Modelling of NPC's as interacting Statecharts

Glenn De Jonghe

Master Student Software Engineering at University Of Antwerp

Abstract

Since video games nowadays got more complex, writing consistent and re-usable code for game AI has become very hard. In this paper we will look at the modelling of Non-Playable Characters using the statechart modelling language and will see that this method has many advantages. This is shown in a small tanks game, developed for the purpose of this paper, where the player has to battle one or more NPC tanks with statechart-modelled behaviour. We will also inspect if modelling other components of the game has any benefits. For instance a completely modelled environment would make it possible for the NPC's to interact with it solely using events.

Keywords: Model Driven Engineering, Non-Playable Characters, Game AI, Statecharts

1. Introduction

NPC's in simple games such as board games or old 2D games do not require much effort to be specified and can be easily done within the programming language itself. In modern games however this is not the case. That's why we will use a visual modelling language at a higher level of abstraction which leads to an easier-to-understand and consistent design.

Game AI models react to events and changes in the state of the environment, and thus is the statecharts formalism an appropriate modelling language. As modelling environment AToM^3[2] has been used in combination with the statechart compiler and DCharts formalism of Huining Feng[3].

As this paper continues we will first look in section 2, at the related work this paper is based on. In section 3 we will show which components next to the AI are also accompanied with a model and how we decided this. In section 4 we will go in more detail about the modelling of the NPC's and we end the paper with a conclusion in section 5.

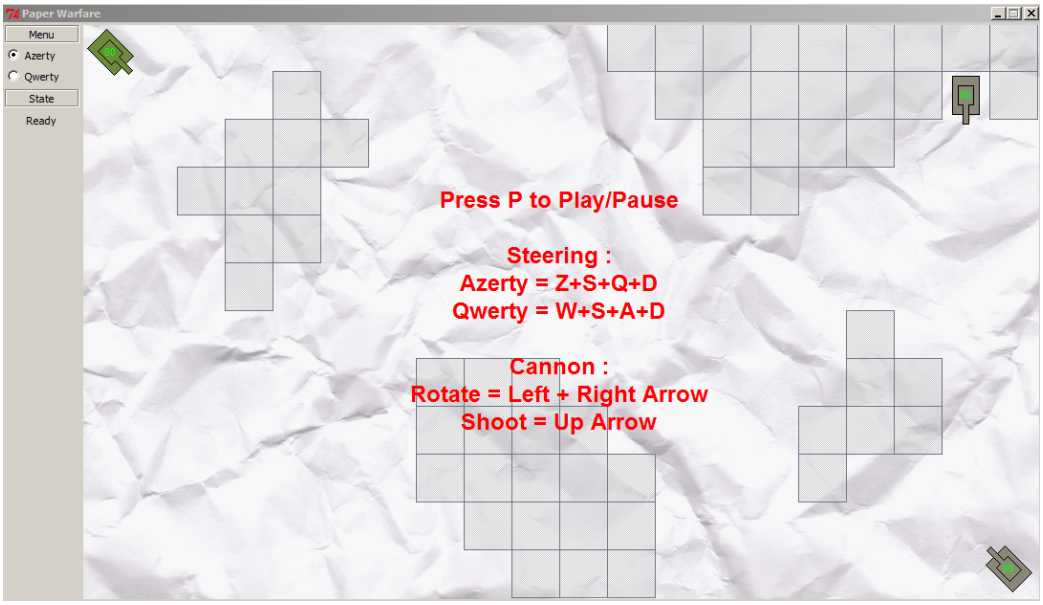

Figure 1: Screenshot of the game which was developed for this paper.

2. Related Work

The behaviour of the computer-controlled tanks in the example game (see figure 1) are modelled using the information given in *Model-based Design of Computer-Controlled Game Character Behaviour*[1]. In that paper the AI is modelled using a layered approach as you can see in figure 2.

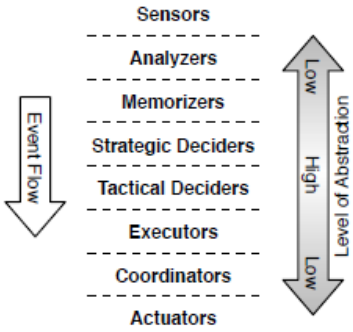

Figure 2: AI model architecture

2.1. Sensors and Analyzers

The sensors extract information from the state of the tank, which evolves during the game. Some games actually model this state in great detail. These games are typically called simulators where the physical interactions of the components with the environment are very important. When only the behaviour of a pilot is desired as is the case here, one can model at a much higher level of abstraction.

The pilot has access to abstracted information such as the position of the tank, in which direction the tank is facing, what speed it is going at, and at what angle the cannon is currently positioned. (Note that in the described paper, the cannon is called turret.) Furthermore it can decide where an enemy attack originates from.

A tank pilot (or a computer player) pursues a specific high level goal and performs actions that work towards the achievement of that goal. High level goals usually remain the same as long as no significant changes in the tank's state or in its environment occur.

Some significant events can only be detected or calculated based on the state of several tank components, this is done by the analyzers.

2.2. Memorizers

A tank pilot does not only react to current events, but also makes decisions based on events/state from the past. In order to remember interesting state or events for future strategical decisions, we need to add state to the model that acts as the tank pilot's memory. While occurences of events can be memorized using enumerations and booleans, it can also easily be done with the use of states a statechart. Remembering complex state however, for instance geographical information, is less trivial, and usually requires the construction of an elaborate data structure that stores the state to be remembered in an easy-to-query form.

2.3. Strategic Deciders

At this point events come in based on the environment, current state of the tank and memory, which make it is possible to model the high level strategy of the tank pilot. At the highest level of abstraction, a tank pilot switches between different operating modes based on events. He starts in exploring mode, and switches to attacking mode once the enemy position is known. This can be added with information on fuel level and sustained damage. If the fuel level is low and the position of a tank station is known, the

pilot will move to there. If none is known, it will have to continue exploring until it finds one. The same counts for heavily damaged tanks. When during an attack the sustained damage gets too high, the pilot will flee and look for a repair station. All these mode changes are announced using events, so the following layers can handle them.

2.4. Tactical Deciders

The high-level goals sent by the pilot strategy component have to be translated into lower-level commands that can be understood by the different actuators of the tank, such as the motor and the cannon. This translation is not trivial, since it can require complex tactical planning decisions to be made. In addition, the planning should take into account the history of the game, which means it needs to consult the memorizers for important game state or events that happened in the past.

For each strategy there should be a different planner component. The exploring planner for instance will need to generate destinations to explore, while the attack planner will try to follow the enemy, aim at it and shoot it. When new destinations are set, a pathfinder component will use information previously gathered concerning the environment to generate an efficient route.

2.5. Executors

The executors map the decisions of the tactical deciders to events that the actuators can understand. The mapping of events is constrained by the rules of the game. A steering component translates the route given by the pathfinder into events which low level components such as the motor can understand.

2.6. Coordinators

Executors individually map tactical decisions to actuator events. This mapping can result in inefficient and maybe even incorrect behavior when the effects of actuator actions are correlated. In such a case it can help to add an additional coordinator component that deals with this issue.

For example, while attacking, the turret should turn until it is facing the enemy tank and then shoot. However, the optimal turning strategy depends on whether the tank itself is also turning or not.

2.7. Actuators

The actuators are part of the last layer in the event flow. They accept simple commands such as moving the tank forward or rotating the cannon.

3. Models

A component with modelled behaviour consists out of a controller, a static part and a dynamic part. The dynamic part is essentially the execution of the compiled statechart code, the static part contains stored information, and the controller is an interface to these two. The statechart can update values of the static part through this interface, while the static part can generate events for the statechart.

To determine whether it's beneficial to model a component's behaviour, some reasoning and perhaps a simple test suffices. It is important to know that the performance highly depends on the used statechart compiler and that you could get very different results with other compilers.

After some testing it seems that Feng's statechart compiler doesn't handle transitions very well if they are only triggered by guards instead of events. This is rather unfortunate since this kind of transitions is very desired in the modelling of our AI. The number of NPC's that can be spawned at the same time, now highly depends on the specifications of the computer.

Furthermore it was concluded that only providing a model for the behaviour of a player controlled tank, a computer controlled tank and the environment as a whole seems a good choice. In later sections will be showed why. At any given time where a class is mentioned, it talks about a class in the example game if not explicitly mentioned else wise.

3.1. Environment

The class *TanksField* represents the environment. As you can see in figure 3 it repeatedly updates all objects in the game. For example it updates the position of the bullets flying around as well as collision detection, which brings up the question again of when to use statecharts. We could've easily made another statechart for the behaviour of a bullet and let it update its position on receiving a certain event from the *TanksField*. This gives exactly the same behaviour as calling a method on the bullet which directly updates it position, with the difference that the latter is way more performant (imagine hundreds of bullets each waiting for events) while there is absolutely no benefit using a statechart.

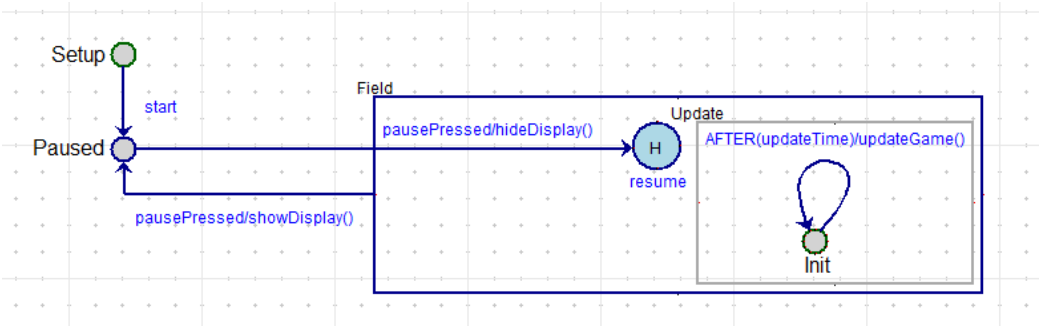

Figure 3: Model of *TanksField*.

Moving on we also see that the statechart is used to pause the game and at the same time display a message using the call *showDisplay()* on the controller. For the reader who isn't familiar with DCharts in AToM^3, gray boxes are orthogonal states, blue are composite states and a green border indicates the initial state. When we arrive at the tank-statecharts only certain orthogonal components will be shown since the complete model is too large and contains many similar submodels.

On a sidenote it should be said that *updateGame()* should actually send events to the tanks for updating purposes, but since this truly destroyed the performance in our example game for some unapparent reason, it is implemented else wise.

3.2. Player

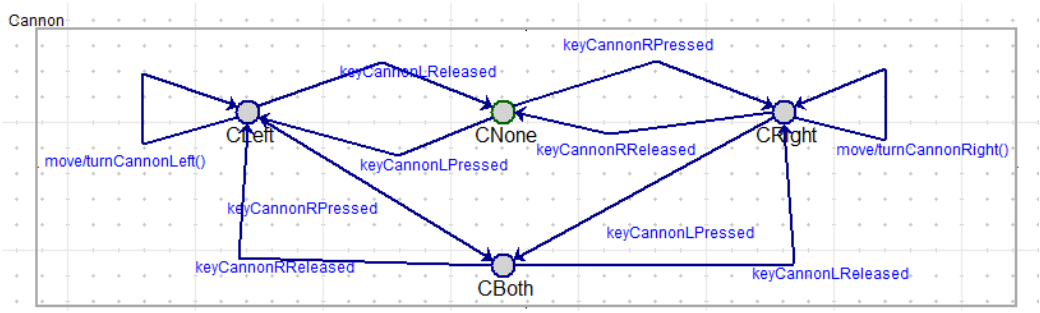

Figure 4: Submodel of *PlayerTank* which controls the rotation of the cannon/turret.

In *PlayerTank* happens all input handling that controls the tank. In figure 4 we see how controlling the cannon works. For instance, when the

left arrow key is pressed an event *keyCannonLPressed* is generated and will bring the statechart in state *CLeft*, which has as meaning that the cannon is turning left. Now each time a *move* event is recieved the cannon will be slightly rotated to the left. This *move* event is used to keep a consistent speed for moving and turning.

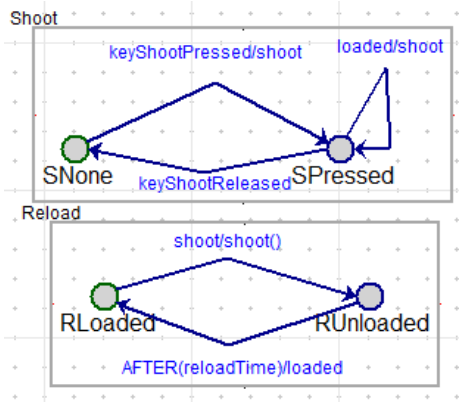

Figure 5: Submodel of *PlayerTank* which controls the shooting.

In figure 5 we see what happens when the shoot key is pressed, it generates a *shoot* event which will call the *shoot()* method if the tank is not in the reloading state. If a shot is fired, the tank reloads, and generates a *loaded* event when its ready. If the shoot key is still pressed, a new bullet will be shot.

4. Computer Controller Tank

The AI is modelled using the information gathered from the related work, which means we will also have the same layered approach. Since some components are very trivial or similar to others we will only talk about the more interesting ones.

In figure 6 we can see how the NPC checks, using info gathered form the state and sensors, whether or not he can see an enemy and sends an appropriate *enemySighted* or *enemyOutOfSight* event down the flow.

In figure 7 we see a memorizer where the NPC combines information to pinpoint the position of the enemy. If an enemy gets sighted, we advance to the *Tracked* state where we repeatedly check if the position of the enemy has changed. If so a *newDestination* event is sent with as destination the

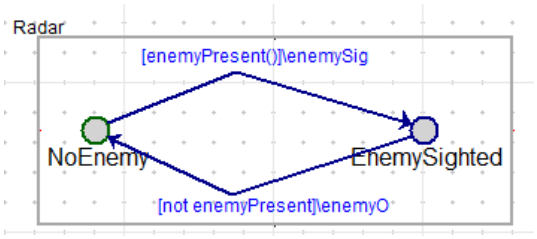

Figure 6: Radar

new position of the enemy. The pathfinder component, which we will discuss later, will then generate points through which the tank can arrive at the destination. If the enemy tank gets lost, the pilot will still move to the last known location increasing the chance of finding the enemy again. If no more points to navigate to are left, meaning no more new knowledge about the enemy has come in, the enemy is considered lost and an *enemyLost* event will be generated putting the *Enemy Tracker* back in the *EnemyLost* state.

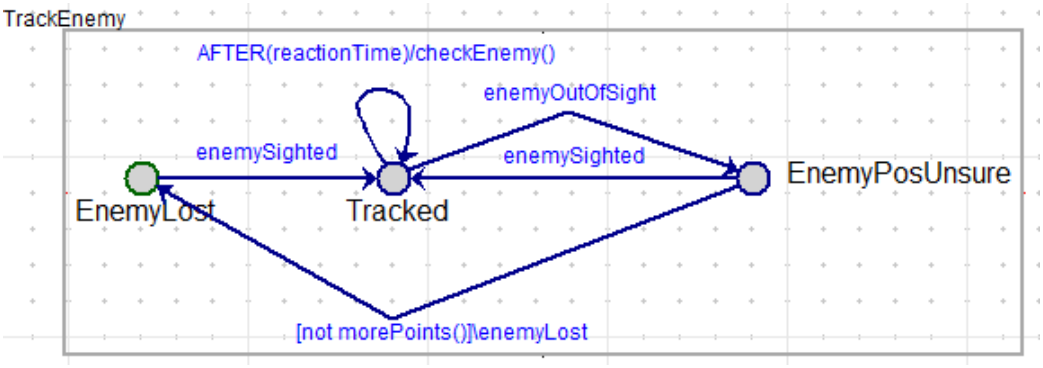

Figure 7: Enemy Tracker

In figure 8 we see the *Pilot Strategy* of our NPC. Here we actually combined the strategic decider with the tactical decider components for planning exploration and attack. We can see that our NPC starts exploring the map, generating new explore destinations when the previous one has been reached.

When an *enemySighted* event comes in the tank will go in attack modus. From that moment on he will try to follow and aim at the enemy by changing its target (for the cannon) and destination every time a *enemyPosChanged* event comes in. If the cannon is loaded and pointed directly at the target, a bullet is fired. This goes on until the enemy is lost which brings the tank

back to exploring.

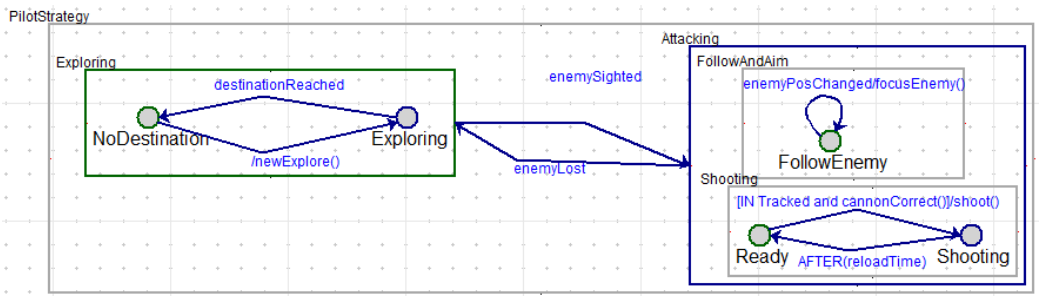

Figure 8: Pilot Strategy

In figure 9 we see the *Path Finder* component. This tactical decider determines an optimal route using points when a *newDestination* event is received. When receiving a *pointReached* event the component will look if there are any more points left in the route, if so a *newPoint* event is sent. If not, we can conclude that the destination is reached and send an event accordingly.

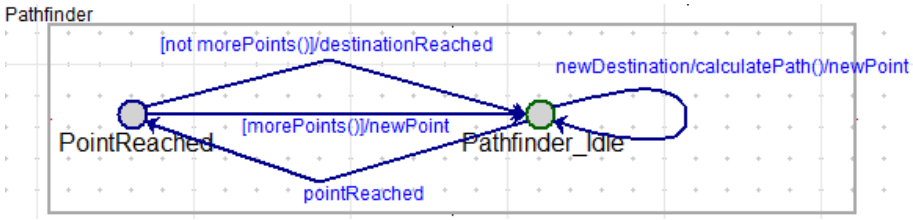

Figure 9: Path Finder

In figure 10 we see the *Steering Strategy* component. This executor will shoot in action when a new target point is set. It checks where that point is located in relation to itself. It then propagates events to adjust it's position, angle and cannon angle to the actuators. After a defined reaction time it will repeat this process because the relation between the tank and the target point will have probably changed.

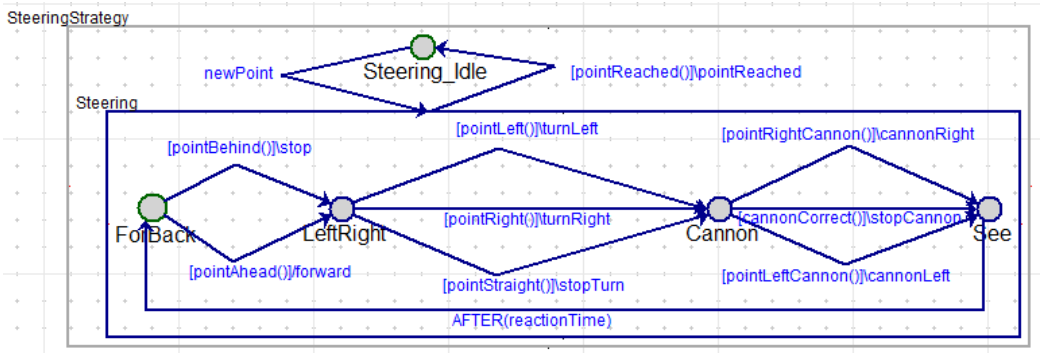

Figure 10: Steering Strategy

In figure 11 we see a combined coordinator and actuator for the rotation of the cannon. The actuator part is very similar to the components we saw in *PlayerTank*. The extra purpose of the coordinator part is to get the cannon back in a straight position after an attack. Once in the right position it goes back to the idle mode until another enemy is sighted.

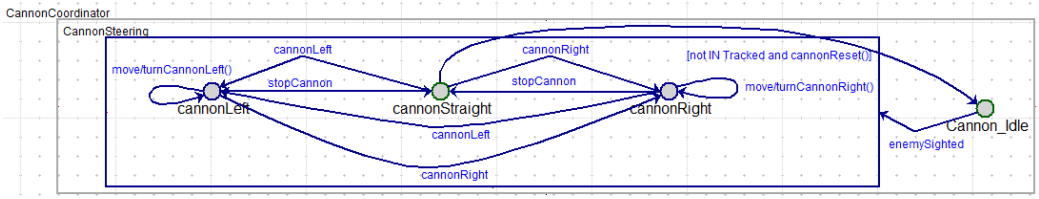

Figure 11: Cannon Coordinator and Actuator

5. Conclusion and Future Work

Modelling with statecharts definitely is a good way to obtain complex AI in a structured and easy-to-understand manner. Also in general when keeping track of certain states is needed, like for instance what key is being pressed at the moment or whether or not a game is paused, statecharts tend to be useful. However they do degrade performance and it is recommended to not use them for every possible use-case.

Future work could consist of extending the AI model and thus making the NPC more intelligent but this paper only had as goal to test the ability of statecharts which we already have completed.

References

[1] J. Kienzle, A. Denault, H. Vangheluwe, Model-based Design of Computer-Controlled Game Character Behaviour.

[2] AToM^3, `http://atom3.cs.mcgill.ca/`.

[3] H. Feng, DCharts, a formalism for modeling and simulation based design of reactive software system, `http://msdl.cs.mcgill.ca/people/tfeng/thesis/thesis.html` (2004).