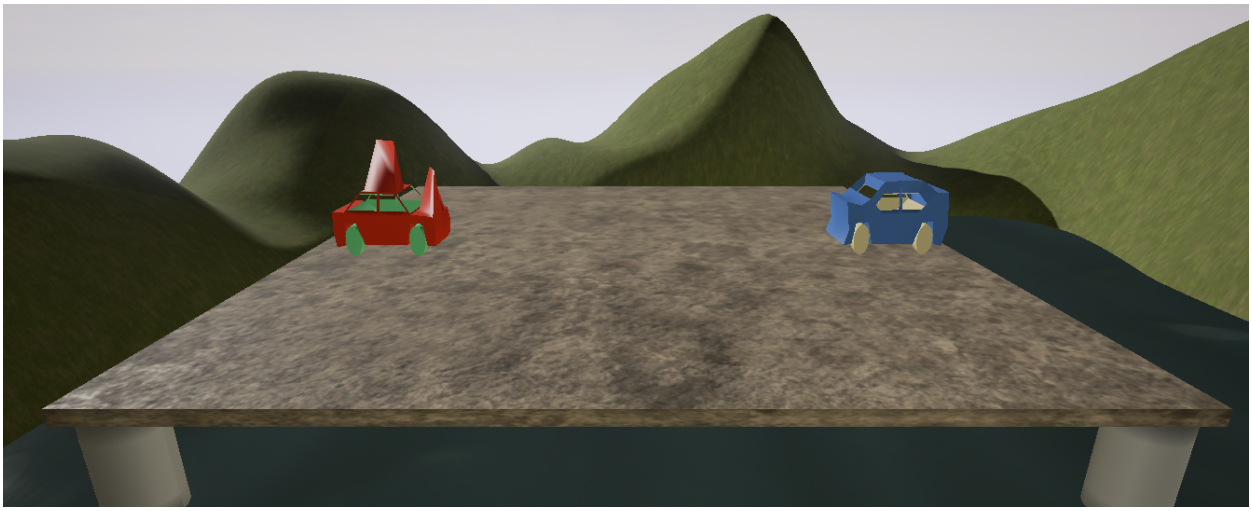

A Case Study:

Reactive AI Behavior for Enhancing the Game Experience

utilizing Unreal Engine 4

Marcus Nygren
Emma Forsling Parborg

TNM095 Artificial Intelligence for Interactive Media, M.Sc. Media Technology,
Department of Science and Technology, Linköping University

November 6, 2015

Abstract

This report aims to design a game robot AI to maximize fun and enjoyability. As a consequence, it looks at what technical implementation is needed to make a behavior tree truly reactive utilizing Unreal Engine 4. The result is a game where the player can manipulate and analyze AI behavior to its favor, and the AI acts in an effective, challenging and believable manner for the hectic human-vs-AI robot game SumoBot Wars.

Keywords: AI Game Design, Reactive Behavior Trees, State Machines, Unreal Engine 4.

Contents

1 Introduction

In the world championships of the sport SumoBot, players pre-program AI robots to compete in 1-vs-1 hectic and strategic tournament-style duels. A game is won when one robot has managed to push its opponent outside of the sumo bot arena. [1]

The AI behavior for the robots can range from basic to very advanced. A basic behavior could be an AI which uses the middle of the ring as a safe zone and then rushes the player from that position [2]. An advanced behavior could be represented by an AI that analyzes the situation and environment, and changes its behavior dynamically.

This reactive behavior has been translated into a digital version of SumoBot, SumoBot Wars, utilizing behavior trees in Unreal Engine. This report demonstrates a case study of how an AI robot can be designed to make a game enjoyable, dynamic and increasingly challenging.

1.1 Motivation

The AI game design can be the difference between a good and a bad experience for a player. If the AI is too easy to combat, it will be perceived as boring. If the AI is too complex, the player will feel outsmarted. In a game which demands progression, the AI needs to be varied and increasingly difficult at the same time [3]. This presents a tough challenge to both the AI programmer and designer. In what ways can an AI improve the gaming experience? Which technical method should be used? How should the robot behave, and when?

This report explores how these challenges can be solved with playfulness in mind, to dramatically improve the game experience.

1.2 Purpose

The purpose was to design an AI that would adapt its behavior to always provide an exciting experience for the player in the game SumoBot.

1.3 Limitations

In the sport SumoBot the players program their own AI. For the game SumoBot Wars, the players should not need to possess coding skills to be able to play the game. Hence, one limitation for this project is that the enemy behaviour needs to be preprogrammed and therefore the AI was designed through the lens of maximizing fun and enjoyability for the player.

For this project Unreal Engine 4 has been used exclusively as the game engine, primarily because of it's built-in behavior tree editor, therefore other game engines have not been investigated.

2 Theory

In this chapter the progression of methods for choosing and building a game AI are highlighted together with what factors make an AI fun and enjoyable.

2.1 Hiarchical logic for designing game AI

Today most game designers and developers have converged towards designing game AI using hierarchical logic [4].

There is a matrix of possible techniques. Academics have been thinking about AI for decades, and game developers for a long period of time. Hierarchical logic is a generic term, with many sub-implementations available. Three methods in particular are standard in modern game industry: scripting, Hierarchical Finite State Machines, *HFSM*, and Planners of the type Hierarchical Task Network, *HTN*. These methods have both advantages and disadvantages, making them non-suitable for particular kinds of games, like a game where sudden events occurs. Therefore, reactive behavior trees has also been investigated.

2.1.1 Scripting

Writing scripts to trigger actions is mainstream knowledge among programmer. However, for designers it can be difficult to understand the script and analyze what is performed and thereby see the plan ahead. A more suitable solution can therefore be derived by using a Hierarchical Finite State Machine, *HFSM*. [4]

2.1.2 FSMs vs. HFSM

A Finite State Machine, *FSM*, contains a finite number of states, each with its own behavior. To transition between states, an input or event needs to be triggered [5]. This enables full control at the lowest level for the designer. When the system expand, reoccurring behavior is common and can be solved by creating a HFSM instead. A HFSM is a FSM where states can be decomposed into other FSMs [6], since states are allowed to be nested into hierarchies.

By utilizing HFSM a goal-driven AI can be created, however the size of the state machine would increase rapidly and a lot of work would have to be done for all different transitions. This problem can be solved by using planners instead.

2.1.3 Planners and HTN-based planners

Planning is a formalized process of searching for sequence of actions to satisfy a goal. The planning process is called plan formulation. [7] While a FSM tells an AI exactly how to behave in every situation, a planning system tells the AI what its goals and actions are, and lets the AI decide how to sequence actions to satisfy goals. Jeff Orkin describes that a planning system gives AI the knowledge they need to be able to make their own decisions about when to make a transition from one state to another.

Planners uses search to create modular behavior to achieve a particular goal, in the game. This provides a certain amount of automation.

F.E.A.R. was the first game known to use planning techniques, based on the work of Jeff Orkin. It used the STRIPS planning algorithm from academia in 1970. [8] Planning can be successful in open-world games, but more and more games are leaning back towards either HTN planners or behavior trees.

HTN planners are based on hierarchies of tasks that can be broken down recursively, like a plan that starts with the big picture and gets refined into actionable steps. They are good because they use search for automation and are goal-directed by default. However, Champandard argues they are not great at making queries while planning, or monitoring the plan once it has started and see if there are any errors. As a result, the planner typically has to run often to check for changes [4].

2.1.4 Behavior trees

Alex J Champandard describes the dynamic between the different forms of hierarchical logic like the dynamics for "rock, paper, scissor" [4], where each of the feature solve the others problems. Champandard argues that, while not being a silver bullet, behavior trees take the best from these methods, achieving the best possible compromise in many situations.

Behavior trees are sometimes referred to as reactive planner [9], however Peter Mawhorter makes a clear distinction between behavior trees and reactive planning [10].

Champandard's motivation is that a behavior tree can follow up on decisions over time, make sure that the plan executes correctly, and deal with failures and re-plan locally. It is goal-driven and reacts to sudden changes.

For a game designer, this behavior trees gives a better overview of how the AI work, since the decisions and can be traced easily throughout the tree structure. By utilizing Unreal Engine, these behaviors can be displayed during gameplay. For these reasons, behavior trees works out extremely well in many game AI applications, which explains its popularity.

2.2 Evolution of behavior trees

In this section the first generation and second generation of behavior trees are introduced. The last section focuses on behavior trees limitations when reactive behavior is desired, presenting a hybrid model as a possible solution.

2.2.1 Behavior trees - first generation

In a basic behavior tree, two different nodes exist within the tree: *Composite* and *Leaf* [9, p. 334-371]. A leaf node is a node in the tree that does not have any children. Composite nodes are inner nodes in the tree that keep track of a collection of child tasks and their behaviour is formed by its children. The composite nodes can be either a *Selector* or a *Sequence*. A selector selects one of its children and executes it until it returns success. A sequence goes through all of its children until one of them returns failure, then it aborts the sequence. A sequence will return success if all children succeed.

2.2.2 Behavior trees - second generation

The second generation of behavior trees adds three new functionalities [11, 12, p.58-60]: The first functionality, decorator, can be used to determine if a child node should repeat processing or if it should be terminated.

The second functionality, the look-up table, is the memory for the behavior tree. In this memory, information can be stored and changed during run-time by the child nodes or from events triggered in the game, in order to make the tree more aware of if certain events have occurred.

Lastly, the concurrency functionality allows nodes to be executed in parallel. This new functionality enables the AI to perform multiple tasks at the same time.

2.2.3 Limitations with behavior trees - introducing a hybrid system

There exist some limitations with behavior trees when reactive behavior is desired [9, p.370-371]. This is most notable when an external event occurs, that should immediately alter the game behavior. There are two major problems:

1. **Clutter when changing the overall game behavior.**

   Instead of adding extra decorators in all branches, and thereby cluttering the behavior tree and introducing repeatability, a better solution would be a state machine with different behavior trees as explained below.

2. **Non-immediate interruption of behavior.**

   A behavior tree can not constantly check if a certain condition, like "Has a storm occured?", is met. Only when a task has been completed (with success or failure), it will leave the node and possibly do a conditional check to discovering e.g. a storm. If the player notices this delay, it can lead to a decreased game experience since the AI does not feel responsive. A state machine has the advantage that it immediately alter which branch or behavior tree that should be active. Depending on the game and task complexity, this can be essential.

There are two ways this state machine can be implemented. The first solution is to have a state machine located outside the behavior tree, with the functionality of activating different trees depending on the external events. The second solution is to have tasks in the behavior tree, located down in the branches, acting like a state machine. This "state machine" would then attach different behavior trees depending on if an external event has occurred.

The downside with introducing a state machine is that it becomes an extra burden for the AI authors and toolchain developers since two types of authoring have to be handled, state machine and behavior tree.

2.3 What makes a fun and enjoyable AI?

When trying to create an enjoyable AI, the first challenge is to determine what makes a game enjoyable and how can it be incorporated it into the AI [13, p.36-37]. According to Mehd Khosrow-Pour, this is a subject which have been researched over time and many so called *fun factors* have been derived. When trying to incorporate these fun factors into the AI, Khosrow-Pour mentions that these factors may not provide a direct guidance in how the role of the AI should be defined, since it may differ depending on what game it is, but they can help frame the discussion. Khosrow-Pour concludes that a successful AI system is distinguished by two attributes: realism/human-likeness and high-skill level. These attributes are effected by the believability and effectiveness of the AI agent, in which adaptivity needs to be embedded in both.

Khosrow-Pour mentions one important aspect to consider for the AI believability. The AI should not be given "God abilities", since the AI could be perceived to be cheating. Instead, Khosrow-Pour mentions the basic principles for building a believable agent, that Jones et al. proposed [13, 14]. These principles describes

that the AI should have roughly the same data processing system, motor systems and sensory as the human player. It should also have the same knowledge as the player.

Khosrow-Pour states that believability is not enough to make the AI fun to play with, and this is were the effectiveness comes in. If the AI should be enjoyable it has to possess reasonable skills, since it's important to have a challenging and effective AI.

These two aspects does however not always go hand in hand, and Khosrow-Pour makes the example that a skillfull agent is not always believable instead the opponent sometimes have to be weaker in order to be believable.

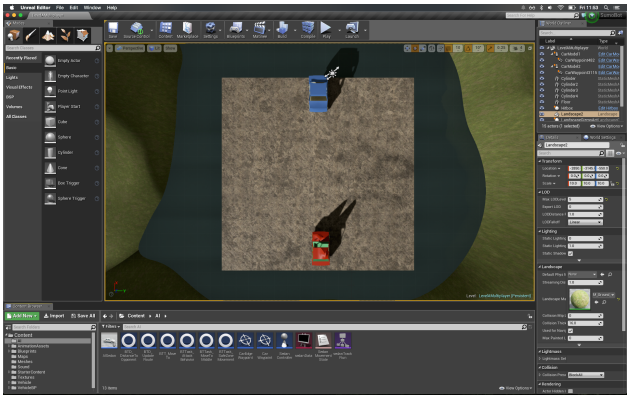

Figure 1: Unreal Engine 4 was chosen as the game editor. The image shows the standard user interface, with the game SumoBot running.

3 Method

For this project the method Behavior trees is implemented in game engine Unreal Engine 4, see Figure 1, when creating the AI. In this section the behavior chosen for the AI is described as well as which method utilized for implementing reactivity. This section will also describe how the behavior trees differ in Unreal Engine.

3.1 Choice of behavior

The inspiration for the robot's standard behavior is based on the basic behavior that was observed in the sport SumoBot [2]. Namely, stay in the middle, wait, and rush at the other player. By also adding emotions to the AI, desperation and aggressiveness, the AI behavior will alter to also be more or less passive, leaving the player a sense of control at the same time that the AI is less predictable.

The robot's behavior is determined by a behavior tree with different branches corresponding to different aggressiveness levels. The branches includes behavior for a defensive, medium, and daring AI, see Figure 2. As soon as the aggressiveness of the AI has lowered, another branch will be activated. The Aggressiveness value is changed on every collision with another car, by checking out which car took the most damage, and thereafter alter the value in the look-up table connected to the behavior tree.

The Desperation value will change, depending on the level, when the level starts. It is technically possible to change the Desperation value in-game, which was desired.

To give the AI nor more or less knowledge about the game than the player, the decision was taken to not give the robot the same abilities as in the sport SumoBot. As the player has knowledge about the whole map and the enemy

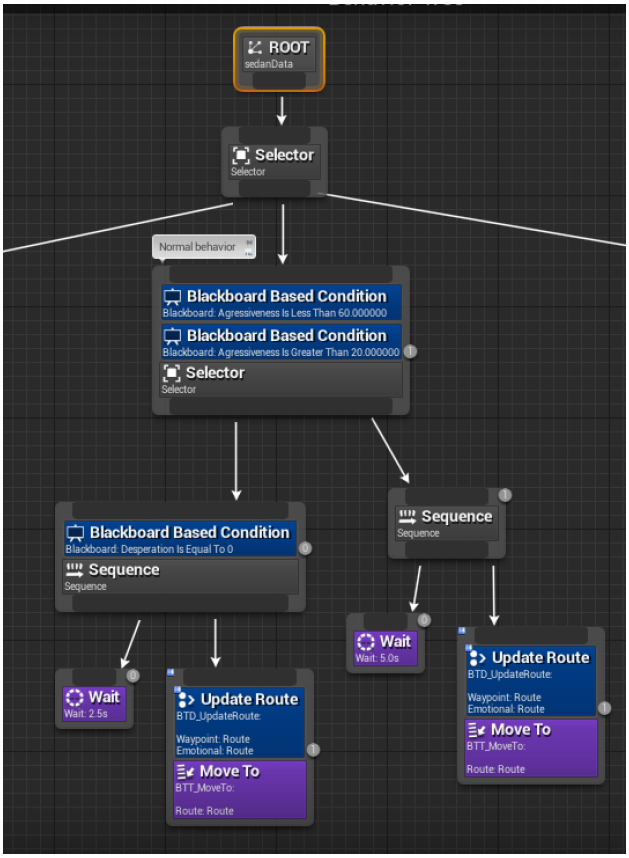

Figure 2: The image illustrates one of the branches, representing one of the AI behaviors, in the behavior tree. The blue node *Blackboard Based Condition*, represents the decorators used to check if different behaviors should be triggered.

position, the AI is not equipped with a motion sensor like it often is in the sport SumoBot.

3.2 Choice of method

The finished result uses a hybrid system for deciding behavior. A behavior tree of the second generation is used as well as the simplest kind of state machine approach, where decorators down the branches has a conditional check to see if the Desperation value should force a certain behavior, see Figure 2.

3.3 Behavior Trees in Unreal Engine

The implementation of Behavior Trees in Unreal Engine have three distinct differences from standard behavior trees [15]. The first is that the behavior trees are event-driven. This means that the tree passively listens for events that may trigger the tree, instead of constantly, for every frame, loop through the entire tree to check to check if any changes have occurred. The advantages with this approach is that the event driven architecture grants improvements to both performance and debugging

The second difference is that conditionals are not leaf nodes. Instead conditionals are used as decorators which are placed at the root of the sub-tree as a way to stop the flow from continue down the branch if the conditions are not fullfilled.

Lastly, the third distinct difference with Unreal Engines behavior trees are that the node *Simple Parallel* can be used as a complement for concurrency behavior. These parallel nodes are a way to conceptually perform several tasks at the same time, however they still run on the same thread. When comparing this to multi-threading in which tasks truly are executed at the same time, the parallel nodes do display the sequences at the same time even though they technically don't perform them so.

Other changes that is good to be aware of

is that the look-up table is very similar to its theory and in Unreal it is simply called a *Blackboard*.

4 Result

In Figure 3 and 4, the derived behavior tree, created in Unreal Engine, is displayed. Safe, normal or crazy behavior is changed real-time depending based on the AI's recorded aggressiveness, which in turn is changed by collisions. Desperation is determined based on what the score is. These two values, Desperation and Aggressiveness, will impact the AI movement depending on which behavior that is active. By doing so, the AI is constantly trying to maximize fun and enjoyability for the player.

In the first level the AI starts out feeling safe, meaning it will wait, go towards the middle of the ring, and then charge towards the player, and repeat. If the player hits the AI, the AI's aggressiveness value will increase, eventually going from safe behavior to normal behavior, which means the waiting time between attacks will not be as long.

In the normal behavior, the player will either calm down the AI or make it more aggressive, depending on if it hits the AI hard or softly. When furious, the AI will go into crazy behavior, meaning it will give the player no rest between attacks. The player can use this to it's advantage, since the AI will be so aggressive it might even move itself out of the map if the player lures the AI towards an edge. On the other hand, in the safe behavior, the player has a window of opportunity to make a big charge towards the AI when it is in waiting mode. Therefore it is up to the player how the AI will behave. This dynamic means that the player will eventually find a style of playing the game it is comfortable with, manipulating the AI to it's own favor. This is an important aspect to maximize fun and enjoyability for the player.

Using decorators as conditionals within the behavior branches, sub-behaviors can be introduced to set up special rules for e.g. how the AI desperation should influence the safe, normal and crazy behaviors. This can be used in several ways. To give the AI even more personality and make it more challenging throughout the levels, the desperation value is utilized. This value increases for each level, depending on what the score is. Early in the game the AI desperation is low, but when the player has reached the final level, the AI has reached maximum desperation - since the AI by all means does not want to lose. For example, in the last level, the AI desperation is so high that the enemy robot will make a false start, charging towards the player even before the traffic light has hit green!

5 Discussion

In this chapter, improvements for the development of the game and the AI is discussed.

5.1 Analysis of method

Based on section 2.2.3, since the behavior tree is not very complex, it was a valid choice to use the simplest form of state machine. For future work, a more thorough implementation should be used, see section 5.3.

The game would benefit from researching

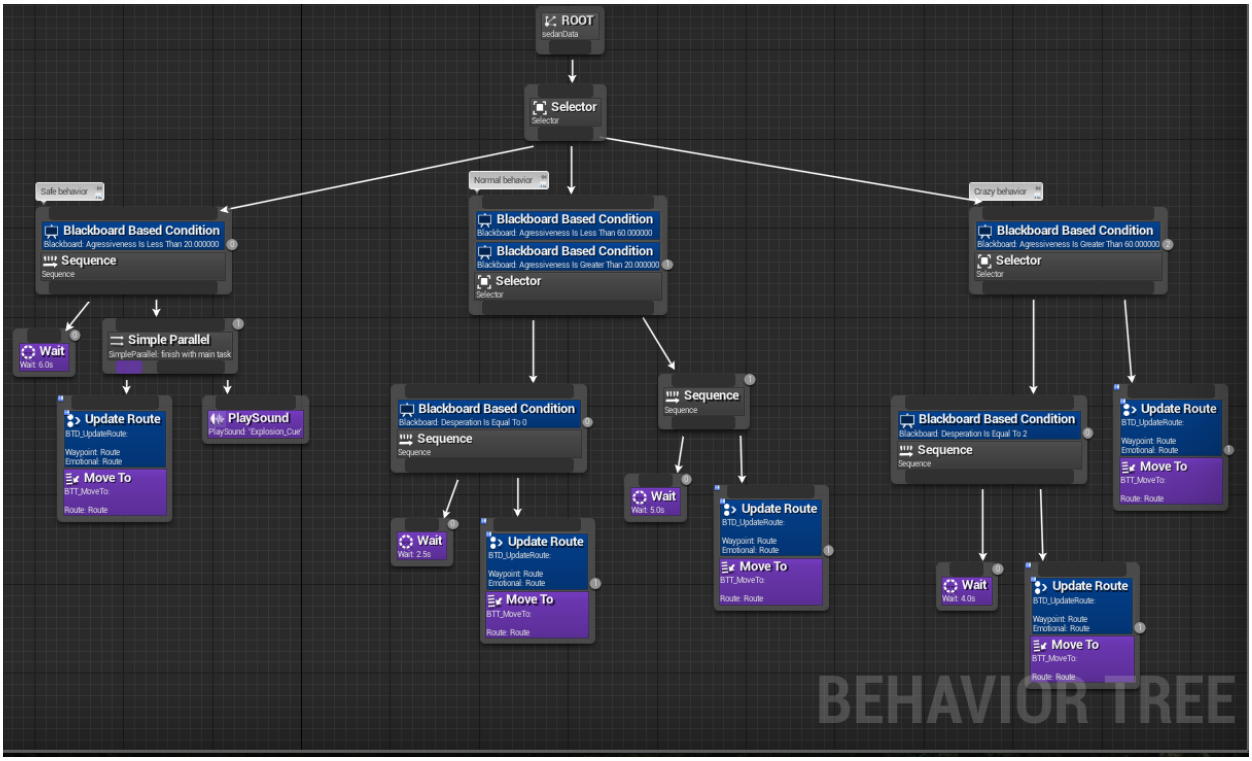

Figure 3: The behavior tree created in Unreal Engine. In this image the Composite nodes, *Sequences* and *Selectors*, are illustrated. At the top of the image, a selector is used to determine which behavior to activate: *safe*, *normal* or *crazy*. The blue nodes called *Blackboard based conditions*, are decorator nodes, which are used to check the input with the look-up-table in order to determine whether or not the branch should be executed. In the left branch, the node *Simple-Parallel* is utilized. This is a concurrency node, which is used to execute parallel activities.

the wider field of emotions in AI: implementation, design and behavior. For this project, accuracy and maximizing fun and enjoyability is the most interesting aspects. Today's implementation of fun, enjoyability, desperation and aggressiveness is very primitive. During game demos, user feedback has been positive about how the AI behaves with regards to the parameters in section 2.3. However, it is not in full bloom.

5.2 Analysis of result

Giving the player more control would have been desirable. It is questionable if it is ethically correct to bereave the player the possibility to decide for themselves what an enjoyable AI is for them, since this is a goal for the game. If there would have been a solution to this limitations,

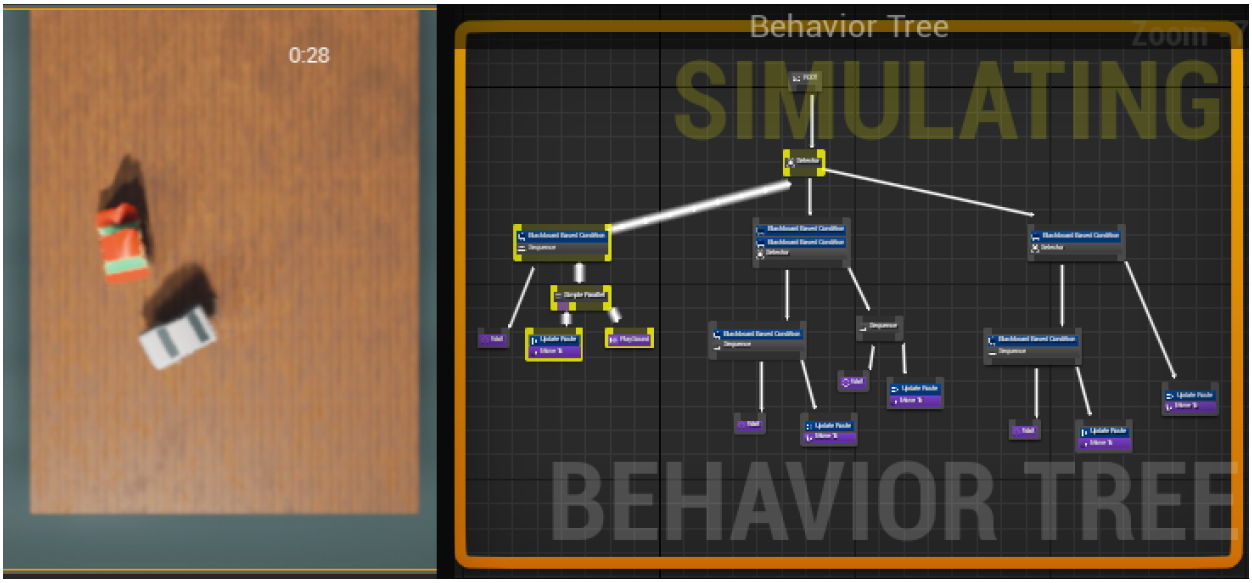

Figure 4: The game running in the developer mode of Unreal Engine 4. To the left: the game, with a player fighting the enemy AI. To the right: the behavior tree debugging mode, showing active behavior. The result can be seen on YouTube: https://youtu.be/MxOh64meF_A

the game designers would not have to compromise when designing the AI to fit a certain kind of player, or making the AI as general as possible.

The Behavior Tree tool in Unreal Engine 4 can be used instead of writing code, with a number of advantages, see section 3.3. Unfortunately, the tree can only be built inside the engine, so the tool will never be presented to the player. Thereby the player will not be able to create its own AI as in the sport SumoBot. In section 5.3, a work-around for the technical limitations is proposed.

5.3 Future work

In the future, the players could "build" their own AI personality, by adjusting the AI settings for when a predetermined behavior should be triggered via a GUI.

Another solution would be to have the AI develop itself, via genetic algorithms and neural networks. The game could analyze how the player behaves, e.g. how it moves through the map, to analyze if is is happy or sad about the AI's behavior.

Another improvement would be to add more complex behaviors and tasks. If so, the game levels could easily introduce sudden changes, such as ice, storm, that should alter the overall game behavior. Technically, this would mean that a real state machine needs to be introduced. This would give a more robust hybrid system,

with a behavior tree implementing a state machine.

6 Conclusion

This project serves as a case study, showing that it can be good idea to implement AI emotions in a human-vs-AI game if the goal is to give the player a better game experience.

While the project result is not a production-ready AI game design, it clearly shows that the methods and techniques highlighted in this report has big potential for appreciated AI game design. It is with confidence, that game designers and programmers are encouraged to explore the field further and deeper.

References

[1] Bc. Jan Tomášek & Bc. Štěpán Havránek. Robot-sumo. http://bit.ly/robot-sumo, 2013.

[2] Henrik Kniberg. How a team of 2 kids + adult rookies won a robot sumo competition. http://bit.ly/robot-sumo-competition-blog, October 2015.

[3] Guy W. Lecky-Thompson. *AI and Artificial Life in Video Games*. Cengage Learning, Boston, MA, USA, 2008. ISBN 978-1-5845-0616-4.

[4] Alex J. Champandard. Behavior trees for next-gen game ai. http://aigamedev.com/insider/presentation/behavior-trees/, December 2008.

[5] David R. Wright. Finite state machines. http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf, Summer 2015.

[6] Penny Baillie de Byl. *Programming Believable Characters For Computer Games*. Charles River Media, Boston, MA, USA, 2004. ISBN 978-1584503231.

[7] Jeff Orkin. Game developers conference 2006 - three states and a plan: The a.i. of f.e.a.r. Monolith Productions / M.I.T. Media Lab, Cognitive Machines Group, March 2006.

[8] Alex J. Champandard. Planning in games: An overview and lessons learned. http://aigamedev.com/open/review/planning-in-games/, March 2013.

[9] Ian Millington and John Funge. *Artificial Intelligence for Games, 2nd Edition*. CRC Press, Burlington, MA, USA, 2009. ISBN 978-0123747310.

[10] Peter Mawhorter. Behavior trees and reactive planning. http://webstaff.itn.liu.se/~piede/courses/tnm095/modules/behavioral%20animation/slides/BT%20and%20reactive%20planning-2010.pdf, October 2010.

[11] Alex J. Champandard. Understanding the second-generation of behavior trees – #altdevconf. http://aigamedev.com/premium/tutorial/second-generation-behavior-trees, March 2012.

[12] Marco Gonzalo and Marco Antonio Gómez-Martín. *Artificial Intelligence for Computer Games*. Springer Science & Business Media, New York, NY, USA, 2011. ISBN 978-1-4419-8188-2.

[13] Mehdi Khosrow-Pour. *Contemporary Advancements in Information Technology Development in Dynamic Environments*. IGI Global, Hersey, PA, USA, 2014. ISBN 978-1-4666-6253-7.

[14] P.E. Nielsen K.J. Coulter P. Kenny & F. Koss R.M. Jones, J.E. Laird. Automated intelligent pilots for combat flight simulation. *AI Magazine*, 20(1):27–41, 1999.

[15] Unreal Engine 4 Documentation. How unreal engine 4 behavior trees differ. https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/HowUE4BehaviorTreesDiffer/index.html, October 2015.