
**UNIVERSITE PAUL SABATIER**
**Formation Doctorale en Informatique**
**Master 2 Recherche IT (parcours AVI)**

**Année 2008/2009**
**Laboratoire IRIT**
**Equipe VORTEX**

# Machine learning by implicit imitation in virtual worlds

Ahmad ABDUL KARIM

**Directeur de recherche : C. Sanza**
**Responsable de stage : P. Torguet**

**Keywords:** Machine learning by reinforcement, implicit imitation, fuzzy logic, learning system classifiers, Virtual world, MMORPG.

**Abstract:** In on-line persistent virtual worlds like MMORPG's: Massively Multiplayer Online Role Playing Game, users control their avatar when they are connected, but this avatar disappears from the world when they disconnect. The goal is to allow to an avatar to continue evolving in the virtual world when the user is disconnected, by reproducing as faithfully as possible the actions of the human controlling it. We propose the use of the machine learning by implicit imitation concepts to generate a behavior similar to that of the user.

**Mots Clés :** Apprentissage par renforcement, imitation implicite, logique floue, système de classeurs, Monde virtuel, MMORPG.

**Résumé :** Dans les mondes virtuels on-line persistants comme les MMORPGs : Massively Multiplayer Online Role Playing Games, les utilisateurs contrôlent leur avatar lorsqu'ils sont connectés, mais cet avatar disparaît du monde quand ils déconnectés. L'objectif est de permettre à un avatar de continuer à évoluer dans le monde virtuel lorsque l'utilisateur est déconnecté, et ceci en reproduisant le plus fidèlement possible les actions de l'humain qui le contrôle. Nous proposons l'utilisation des concepts d'apprentissage par imitation implicite pour générer un comportement proche de celui de l'utilisateur.

# Table of Contents

# Table of Figures

# Chapter 1: Introduction

*L'industrie des jeux vidéo est l'un des industries qui ne s'arrête pas de grandir depuis 1958 jusqu'à maintenant, et pour maintenir cette succès les développeurs des jeux vidéo ont été obliger de rechercher d'autres façon de modéliser le comportement des personnages non-jouables ou les PNJ (comme les ennemis), autres que les vieilles méthodes de la création des ennemis inintelligents qui ne peuvent marcher que sur un chemin prédéterminé avec peu d'interactions, pour ça les concepts d'intelligence artificielle ont commencé à apparaître dans les jeux vidéo pour créer des PNJ avec de la personnalité, des environnements vivants, et des mondes virtuels immersifs où les joueurs sentent qu'ils sont en interaction avec de vrais humains, pas des personnages scriptés.*

The video game industry is one of the most ever growing industries these days, it used to be for niche markets only, but now it is for mainstream markets: you can find a video game related products almost everywhere. It all started in October 18, 1958 when William Higinbotham, a nuclear scientist, created *Tennis For Two*, it was never commercially released because it was not meant to be an actual video game, more of an electrical experiment, but it was one of the first forms of a video game.

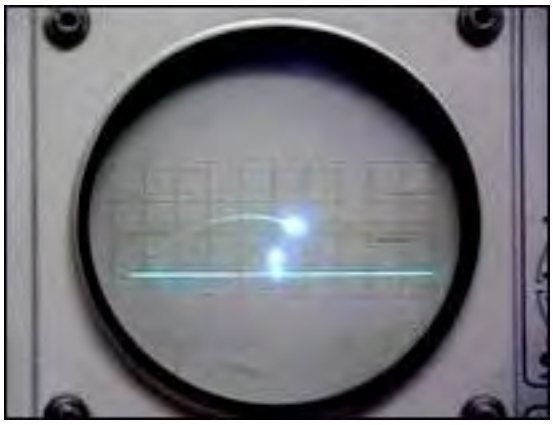


**Figure 1: Tennis For Two**

Another milestone and one of the first video games ever was called *Pong*, which is a two-dimensional sports game that simulates table tennis. Created by Allan Alcorn in 1972, you could only play it on the arcade machines, until 1975 when Atari, Incorporation created a home version of it that was sold to the mass public, *Pong* sales soared when the unit was released in that year; it was one of the first home consoles ever.

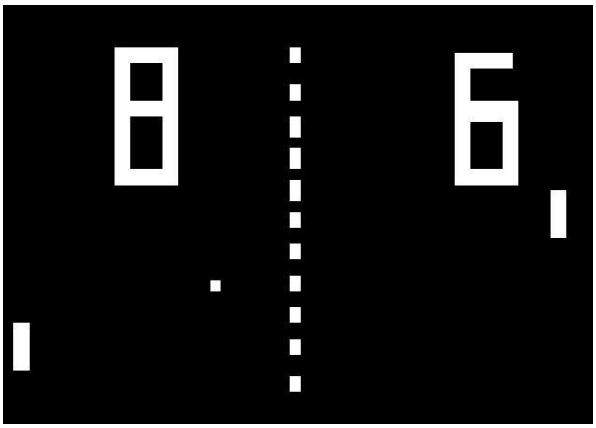


**Figure 2: Pong**

From that day on, there was an explosion of new video games (*Pac-Man*, *Mario* series, *Metal Gear* series…), new consoles (Super Nintendo Entertainment System, PlayStation, Xbox…) and new companies constructed around video games development (Electronic Arts, Ubisoft…), all of them trying to adapt to the demands of the consumers. Video game industry, has another effect: it is one of the driving forces of the most of the new technological advancement in CPU's, GPU's, sound cards, storage devices like CD-ROM's, DVD's….because video games are always demanding more and more of raw power in order to achieve the desired quality.

At the beginning of video games era it was acceptable to have a few squares on the screen to represent the user character and the environment, or enemies that move on a simple path aimlessly with really simple behavior (attacking the user character directly without any kind of intelligence).
On the other hand these days video games are more immersive by replicating real environments, with complex stories, and engaging game play like (*Crysis* on the PC, *Gears of war 2* on the Xbox 360, *Killzone 2* on the PlayStation 3)


**Figure 3: Crysis**

The improvements on video games were not only in creating more photo realistic games, because gamers started demanding more engaging game play and environments, they wanted to feel that they are in a living and breathing environment, and since those gamers are the ones buying the video games and consoles, developers were forced to search for other ways to model the behavior of the Non-Playable Characters or NPC's (like the enemies) other than the old methods of creating unintelligent enemies that can only walk on a pre-determined path with limited interactions, that is when the Artificial Intelligence concepts started to gain momentum in order to challenge the players with smart enemies, to make NPC's with impressive personalities and to create vivid environments…and by that developers were capable of luring the gamers into buying their games with the promise of more immersive virtual world where gamers will feel that they are interacting with real humans rather than brainless scripted characters.
So artificial intelligence is becoming a really important component in video games, to model complex behaviors and to create that immersive environment the gamers will interact with it more naturally.

In the first chapter of this paper I will shed light on the different kinds of algorithms used in the artificial intelligence domain, how they are used in the industry in general and more specifically in video games, this chapter will also contain some of the most influential imitation experiences that I based my work on, while in the second chapter I will start talking about my own machine learning by imitation experience, with some interesting results and conclusions.

# Chapter 2: State of the Art

*Il existe deux grandes familles de techniques pour résoudre les problèmes d'IA en général, et les problèmes d'apprentissage spécifiquement, elles sont :*

- *Les méthodes ascendantes : le système utilise la connaissance complète d'un problème donné, avec toutes les situations possibles, les actions et les récompenses, ainsi que parfois la connaissance d'une expert, pour créer ce système de haut niveau qui est capable de résoudre certains problèmes de manière efficace, exemple de telles méthodes : machine à états finis, arbres de décisions, systèmes experts, réseaux de neurones…*
- *Les méthodes descendantes : le programmer fournit le système avec des outils suffisamment efficaces pour résoudre le problème, comme : algorithmes de planifications, outils de traitement de l'image, certaines méthodes de la robotique…, mais sans lui donner la solution explicite, et le système a besoin de combiner les outils et les utiliser bien pour trouver une solution, exemple de telles méthodes : algorithmes génétiques, apprentissage par renforcement…*

*Les méthodes descendantes vont nous aider à créer notre système d'imitation, surtout les systèmes de classeurs qui sont un mélange des techniques d'apprentissage par renforcement et des algorithmes génétiques.*

## 2.1. Techniques

There are two main families of techniques for solving AI related problems in general and Machine Learning problems in specific, they are: the *Top – Down* methods and the *Bottom – Up* methods.

In the *Top – Down* methods, the system uses the complete knowledge of a given problem, with all the possible situations, actions and rewards, plus sometimes the knowledge of an expert, to create this high end system which is capable of solving certain problems effectively. So it is deterministic, because every possible situation is handled and a proper action is assigned to it, and the behavior of the system is predictable, but because it is deterministic, the system designers need to know this *Complete Knowledge* and that is not always possible, moreover, the system is static, so the idea of evolution is not always present.

In the *Bottom – Up* Methods, the designer gives the system enough tools to solve the problem, like: path finding algorithms, image processing tools, some robotic methods…, but without giving it the solution explicitly, and the system needs to combine these tools and use them properly to find a possible solution, so it needs to be dynamic enough to cope up with the different problems and be capable of using the right tools at the right situation, so the designer do not have to assign to every single situation the proper action explicitly, and the system at the end is dynamic enough to be used in totally different environment with only little changes in the main structure. But one of the disadvantages of this method is that finding the solution is not always guaranteed, and converging for a solution is sometimes slow.

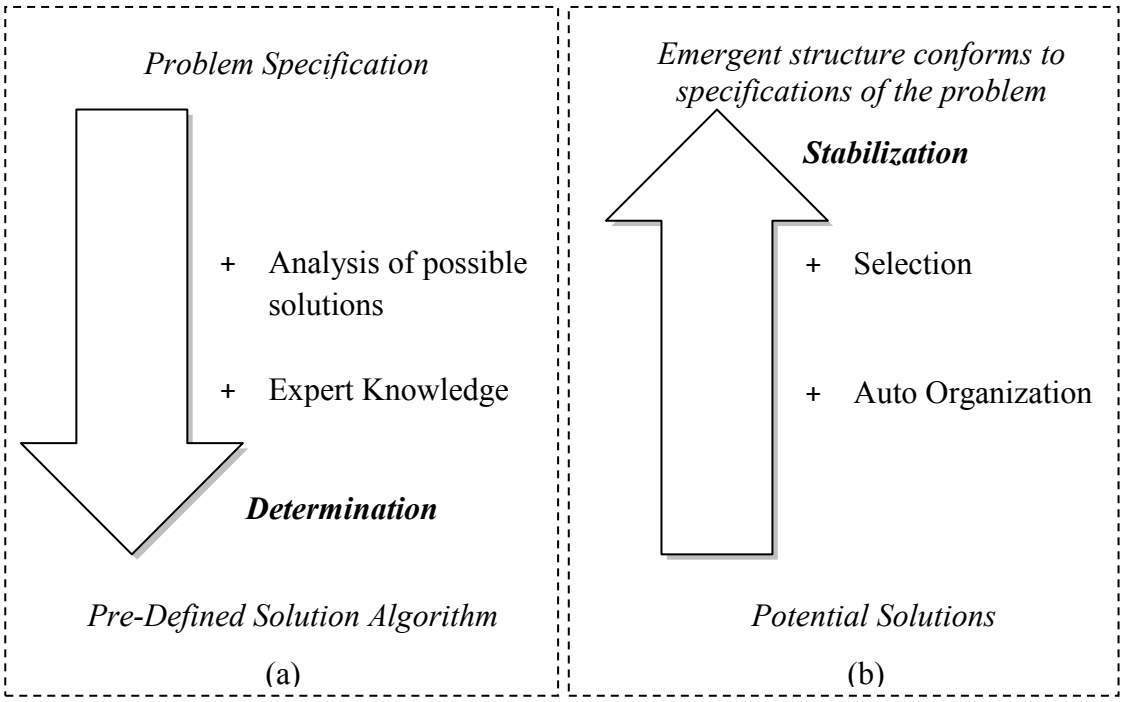



**Figure 4: AI Techniques**
**(a) The "Top-Down" approach, (b) The "Bottom – Up" approach.**

## 2.1.1. Top-Down methods

### 2.1.1.1. Finite State Machine

A Finite State Machine or (FSM) is an oriented graph with *States*, and *Transitions* between those states plus *Actions*.

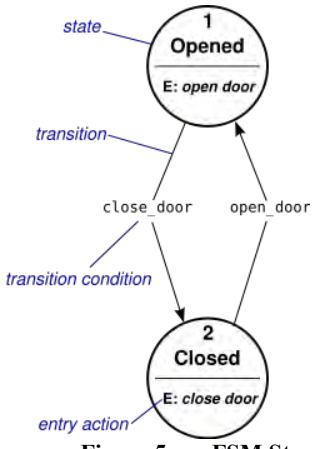



**Figure 5: an FSM Structure**

A *current state* is determined by past states of the system, and it reflects the input changes from the system start to the present moment. A *transition* indicates a state change and is described by a *condition* that would need to be fulfilled to enable this *transition*. An *action* is a description of an activity that is to be performed at a given *state*.

We can find FSM implementation in a vast number of applications, like in language processing tools: Spell Checking, Stemming, recognizing regular grammars…and in Pattern recognition, Classification and, of course, Video Games.

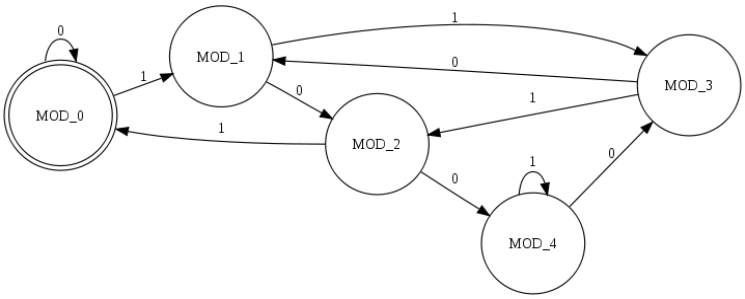



**Figure 6: Example of an FSM**
**Capable of detecting if a binary number is divisible by 5, the double circled state is the accepted one.**

The Turing machine which was described in 1936 by Alan Turing (Turing, 1936), is the base model of the current finite state machine, the only difference, is that in the Turing machine there is an extra tape to save the previous states, and the previous characters encountered, so the transitions depends on this tape and on the input at the same time, while in the current FSM model, what matters is the current state and the current input only, so it does not memorize the previous states.

There is no specific algorithm to construct a finite state machine because at the end it depends on the environment, system designer and/or system specification…but the resulting FSM is always a state graph with the possible transitions and the actions associated.

#### 2.1.1.2. Decision Tree

Decision Tree is a part of: *Knowledge Discovery in Databases* (KDD) research domain, which refers to the broad process of finding knowledge in databases, or in other words: *Data Mining*.

A decision tree is a tree-like graph or model of decisions and their possible consequences, including event outcomes, resource costs, and utility.

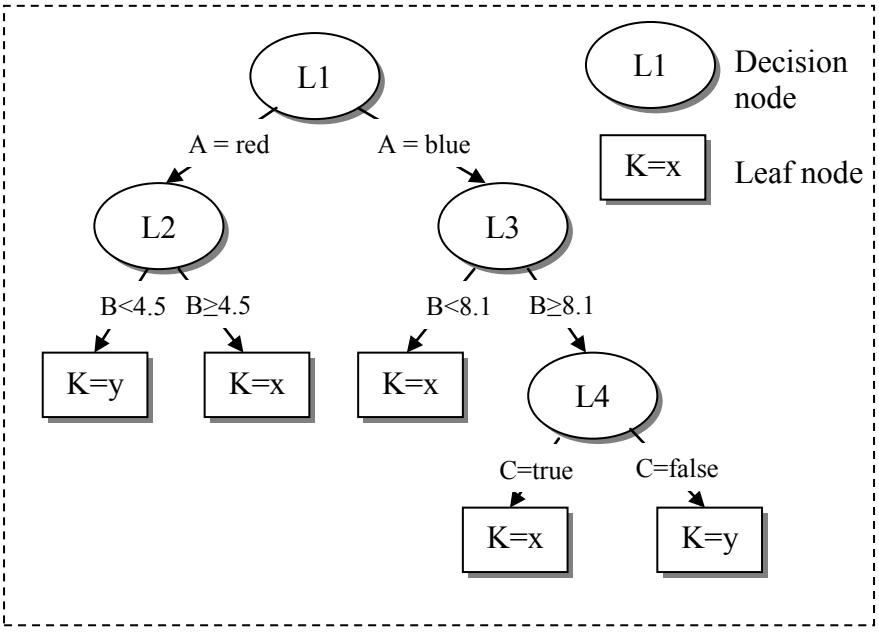



**Figure 7: Example of a Decision Tree**

Decision trees are commonly used in operational research (project planning, designing the layout of a factory for efficient flow of materials…), especially in decision analysis to help identify the strategy which most likely will reach the goal.

One of the most famous algorithms to build such decision tree is called *ID3* (Quinlan, 1986) with its extensions like *C4.5* (Quinlan, 1993) and *C5.0*. The basic idea is the same: the tree is constructed from a set of training data using the concept of information entropy, where the algorithm tries to choose the attribute that will potentially minimize the tree depth and create the purest leaves, i.e. containing majority of examples with the same class. These kinds of algorithms are used in Machine Learning problems for creating a classification tree out of the observations.

#### 2.1.1.3. Artificial Neural Network

An Artificial Neural Network (ANN), often just called a *Neural Network* (NN), is a computational model based on biological neural networks. It consists of an interconnected group of simple processing elements (artificial neurons) that processes information through a simple computational approach: the propagation of the information in an interconnected network of simple processing units.

The ANN's can take endless structures based on the application we want. The three main layers are: the input neurons, the output neurons and the hidden layers; the flexibility of the ANN's is in these hidden layers, where we can use multiple layers each one of them consists of the number of neurons that we want, and all of these neurons are interconnected with each other to form the network, this structure is optimized during the learning phase.

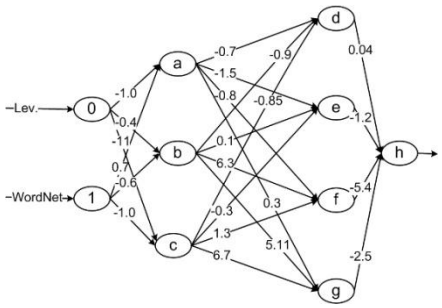



**Figure 8: Example of an ANN**

The power of the ANN's comes from its ability (with the right structure) to learn everything the user wants it to learn, where during this learning phase the weights and the values of the connections between the neurons changes to achieve the minimum error committed between the output of the network and what the user excepts the output to be.

So Neural Networks are non-linear statistical data modeling tools. They can be used to model complex relationships between inputs and outputs or to find patterns in data, or (and most importantly) learn anything the user wants it to learn.

The ANN model that we have been talking about is called the *Feedforward Neural Network* which is inspired form the way the human mind works and store data.
The first artificial neural network was invented in 1958 by psychologist Frank Rosenblatt (Rosenblatt, 1958). Called *Perceptron*, it was intended to model how the human brain process visual data and learn to recognize objects.

#### 2.1.1.4. Expert Systems

An Expert System, also known as a *Knowledge Based System* (KBS), is an attempt to reproduce the performance of one or more human experts, most commonly in a specific problem domain like in the medical domain, in math or engineering… and it consists mainly of 2 main components: the so-called *Knowledgebase* and the *Inference Engine*.

The *Knowledgebase*: it is the digital representation of the human expert knowledge in the form of a series of rules, each one of these rules consists of: a condition and an action

An example of such rules, in the medical domain:

*If* symptom S1 and Symptom S2 *Then* Disease D7
*If* symptom S8 and Symptom S3 and Symptom S1 *Then* Disease D6

This Database could contain sometimes from 10 rules to even more than a 1000, depending on the subject being treated.

The second most important component is the *Inference Engine*: it is the brain of the system with a specific set of goals: getting answers from the *knowledgebase* or formulating new conclusions…using a simple process: the system asks the user some simple questions (sometimes simple Yes/No questions) and from the answers obtained plus the *Knowledgebase*, the *Inference Engine* tries to find the optimum rules that correspond to the user input, and after a certain time, it gives him the best conclusion that it found.

The power of the Expert Systems comes from what is called *Certainty Factors*: whenever an expert writes a set of rules he supplies the system with a percentage to indicate how much he is confident about each one of these rules, and the *Inference Engine* uses these Certainty Factors to select the optimum rules to be used, and at the end of the process when it supplies the conclusions, it provides them with some new Confident Values deduced from the database.

One of the first expert systems created was called *DENDRAL*; it was invented at Stanford University in the 1960s. The development of this system started in 1965; it uses the mass spectra or other experimental data together with a *Knowledgebase* of chemistry, to produce a set of possible chemical structures that may be responsible for producing the data.

Another famous system was called: *MYCIN* it was based on *DENDRAL* and developed in the 1970s at Stanford University as well, its *Knowledgebase* consists of ~600 rules, it would query the physician running the program via a long series of simple yes/no or textual questions, and At the end, it provides a list of possible culprit bacteria ranked from high to low based on the probability of each diagnosis, with its recommended course of drug treatment.

### 2.1.2. Bottom-Up" methods

#### 2.1.2.1. Genetic Algorithm (GA)

It is an algorithm that mimics evolution and natural selection to solve a given problem, it is based on the concept of the *Survival Of The Fittest* introduced by Dr. Charles Darwin in his famous book *On the Origin of Species* first published on November 1859, where he talked about the idea of the natural selection process in our world and he made some key observations and some interesting theories, that became the main concepts in the GA's:

- Populations remain roughly the same size, with small changes.
- Individuals less suited to the environment are less likely to survive and less likely to reproduce, while individuals more suited to the environment are more likely to survive and more likely to reproduce.
- The individuals that survive are most likely to leave their inheritable traits to future generations.
- This slow evolution process results in populations that adapt to the environment over time, and ultimately, after interminable generations, these variations accumulate to form a new varieties, and ultimately, new species.

Now, each iteration of the evolution consists of 3 main stages: *Evaluation*, *Selection* and *Modification*. The Genetic algorithm starts with an initial population either created randomly or with a method related to the problem domain, each individual of this population is an abstract representation of a candidate solution in the form of a chromosome (a series of genes), the most sample representation is a series of bits.

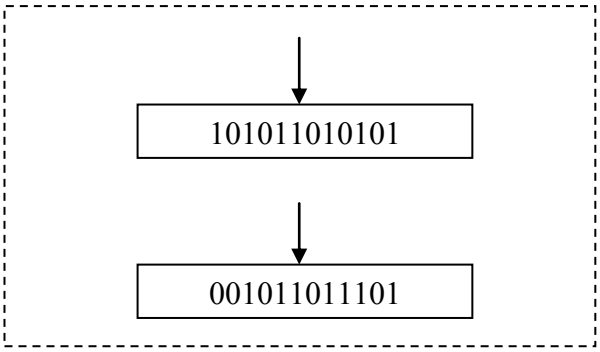



**Figure 9: 2 examples of an individual (chromosomes)**

*Evaluation*: a fitness function is used to evaluate every individual of the population (sometimes we use several functions for this evaluation); this fitness function is based on the problem we need to solve, and it gives a numerical value for each individual.
This numerical value could be the error committed by this chromosome; in this case the Genetic algorithm will try to minimize it, or it could be the gain expected; in this case the GA will try to maximize it.

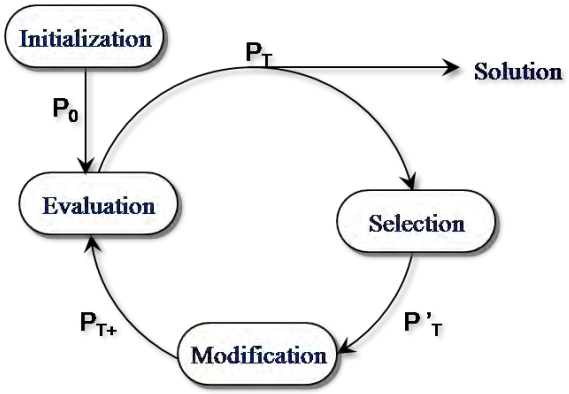



**Figure 10: an example of the iterations in a Genetic Algorithm**

*Selection*: the algorithm will choose a proportion of the current generation to breed and create a new generation; this selection is based on the fitness value, where fitter solutions are more likely to be selected, the most famous and well studied methods for selection are *Tournament selection* and *Roulette Wheel Selection*:

- In *Tournament selection* a K number of individuals are chosen randomly from the current population and the best individual (based on its fitness) is kept, this is repeated several times until we get the number of individuals needed.
- In *Roulette Wheel Selection*: it uses the idea of the Roulette in a casino. Each individual is represented on this roulette by a section proportional to it fitness function (the fitter it is, the bigger the sector is), and we do a probability simulation of the balls spinning for the selection.

*Modification*: now it is time for generating the next generation using the individuals selected by the previous step, using two operations *Crossover* and *Mutation*:

- *Crossover*: it mimics the way the genetic attributes are transferred from parents to children: a Crossover point is chosen (sometimes several points) on the chromosome and the genes are swapped between the two parents,

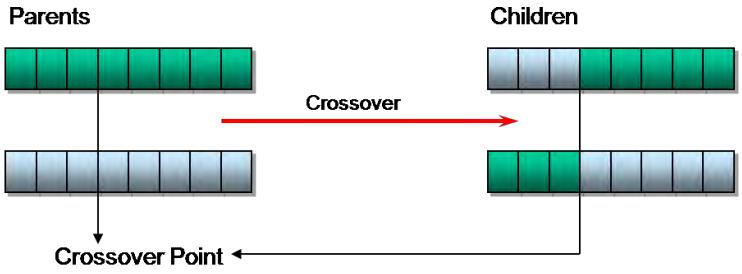



**Figure 11: Crossover**

The *Crossover* operation is regarded as a way of preserving the best genes for the next generations.

- Mutation: A random gene is chosen (sometimes several) on a random parent and will be mutated (changed) into other values.


**Figure 12: Mutation**

The idea behind the mutation is to maintain the diversity in the population and to avoid local minima by preventing the population of chromosomes from becoming too similar to each other (if we only used the crossover operator).

*Termination*: there are several criteria for termination:

- A solution is found that satisfies minimum criteria.
- Fixed number of generations reached.
- Successive iterations no longer produce better results.
- Manual inspection.
- Or a Combinations of the above.

GA's are useful and efficient when:

- The search space is large, complex or poorly understood.
- Domain knowledge is scarce or expert knowledge is difficult to encode.
- No mathematical analysis is available.
- Traditional search methods fail.

Like *Travelling Salesman Problem*, *Game Theory*, *and Bioinformatics*…

Genetic Algorithm is part of evolutionary algorithms (evolutionary computation), that was first introduced in 1954 by Nils Aall Barricelli, who was using the computer at the Institute for Advanced Study in Princeton, New Jersey, but this kind of algorithm became more common in the early 1960s, and the methods were described in books by Fraser and Burnell (1970) and Crosby (1973), Fraser's simulations included all of the essential elements of modern genetic.

#### 2.1.2.2. Reinforcement Learning

Reinforcement Learning is based on the idea that an agent (or an artificial system) is always in interaction with the environment surrounding it, so it can perceive the *State* of the environment, influence it by executing an *Actions* (also called: a Decision or a Control), and at the end, expects to receive some kind of *Reward*.

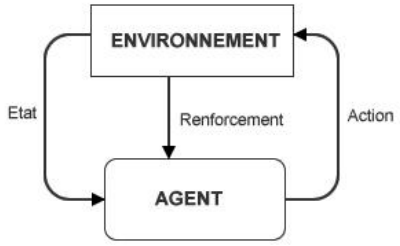



**Figure 13: Reinforcement Learning Cycle**

In this kind of systems, the long-term goal of this agent is to select the best action, based on the current state, which will maximize his reward system, so he always tries to map the world State with an Action that he thinks will get him the maximum Reward.

The environment is typically formulated as a finite-state Markovian decision process (MDP) (Bellman, 1957), which is defined as a tuple <S, A, T, R >:

- S : a finite set of States or possible worlds
- A: a finite set of Actions.
- T: S x A → P(S), a state Transition function, where P(S) is the probability distribution of all the possible states, so it models the probability to be in a new state (*s`)* when we execute the action (*a)* in the state (*s)*.
- R: S x A x S → R, a Reward function.

So on every time *t*, we are in a state *s(t)* from S, and the agent chooses the action *a(t)* from A, after executing this action we are in a new state *s(t+1)* (based on the transition function), and the agent gets a reward *r(t+1)* = R(s(t), a(t), s(t+1)), So the new state depends only on the old state and the action taken by the agent.
Most of the time, the agent gets his reward when he achieves his objective, so in order to achieve it he may executes numerous successive actions, but sometimes he will not get the reward until the execution of the last action.

There are two kinds of MDP's:

- Deterministic: where for each couple of (action, state) there is only one single possible transition to a new state.
- Non-Deterministic (or stochastic): where for each couple of (action, state) there is a list of transitions to new states.

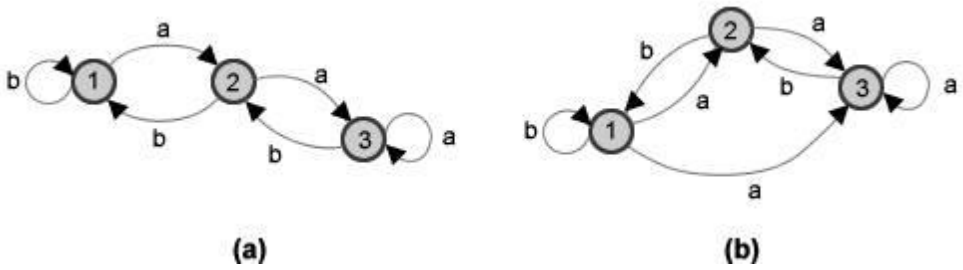



**Figure 14: 2 examples of an MDP**
**S = {1, 2, 3}, A = {a, b} (a) Deterministic, (b) Non-Deterministic**

What makes this kind of system different form the other Machine learning techniques is that: it does not use any kind of correct input/output examples, neither some kind of explicit action correction method; so everything is On-line and depends directly on the case being studied and on the environment that surrounds the agent, whose only way of knowing the benefits of an action is executing it directly on the environment.

This system is really dynamic, that is why we need to manage the trade-off between the Exploration/Exploitation:

- In Exploration (visiting uncharted territory): the agent always tries new actions, so his choice is not influenced by the reward that he thinks he would get, for this reason it is a totally random choice, which helps the exploration of the whole environment.
- In Exploitation (based on current knowledge): the agent always chooses the actions believed to be optimal, based on the model built by him (the tuple <S, A, T, R>), this help the convergence to a Plan which he thinks it is optimal.

The most common method for achieving this trade-off is the *e-greedy* method in which the agent chooses a random action for a fraction of the time ($e$), where $0 < e < 1$. Typically, $e$ is decayed over time to increase the agent's exploitation of its knowledge.

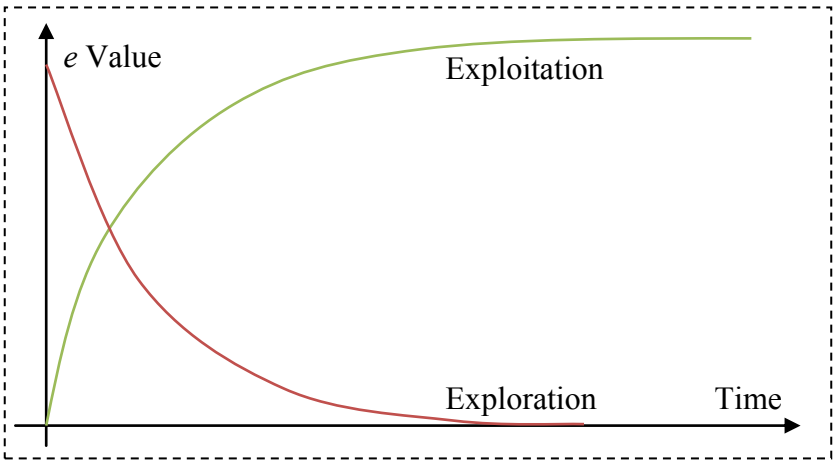



**Figure 15: Exploration / Exploitation using the e-greedy algorithm**

There are several famous reinforcement learning algorithms and techniques like: Q-Learning, SARSA (State-Action-Reward-State-Action) and most importantly *Learning classifier system* or LCS.

#### *2.1.2.2.1.1. Learning classifier system*

LCS is a machine learning system, which uses reinforcement learning techniques plus genetic algorithms, to create this adaptive system that learns to perform the best action (or to take the best decision) based on the input from the environment surrounding it. The LCS consists of a collection of rules (each called classifier) just like an expert system, with the same structure for each rule <condition, action>, but the main difference between the learning classifier systems and the expert systems lays in the *knowledgebase*: while it is static in expert systems, it evolves in the LCS's based on the reward that the system gets. This is realized by using the genetic algorithms as one of the engines for its evolution.

The LCS was first introduced by John Holland in 1986 (Holland, 1986), and his model is the base structure for most of the LCS's to follow, and it consists of:

- Input interface: it is a set of sensors, and it generates a message which describes the environment that the system can perceive with these sensors.
- *Knowledgebase*: the rules (classifiers).
- Message list: it will hold all the messages from the input interface and from the classifiers as we will see later.
- Output interface: it will execute the chosen action.
- Reward distribution system: it will distribute the reward on the *knowledgebase* rules using algorithms like the *Bucket Bridge* algorithm.
- The Genetic algorithm.

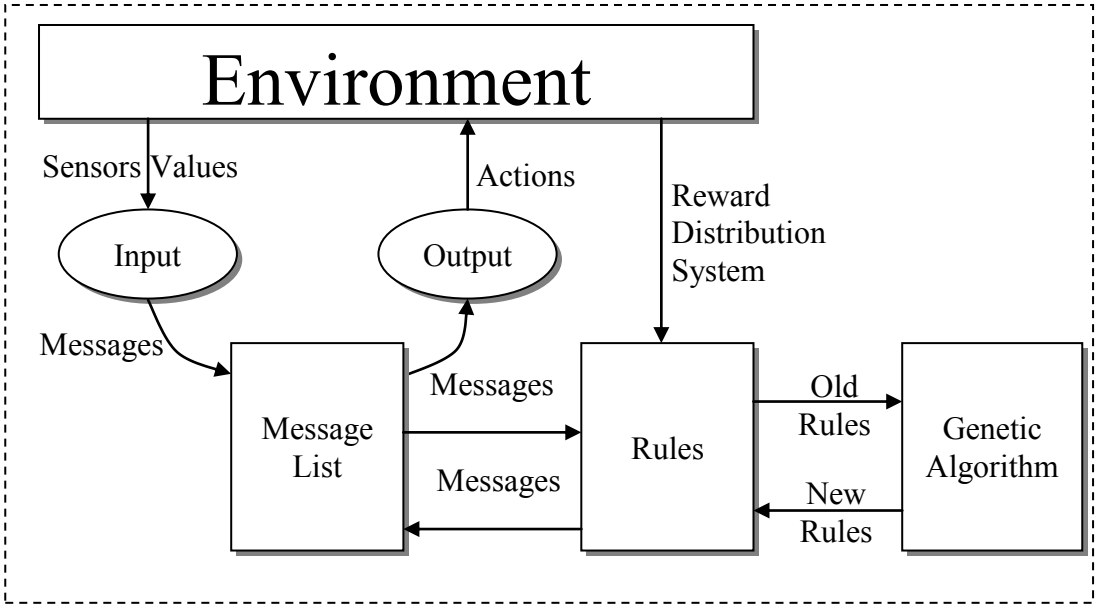



**Figure 16: First Learning Classifier System model**

In Holland model, the rules are binary so both the condition part and the action part (or what he calls the message part) are coded by a list of bits (0 and 1), example of a classifier:

Condition (01011010): Action (10101)

But that is not all; he added to the condition part a new symbol # which means *whatever*: it can take the value of 0 and 1 at the same time, in that way we will have rules more generalized than others; that is when their conditions part contains more #'s.
As an example: The message 010101 can satisfy the following conditions: 010101, 0101#1, ##0101 and even ######.
So the condition alphabet = {0, 1, #} and the action alphabet = {0, 1}, to ease the work of the genetic algorithm during the *Crossover* and *Mutation* process (the evolution process).

The way things work inside this system is quite simple: the input interface will take the value of the sensors and code them in binary, after that this message is compared to the condition part of the rules, if it is fulfilled, then the rule can add its own message to the list of messages and so on, until when the *inference* cycle results in an action (or a decision), thus the output interface will execute it on the environment. After executing this action the system can receive a reward from the environment, this reward will be distributed on the rules that triggered it using the *Bucket Bridge* algorithm.

Bucket Bridge algorithm (Goldberg, 1989)
It is one of the most popular algorithms for the reward distribution process; it is used to share the reward between the rules that triggered the action, it consists of two parts: an auction and the reward distribution:

- Auction: when the condition of a rule is satisfied, and the force of the rule is big enough, it will pay a percentage of its force and place it in the bucket (a tax).
- Reward distribution: when a reward is received from the environment, this reward is added to the same bucket. The bucket content will be distributed on the rules that helped in filling it at the first place, and this sharing could be uniform or based on the time between paying the auction tax and receiving the environment reward.

So the force of each rule will evolve based on the interaction of the system with the environment, but these rules are still static and their condition part and action part are not changing, that is why Holland has introduced the idea of using genetic algorithm; it will take the rules and treat them like a population of chromosomes, and by applying the *Crossover* and the *Mutation* operations we will have new rules that are most of the time better in modeling the interaction with the environment than the starting rules. And he did not have to modify the genetic algorithm, because at the end the rules are coded in binary, and he used the force of these rules as the fitness function, the only modification he did was: adding the # symbol to the mutation operator.

And to make the system more adaptive there are several more enhancements like:

1. If the rule is not activated for a long time it will start to lose a percentage of its force, and at the end if this force is weak enough, it will be deleted from the *Knowledgebase*.
2. If the message sent from the input interface did not fulfill any rule, a new rule will be created randomly to satisfy this message, with a force equal to the average of either all the rules in the data base or the rules which has the same action (message); this is what we call *Covering Operator*.

Based on this LCS system there are three learning classifier systems: the ZCS (Wilson, 1994), the XCS (Wilson, 1995) and the ACS (Stolzmann, 1998). All of them share most of the features of the Holland LCS but with a main difference: they all removed the message list, so the selected rule will execute its action directly.

I. ZCS or zeroth level classifier

It is a Holland LCS but without the message list, other than that it operates in the same way: the input interface send the sensors value to the *Knowledgebase*, then a list called *Matching Set* is created which consists of the rules with the condition satisfied by this input, the rules in the *Matching Set* are grouped based on the actions (so all the rules with the same action will be grouped together) and the action of the group with the highest force will be executed, this group will pay the auction tax and wait for the environment reward to share the bucket contents using the same *Bucket Bridge* algorithm. The ZCS uses the same concept of genetic algorithm and the other enhancements.

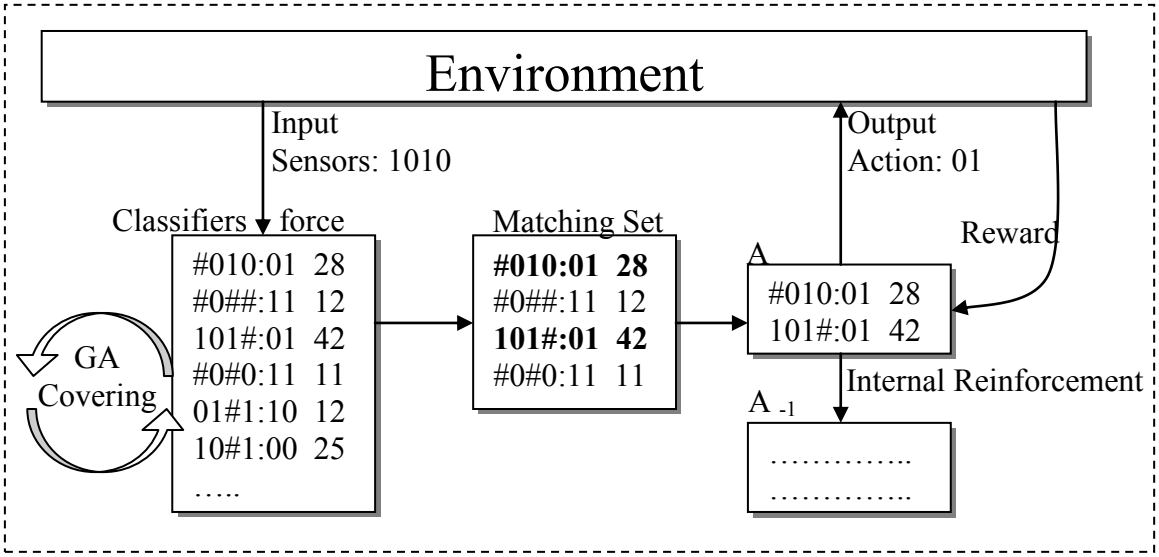



**Figure 17: ZCS**

The ZCS is often called *strength based* LCS, because what is matter is only the strength of the rule, and it has the capability of solving the same problems like Holland LCS and more, even with its simplified model.

II. XCS

It is a ZC, but the force of the rule is replaced by three parameters: the *reward prediction*, the *prediction error* and the *fitness*. The *reward prediction* estimates the reward perceived form the environment when the classifier action is executed. This parameter is used during the action selection. The *prediction error* is computed by comparison between the prediction and the real reward. Finally, the *fitness* measures the prediction accuracy: it is computed in function of the reverse of the error. It is used during the action selection and the genetic algorithm selection.

So the XCS tries always to select the rules with the right prediction not the highest one: because sometimes the rewards that the system gets from the environment are not alike, so a rule can receive a high reward although it was used only once, in a ZCS this rule will be selected more often after receiving that reward although it might not be the best choice (other rules could be the best choice but they did not receive high rewards), while in the XCS this will not happen and the system will try to select the best rule not the one with the highest force, that is why it is called *accuracy-based* LCS.

III. ACS or Anticipatory Classifier System

In this LCS each rule has a third part called the *effect* part, were the rule tries with it to predict the consequence of its action on the environment, so the classifier structure becomes: <Condition, Action, Effect>, this *effect* has the same alphabet of the condition part, that means: it could contains # in it, but here the # means that the value of the sensor will not change.

In the ACS each classifier has two variables: the *reward prediction* and the *anticipation quality*, so the force of the rule, and its fitness, are based on these 2 variables. The *anticipation quality* is always increased or decreased based on the difference between the real value of the sensors after the execution of the action and the *effect* part of the classifier.

Next to these three main LCS's, There are a lot of proposed modifications, like:

- GCS or *Generic System Classifiers* (Olivier & Stephane, 2004): As we saw in all the previous LCS's the classifiers are binary, so we will have to code everything in binary (0 and 1) in order to use them effectively, but that is not always possible, for that there are GCS's where the condition part and the actions are really generic and can contain: integers, ranges, floats and sometimes strings… which makes the rules easier to read and easier to code, but in this case we will have to modify the GA's and generalize the evolution operators in order to use them on these generic values.
- MCS or *Multiple Classifiers System* (Kam Ho, 1992): this system is composed of several LCS's of the same or sometime different types, in it, some LCS's tries to model a behavior, while other LCS's are the one choosing which LCS to be activated

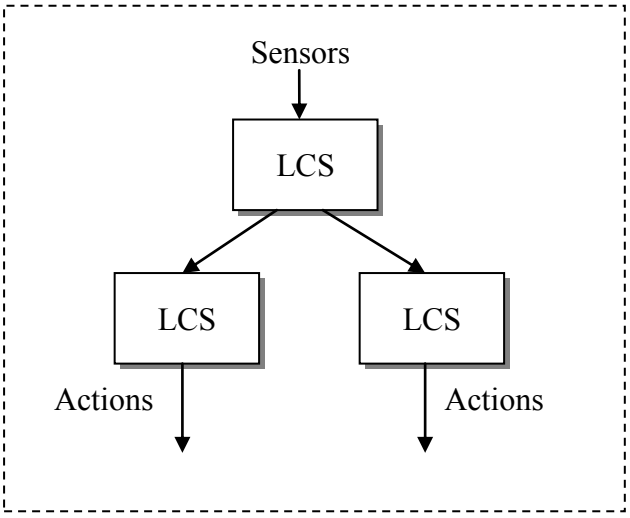



**Figure 18: MCS**

- αCS (Sanza, 2001): in a game like football the entities should cooperate with each other, so a normal LCS's cannot be used directly, because there are some team objectives that cannot be modeled with the use of a simple fitness function for each individual. So in the αCS each LCS can communicate with the others, they can share some classifiers, and when they cooperate with each other they will receive a greater reward.

The LCS's are used in a variety of applications like learning in the presence of noise, incorporation of continuous-valued actions, learning of relational concepts, on-line function approximation and clustering…

## 2.2. Implementation of AI Techniques in video games NPC's

Most of the games these days uses the *Top-Down* methods to model the Non Playable Characters (NPC) behavior in video games like the behavior of enemies, allies, normal people that populate the environment….etc, because those methods are easier to use and to understand than the *Bottom-Up* methods, that is why most developers are using a combination of these *Top-Down* methods in order to achieve behaviors that feels as natural as possible, on the other hand, there is not a lot of video games that incorporate *Bottom-Up* methods in behavior modeling especially genetic algorithms, because these methods are still immature when it comes to video games logic but that did not stop some developers of incorporating them in some really important aspect of their games, I am going to write about a couple of examples of these AI techniques in video games.

- Finite State Machines, decision trees, and expert systems are widely used in most of the current generation games, and from a point of view of an AI designer those methods are the same: they share the same main features and structure (we can represent the three of them with a set of rules each one of them has the structure of IF-THEN-ELSE), but they differ in the way they are built, the way they visualize the final results, and the way they handle the input. They are widely used because they are easy to design and to visualize. Although sometimes they become troublesome when the graph in an FSM or a decision tree becomes really complex or the *Knowledgebase* is huge, but in the end they are not that hard to generate and to implement, and up-till now the results are really promising, examples:
    - Example of an FSM in a first person shooter called *Bet on soldier* developed by Kylotonn in 2005:

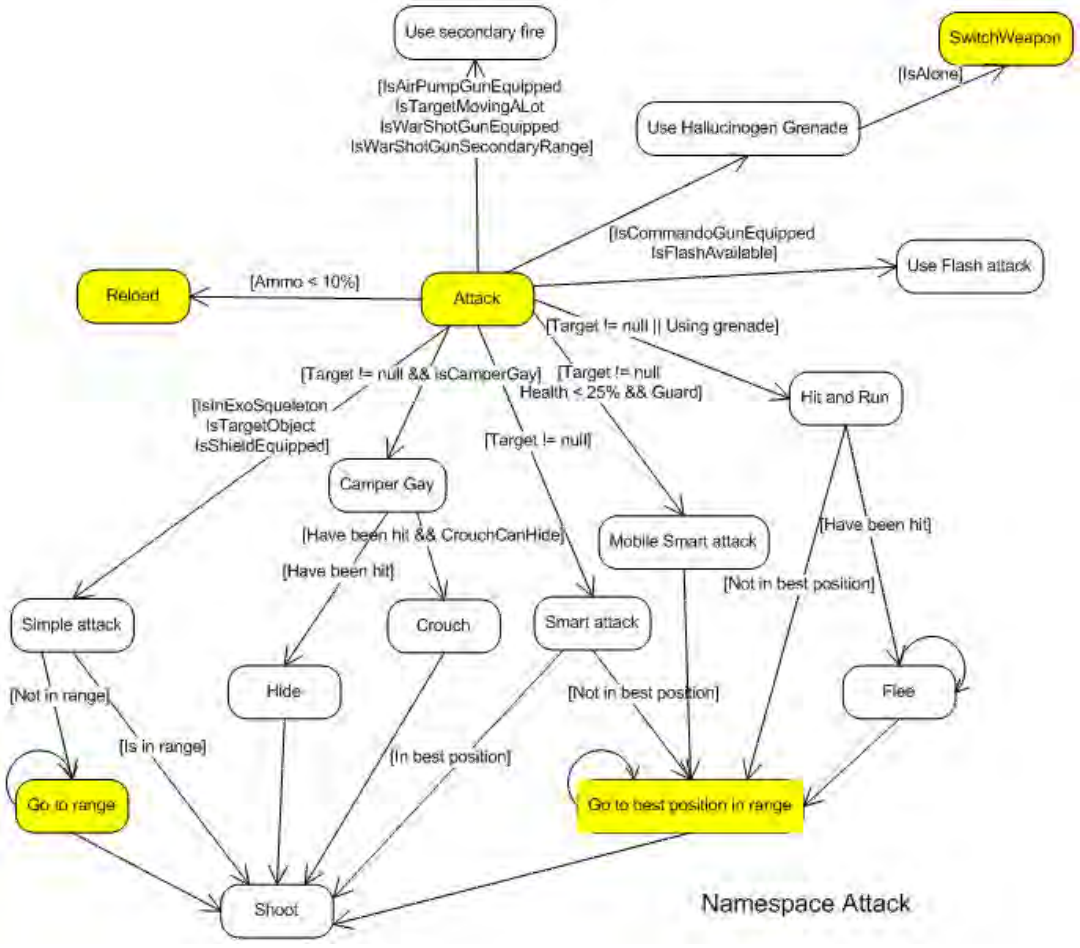



**Figure 19: an FSM from a video game: Bet on soldier**

  - In 2005 Monolith Productions developed a game called *F.E.A.R.* (First Encounter Assault Recon) a first person shooter, they used an FSM to create really challenging enemies that take cover, adapt to the ever changing environment and to the behavior of the player, and the enemies were even capable of doing some really intelligent flanking maneuvers, and fight the player using team work behavior, Monolith won several awards in that year because of their really advanced AI. In 2009 they developed a sequel called *F.E.A.R. 2: Project Origin* with even more intelligent AI: far more challenging and aggressive using the same FSM's concepts.
  - The *Age of Empire* series developed by Ensemble Studios always uses expert systems as the force driving the behavior of the enemy and it uses CLIPS as the *Inference Engine*, which is a public domain software tool, CLIPS stands for "C Language Integrated Production System"

- As for neural networks, there are not a lot of games that uses ANN concepts in the behavior modeling, but one of the most famous one is *Black & White* developed by Lionhead Studios in 2001: The player takes on the role of a god ruling over an island populated by various tribes, he needs to protect his population and lead them into prosperity, beside that, he teaches a large animal-like creatures to do his/her bidding… This game combines neural networks and decision trees, in order to create this realistic experience. The game uses *Perceptrons* to learn the player desires based on reward and punishment, and combined with their own state, the creatures are capable of making decisions all alone, these decisions were memorized trough out the game to create a consistent experience, as for decisions trees, they were used to return the feedback to the player, generate beliefs about the types of objects…etc. In 2005 they developed a sequel called *Black & White 2* where they used more reinforcement learning techniques: enemies in this game try to attack in different ways using different tactics, and waits for the results (reward and loses) of these different tactics, until they find the tactic that would win the battle. And they are dynamic: adapting to the changes of the player tactics, all of that created a feeling that the player is playing against real humans, not scripted ones. These artificial creatures in *Black & White* hold the Guinness world record for most intelligent beings in a game.
- In 2008 Lionhead Studios developed a game called *Fable II*, it is an action role-playing game where the player controls a character that has a dog, one of the most notable invention of this game was that the dog is completely independent, the user cannot control it directly, he needs to interact with it like a real dog in order to control it. This dog AI is based on BDI architecture (belief, desire, intention) the same used in their *Black & White* games:
  - Desires: The desires of the creature are represented as a set of continuous numbers that evolve over time. These are used to select the next goal once the current one has been satisfied like eating, pooping, peeing, and possibly affection and eating, etc.
  - Actions: like moving in front of the player, lying down, protecting the player from other enemies, playing fetch with the player…
  - Beliefs: A decision tree is used for understanding objects (like in *Black & White*). And when you give feedback to the dog, it gets an extra data sample to build the tree.

  As a result, you can train your dog in the typical way, by rewarding it or punishing it by means of a basic YES, or NO emotions of your avatar, all of that to create this virtual being that feels more like a real dog than computer controlled dog.

There are lots and lots of more examples of the utilization of artificial intelligence methods in video games, just to name a few:

- *Prince of Persia* developed by Ubisoft in 2008, it is an action adventure game, where the player is accompanied by a girl named *Elika* whom is controlled entirely by the artificial intelligence,
- *Far Cry 2* developed by Ubisoft in 2008, it is a first person shooter where enemies are purely dynamic, never spawned. Instead, they simply exist in the world, sometimes called in from other areas by signal flares.
- *Grand Theft Auto IV* developed by Rockstar North in 2008, they have created a living and breathing simulation of New York City. This type of games is called a *Sandbox* where you can go anywhere, do anything, and interact with anyone, and it is heavily based on AI, so everything in the city is dynamic: the pedestrians, the traffic, the shops….
- ...

At the end, AI is becoming really important in video games, and that is noticeable with the growing number of games with Cooperative AI like *Army of Two*, *Killzone 2*, *Gears of War 2*…*Sandbox* games like: *Saints Row 2*, *Mercenaries 2*…So developers are starting to experiment with new ways of implementing AI methods to create the best video game experience in the markets.

## 2.3. Fuzzy Logic

In contrast to binary logic (normal logic) where a Boolean proposition can be either false or true (0, 1 respectively), in the fuzzy logic, the degree of truth of a statement can range (inclusively) between 0 and 1. So normally in binary logic the element either belongs to a set or not, but in fuzzy logic this membership is gradual and defined by a *membership function*, and this set is called a *fuzzy set*.

Example: let us take a random set, like the definition of an object if it is "SMALL" or not:

- In binary logic: if the object size is smaller than (5) for example then it is "SMALL" → small = true, but the problem is, if the size is (5.00001) it will not be considered "SMALL" → small = false, so the transition is really rough.
- In fuzzy logic: we could define a membership function where the size (4) is considered 75% "SMALL" and the size (5) is 50% "SMALL", in this way we have a transition between the states a lot smoother with this fuzzy set.

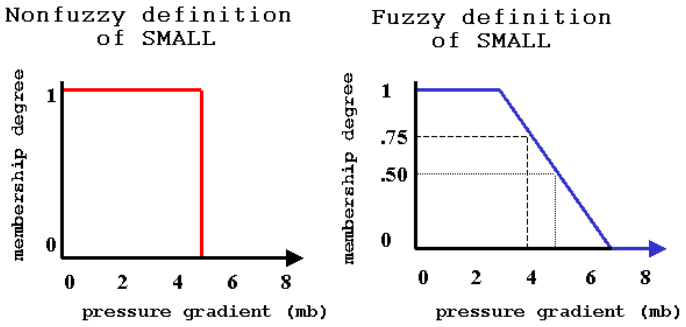



**Figure 20: Membership functions**

The main use of these concepts is during the fuzzy logic reasoning, where (like in knowledge base systems) we have rules of the type:

*If* variable *is* property *Then* action

But in this case, the condition part of the rule is not Boolean (true or false), but rather how much the variable belongs to the property needed, let us take a simple temperature regulator that uses a fan, the fuzzy rules could look like this:

*If* temperature *is* very cold *Then* stop fan
*If* temperature *is* cold *Then* turn down fan
*If* temperature *is* normal *Then* maintain level
*If* temperature *is* hot *Then* speed up fan

As you may have noticed there is no "if - then – else" structure, All of the rules are evaluated, because the temperature might be "cold" and "normal" at the same time to different degrees, and the final action taken depends on these degrees.

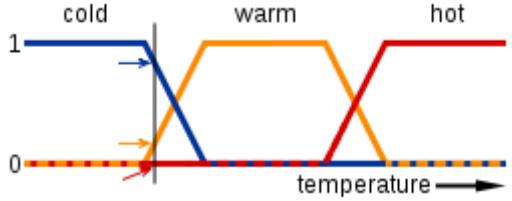



**Figure 21: 3 fuzzy sets for the temperature: cold, warm and hot**

The AND, OR, and NOT operators of Boolean logic exist in fuzzy logic also, usually defined as the minimum, maximum, and complement. So for the fuzzy variables x and y:

- NOT x = (1 - truth(x))
- x AND y = minimum (truth(x), truth(y))
- x OR y = maximum (truth(x), truth(y))

There are even operators not found in normal logic like the average, median….

Fuzzy logic is used in vast types of applications these days like: Air conditioners, Washing machines and other home appliances, Digital image processing, Pattern recognition and Video game artificial intelligence…and it was first proposed by Lotfi Zadeh. He published his seminal work on fuzzy sets in 1965, in which he detailed the mathematics of fuzzy set theory. In 1973 he proposed his theory of fuzzy logic.

## 2.4. Imitation

### 2.4.1. Introduction

Imitation is a really powerful tool that the human brain uses to augment its knowledge and to discover solutions, because from an abstract point of view, the imitation strategy can reduce the search space for the appropriate solution. That is why the imitation is looked at as a way of enhancing machine learning in multi-agent systems: where the agent uses the knowledge of the past cooperative teachers, or other different agents to augment its ability of learning how to survive, to operate in the same environment and/or how to interact with it.

In the machine learning by imitation we have the agent who is the *observer*, and one or several other agents with the role of *mentors*, and there are two main types of imitation:

- Explicit imitation: the *mentors* take the role of the teachers (expert ones), and their goal is to make the *observer* imitate them, by directly teaching him what they think the right set of actions and decisions are, but in this case the main assumption is that the *mentors* are cooperative and ready to share their knowledge with the *observer*, and that the system is capable of incorporating this kind of direct communication, (Atkeson & Schaal, 1997; Lin, 1992; Whitehead, 1991).
- Implicit imitation: in this case there is no direct communication between the *observer* and the *mentors*, and the agents are not forced to play the role of the teacher explicitly, on the contrary, the *observer* tries to learn simply by watching the behavior of other agents. This is important because sometimes it is not possible for a *mentor* to alter its behavior to teach the other agents, or it could be unwilling to do that because they are in a competitive situation, (Price, 2003; Métivier, 2004; Thurau, Sagerer, & Bauckhage, 2004; Tambellini & Sanza, 2006).

There are a lot of other differences between the two strategies, like in implicit imitation the agent is not forced to imitate the *mentors* 100%: he can use the observations he made to enforce his own system, while in explicit learning the *observer* is expected to be whatever the *mentors* wants him to be. Another difference is that in implicit imitation the objectives of the *mentors* can differ from those of the *observer* and that is why the imitation cannot be exact, and the *observer* in this case needs to imitate the more interesting behaviors concerning its proper objectives, but of course the closer they are to his objectives, the more utility can be derived from implicit imitation.

We need to distinguish between two settings: *homogenous* settings where the set of action and abilities between the *observer* and the *mentors* are the same, in this case the mapping between the *mentors* and the *observer* actions and decisions is straight forward while in *heterogeneous* settings, there might be differences, which could lead to odd situations where the *mentor* is capable of interacting with the environment differently or he has different capabilities, in this case the *observer* needs to make adjustments during the imitation process to adapt everything to his own model.

Now, in any kind of imitation there are 3 main steps: Recording stage, Machine learning stage and the Re-playing stage.

### 2.4.2. Recording Stage

In this stage, each researcher records a different kind of data, which is based on the type of the environment being used, the type of imitation, capabilities of the agent…etc

- In Bob Price (Price, 2003) research, the environment that he was testing on is a simple 2D grid with different types of cells: empty, obstacle, start, finish…He was using some of the famous maze configurations like: wood1, maze4…

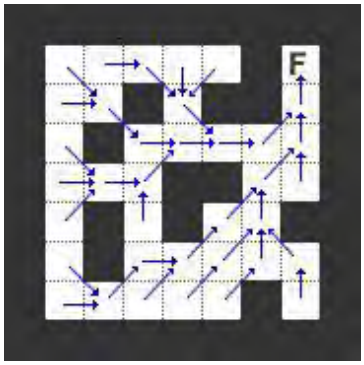



**Figure 22: Maze4 and the optimal behavior**

In this environment, the objective of the agent is to reach the finish cell, while only perceiving the 8 cells surrounding him.

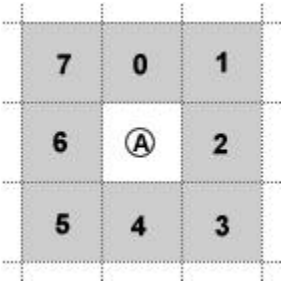



**Figure 23: Agent perception**

In his research, Price uses the implicit imitation strategy with homogenous settings, with only one *mentor*. By definition this *mentor* is not cooperative but he has the same objectives as the *observer*, this *mentor* is either controlled by the user directly or a by a system of machine learning by reinforcement, and of course in the same environment we have the *observer* doing the imitation. As for the data being registered; it is the state of those 8 cells.

- Marc Métivier (Métivier, 2004): he uses the same environment like Price, with the same data being recorded but he introduced a new concept: *guidance interaction*. In Price research the two agents can exist in the same time on the grid (although not interacting with each other but it is possible), while in Métivier research an external agent can take over the control of the *observer*, which means we have only one agent who is the *mentor* and the *observer* at the same time.

- This agent can be controlled by either the user (the *mentor*) or controlled directly by the system doing the imitation (the *observer*).
- At any moment during the simulation, if the user stops controlling the agent, the observe takes control directly.

Thus both the *mentor* and the *observer* share the same observations of the environment and the behavior transfer between them is straight forward. The imitation in Métivier research is implicit with homogenous settings, and with the same objectives.

- In (Thurau, Sagerer, & Bauckhage, 2004): they use a famous FPS (First person shooter) game for their research, called *Quake II* from ID software[1], where users shoot each other in a 3D environment (no teams, each player is on his own), and there are weapons, power-ups and health packs laying all around the arena; the idea behind using this kind of popular game is that users normally organize what they call *LAN-party* where hundred or sometimes thousands of gamers compete against each other at the same time, so the amount of data is a lot more significant from an in-house developed game, on top of that, most of the players have developed some really important expertise for their research.

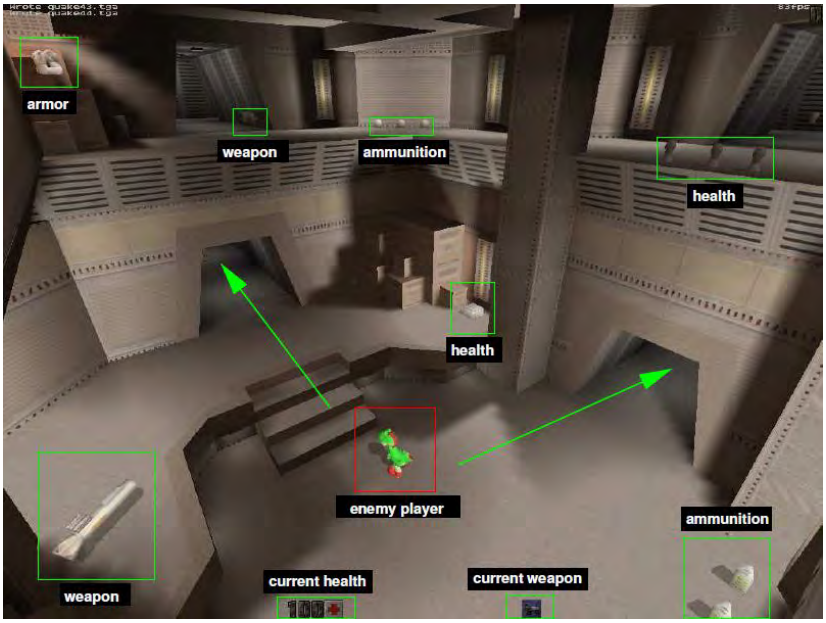



**Figure 24: Quake 2 environment**

They use an implicit imitation strategy with several *mentors* (the gamers) with homogenous settings, those gamers were not aware that they were recording their behavior, so in machine learning terms: they were not cooperative. The data they were registering contains information about the exact locations (x, y, z) the player assumed, nearby items and other players. Temporary entities like sounds and flying projectiles are also included. They did not have to visually analyze the game scene, since all necessary information was already available on a cognitive higher level.

- In (Tambellini & Sanza, 2006): they use a similar approach (an FPS game), but with a in-house developed game thanks to Kynapse Artificial, which is an Intelligence middleware framework[2], another difference is that the user does not control the virtual character (avatar) directly, but he gives it high level orders only like: Wander, Follow, Attack, Flee, Hide,....they use an implicit imitation strategy with also homogenous settings, and the recorded data was:

- Number of visible enemies: the number of enemies the character can see according to a fixed visibility distance accuracy and vision cone angle.
- Number of enemies around: the number of enemies seeing the player (at front or at back).
- Player life (between 0 and 100).

[1] http://www.idsoftware.com/
[2] Kynapse Artificial Intelligence middleware, http://www.kynogon.com

**Figure 25: Testing environment in (Tambellini & Sanza, 2006)**

### 2.4.3. Machine Learning Stage and Re-Playing Stage

Now after the recording stage, we need to use this data recorded the proper way:

- Bob Price: in his research, imitation is an extension of reinforcement learning, and may be considered as a knowledge transfer process between those two agents, so the *mentor* data (acquired by the previous stage) is used to build the environment finite-state Markovian decision process (MDP) from the *mentor* point of view of course, (which, as we saw, is defined as a tuple <State, Action, Transition, Reward >).

After that, when the *observer* is building his own representation of the MDP environment (that mean his own transition function) in the re-playing stage, he is incorporating the *mentor* model in his decision for the next action, the paradigm here is that the *mentor* behavior should be somehow optimal, so when the *observer* uses the transition function of the *mentor* to decide the next action, this action should serve his own objective. But sometimes this is not the case, thus the blind imitation can really slow down the progress of the *observer*, that is why in Price research, even if the behavior of the *mentor* is not optimal, the *observer* is going to try to use it at first, but if he finds that it is not serving him well (based on the reward he is receiving from the environment), he is going to lower his dependency on the *mentor* model, and start exploring other options, so he could suppress the *mentor* by finding the optimal policy for accomplishing his objectives (in this case reaching the finish cell).

- Marc Métivier: in his research, the imitation is based on reinforcement learning too, he even consider it as reinforcement learning but without reward. He uses learning system classifier (LCS) as a base for the imitation, with 2 main approaches: without a *mentor* model, or with a *mentor* model, and in both ways and during the recording stage, a rule based system is deduced from the *mentor* behavior (set of rules with their force); the difference lies in the re-playing stage.

  - Without a *mentor* model or Guidance only: in this case we have an LCS to represent the *observer* behavior, and during the LCS action selection process, the *mentor* imposes his action on the *observer,* that means: if there is a rule activated in the *mentor* rule base system (if the condition part of the rule is met), this rule action will always be executed no matter what is the answer of the *observer* LCS, and if there is nothing activated then the LCS will behave normally.

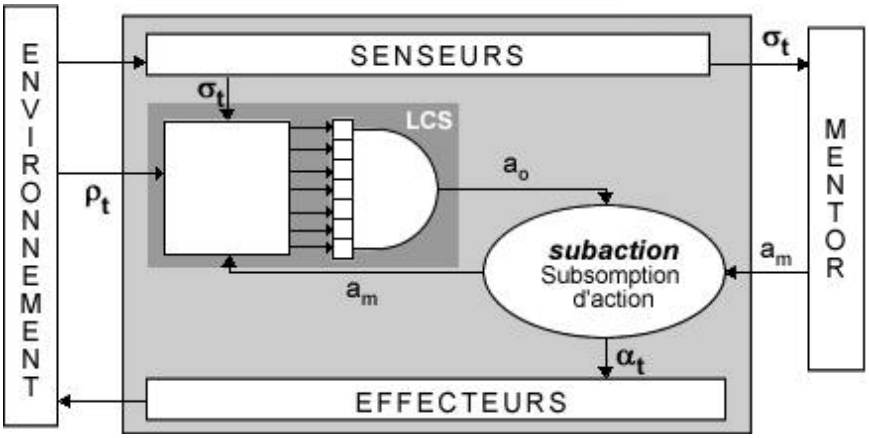



**Figure 26: Marc Métivier Guidance only system**

- With a *mentor* model: here we have a two LCS's (or what they call a DoubleCS system): one for the *observer* and the other for the *mentor*.
The *mentor* LCS is actually the rule based system from the recording stage with the gain or confidence…and it evolves only during the recording stage, as for the *observer* LCS it is totally independent and it evolves during the re-playing stage, and here we have also two other possibilities, when it comes to selecting the most consistent action:

1 – Each LCS will calculate the matching set of actions in the normal way, then another step is used to combine both matching sets (if the same action is selected in both LCS's, the system will take the max of the force), but only the *observer* LCS will be updated based on the reward.

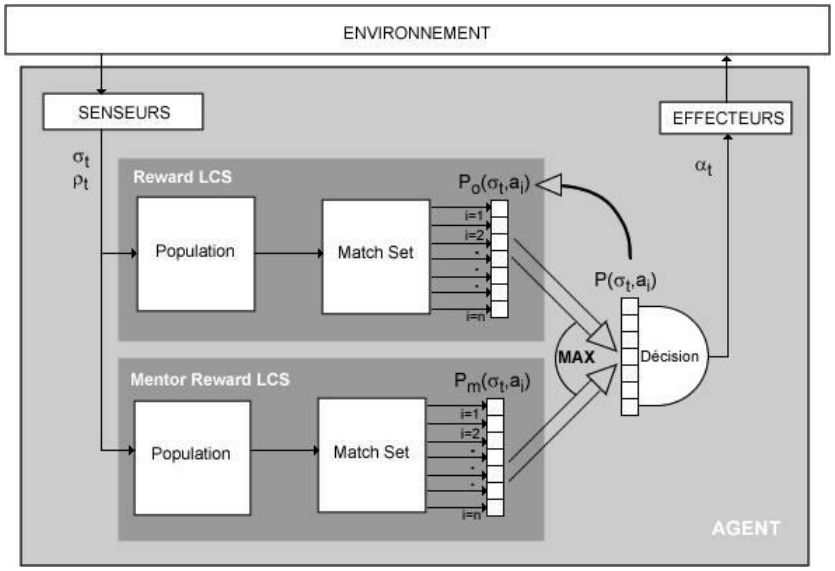



**Figure 27: Marc Métivier imitation system with mentor model (1)**

2 – The same idea of the previous system, but here the system will select only one action (the one with the max force) from the *mentor* LCS and it will add it to the matching set of the *observer* LCS, so this action will have the same probability to be selected like the other actions.

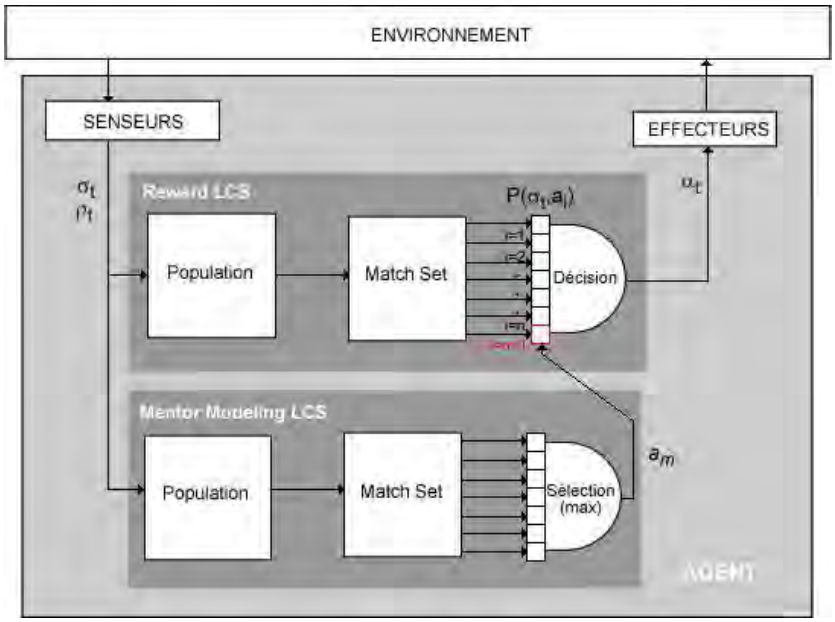



**Figure 28: Marc Métivier imitation system with mentor model (2)**

- (Tambellini & Sanza, 2006): they built a decision tree based on the recording stage data using the *ID3* algorithm, were the stopping condition is either the depth of the tree, or the number of examples each leaf is covering (to avoid over fitting when the leaf covers only 2-3 examples), with one hypotheses: The recording time is only 10 minutes, so that the gamer does not have time to change his way of playing, thus avoiding incoherence during the tree building, so in this way they avoided the problem of contradiction (two different actions with the same kind of object observed), which would have bad effects while the tree is being constructed.

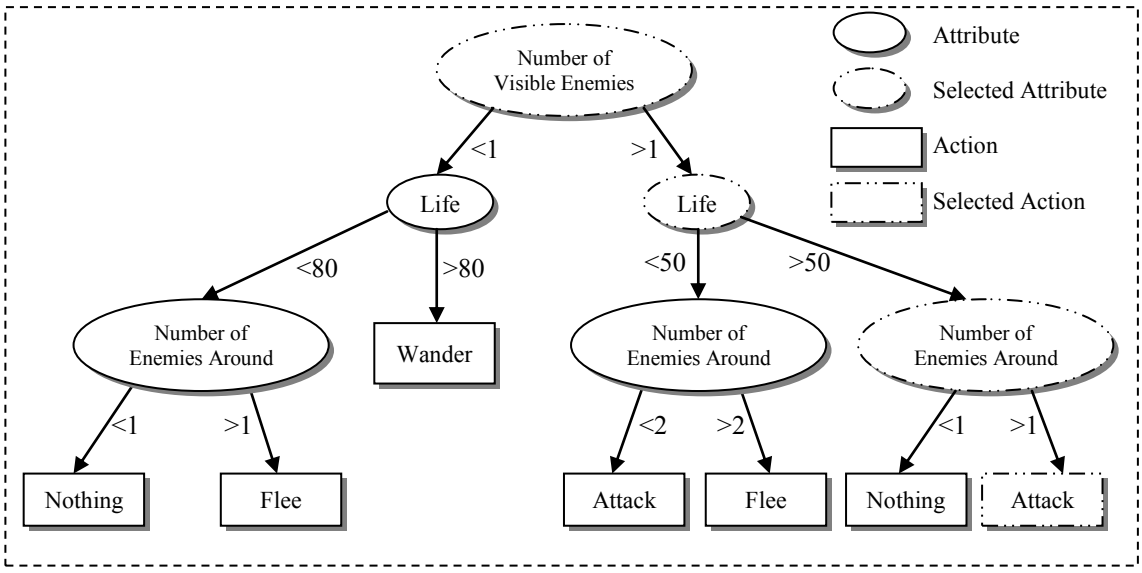



**Figure 29: Resulting Decision tree in (Tambellini & Sanza, 2006)**

- (Thurau, Sagerer, & Bauckhage, 2004): they were concentrating on two different aspects of imitation: first, producing natural reactions, and second, producing natural movements.
For producing natural reactions or *reactive behavior* (which is the reaction of the *observer* based on the audio-visual perception) like: aiming and shooting on an enemy, predicting the enemy place based on audible cues, and some movement like jumping, hiding…And for this behavior, they use Neural-Networks, and more exactly Multi layered Perceptron (MLP) using the examples of the recording stage, with several networks, each one representing a different situation the *mentors* has encountered and sometimes a different action. But they found that for long term goals (like achieving high killing rate, surviving for long time…), these goals should be treated separately and be implemented independently from the imitation process, because they depend more on the motive of the player and his objectives.
As for producing natural movement or *Motion Modeling*, they were breaking the movement to its primitives by using clustering algorithms like the *k-means* to regroup similar primitive motions with each other, then combining several motion primitives to achieve more natural movement based on direct observations, and they were capable of managing 20-30 motion primitives per second.
In the re-playing stage: the agents were making more natural motions and even doing some movements learned from experienced gamers (like climbing up the stairs and jumping toward a weapon that cannot be reached in the normal ways).

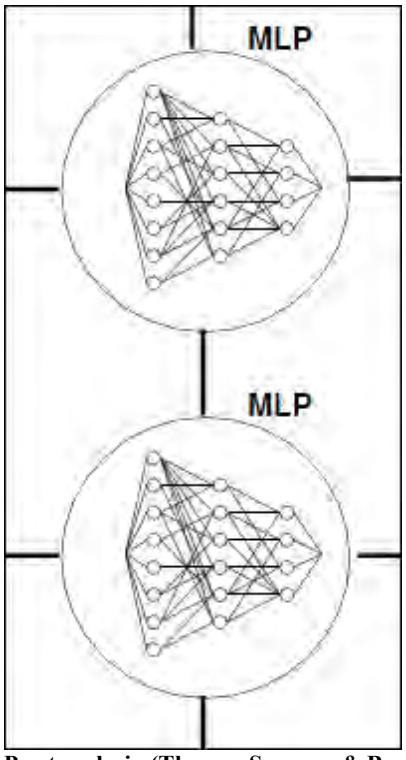



**Figure 30: MLP networks in (Thurau, Sagerer, & Bauckhage, 2004)**
**Each network could represent a different action or decision.**

### 2.4.4. Conclusion

In all of these researches, the imitation was really speeding up the machine learning stage and sometimes eliminating the tedious work of scripting in video games, or if not, it was not making the process slower, and one of the important conclusions: the implicit imitation model is more intuitive because it does not impose anything on the *mentors* (like changing their behavior to teach the other agents…), so at the end the imitation is a way the *observer* accelerates its learning process.

# Chapter 3: Imitation Experience

*Cette étude est basée sur le processus d'imitation dans le contexte des jeux vidéo, plus exactement : l'imitation implicite dans les MMORPGs (Massively multiplayer online role-playing game), et nous testons notre système d'Intelligence Artificielle dans une simulation de ville 3D.*

*L'imitation est implicite, et elle est basée sur le concept de Métivier : l'interaction par guidage (Métivier, 2004), où, à n'importe quel moment de la simulation, l'utilisateur peut se connecter à l'environnement et contrôler l'avatar (sa représentation dans le monde virtuel), et, quand il se déconnecte, le système va prendre le contrôle de cet avatar et le processus d'imitation va commencer.*

*L'environnement est une simulation de ville 3D, avec différents types d'objets et de surfaces. De plus nous utilisons le moteur physique Bullet pour animer ces objets et simuler les interactions.*

*Comme tous les systèmes précédents, il y a 3 étapes : l'enregistrement, l'apprentissage et le rejeu.*
***L'enregistrement :*** *dans cette étape nous simulons la perception humaine : notre avatar a un champ de vision sous la forme d'un cône 3D, il peut voir tous les objets présents dans cette zone, et il peut même détecter le type de surface sur laquelle il évolue.*
*A chaque pas de simulation, l'avatar construit la liste des objets avec lesquels il peut interagir, chacun de ces objets ayant ses attributs qui définissent son état.*

*Donc comme on peut le constater, nous avons l'état des objets, chacun avec plusieurs attributs, avec des types de données hétérogènes (mètre, degré, Booléen ....), ce qui va rendre la phase d'apprentissage complexe et difficile, c'est pourquoi nous utilisons le concept d'ensembles flous pour représenter ces attributs.*
*L'idée d'utiliser l'ensemble flou est très simple : unifier tous les types de données c'est-à-dire remplacer la valeur de chaque attribut par une variable bornée entre [0..1] qui représente l'importance de cet attribut pour notre avatar.*

*Avec l'utilisation de la logique floue, nous avons éliminé l'une des complexités du système, et pour le simplifier encore plus, nous avons ajouté un attribut supplémentaire qui permet de regrouper certains objets, par exemple : si l'avatar voit 4 objets du même type à sa droite, nous allons choisir le plus proche comme un représentant de ce groupe et changer la valeur de son attribut Count : de la valeur par défaut qui est 0.5 normalement (c'est-à-dire 1 objet), à une valeur de floue qui reflète l'importance de 4 objets pour notre avatar.*
*Enfin, à chaque pas, l'avatar fait une observation au format : Liste d'état des objets + l'action effectuée + un ID*
*Les objets dans cette liste sont triés selon les types d'objets et leur orientation, et un ID est généré à partir de cette liste, cet ID représente les objets dans cette liste, et nous l'utilisons dans la plupart des étapes suivantes, parce que les comparaisons entre des chiffres sont bien plus rapides que les comparaisons entre des strings.*

***L'apprentissage :*** *c'est l'étape principale de notre système, et les opérations qu'elle utilise sont : Filtrage des données, l'apprentissage : Espace de Version, Génération de la force des règles, Fusion des règles.*

- *Filtrage des données : dans chaque pas de simulation, l'avatar va faire une observation, qui s'élève à un grand nombre d'observations dans un petit laps de temps (la simulation fonctionne avec 60 images par seconde en moyenne), pour ça, nous allons comparer l'observation de l'image courante, avec l'image précédente, et nous la prenons en compte si elle est différente. En revanche, pour ne pas perdre de données, nous enregistrons la durée pendant laquelle l'observation ne change pas.*
- *Espace de Version : le cœur de notre système est un algorithme d'apprentissage qui s'appelle L'espace de version ou EV (Mitchell, 1997). Cette algorithme construit deux type d'hypothèses GB et SB, et nous nous intéressons à SB qui consiste à construire les hypothèses les plus spécifiques et cohérentes : les hypothèses qui couvrent seulement les exemples positifs, et si elles sont réduites un peu plus, elles deviendraient incompatible parce qu'elles vont exclure un de ces exemples positifs, la construction de ces hypothèses commence avec l'hypothèse la plus spécifique : le premier exemple positif, après ça, pour chaque exemple positif, l'algorithme généralise les hypothèses un peu pour couvrir ce nouvel exemple, par contre pour les exemples négatifs, il va éliminer les hypothèses qui le couvrent. En résumé il commence à partir des hypothèses plus spécifiques vers des hypothèses plus générales.*

  *Mais ce processus d'élimination de l'algorithme original peut avoir des effets négatifs sur notre système, par exemple:*
  *Si ont prends deux règles, d'un EV :*
  *Si Objet = véhicule et distance entre [0.3, 05] Alors marcher*
  *Si Objet = véhicule et distance entre [0.1, 02] Alors arrêter*
  *Et une nouvelle observation :*
  *Objet: véhicule, distance: 0.4, action: arrêter*
  *Normalement l'algorithme d'espace de version va éliminer la première règle et la deuxième règle sera:*
  *Si Objet = véhicule et distance entre [0.1, 04] Alors arrêter*
  *Mais de cette manière, nous avons perdu des informations vraiment importantes, de l'autre côté, si nous avons essayé de résoudre le problème dans la même VS, ce qui signifie avoir d'intersection entre les règles:*
  *Si Objet = véhicule et distance entre [0.3, 05] Alors marcher*
  *Si Objet = véhicule et distance entre [0.1, 04] Alors arrêter*
  *Cette contradiction va affecter gravement l'algorithme d'espace de version au cours des prochaines généralisations et éliminations.*
  *Modification d'espace de version : nous proposons une modification sur l'algorithme original pour éviter les problèmes précédents : en cas de contradiction, nous allons créer un autre EV avec le même ID, cet EV va traiter tous les exemples qui causent les contradictions, et s'il y a de contradiction dans ce nouvel EV, nous allons créer un autre EV. Enfin, le système peut contenir 2-3 espace de versions avec le même ID.*
  *Comme ça, nous avons maintenu le montant maximal de l'information, et comme effet secondaire de cette modification, nous avons pratiquement supprimé l'étape de l'élimination de l'algorithme original.*

- *Génération de la force des règles : cette force sera basée sur deux critères : le pourcentage d'observations que chaque règle recouvre, et la durée de chaque observation.*
- *Fusion des règles : l'idée de cette étape est de minimiser le nombre de règles : le principe est de toujours conserver la règle la plus générale.*

***Le rejeu :*** *à partir de l'étape d'apprentissage, nous avons construit une base de règle qui modélise le comportement du mentor, nous allons utiliser cette base de règle dans un système de classeurs : un ZCS, mais un ZCS générique parce que les règles ne sont pas binaire. Nous allons utiliser un seul LCS pour représenter le comportement de l'observé, tout comme le modèle de Marc Métivier en l'absence d'un mentor (guidage seulement) (Métivier, 2004), de cette façon, les décisions que le mentor a prises au cours de la phase d'enregistrement influence directement le comportement de l'avatar.*
***Observations :*** *les observations suivantes seront basées sur l'imitation de notre mentor, quand il est en train de traverser la rue, et nous avons testé notre système en trois différents scénarii :*

- *Scénario 1 : Nous avons supprimé tous les véhicules et autres piétons, donc les rues étaient vides.*
- *Scénario 2 : Nous avons ajouté les véhicules.*
- *Scénario 3 : Nous avons ajouté les piétons, et l'environnement contient tous les objets.*

***Résultat :***

- *Scénario 1 : l'avatar a compris l'importance des feux de trafic, et il a commencé à s'arrêter quand les feux de trafic est rouge.*
- *Scénario 2 : Étant donné que le nombre de véhicules est beaucoup plus important que le nombre de feux de trafics, les véhicules ont plus d'influence que les feux de trafic. C'est pour cela que parfois il traverse la rue quand il n'y a pas de véhicules alors que le feu était rouge.*
- *Scénario 3 : même résultat que scenario 2 car les autres piétons sont beaucoup plus nombreux que les feux de trafics.*

*En plus des résultats précédents, nous avons fait plusieurs observations :*

- *Dans tous les scénarios précédents, notre système converge vers le même nombre de règles (≈13 règles en scenario 1, ≈40-50 en scenario 2, ≈170-180 en scenario 3).*
- *Plus l'environnement est complexe, plus nous nous avons un problème de bruit : le système devient moins capable de découvrir les objets importants, tous les objets ont le même degré d'importance, ils ont tous été capables d'influencer le comportement de notre avatar par la même force.*

***Conclusion :*** *notre système a été capable d'imiter le mentor, mais nous avons essayé de créer un système général, pour cela nous avons le problème de bruit, ou parfois l'avatar prend des décisions qui apparaître illogique, mais il ne faut pas oublier qu'il prend en compte tout ce qu'il voit. Mais en général, l'imitation a été un succès et le système final par bien modéliser les comportements du mentor.*

***Implémentation dans un MMORPG :*** *Un MMORPG ou Massively multiplayer online role-playing est un genre des jeux de rôles où le joueur contrôle un avatar dans un monde virtuel persistant : le monde continue d'exister et évoluer alors que le joueur est hors jeu, ce qui crée un monde qui n'arrête pas d'évoluer. L'idée c'est d'utiliser notre système pour que le monde soit toujours peuplé par des avatars : au moment où l'utilisateur se déconnecte, notre système prend le contrôle de l'avatar et essaye de rencontrer des nouvelles personnes ou de gagner de l'expérience pour l'utilisateur.*

## 3.1. Introduction

This paper is based on the imitation process in video games context, more exactly implicit imitation in an Massively multiplayer online role-playing game or MMORPG (more about this type of games later), and the final AI imitation system will be first tested in a city simulation.
So the imitation is implicit, and it will be based on Métivier concept of: *guidance interaction* (Métivier, 2004), where, at any moment of the simulation, the user can connect to the environment and control the *Avatar* (his representation in the virtual world), and when he disconnects, the system will take control of this *Avatar* and the imitation process will begin, so we have only one *mentor* who is the *observer* at the same time, and they both have the same capabilities *(homogenous setting*) and share the same objectives.

## 3.2. The environment

The test environment will be an in-house developed 3D city simulation, with different kinds of objects like vehicles, pedestrians, traffic lights…and different kinds of surfaces like sidewalks, crossroads, streets…We use 3D models to represent these objects and surfaces plus we use also MoCap (Motion Capture) for the pedestrian's movements, because it is more natural this way.

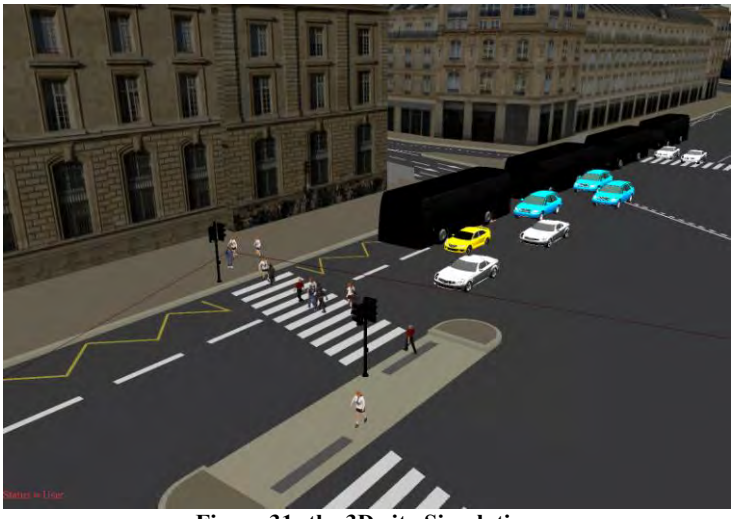


**Figure 31: the 3D city Simulation**

This simulation is as close as possible to reality and this was achieved by using physics as the force that drives most of these objects and interactions, like the interactions between the pedestrians and the environment (going up the stairs, bumping with each others), the movement of the vehicles (suspensions, breaking forces)…..etc all of that thanks to an open source physics engine called *Bullet*[3].

[3] http://www.bulletphysics.com/

But what are the benefits of using this kind of simulations? The idea behind using this kind of environment is: it is complex enough to test the imitation system before using it in a more complex video game, and because it is in-house developed, there is no need to understand how things work in the game engine before implementing the system.

## 3.3. Machine learning by imitation system

Like the previous systems, we have the same 3 main stages.

### 3.3.1. Recording stage

During this stage, we simulate the way the human perception works, so our *Avatar* has a field of view in the form of a 3D Cone, just like humans, everything inside this cone the *Avatar* can see, and on top of that he can even sense if he is on a surface like a crossroad or a sidewalk…

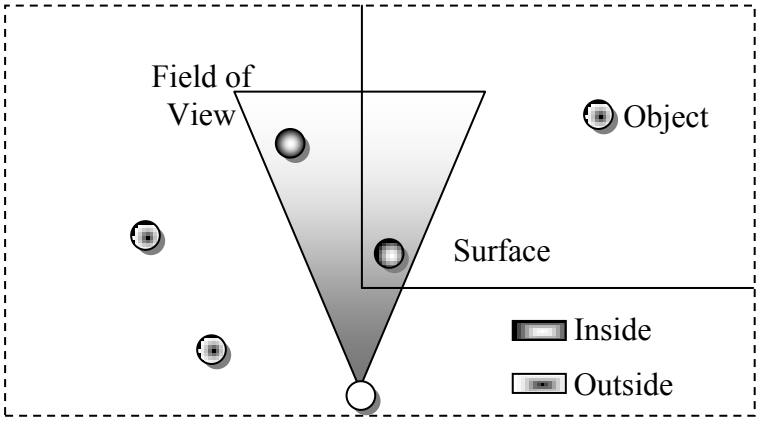



**Figure 32: Avatar Field of View**

So at any given time during the simulation we have a list of interesting objects that the *Avatar* can interact with. Each one of these objects has its own set of attributes that represent its proper state. Some of these attributes are general and common between all the simulation objects like:

- Type and Name of the object.
- Distance: the distance between the *Avatar* and this object, measured using the distance units of the OpenGL[4].
- Orientation: is the object in front of the *Avatar* or at his right or left, this is a variable with 3 states.
- Angle: the angle between the velocity vector of our *Avatar* and the velocity vector of the object if he is a moving one, measured in degrees.
- …

And we have also the object specific attributes like:

- For surfaces: is the *Avatar* on this surface? , this is a Boolean.
- For traffic light: is it red? Also a Boolean.
- For vehicles: speed, force, weight, each one with its respective unit.
- …

[4] The unit in OpenGL could be meters, inches, kilometers, or leagues…it is up to the user to define the relation between the units in OpenGL and their meaning in the real world.

So as we can see, we have the state of the objects, each with lots of attributes, with really different units (distance, degree, Boolean….), and this is going to make the learning stage complex and difficult, that is why I use the fuzzy sets concept to represent these attributes.
The idea behind using the fuzzy sets concept is to unify the units of all the attributes, by replacing all of these different values of these attributes with only a value from the interval [0..1], this value will represent the importance of this attribute to the *Avatar*.

Let us take an example: The Distance, the more the value of this attribute is close to 1 the more the object is closer to our *Avatar*, that means it is more important, and this importance depends on the type of the object of course, thus the membership function for a static object is not the same for a moving one.

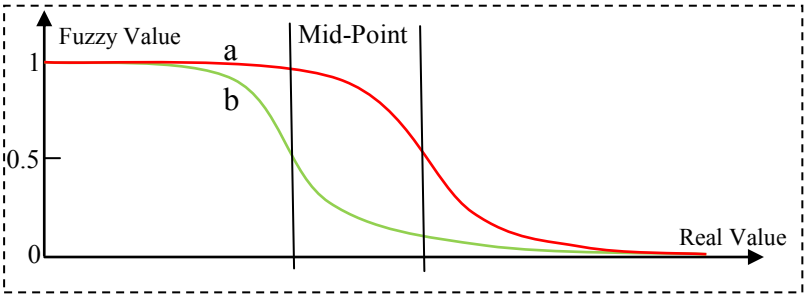
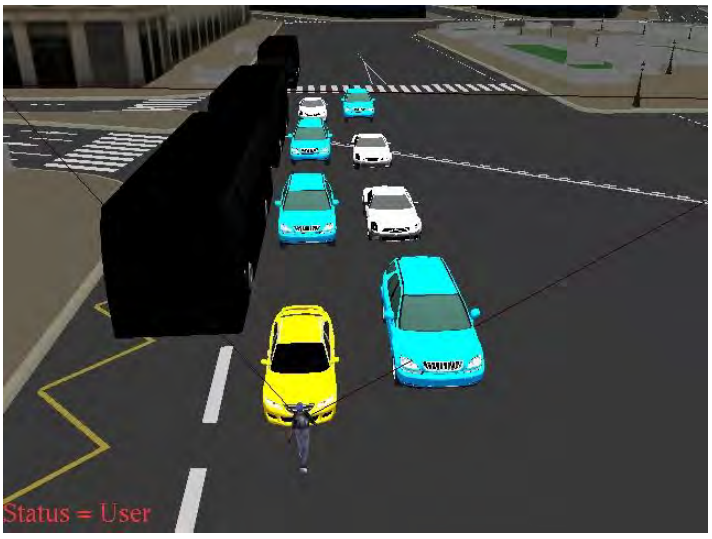



**Figure 33: The Distance membership function**
**Moving object (a), Static object (b)**

And this is the same for the other attributes:

- Orientation: can only take the following values: left = 0, in front = 0.5, right = 1.
- Angle: a value between [0...1], it has the same significance of the distance.
- For the Boolean attributes: either 0 or 1.

So by using this kind of representation I have eliminated one of the complexities of dealing with different kinds of objects and attributes. Now, to lower the complexity of the recorded data even more I used an extra attribute that represents the Count: so for an example if the *Avatar* sees 4 objects of the same type on his right, I am going to choose the closest one as a representative of this group and I am going to change the value of its count attribute, from the default value which is 0.5 normally (that means 1 object), to a fuzzy value that reflects the importance of 4 objects to the *Avatar*.




| Vehicle | | |
|---|---|---|
| **Type** | **Value** | **Interpretation** |
| Distance | 0.980000 | Really close |
| Orientation | 0.500000 | In front |
| Angle | 1.000000 | 180 degree |
| Count | 0.861496 | 4 cars |
| Advancing | 0.000000 | no |

**Figure 34: Example of fuzzy set utilization**

So, in each frame, the *Avatar* makes an observation (the *recorded data*) that consists of:
A list of objects state + the action carried out + an ID.

Before talking about the ID, I should mention that inside this list, objects are sorted based on two criteria: their type, and their orientation. In other words, if the *Avatar* sees the same objects and he is on the same surface but the order of his perceptions is different, the final list of objects state is always the same, that will be really helpful during the comparisons in the later stages either with the previous observations or with the condition part of the resulting rules.

Now, the ID; it is a numerical representation of the objects inside the list of objects state, and it is generated based on the object type, for an example: an object with the type = vehicle, the system will assign always the number 2, for the pedestrians the system will assign the number 5 and so on. So if I have two vehicles and one pedestrian the ID will be 225, and because the objects are always sorted, I will always have the same ID (225) for any kind of configurations of these three objects, this way I reduced the complexity of the comparison of the two lists of objects (comparison of strings) into just comparing integers, thus optimizing lots of steps (we will see the importance of the ID in most of the following steps).

An example of an observation:
**Objects state:**
Name = TRL1, Type = TrafficLight
MType = Distance, MValue = 0.980000, MType = Orientation, MValue = 0.500000,
MType = Angle, MValue = 0.153760, MType = Count, MValue = 0.500000,
MType = **isred**, MValue = 1.000000.

Name = Car1, Type = Vehicle
MType = Distance, MValue = 0.564444, MType = Orientation, MValue = 0.500000,
MType = Angle, MValue = 0.914184, MType = Count, MValue = 0.551247,
MType = **Advancing**, MValue = 0.000000.

Name = Car1, Type = Vehicle
MType = Distance, MValue = 0.624444, MType = Orientation, MValue = 1.000000,
MType = Angle, MValue = 0.914184, MType = Count, MValue = 0.861496,
MType = **Advancing**, MValue = 0.000000.

Name = Surface, Type = CrossRoad
MType = Distance, MValue = 0.857778, MType = Orientation, MValue = 0.500000,
MType = Angle, MValue = 0.000000, MType = Count, MValue = 0.500000,
MType = **on**, MValue = 0.000000.

Name = Surface, Type = SideWalk
MType = Distance, MValue = 0.997778, MType = Orientation, MValue = 0.500000,
MType = Angle, MValue = 0.000000, MType = Count, MValue = 0.500000,
MType = **on**, MValue = 1.000000,

**ID** = 23346
**Action** = STOP

Common Attributes
**Specific Attributes**

### 3.3.2. Machine Learning stage

This is the main stage in our system, and it consists of 4 sub-stages: Data Filtering, Machine learning (Version space algorithm), force generation and rules fusion.


**Figure 35: Machine learning Steps**

#### 3.3.2.1. Data filtering

As we saw in the previous step, each frame the *Avatar* will make an observation, which amount to a huge number of observations in a small period of time (the simulation runs with 60 frames per second as an average), so in the filtering stage we compare the data of the current frame with the observation of the pervious frame, and if they match completely, or the difference is negligible, we will not process the current frame observation, but by doing that we could lose important information, so we added a new variable to the observation data which will represent the time of the observation: the time during which this observation (objects state, action, ID) stayed the same, this will be helpful during the force generating of the rules.

#### 3.3.2.2. Machine Learning: Version Space algorithm

This is the core of our system, and it is based on a famous machine learning algorithm called *Version Space*, or VS (Mitchell, 1997).

Let us take a 2D coordinates universe with positive and negative examples, with a clear objective: finding the rectangle that separate those two kinds of examples completely.

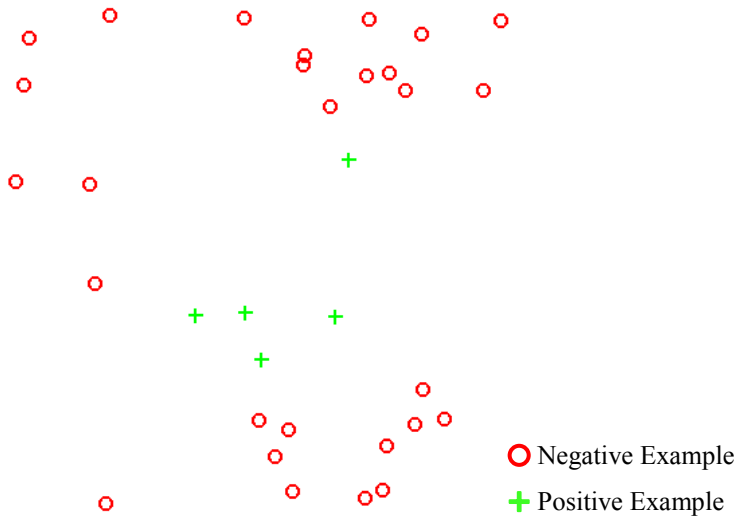



**Figure 36: Features Space with examples**

The idea of the VS algorithm is quite simple; it will try to create two types of hypotheses:

- The most specific and consistent hypotheses (i.e., the specific boundary SB): which is the hypotheses that covers the positive examples only, and if it is reduced any further, it will exclude one of those positive training examples, and hence becomes inconsistent; in our example it is the smallest rectangle.
- The most general and consistent hypotheses (i.e., the general boundary GB): which is the hypotheses that covers the positive examples also, but tries to cover as much as it could of the remaining feature space without including any negative example otherwise it will become inconsistent; in our example it is the largest rectangle.

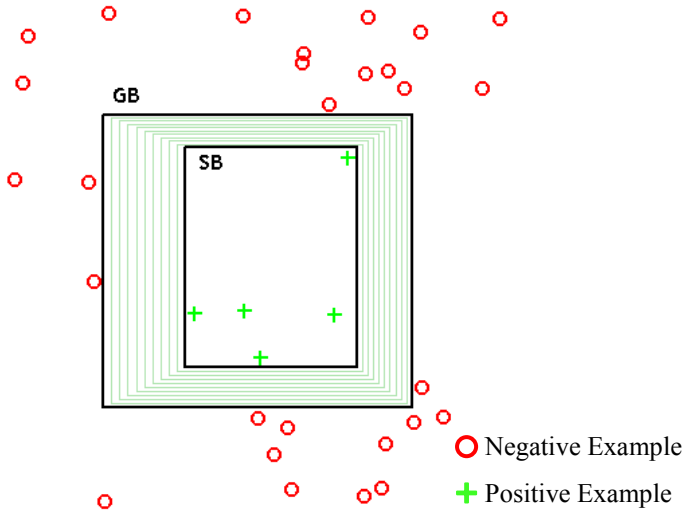



**Figure 37: Version Space Result**

So, in our example, we will have two different rectangles that separate the positive and negative examples completely.
The notion of Version Spaces was introduced by Mitchell in 1997 as a framework for understanding the basic problem of supervised learning within the context of solution search.

In the specific boundary (SB) construction, the algorithm starts from the most specific hypotheses, which is the first positive training example that it encounters, then during the course of the algorithm, for each positive example it will generalize the hypotheses a little bit to cover this new example; as for the negative examples, it will eliminate the hypotheses that cover it, so it starts from the most specific hypotheses up to a more general ones.

As for the general boundary (GB) construction, it is quite the opposite, the algorithm starts from the most general hypotheses, the one that covers all the feature space, then during the course of the algorithm, for each negative training example, it will specialize the hypotheses that cover this negative example so it will not cover it anymore; as for a positive example, it will eliminate the hypotheses that does not cover it, so it starts from the most general hypotheses down to a more specific ones.

But why I chose this *Version Space* algorithm? The idea behind using this algorithm is that it is general: it does not make any assumption on the type of the space being processed, and on top of that its powerful enough to generate the right hypotheses from the data recorded which will work in our system.

In our system, I am only going to use the SB part of the algorithm, because the problem is really complex (5-6 objects in each observation with 5-6 attributes each, which means about 25-36 dimensions to treat) , and starting from the general hypotheses down the specific ones (the GB part) resulted in an enormous count of rules (2000 – 3000 rule) and they were taking lots of time to be generated (2-3 minutes for each new example at first, then 20-30 minutes for each example afterward). On the other hand the resulting hypotheses from the SB construction were really capable of capturing the behavior of the *mentor*, with only 100-200 rules (with the same number of observations that generated the previous 2000 – 3000 rules) and constructing them is even *on-line*: that means the imitation system is being constructed directly during the recording stage.

Now, before continuing, I need to define the concept of positive and negative examples in our simulation: The examples will be classified as positive and as negative based on the action part of the observation, that means if I have three actions: *walk*, *stop*, *jump*, and the example being processed contains the action *walk*, then it will be positive for the *walk* hypotheses and negative for the hypotheses of the two other actions.

After clearing out that point let us begin talking about what happens during the SB construction. When the imitation system receives two positive observations with the same ID, which means two observations with the same objects and the same action, it will try to combine them in a new rule as follows:

- At first, if the system receives the following observation:
  Object: vehicle, distance: 0.3, action: walk
  The new generated rule will be:
  *If* Object = vehicle and distance in [0.3, 03] *Then* walk
- Now for a new observation:
  Object: vehicle, distance: 0.5, action: walk
  The rule will be generalized:
  *If* Object = vehicle and distance in [0.3, 05] *Then* walk
- At the end if the new observation:
  Object: vehicle, distance: 0.4, action: walk
  Nothing will change

For each rule I will store two things: the number of observations that this rule covers and the time of these observations, so in the previous example the number of observations that the rule covers is 3 and the time of these observations is the sum of the time of each observation that we saw during the data filtering step.

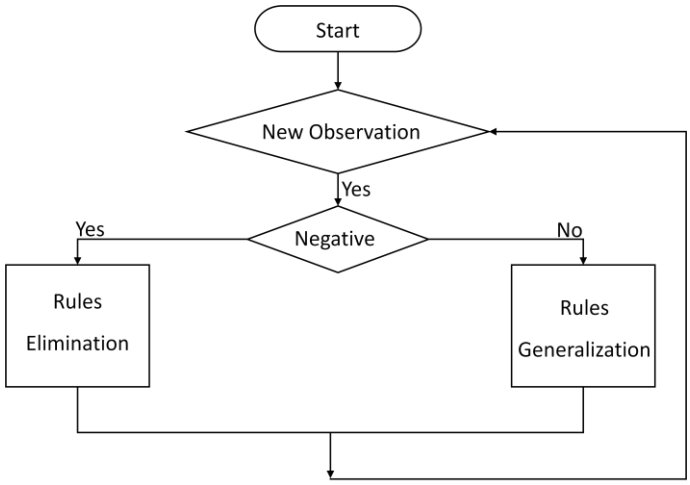



**Figure 38: Version Space, SB building algorithm**

During this machine learning process, the observations with the same ID will be processed together, so I will have a *Version Space* for each ID, and for each new observation I will choose the right VS based on the ID (if nothing found I will create a new one), then inside this VS I will generalize the rules with the same action and start the elimination process of the other rules with different actions (positive and negative examples).


**Figure 39: Working memory during the VS building process**

But this elimination process of the original algorithm can have some bad effects on our system, for an example:
Let us take 2 rules from a *Version Space*:

*If* Object = vehicle and distance in [0.3, 05] *Then* walk
*If* Object = vehicle and distance in [0.1, 02] *Then* stop

And a new observation:

Object: vehicle, distance: 0.4, action: stop

Normally the VS algorithm will eliminate the first rule and the second rule will be:

*If* Object = vehicle and distance in [0.1, 04] *Then* stop

But in this way we have lost some really important information, on the other hand if we tried to resolve the issue in the same VS, which means having intersecting rules:

*If* Object = vehicle and distance in [0.3, 05] *Then* walk
*If* Object = vehicle and distance in [0.1, 04] *Then* stop

This contradiction will affect the VS algorithm dramatically, during the following generalizations and eliminations.
But why we have this contradiction? It comes from the user, because during the course of the simulation he can take different actions based on the development of his *Avatar* or maybe he is experimenting with different strategies, that is why some researches limited the recording stage time (Tambellini & Sanza, 2006) so the user will not change his behavior, but we cannot do that in our case, and we need to deal with this contradiction, so our *Avatar* should adapt to this behavior changing.

#### *3.3.2.2.1. Version Space Modification*

I propose a modification on the VS algorithm to avoid the previous dilemma.
In the case of contradiction, I will create a new *Version Space* with the same ID which will process all the examples that causes contradiction with the original VS, and in the case of contradiction in the new VS, I will create a new one and so on. So the system could contain 2-3 VS's with the same ID but with rules that have contradiction with rules from the other VS's, this way I have maintained the maximum amount of information, and as side effect of this modification I practically removed the elimination step of the original algorithm.

| Espace de version ID = 223 | | | Espace de version ID = 223 | |
|---|---|---|---|---|
| Action 1 | Action 2 | Action 3 | Action 1 | Action 2 |
| Des Règles | Des Règles | Des Règles | Des Règles | Des Règles |

**Figure 40: Working memory during the Modified VS building process**

And the new algorithm will be:

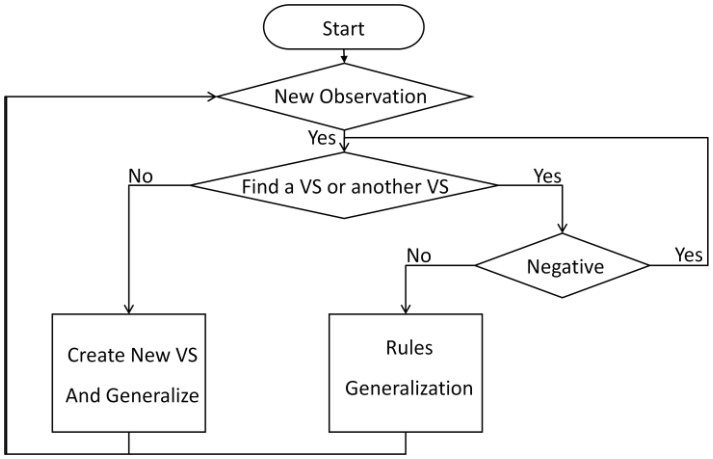



**Figure 41: Version Space, Proposed modified SB building algorithm**

So in the previous example we had 2 rules:

*If* Object = vehicle and distance in [0.3, 05] *Then* walk
*If* Object = vehicle and distance in [0.1, 02] *Then* stop

And the new observation was:

Object: vehicle, distance: 0.4, action: stop

In this case we will create two VS's, one with the following rules:

*If* Object = vehicle and distance in [0.3, 05] *Then* walk
*If* Object = vehicle and distance in [0.1, 02] *Then* stop

And the other with the next rule:

*If* Object = vehicle and distance in [0.4, 04] *Then* stop

Now, for a new observation like:

Object: vehicle, distance: 0.6, action: stop

We will only generalize the rule in the second VS so we will have:

*If* Object = vehicle and distance in [0.4, 0.6] *Then* stop

And so on.

At the end of this stage we will have rules that represent the behavior of the user.

#### 3.3.2.3. Force generation

Now that we have rules from the previous step, we need to generate the force for each of these new rules to be used in the re-playing stage.

This force will be based on two things, the number of the observations that each rule covers, and the time of these observations, so the force will be calculated from the percentage of the observations it covers from the total number of the observations of the recording stage, and the percentage of their time.

#### 3.3.2.4. Rules Fusion

The idea behind this step is to minimize the number of rules: so if we have two rules and one is more generalized than the other, we will keep the generalized one and update its force with the force of the specific one which will be eliminated.

This could happen during the same recording session, or during the fusion between the rules generated from a new recording stage with an already working imitation system created in a previous session.

Example of a rule and the interpretation of the fuzzy values:

| Actual rule: | Rule interpretation: |
|---|---|
| *If* ( | *If* ( |
| Object Type = TrafficLight | Object Type = TrafficLight |
| Distance in [0.624444..0.680000] | Distance: not that much far. |
| Orientation in [0.000000..1.000000] | Orientation: it does not matter. |
| Angle in [0.908163..0.989796] | Angle: looking at it directly. |
| Count in [0.500000..0.500000] | Count: only one. |
| Isvred in [0.000000..1.000000] | Isvred: light for vehicle: it does not matter. |
| Ispred in [1.000000..1.000000] | Ispred: light for pedestrians: red |
| ) and ( | ) and ( |
| Object Type = SideWalk | Object Type = SideWalk |
| Distance in [0.000000..0.980000] | Distance: not quite important |
| Orientation in [0.500000..0.500000] | Orientation: just in front of him |
| Angle in [0.000000..0.000000] | Angle: not important for surfaces. |
| Count in [0.500000..0.500000] | Count: only one. |
| On in [1.000000..1.000000] | On: is on it. |
| ) and ( | ) and ( |
| Object Type = CrossRoad | Object Type = CrossRoad |
| Distance in [0.891111..0.920000] | Distance: really close. |
| Orientation in [0.500000..0.500000] | Orientation: just in front of him |
| Angle in [0.000000..0.000000] | Angle: not important for surfaces. |
| Count in [0.500000..0.500000] | Count: only one. |
| On in [0.000000..0.000000] | On: not on it. |
| ) | ) |

*Then*

STOP

Number Of Observations = 19 → number of observations that this rule cover
Number Of Milliseconds = 50626 → the time of these observations in milliseconds
Force = 361.330383 → the force of the rule

### 3.3.3. Re-playing stage

After the previous machine learning stage we will have a *Knowledgebase* that consists of the rules that our *Avatar* has learned from his observations, we will use this *Knowledgebase* to construct a learning classifier system: ZCS, but not a normal ZCS because the rules in our case are not binary, so in other words it will be a *Generic* ZCS, and we are going to use only one LCS to represent the behavior of the observer just like Marc Métivier Without a *mentor* model (Guidance only) (Métivier, 2004), in this way the decisions that the mentor took during the recording stage will influence directly the behavior of the *Avatar*.

In addition of using a ZCS we need to define a couple of things in order to adapt this LCS to our environment:

- # symbol: In the definition of the LCS's, the condition alphabet contains the symbol # which means *whatever*, but our system is generic, thus the # symbol will equal to the whole fuzzy values range (# = [0..1]), in this way all the fuzzy values will be accepted by this range.
- Environment Reward: the reward is based on our 3D-city simulation, that is why we tried to make the reward system to feel as natural as possible, for example:
    - The *Avatar* will receive a negative reward if he was involved in an accident with a vehicle
    - The *Avatar* will receive a positive reward if he crossed the street on the right place (on a cross road) respecting the right conditions (the traffic light is green for the pedestrians)
    - ….
- Rules Force: since the reward could be negative, we will have to insure that the force of the rules stays always positive, so we need to delete the rules with negative force.
- Genetic Algorithm: Because we have only one LCS, we did not use a genetic algorithm, in order to maintain the behavior of the mentor intact.

At the end, we have created a system that controls the behavior of our *Avatar* based on what he learned from the mentor during the recording stage, with the force of the rules evolving based on the *Bucket Bridge* algorithm.

## 3.4. Observations

The following observations will be based on imitating our mentor while he is crossing the street, although it looks like a simple problem, but as matter of a fact there is a lot of things to account for like: the traffic light, the other pedestrians, the vehicles…beside, we should not forget that the observer should infer the importance of the color of the traffic light all alone, so this small problem is a really a good test bed for our system, we tested our system in three different scenarios:

- Scenario 1: We removed all vehicles and other pedestrians, so the streets were empty.
- Scenario 2: We added vehicles.
- Scenario 3: We added the pedestrians, so the environment contained everything.

In all of those scenarios our *mentor* was behaving normally: he was always stopping when the traffic light is red.

### 3.4.1. Results

| Scenario No. | Number of data recorded | Time of recording |
|---|---|---|
| 1 | ≈ 500 observation | ≈ 3 minutes |
| 2 | ≈ 1900 observation | ≈ 5 minutes |
| 3 | ≈ 35000 observation | ≈ 40 minutes |

- Scenario 1: our *Avatar* was capable of understanding the importance of the traffic light, so after only 2-3 street crossing of the *mentor*, and during the re-playing stage he was stopping when the traffic light is red, so it was a successful imitation.
- Scenario 2: Since the number of vehicles is a lot more significant than the number of traffic lights (5-8 vehicles for 2 traffic lights). Because of that, the vehicles were a lot more important in affecting his behavior than the traffic lights, that is why sometimes he was crossing the street when there is no vehicles although the traffic light was red.
- Scenario 3: like in scenario 2, the other pedestrians and vehicles will be more important to him than the traffic light, so sometimes our *Avatar* was stopping when he sees that the other pedestrians are stopping or when there are some vehicles in front of him, and like the previous scenario sometimes he was crossing the street when there is no vehicles although the traffic light was red, and some pedestrians already stopped.

Plus the previous results we made several more observations:

- In all these scenarios the system at the end will converge to the same amount of rules (≈13 rules for scenario 1, ≈40-50 for scenario 2, ≈170-180 for scenario 3), the only benefit of controlling the *Avatar* longer than that: fine tuning the force for each rule, and the duration mentioned in the previous results, are the minimum durations that we found in order to get the best possible imitation. The difference of the number of rules between the scenarios is due to the complexity of the environment.
- The more the environment is complex, the more we encountered a noise problem: the system became less capable of discovering the important objects (the traffic light in

our case), so all objects have the same degree of importance, so they were all capable of influencing our *Avatar* action with the same force.

- *Overfitting*: the system will try to behave like the *mentor* as much as possible, but that will create problems when placing the system in a different environment like using the *Knowledgebase* of scenario 3 in scenario 1: In this case none of the rules conditions will be satisfied (there is neither vehicles nor pedestrians) due to the noise problems that we talked about earlier, and the system will use the *Covering operator* of the LCS to create new classifiers most of the time, as if it did not learn anything. But if we used the *Knowledgebase* learned from scenario 1 in scenario 3: we will have a flawless execution, because the *Avatar* in scenario 1 learned the importance of the traffic light.
- The system will imitate the user even if he made the wrong choices (like crossing the street on the red light), so, when we did several other experiments with a *mentor* not behaving logically, our *Avatar* was imitating him, by taking those wrong actions. And, although the LCS will try to penalize the rules that could lead to such actions, those decisions will be made at least 1-2 times, before these rules lose their force.

## 3.5. Conclusions

At the end, our system was capable of imitating the *mentor* to some extent, but we tried to make the system as much general as possible, so for instance, in crossing the road problem we did not specify the important objects for the system and we left it to infer that alone, because of that sometimes we would have some odd situations where the *Avatar* takes illogical decisions (from an external point of view) during the re-playing stage; that is because he takes into account everything he sees.

But in general the imitation was successful and the final *Knowledgebase* really modeled the *mentor* behavior.

## 3.6. Implementation in an MMORPG

Massively multiplayer online role-playing game or MMORPG is a genre of computer role playing games or RPG's where the player controls an avatar in a virtual world imagined by the developer. The difference between an RPG and an MMORPG is the huge number of players playing at a certain time and most importantly the persistent world: it continues to exist and evolve while the player is away from the game, which creates a virtual world that does not stop of evolving when the player disconnects.

This kind of games gained a lot of momentum lately, not because players fight against enemies, environment hazard or even other human controlled avatars, but because it is a social experience where the player meets other people, teams up with them to do more difficult quests and raids…So the social networking aspect of this kind of games is really important, and one of the driving forces of the players to connects and play this kind of games.

One of the most famous MMORPG is called *World of Warcraft* or WoW developed by Blizzard Entertainment, it was released in 2004 and in 2008 there were more than 10 million subscribers to this vast universe (about 62% of the massively multiplayer online game market), with about 1-2 millions of active users at the same time.

**Figure 42: World of Warcraft**

Beside normal MMORPG's like WOW or EVE Online (a space simulation based MMORPG), there is a sub genre of MMORPG where the player avatar is in a real virtual world (non fictional one), he can explore, meet other avatars, socialize, participate in individual and group activities, and create and trade virtual properties and services with one another just like in the real world, they are called Social Virtual Worlds, and one of the famous is called *Second Life* developed by Linden Labs, it was released in 2003 and in 2008 there was more than 8 million subscribers but with less than 70 thousands active user at the same time (Varvello M. & Picconi F. , 2008).

**Figure 43: Second Life**

The problem with these kinds of games is the difference between the number of subscribers and the number of active users at once, so for an example: although there are about 8 million subscribers in *Second Life*, with only 70 thousands active users, this massive virtual world feels empty most of the time, furthermore most of the population is concentrated around famous places because there is more chances to meet other avatars.

One of the uses of our imitation system is to solve this problem: to keep all 8 million subscribers on-line in a game like *Second Life*, to keep them always in the virtual world, in this way the world will be always populated. So while the player is navigating in the virtual world and interacting with other players, our system will be learning the places he likes, the criteria of the other peoples that he interacts with, and the way he interacts with them (playing games with them, dancing….), and when our system controls the user's *Avatar*, it will try to search for people that could interest the user, and it will try to interact with them, thus when the player re-connects to this virtual world, our system will give him a list of the people it met, the activities it did, and a summary of their characteristics, so that the user could have some new friends that might interest him (Hu S. Y. & Jiang J. R. , 2008). Beside, our system, through the use of LCS's and the concept of exploration / exploitation, could try to go to new places which the user would never think about going to. As a result, this virtual world will be always populated, and moreover most places will be populated too.

And we can also use our system in MMORPG's like *World of Warcraft*, hence the avatar will gain experience, some fighting skills, or other items while the user is off-line. However the utilization of Bots[5] to control the user avatar is forbidden, because the users might abuse the use of it to gain levels and experience by letting the system control their avatar for consecutive days. In other games, like EVE online, *Avatars* Still gain experience when they are not connected (e.g. they learn new abilities) and it could be easier to integrate our system in such games.

[5] Computer controlled character that the player compete against in video games, and mostly in FPS's

## 3.7. Future work

There are several propositions in order to either make the system more capable of modeling the *mentor* behavior, or to use it in a new type of applications:

- Learning the behavior of two users and interpolating between them, for example: in the street crossing problems the system can learn the behavior of a young user who will not pay much attention to the traffic light but crosses when the street is empty (Risky behavior), and another old user who always crosses depending on the traffic light always (Safe behavior), so we would have two different behaviors that we could interpolate between them to create several other behaviors that has some degree of risk and safety in them.
- In order to make our imitating system more powerful we can use Marc Métivier with *mentor* model (Métivier, 2004): using his DoubleCS system, we can separate the *mentor* LCS of the *observer* LCS, so in this way our *observer* behavior will only be influenced by the behavior of the *mentor*, not controlled totally by it.
- Making the learning stage more context sensitive: filtering the important objects in a certain situation, this way, we will have several LCS's each for a specific problem, and the decision for the right LCS could be realized by using another LCS (an MCS structure).

# Bibliography

Atkeson, C. G., & Schaal, S. (1997). Robot learning from demonstration. *Proceedings of the Fourteenth International Conference on Machine Learning*, (pp. 12-20). Nashville.

Barricelli, N. A. (1954). *Esempi numerici di processi di evoluzione [Numerical examples of evolution processes]*.

Bellman, R. (1957). A Markovian Decision Process. *Journal of Mathematics and Mechanics 6* .

Crosby, J. L. (1973). *Computer Simulation in Genetics.* London: John Wiley & Sons.
Darwin, C. (1859). *On the Origin of Species.*

Fraser, A., & Burnell, D. (1970). *Computer Models in Genetics.* New York: McGraw-Hill.

Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning.* Massachusetts: Addison-Wesley.

Holland, J. H. (1986). *Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In: Michalski RS, Carbonell JG, Mitchell TM (eds) Machine learning, an artificial intelligence approach.* San Francisco: Morgan Kaufmann.

Hu S. Y. & Jiang J. R. (2008). *Plug : Virtual Worlds for Millions of People.*

Kam Ho, T. (1992). *A Theory of Multiple Classifier Systems And Its Application to Visual Word Recognition.*

Lin, L.-J. (1992). *Self-improving reactive agents based on reinforcement learning, planning and teaching.*

Métivier, M. (2004). *Méthodes évolutionnaires & apprentissage: Apprentissage par imitation dans le cadre des systèmes de classeurs.* Ph.D. thesis, l'Université RENE DESCARTES - PARIS 5.

Mitchell, T. M. (1997). *Machine Learning.* Boston: McGraw-Hill.

Olivier, H., & Stephane, S. (2004). *Generic Classifiers System and Learning Behaviours in Virtual Worlds.* Proceedings of the 2004 International Conference on Cyberworlds.

Price, B. (2003). *Accelerating Reinforcement Learning with Imitation.* Ph.D. thesis, University of British Columbia.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers.

Quinlan, J. R. (1986). *Induction of Decision Trees.*

Rosenblatt, F. (1958). *The perceptron.* Cornell Aeronautical Laboratory, inc.

Sanza, C. (2001). *Evolution d'entites virtuelles cooperatives par systeme de classifieurs.*

Stolzmann, W. (1998). *Anticipatory classifier systems.* Morgan Kaufmann.

Tambellini, W., & Sanza, C. (2006). Behaviors Generation with Artificial Intelligence in Video Games. *International Digital Games Conference*, (pp. 217-220). Portalegre, Portugal.

Thurau, C., Sagerer, G., & Bauckhage, C. (2004). Imitation learning at all levels of Game-AI. *International Conference on Computer Games: Artificial Intelligence, Design and Education*, (pp. 402-408).

Turing, A. (1936). *On Computable Numbers, With an Application to the Entscheidungs problem.*

Varvello M. & Picconi F. (2008). *Is There Life in Second Life?*.

Whitehead, S. D. (1991). Complexity analysis of cooperative mechanisms in reinforcement learning. *Proceedings of the Ninth National Conference on Artificial Intelligence*, (pp. 607-613). Anaheim.

Wilson, S. W. (1995). *Classic Fitness Based on Accuracy.*

Wilson, S. W. (1994). *Zcs : A zeroth level classifier.*

Zadeh, L. A. (1965). *Fuzzy sets and systems.* New York: Polytechnic Press.

# Appendix

## Appendix A: Vehicles and pedestrians rule based system

Both the vehicles and pedestrians in our 3D city simulation are controlled by a rule based system: so they have a field of view just like our *Avatar* in the form of a 3D cone, and for each frame we have a list of the objects state that each of them can see and sense, if this list satisfy a rule condition, its action will be executed directly.

The idea behind using a rule based system to model these objects behavior is quite simple: their behavior is really simple as we will see. Their *knowledgebase* does not contain more than 2-3 rules, so it is really easy to maintain.

- **Vehicles:**
  - *If* traffic light is close *and* is red for vehicles *Then* STOP
  - *If* vehicle is close *and* in front *and* not advancing *Then* STOP
  - *If* vehicle is really close *and* in front *and* advancing *Then* STOP
  - *If* pedestrian is close *and* in front *Then* STOP
  - *Else* GO
- **Pedestrians:**
  - *If* traffic light is close *and* is red for pedestrians *and* just looking at it *Plus* on side walk *Then* STOP
  - *If* vehicle is close and in front *Plus* not on side walk *Then* STOP
  - *If* On cross road *Then* WALK FAST
  - *Else* WALK NORMAL