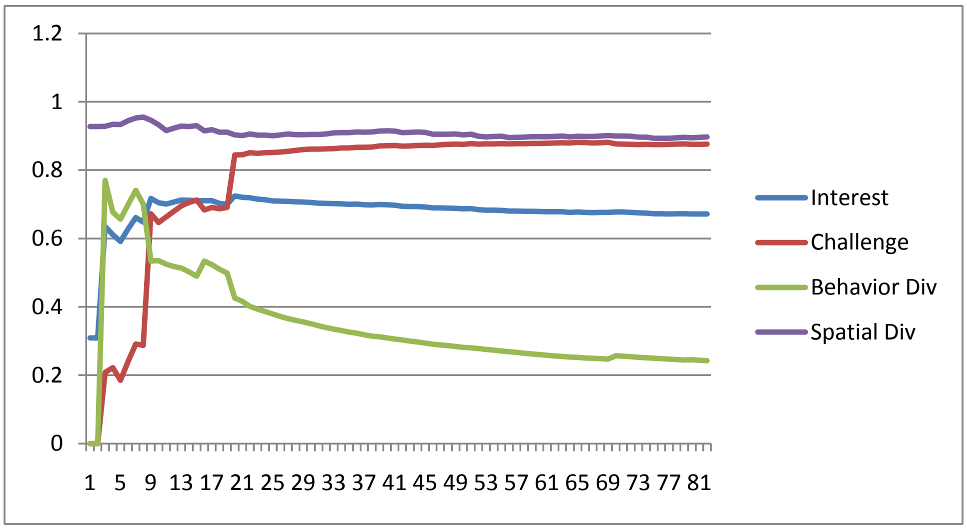


Figure A.2.: Diversities, challenge, and interest for Blink with scatter and no learning

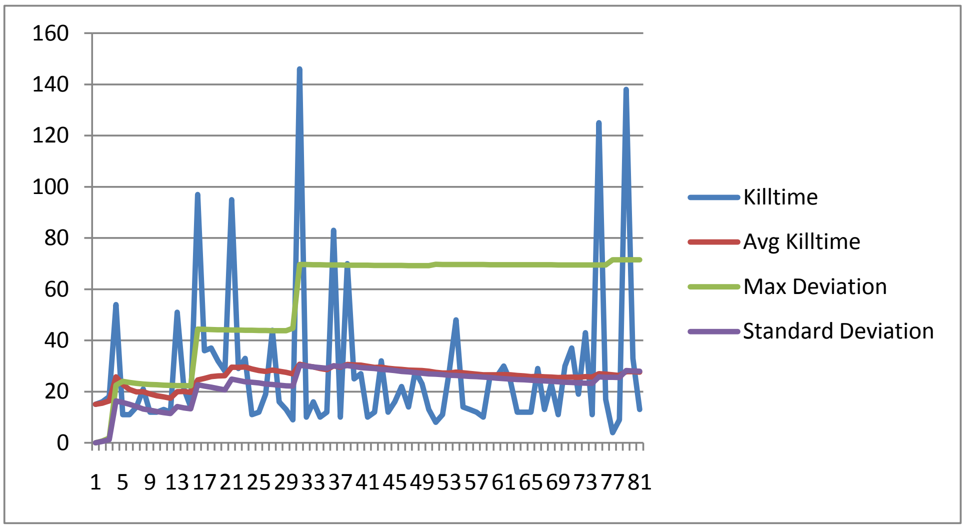


Figure A.3.: Deviations and kill times for Blink without scatter and no learning

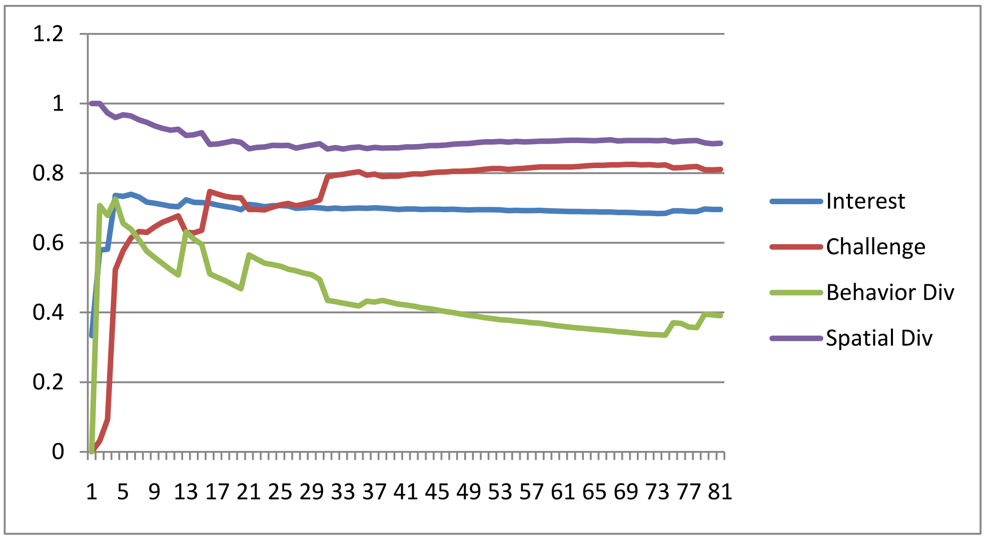


Figure A.4.: Diversities, challenge, and interest for Blink without scatter and no learning

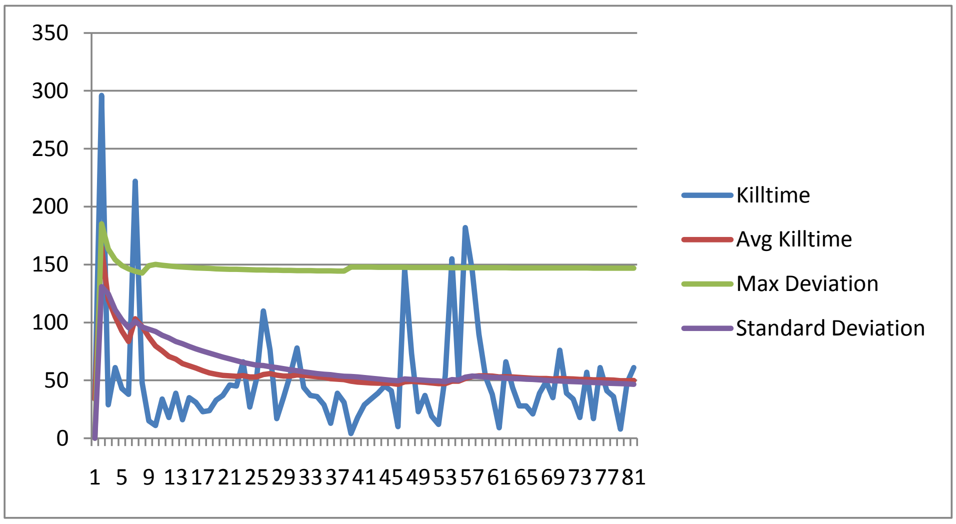


Figure A.5.: Deviations and kill times for Blink with scatter and learning

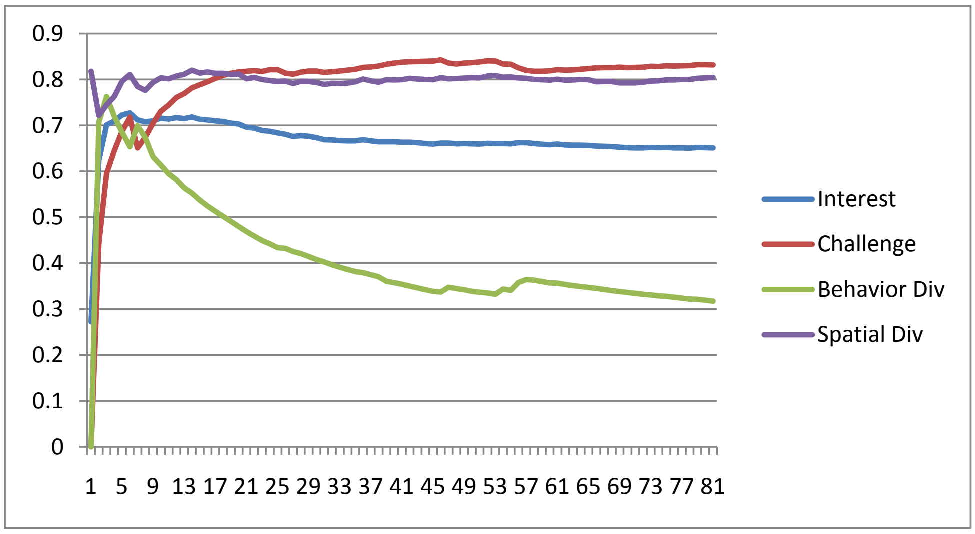


Figure A.6.: Diversities, challenge, and interest for Blink with scatter and learning

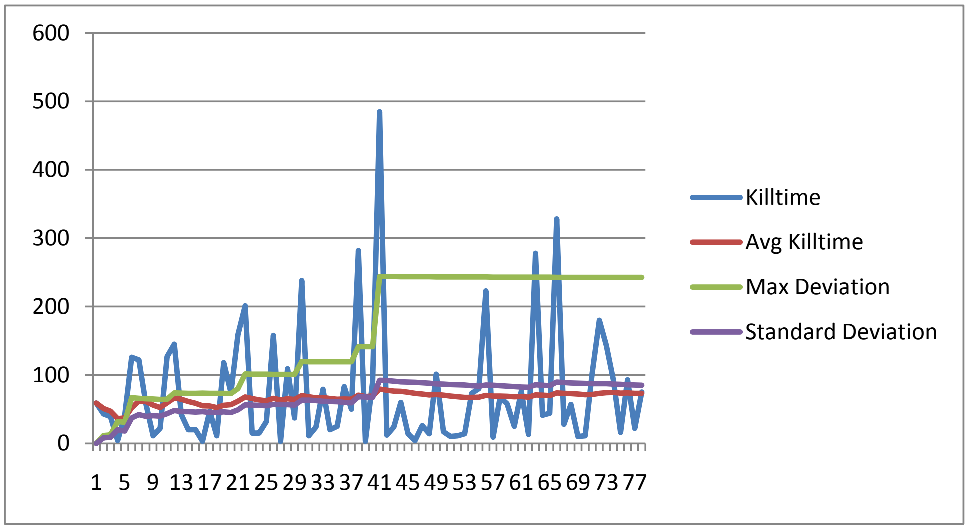


Figure A.7.: Deviations and kill times for Clyde with scatter and no learning

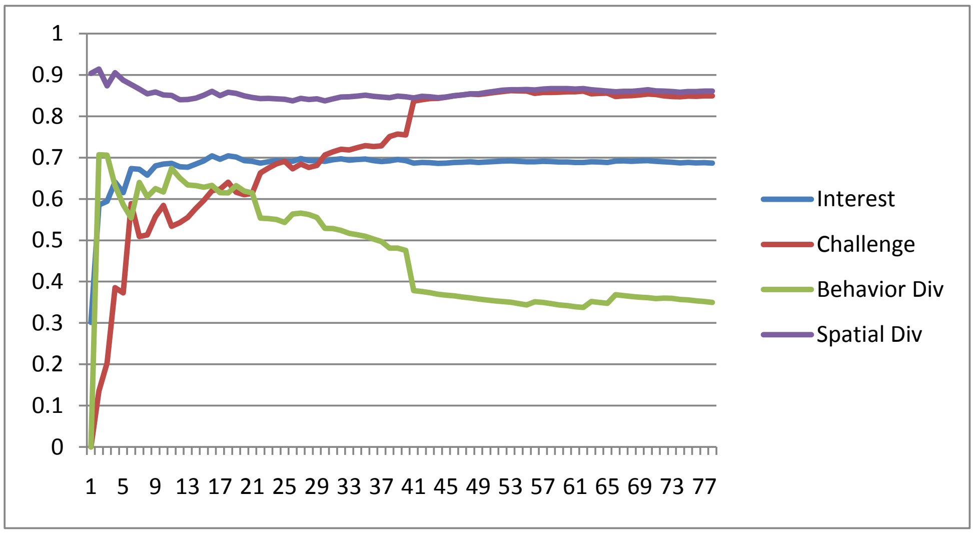


Figure A.8.: Diversities, challenge, and interest for Clyde with scatter and no learning

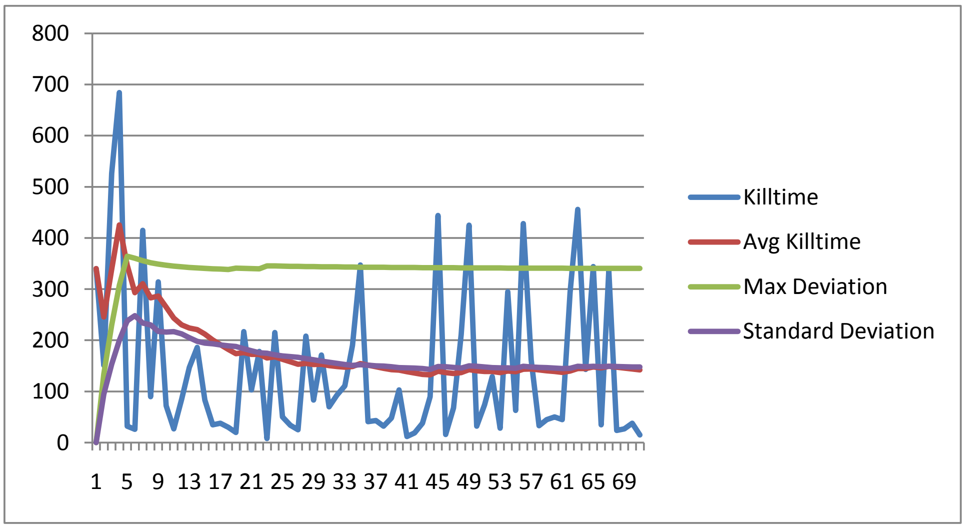


Figure A.9.: Deviations and kill times for Clyde with scatter and learning

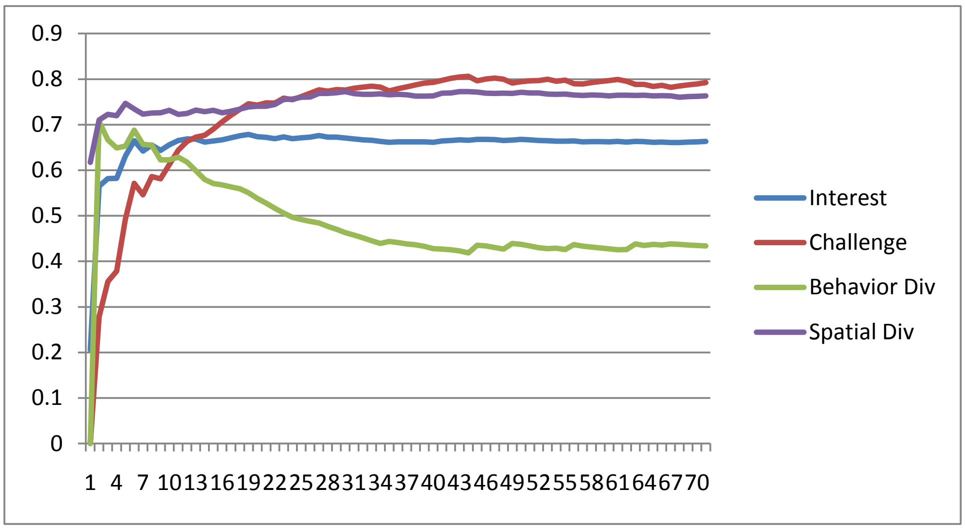


Figure A.10.: Diversities, challenge, and interest for Clyde with scatter and learning

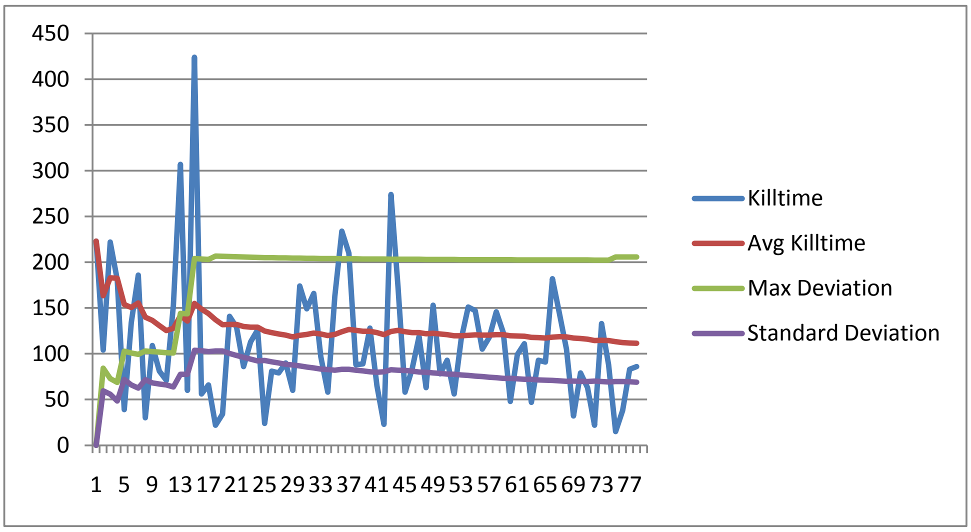


Figure A.11.: Deviations and kill times for Pinky without scatter and no learning

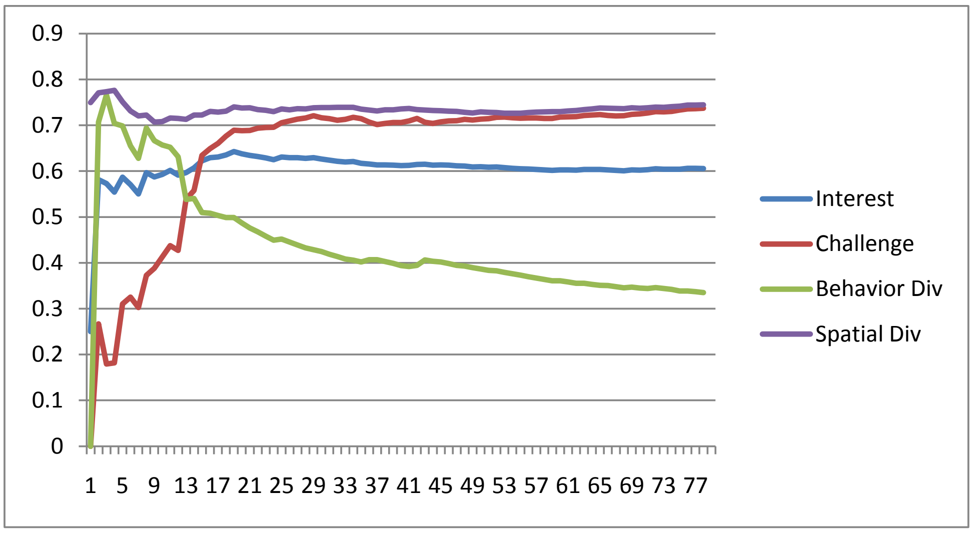


Figure A.12.: Diversities, challenge, and interest for Pinky with scatter and no learning

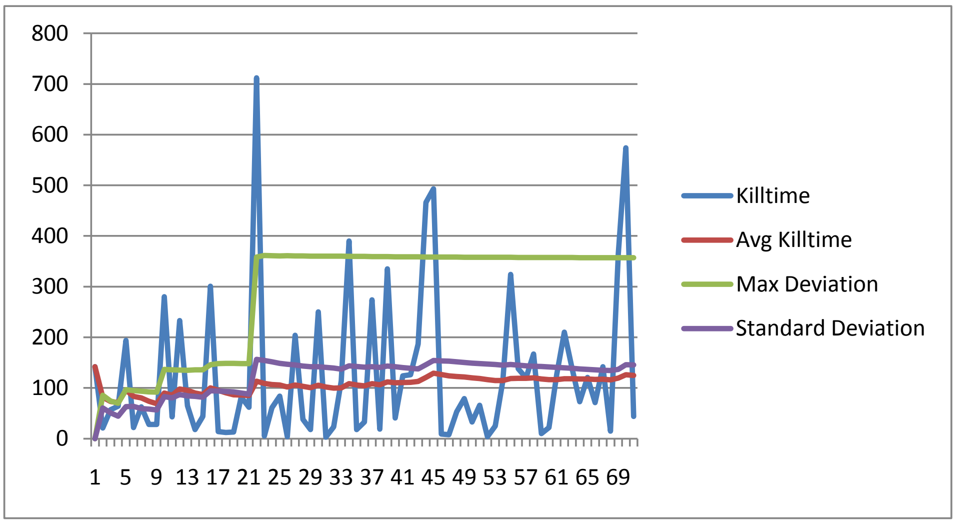


Figure A.13.: Deviations and kill times for purely random

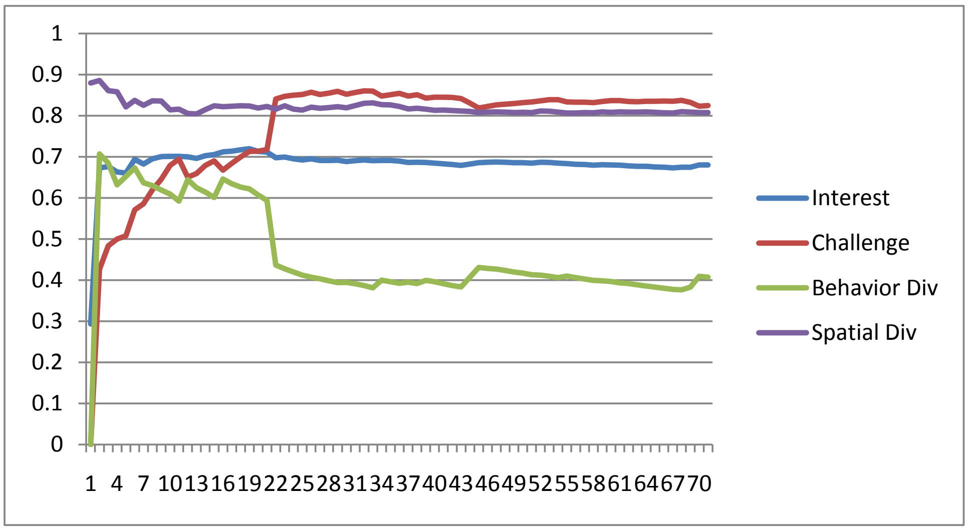
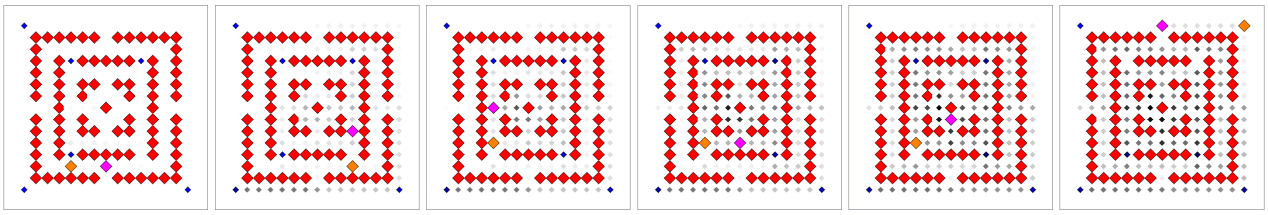


Figure A.14.: Diversities, challenge, and interest for purely random

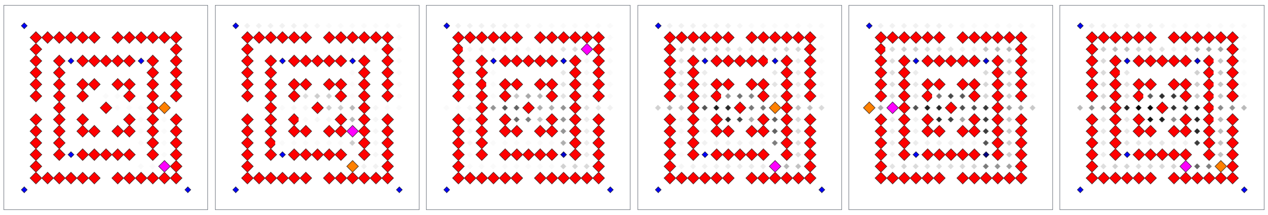
Selected Screenshots

The following is a selection of screenshots from the tests performed. The tiles are the game world at specific times during the course of the game. The grey dots seen in the game world is a heat map, indicating how many times the agent in question (for example Binky in the first set) has visited that tile.

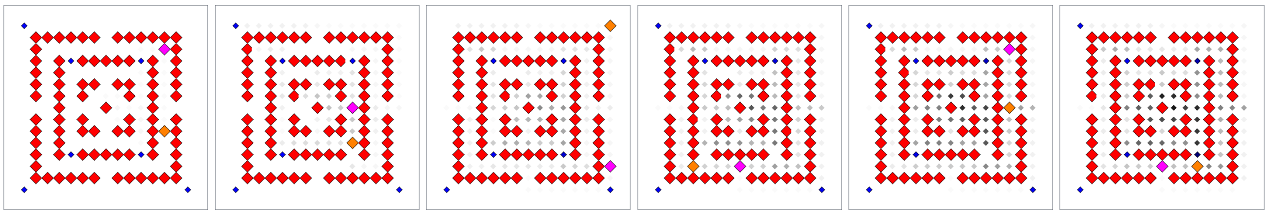
B.1. Blinky, with learning and scatter



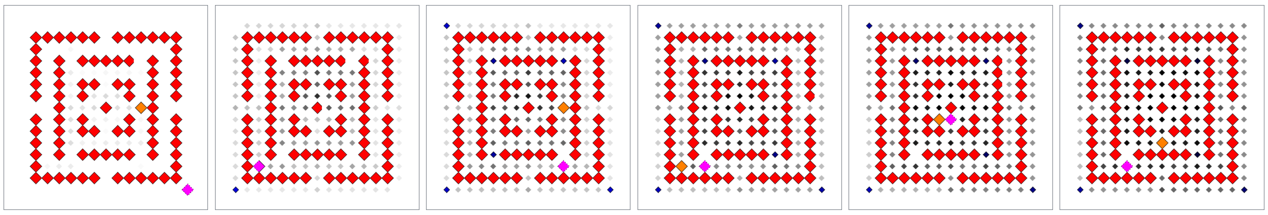
B.2. Blinky, without learning and scatter



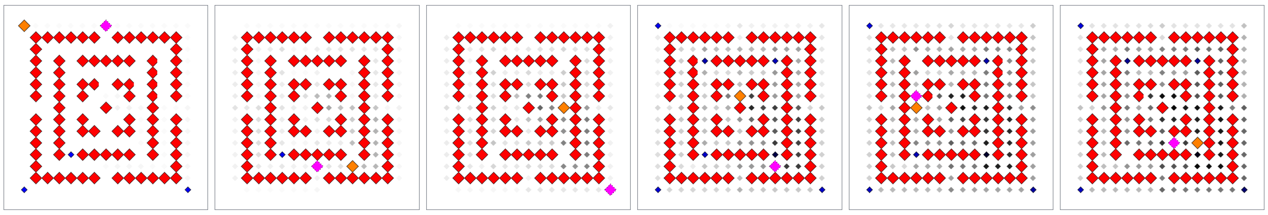
B.3. Blinky, without learning and with scatter



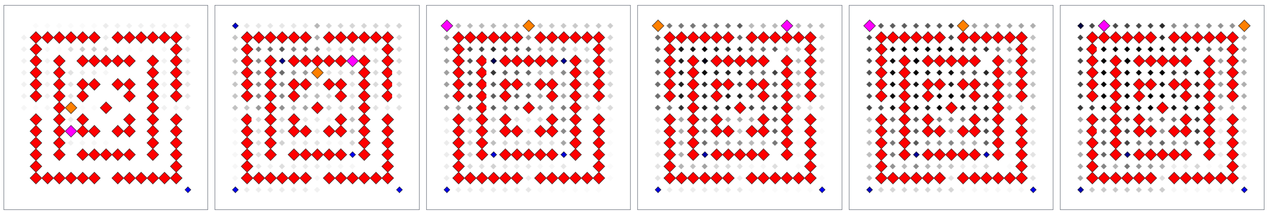
B.4. Clyde, with learning



B.5. Clyde, without learning



B.6. Pinky



B.7. Pure Random