CREATING A DOGFIGHT AGENT

The design and implementation of a fully autonomous agent that can fly an airplane during a one-to-one dogfight

Solinger, David

Technical Report DKS-05-01 / ICE 10
Version 1.0, April 2005
Mediamatics / Data and Knowledge Systems group

Solinger, David (david@ch.tudelft.nl)
Student number: 9784808

"Creating a Dogfight Agent: The design and implementation of a fully autonomous agent that can fly an airplane during a one-to-one dogfight"

Technical Report DKS-05-01 / ICE 10
Version 1.0, April 2005

Mediamatics / Data and Knowledge Systems group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, The Netherlands

http://www.kbs.twi.tudelft.nl

Keywords: ICE project, knowledge based systems, artificial intelligence, agents, dogfight, game AI, flight automation

Abstract

The report describes the design and the implementation of a prototype for a fully autonomous agent, which can fly an airplane during a dogfight. This project is one of the multiple projects that are done within the ICE project of the Knowledge Based Systems group of the Delft University of Technology. Until now, many projects, also within the ICE project, focused on one specific element of flight automation. This project focuses on combining multiple elements of flight automation in one fully autonomous agent. In this project certain methods that are proposed in earlier reports are studied and some of them are used for the dogfight agent. A flexible and modular overall architecture is presented in which, for each element of the flight automation process, different methods can be implemented and replaced easily. The methods that are used for the prototype implementation are presented with a detailed design and a description of the implementation. The prototype is successfully implemented and can really act as a competitive player during a one-to-one dogfight in Microsoft Combat Flight Simulator. An important conclusion is that the performance of the whole agent is dependent on the performance of all the different elements. This conclusion underlines the relevance of this project, because the basis of the project was to split the concept of flight automation into multiple elements, so it is possible to focus on the individual elements.

Preface

This report is the result of my graduation project at the Knowledge Based Systems group of the Delft University of Technology. After spending more than eight years at the study Technical Informatics of the Delft University of Technology, I think it is worth to write some words about this.

In 1996 I choose to start a study Informatics. I was always very interested in science and technology, but a more important reason was: I did not understand anything of a computer. It seemed a miracle to me how a machine could act like some person had programmed it to for as many times as it was desired. During the first year a whole new world opened for me and this resulted in a propaedeuse with honors.

An important aspect of my study was the study organization 'W.I.S.V. Christiaan Huygens'. This active and motivating organization made me enjoy both the social and the scientific sides of my study much more. Computer-programming contests were my favorite events. I participated for many years as a contestant, a member of the organization and as a judge. Thanks to 'Christiaan Huygens', nationally and internationally well-known contests were often organized at the Delft University of Technology.

This graduation project is done at the Knowledge Based Systems group. My special thanks go to ir. Patrick Ehlert and drs. dr. Leon Rothkrantz for their support during this project. Before you start reading this report I like to rememorize some words dr. Peter Kluit spoke during a speech for being 25-years in education. He described a University as a house: whereas the walls are necessary elements of a house, the most useful elements are not the walls, but the empty spaces between them.

Table of Contents

1. Introduction

1.1. Background

It is already more than 100 years ago that the first airplane was build by the Wright brothers. During that period the capabilities of aircraft have been improved enormously. Besides this the amount of other aircrafts in the same area is also increased, so that in some areas in the air it has become 'busy'. Altogether the information load for a pilot has continuously been growing. This is even more the case for military pilots. The capabilities of military aircraft are on the limits of what a human being can handle. Furthermore, a military pilot needs to keep track of both friends and enemies, in the air and on the ground, that are within a range of tens of miles away.

In situations like this humans normally start to look for ways in which a computer can help them. So projects have been started to investigate and design human-support systems for use in a cockpit. The intelligent cockpit environment (ICE) project is a project of the Knowledge Based Systems group of the Delft University of Technology. The goal of the ICE project is to design, test and evaluate computational techniques that can be used in the development of intelligent situation-aware crew assistance systems [Ehlert, 2003]. This report is a description of a graduation project that has been done within the larger ICE project.

1.2. Project description

Within the ICE project different areas of flight automation have been addressed, like: situation recognition, decision-making, flight planning and flight control. This resulted in a number of ideas and methods for modern flight automation. Most of them have successfully been tested, but not in combination with other ideas. The ideas are not combined or the influence of one task on another is not studied. For example, a good situation recognition method can maybe improve the decision-making. In this project multiple ideas and methods of the ICE project are combined in one program. All necessary tasks are added, so that it can act completely by itself, without any human interventions.

This will result in an architecture in which new methods for flight automation can be designed, developed and tested. Furthermore this project focuses on the situation recognition process, the decision-making process and the execution process individually. These parts of flight automation are already addressed before in the ICE project, but the current techniques are not good and/or not detailed enough to be usable in this project.

Projects like this one, also in the ICE project, often follow a corpus-based approach. This means that the automated process is modeled after a human expert, in case of flight automation projects often a human pilot. In this project, a design-based approach is used. Processes are not necessarily modeled after a human pilot, but are designed from paper, just to perform as good as possible. However, a human model might be used for certain tasks when this seems to be better or easier.

One of the most difficult scenarios for flight automation is the dogfight, so that's why this scenario is chosen.

1.3. Project goal

The goal of this project is to create an agent that can 'play' the computer game Microsoft Combat Flight Simulator (MSCFS). In this simulator game the agent has to be a competitive player in a one-to-one dogfight. The opponent of this agent can be either a human player or another computer program. Figure 1 shows how two players take part in the same multiplayer game of MSCFS.

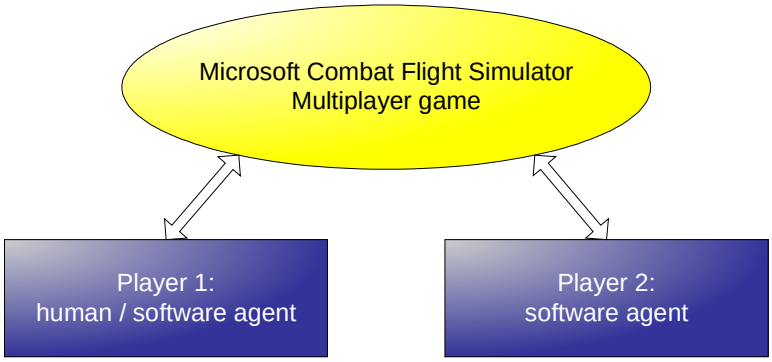

Figure 1: Two players in a MSCFS multiplayer game

The agent should combine methods discovered in the ICE project in a flexible design and a nice implementation so that the program can be used as a test environment for future research.

To create this, a literature study is required. Literature about the ICE project and relevant flight-related studies will be studied. Besides this also projects about intelligent games and simulation will be included in the literature study.

1.4. Project assignment

Altogether the goal of this project, to create an agent for MSCFS, will consist of three parts:

- A literature study about related projects
- A model and a design of the agent
- A prototype of the agent in the form of a C++ implementation

1.4. Requirements

First of all it is important that the bot really acts like an agent. This means that the program must perceive, reason and act completely by itself. The bot may not be helped by a human or by some other computer program while it is playing the MSCFS game.

The bot has to 'play' the simulator game like human players do. This means the bot may not cheat. Human player have access to situation parameters like 'current altitude', 'current speed' and 'range between aircrafts'. So, also the agent will have access to this kind of parameters only. Of course, a computer program, can compute by itself other parameters once the basic ones are retrieved. Also the output of the agent will be the same as the output of a human player. For example: The agent cannot set its current location right away, but it can set the elevator and aileron, in the same way a human player can do this using the joystick.

The MSCFS is a real-time game. Therefore the agent needs to be a real-time player. This means that the program cannot take some minutes to set up a strategy, because then the game would already be lost. Even more, when the agent program is running, the MSCFS is also running and that game consumes only by itself a lot of CPU time. To let everything work smoothly, the program may use a maximum of about 20% CPU time.

The program must be suitable as a test environment for future research. The architecture must be flexible and modular, so that individual parts of it can be changed, extended or replaced by new work. It must also be possible to add new modules if necessary. For the same reason the implementation needs to be of high quality, so that other programmers can understand, change and extend it.

1.5. Report overview

The layout of this report is not very unusual. Within this project preliminary research has been done. This is described in chapter 2. This chapter also describes some basic knowledge about flying. The chapters 3, 4 and 5 describe the overall architecture, the design of the AI part and the implementation. Chapter 6 describes the tests that are executed and the results of these tests. The conclusions are made in chapter 7 and the report ends with a few suggestions for interesting future work in chapter 8 that can be done based on this project.

2. Preliminary research

2.1. Literature study

2.1.1. Introduction

A literature study was done to see what results are already available in this area. There are of course a lot of studies made in the areas of flight automation and intelligent computer games. It is the goal of the literature study to see if it is possible to create a complete dogfight agent that can play the MSCFS. Furthermore it is interesting to see if there do exist methods and techniques that can be useful in creating the agent.

Many papers have been studied. About 20 of the most interesting papers are studied in more detail. This is described in [Solinger, 2004]. The set of studies can actually be divided into three different subjects. Therefore the description of the studies is divided into the same three subjects in [Solinger, 2004] and also in this chapter. The first subject contains papers within the ICE project. The second subject is about Flight Automation in general and the third subject is about intelligence in computer games.

2.1.2. ICE project

The goal of the ICE project is to design, test, and evaluate computational techniques that can be used in the development of intelligent situation-aware crew assistance systems [Ehlert, 2003]. Using methods from artificial intelligence, ICE focuses primarily on the data fusion, data processing and reasoning part of these systems. Special issues addressed in the ICE project are:

- Situation recognition
- Mission or flight plan monitoring
- Attack management
- Pilot workload monitoring

Until now most studies within the ICE project are done on the issue of situation recognition. In different studies, different techniques are studied. For example in [Mouthaan, 2003] a Bayesian belief network is used and in [Capkova, 2002] a method is studied that uses a neural network. In all cases however the situations to recognize are different from the most important situations in a dogfight. In a dogfight most situations will deal with the relation between two aircrafts like being in or under attack. All studies in the ICE project so far deal with situations that involve only one aircraft like 'taxiing' or 'turn right'.

Another interesting study within the ICE project is described in [Andriambololona, 2003]. This study is about decision-making for a dogfight agent. The idea is to set up a decision-tree with decisions modeled after decisions as they are made by human pilots. However the results of this study do not satisfy enough to use the proposed model directly in a real-time dogfight agent. Furthermore the proposed method is not detailed enough which makes it impossible to use this work directly. This all does not prevent this study from being usable as a good starting point for decision-making in a dogfight agent.

The last study within the ICE project that needs to be mentioned here is about the actual construction of an autonomous flight bot [Tamerius, 2003]. This paper describes the construction of an agent that can fly a human-defined flight plan. The agent first checks whether the flight plan is correct and consistent. Thereafter it defines consecutive steps that are necessary to execute the flight plan. The last step is the actual execution of the flight plan. In a dogfight situation a predefined flight plan is not available, but the execution of specific steps is also necessary in a dogfight agent. Similar with [Andriambololona, 2003] the results of this project are not good enough to use it directly in a dogfight agent, but again it can be a good starting point for the execution part of the dogfight agent.

3.1.3. AI in flight automation

Besides the ICE project there are many other projects that focus on flight automation in one way or the other. There certainly are some interesting projects amongst those. In most of the projects a simulated environment is used to test new ideas. The reasons that are brought forward are the same as for the ICE project. In a simulated environment it is much easier, faster, less dangerous and cheaper to test new ideas than in a real-world environment.

There are many different aspects of flight automation and of course research is done in all those different aspects. The most interesting projects with respect to this project about an agent for MSCFS are of course the projects about autonomous agents for flight simulators. Many of the interesting projects are about strategy (or decision-making). Situation recognition is often not named as a big issue in projects about flight agents. Execution is often not even mentioned. Probably this is because automated execution of flight maneuvers is already used for many years in real world situations and is regarded as not-a-problem.

In many projects the dynamic, complex and real-time environment is seen as the biggest challenge of the project. The locations of the airplane and the opponent(s) change constantly, also when no explicit action is taken. Furthermore the environment is real-time which means that there will never be a lot of time for reasoning so that time-consuming algorithms are not usable.

One of the interesting projects is about agents that can fly military missions in a military simulator [Laird, 1998]. The agents are created to create both enemies and friends for human pilots. The agents are created in the Soar-system. Soar is a rule-based system that is designed to create agents for real-time simulations. The decisions in this project are mainly on a higher level than decisions in a dogfight situation. For example a rule in this system can be 'if an enemy plane is in front of us, go into dogfight'. It is not clear how this dogfight will be executed. This is often the case in projects about decision-making in flight automation. In many projects there is no implementation created, so there is no need to specify more detailed and lower-level decisions, but in the Soar project there is a real implementation, so there must be some way for an agent to deal with a dogfight situation. Probably the dogfight capability of the agents is not very enhanced. The goal of the project is more to use it for training of different military missions, than for training of actual dogfights.

3.1.4. AI in computer games

AI in computer games exists since the dawn of video games in the 70's. In recent years the interest in AI in computer games grew for several reasons [Tozour, 2002]. One reason is that because of the increasing complexity of modern computer games, it becomes more difficult to create interesting digital opponents for human players. More time and money is spent on creating a new video game, but also on creating digital opponents in computer games. This has led to the use of more sophisticated AI, with success.

Another important reason for the increasing interest in AI in computer games comes from the academic world. As it is discussed in section 3.1.3 it is far more easy, fast and cheap to test new ideas in a simulated environment, such as a computer game. Despite this, until now the most successful projects of AI in computer games are created with relatively simple AI techniques like finite-state machines, decision trees and rule-based systems. The complexity of a simulated environment may be lower than in a real-world situation, but the complexity of modern video games is already quite high.

An interesting project in this area is the development of the Quake III Arena Bot [Waveren, 2001]. In this project an artificial player is created for the computer game Quake that can serve as an interesting and challenging opponent for human players. Therefore it needs some human-like behavior. The bot, for example, needs to learn the map of a level while playing, just like normal human players do. The Quake III Arena Bot is developed successfully for commercial purposes. It will be of no surprise that it is based on proven and good-to-understand AI techniques like a finite state machine and a rule-based system.

Despite the success of quite basic AI techniques in computer games, there is also a lot of research in the use of more sophisticated AI techniques in computer games. In [Spronk, 2003] neural networks are used in combination with genetic selection. This project shows that with these AI techniques it is possible to create challenging artificial players that can explore new strategies in a strategy game. However the success of projects like these is depending heavily on the simplicity of the computer game. Until now these more sophisticated AI techniques are tried out in more complex computer games only for subtasks.

3.1.5. Conclusions

A lot of research has already been done in the area of flight automation. The ICE project and also other projects about flight automation include research in all different aspects of flight automation, such as situation recognition, decision-making or flight control. Furthermore successful implementations are created for agents that can play a complex computer game like for example Quake. An implementation of a dogfight agent will be a combination of everything above. Many different aspects of flight automation will be included in an agent that can play the computer game MSCFS. Since there are satisfying or at least reasonable results in all these areas it must be possible to actually create a dogfight agent for MSCFS. However the performance of such a dogfight agent cannot be predicted in advance. The only way to find out is to design and develop the dogfight agent and to test it in its prescribed environment.

2.2. AI methods and techniques

There are many different AI methods and techniques used in flight automation or AI in computer games. Some of them are described in this section. This is by no means a complete list, but this is just a list of AI methods and techniques that are found more often in literature about flight automation or AI in computer games. The list below is a little bit based on the lists in [Waveren, 2001] and [Ehlert, 2003]. More about AI methods and techniques can be found in [Russell, 1995].

2.2.1. Finite state machines

A finite state machine is a system that is modelled with a limited number of states of operation, as for example: 'normal flight', 'attack', 'defend', 'fly away'. In each time step one of the states is active, the others are not. Due to changes in the system, or in the environment, the system can change to another state. There are more sophisticated methods to model reasoning in different states, but the simplicity of finite state machines makes them rather popular. Finite state machines are often used, with success, in AI in complex computer games.

2.2.2. Rule-based systems

A rule-based system consists of a lot of rules to describe what to do in predefined situations. It requires a lot of knowledge about all different situations to make explicit rules for the rule-based system, for example: 'IF an enemy is on my right side THEN make a turn left'. It is not very difficult to set up a rule-based system. The only problem is that one needs to define enough rules to deal with all sorts of situations that can occur. Like finite state machines, rule-based systems are also popular because of their simplicity. They are often used in flight automation projects or in complex computer game AI, sometimes in combination with a finite state machine. It is rather easy to keep track of the reasoning of a rule-based system and so to explain and improve strange or undesired behaviour.

2.2.3. Decision trees

A decision tree is actually also based on rules. But in a decision tree rules are put into the form of a tree. In each node a rule will decide which path must be followed in the tree. After some nodes the path will end in a leave. The leaves contain all actions that can be taken. By following the right path the system will end up in the leave with the action that is appropriate to the current situation. Decision trees are often used to create the decision-making mechanism in agents. Again their simplicity makes them easy to use and easy to debug.

2.2.4. Neural networks

A neural network consists of nodes and links between these nodes. Each link has a weight. Before a neural network can be used, it must be trained first. Training is done by giving the network examples. In the training process the weights are updated. There is no need to put explicit knowledge into the neural network as with the previously described methods. This makes neural networks very suitable to use in case no explicit knowledge is available. The biggest disadvantage of a neural network is that it is very difficult to keep track of its reasoning process. The reasoning of a neural network is not transparent. Therefore the reason for strange or undesired behaviour is difficult to find.

2.2.5. Bayesian belief networks

A Bayesian belief network consists, just as a neural network, of nodes and links between the nodes. The links are directed. A node contains a fact and a link contains the probability that a fact will be true when the fact in the other node is true. By reasoning with probabilities it is possible to reason with uncertainty. Bayesian belief networks are often used and they are proven to be successful in a number of applications. The biggest problem is to set all the probabilities in the network. These probabilities can be learned from a large dataset or they can explicitly set by an expert. Anyway the success of the application stands with the quality of the probabilities.

2.2.6. Genetic algorithms

With genetic algorithms the best values for certain properties are learned using natural selection. First multiple versions of a system are created and they are all a little bit different from each other. After some tests the best ones are selected and some similar ones are added to the population. These steps are repeated many times until finally only good versions are left. The advantage of this method is that no extensive domain knowledge is required. One of the disadvantages is that the natural selection can be a time consuming task. It is also not guaranteed that the best solution, or even a good solution, is found.

2.3. Basic flight knowledge

2.3.1. Flight parameters

To fully understand the design and working of the bot, it is necessary to know a little bit of flying itself. Therefore the most important flight parameters are explained in this chapter. More about flight parameters is explained in [Machado, 2002]. There is a distinction between parameters for observation and parameters for action. Parameters for observation tell something about the current situation of the airplane or a combination of more airplanes. Parameters for action are used to have control over the airplane.

The situation of a single airplane can be described with certain parameters. Some of them do not really need an extra explanation like: speed, altitude and location. The two most important parameters of the actual flight are less known by non-pilots.

The pitch is the angle that determines how much the airplane goes up or down. The bank is the angle that tells how much the airplane is rolling over its longitude axis. This rolling will happen when the plane makes a turn. When the airplane flies completely straight, both pitch and bank will be zero.

Figure 2: Pitch (left) and bank (right)

Because this project is about a one-to-one dogfight situation some other important parameters are about how the situation of an airplane is related to the situation of the opponent. The range between the two airplanes is such a parameter. Notice that this range is the only parameter in a three-dimensional space. The aspect angel is the angle between the headings of both airplanes.

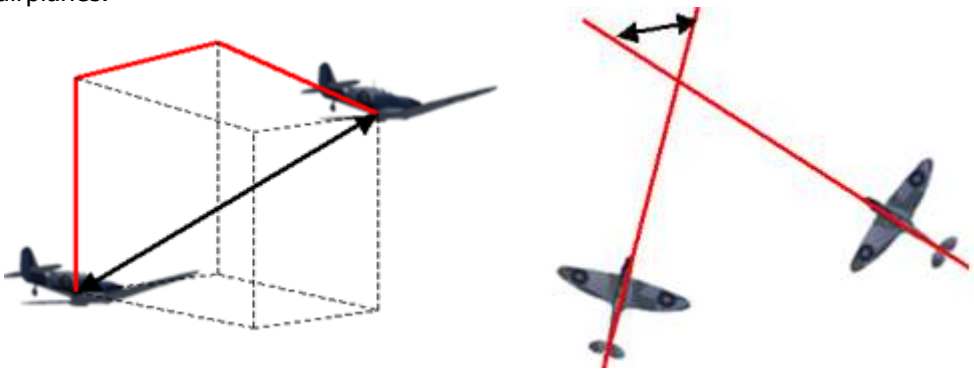

Figure 3: Range (left) and aspect angle (right)

The angle off is the angle between the heading of an airplane and a straight line from the airplane towards the opponent. Notice that this angle off is different in the opponents view, in most cases.

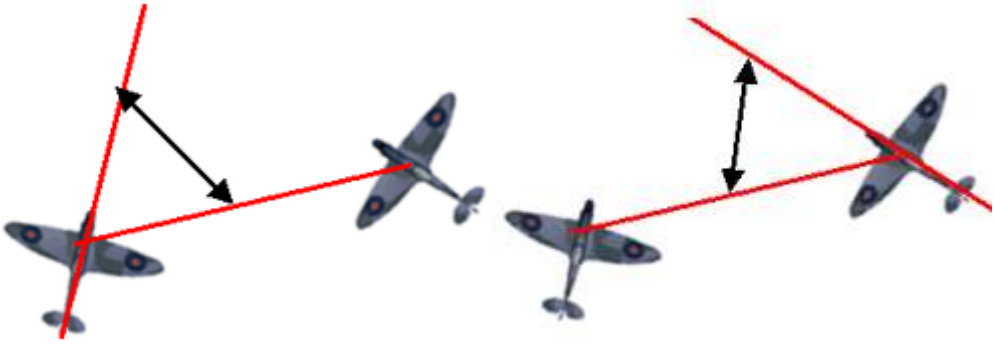

Figure 4: Angle off (left) and angle off in opponents view (right)

2.3.2. Input control parameters

There are three important action parameters in an airplane: elevator, aileron and throttle. The throttle is actually the same as the gas pedal is in a car. More throttle means the airplane its speed will increase or it will be able to fly towards a higher altitude.

The elevator is the action parameter to control the pitch. If the elevator goes up, also the pitch goes up, this means the nose of the airplane is going down. The aileron is the action parameter to control the bank of the airplane. The aileron is carried out by extensions on the wings, so that when activated, one wing will go up and the other will go down. This way the airplane will rotate clockwise or counter clockwise.

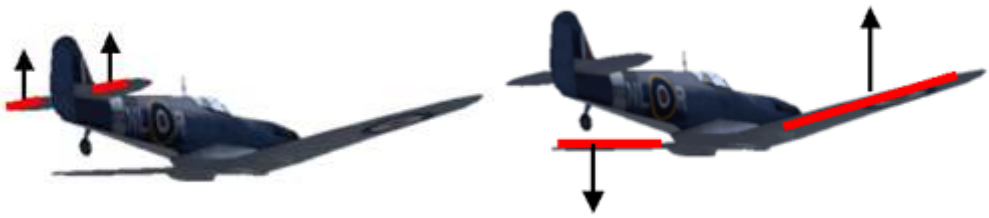

Figure 5: Elevator (left) and aileron (right)

2.3.3. Flight maneuvers

In a dogfight situation an aircraft is constantly making a specific flight maneuver. Deciding which maneuver to make and performing that maneuver are, besides shooting, the only things a pilot, or a dogfight agent, can do to win the dogfight. So the specific flight maneuvers are important and they will be mentioned in different sections of this report. Therefore some basic flight maneuvers are described in this section. There are more flight maneuvers, but in this project only the ones described below are used. Maybe these maneuvers are not exactly as the ones used in real life, but they are all useful in the MSCFS game.

Straight flight

A straight flight is a flight without a lot of movement. So no heavy turns or climbs will occur. A straight flight is a continuous maneuver, so without a real begin or end. It will just end when another maneuver is started. In a straight flight it will be possible to make some adjustments in direction and altitude when it is necessary.

Turn (left & right)

A turn left is in fact the same as a turn right except for the direction of course. Also a turn has no real begin or end. An aircraft can continuously make a turn and will fly in a circle until it runs out of fuel. A turn is as sharp as possible as long as the aircraft will stay on the same altitude. It will not be possible to make a turn less sharp. In a heavy dogfight such a turn will be of little use. When it is necessary to make some smaller adjustments in direction the straight flight maneuver can be made.

Extreme turn (left & right)

Sometimes it is desired to make a turn as sharp as possible, for example when the enemy is very close behind. This can be done with an extreme turn. This maneuver is almost the same as a normal turn, but there is no constraint on altitude. In an extreme turn the aircraft will definitely loose some altitude and some speed.

Looping

In a looping the aircraft will go up by raising the elevator. When this state is hold on, the aircraft will make a loop and end up at about the same location as where the loop started. This can be useful when an enemy is close behind. After making the loop one will be in a position close behind the enemy, unless the enemy makes an intelligent move at the same time of course. There are no adjustments possible in a loop, but it is possible to stop with the loop and go on with another maneuver before the loop is completed.

Split-S maneuver

A split-s is a maneuver to change the direction by 180 degrees. It starts with turning the aircraft around its longitude axis to bring it in an upside-down position. The aircraft is then in the same situation as halfway a loop. The rest of the split-s turn is the same as finishing the second half of a loop maneuver. The aircraft will end up flying normally in the opposite direction with regards to the situation just before the maneuver was started.

Immelmann turn

An Immelmann turn is almost the same as a split-s, but then the other way around. In fact it starts with a half loop. When the aircraft is halfway a loop it flies in the opposite direction with respect to the situation just before the loop started. It also flies upside-down. The Immelmann turn ends with turning the aircraft around its longitude axis, so it turns from upside-down to a regular flight. The difference with the split-s maneuver is that the aircraft will end up at a higher altitude, but with a lower speed.

3. Software design

3.1. Model

3.1.1. The one-to-one dogfight situation

This project is about an agent that can handle a one-to-one dogfight situation. In a one-to-one dogfight situation there are two airplanes flying in the air. For both, the goal is to remain unharmed and eliminate the other. This requires two skills: defending, make sure the enemy cannot shoot you down, and attacking, try to shoot down the enemy. There are no fixed roles. It is just like a soccer match. One time you are in a defending position and the next minute you are attacking.

The whole dogfight is actually about strategy and airplane control. It is necessary to get in a good shooting position first, before it is possible to shoot down the enemy. With modern combat aircrafts this is not always necessary, due to for example help of guided missiles, but the dogfight situation in the computer game MSCFS is more like a dogfight as it was in the time of World War II.

In this project the actual shooting is left out, so we can fully concentrate on the flight behavior and tactics of the agent. The goal of the agent will no longer be to shoot down the enemy, but to get in the best shooting position, which is right behind the enemy and flying in the same direction.

3.1.2. The reasoning model

The agent must act in the situation described before. The model for this agent is based on the model of a reflex agent. This is not a very complex model, but that does not mean it is a bad model. In a 'real-life' dogfight situation, but also for example in many highly reactive sports, like karate, human actors are also trained to use their reflexes, because fast reaction is often more important then choose the best move. Furthermore such a reflex model is not very difficult to implement. This is mentioned more often in this report, but one of the goals of this project is 'that it really works'. Therefore it is a good choice to choose rather simple models to create a working agent in the first place. Then it will be possible to improve this agent later on.

The reasoning process is continuously. The agent performs one reasoning cycle and when this is finished, it starts another one. It keeps on reasoning until the game is finished. One whole reasoning cycle is shown in Figure 6. This figure shows that the agent first gets all its information from the 'environment'. In this case the 'environment' is the simulator. The next step in the reasoning cycle for the agent is to be aware of the current situation. The agent calculates certain angles and ranges and tries to recognize whether it is in an attack, a defend, or another kind of situation. Thereafter the agent can use this information and try to decide a strategy. This strategy consists only of one maneuver that is best to execute at that moment. After this is decided the maneuver must be executed. This is the last step the agent takes in the 'reasoning cycle'. The execution is performed by setting the right values for the input control parameters in the simulator.

This execution influences of course the 'environment'. The agent will wait a few milliseconds before the changes are really noticeable in the 'environment'. Meanwhile the opponent might also perform some action, but maybe this is not the case. This is uncertain for the agent. Whether one or both of the players performed an action or not, the locations, directions, etc of both airplanes change anyway. So after a few milliseconds the agent needs to start the whole reasoning cycle again.

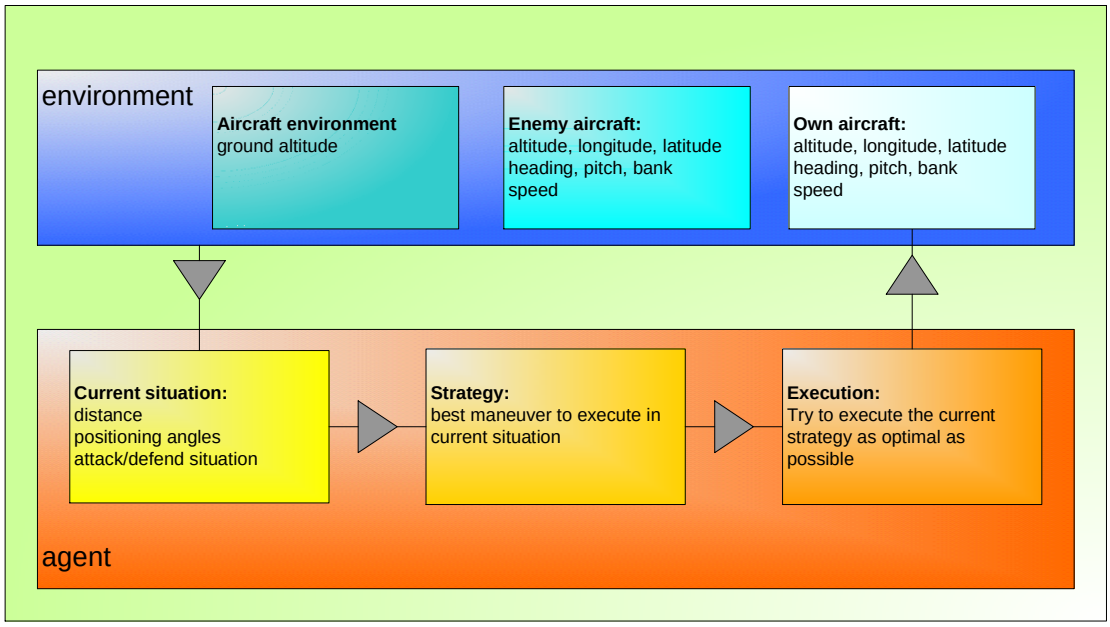

Figure 6: model of the dogfight agent

Environment

The 'environment' in the model stands for the simulator. In this environment two airplanes are flying, the one the agent is controlling and the enemy aircraft. Furthermore there is a surrounding space where both aircraft are flying in. Currently the only interesting parameter in the surroundings is the ground altitude. Both aircraft have the same parameters. The more important ones are shown in Figure 6. All this data can be retrieved by the agent, but the actions of the agent do not change all these parameters. Only the parameters of the own aircraft change because of actions of the agent. But of course the parameters of the enemy aircraft also change constantly due to an external and non-accessible factor: the actions of the enemy.

This environment is rather complex for an agent [Russell, 1995]. First of all the environment is nondeterministic. The agent can decide whatever it wants, but it is not possible to know the actions the enemy is going to take. Even the outcome of the agents own actions regarded to its own aircraft cannot be exactly known. Furthermore the environment is dynamic. Also when the agent does not take an action the environment changes. Both airplanes are always in motion. Another feature of the environment that makes it more complex is that all interesting parameters in the environment are continuous. A less complex feature of this environment is that there are no errors, uncertainty, or missing values in the data. Therefore it is not necessary to use a probabilistic approach in this project. It might, of course, be interesting so see whether a probabilistic approach can be successful for a dogfight agent, but this is beyond the scope of this project.

Agent

The agent consists of the necessary parts for percepting, reasoning and acting. The percepting means for this agent that is retrieves the data out of the 'environment'. The reasoning consists of all steps taken to recognize the situation out of the data retrieved from the environment, decide a strategy based on the available data about the current situation and finally find the right values for the input control parameters of the airplane. Acting means for this agent takes the right values for the input control parameters and sets these in the 'environment'. This will result in the airplane really executing the planned maneuver. The description of the agent will be more detailed in the next sections.

3.2. System architecture

The goal is to create an agent for a one-to-one dogfight. The MSCFS has a multiplayer option that enables this one-to-one dogfight. The MSCFS will run on two different computers with a multiplayer game. These two computers are connected via a TCP/IP network.

On 'computer 1' the agent computer program will run with two different parts: the 'Bot Basics' and the 'Bot AI'. The Bot Basics part will keep track of all the necessary data and it will exchange some of this data with the MSCFS and also with the other player. The Bot AI part does all the intelligent work so that the bot tries to defeat the opponent.

On 'computer 2', a normal human player can also do the work of the Bot AI part. The Bot Basics will still run in order to exchange data with the opponent. This way the agent can fight a human player or another software agent. The other software could be exactly the same, or it could be a different agent. For example someone else could create an agent. In that case the Bot Basics part must be the same, so communication between both players is possible. But the Bot AI part could be very different and by fighting against each other the capabilities of the two different bots could be compared.

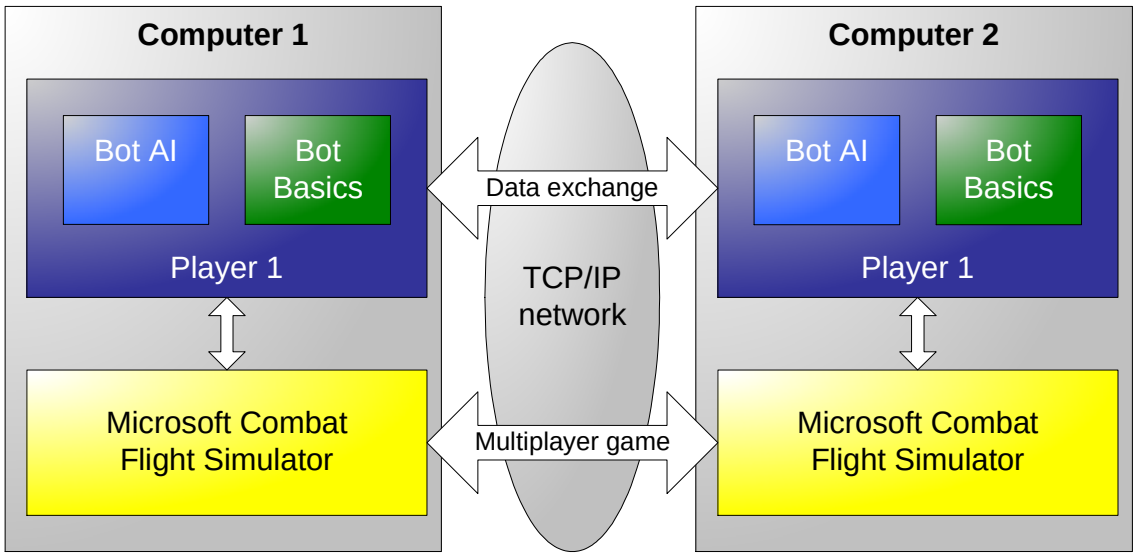

Figure 7: System architecture

The Bot Basics part will always be necessary, also when a human is playing at computer 2. This is because it is not possible to read data about the opponent's airplane out of the MSCFS. Only data about the agent's own airplane, for example the altitude, the speed and the position can be read out of the MSCFS. To let the agent find an appropriate strategy, it must also have these data of the opponent's airplane. The exchange of these data between the two players is one of the tasks of the Bot Basics. Therefore the Bot Basics part will always be active on both computers.

3.3. Agent architecture

In this chapter the architecture of one single agent is described.

3.3.1. Layered architecture

The agent's architecture of is build up in five layers. The abstraction level of the design increases with higher layers. The five layers are grouped into three parts: External, Bot Basics and Bot AI. This layered architecture is based on the Subsumption architecture of Brookes [Brookes, 1989]. The Subsumption architecture is used more often for fully autonomous agents and resembles the behavior of some insects, which can operate successfully with only a basic set of rules and an hierarchical control structure.

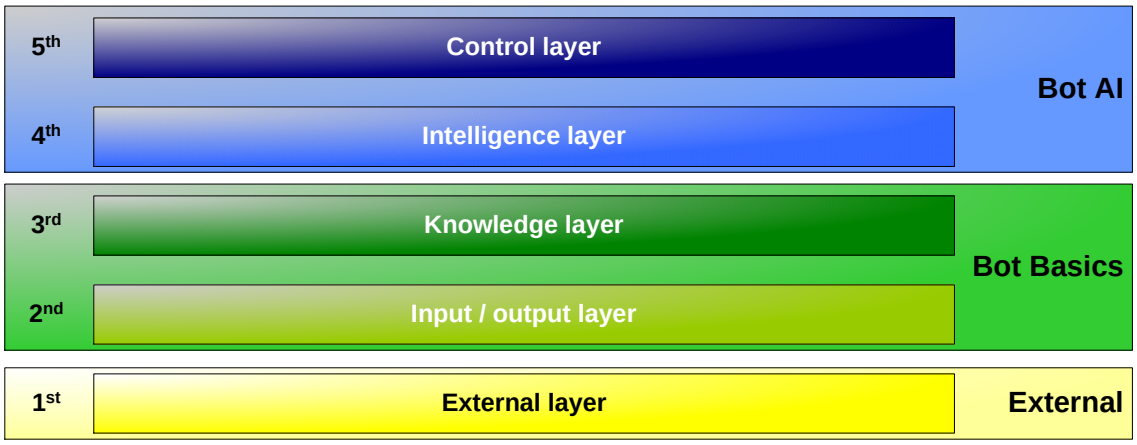

Figure 8: Layered architecture

The External layer is actually not a part of the agent itself, but it is an essential part of the architecture. This layer includes two external programs that are necessary for the agent to function properly: the MSCFS and the opponent's agent software, with or without Bot AI.

In the Bot Basics part there are two layers. The Input/output layer will handle all the IO with the two external programs. The Knowledge layer keeps track of all the dynamic data. There is data about the airplane, the opponent's airplane, and higher-level data produced by other parts of the bot.

The Bot AI part has the two highest layers in the architecture. The Intelligence layer includes all the intelligence of the bot. In this layer all reasoning takes place. There are four different parts in the intelligence layer: Situation Recognizer, Predictor, Decision Maker and Executor. As it is in the real world, there is only one thing higher than intelligence and that is control. The Control layer does not have any intelligence at all, but it controls all other parts of the bot.

Figure 9 shows how all layers are divided into objects. There are ten objects in the architecture. Five of them are active, the other five are passive. The active objects will be implemented in a thread and will remain active as long as the bot program is running. The passive objects will

only act when one of the active objects will ask for it, and they will not be implemented in a separate thread. The two external objects are active because these are separate computer programs. The three internal, active objects are threads. The Controller is a separate thread that keeps control of the agent's work. The Communicator is an active thread, so that it can exchange data with the opponent all the time. Also the Knowledge centre is an active thread. This thread exchanges data with the MSCFS via the FSUIPC object constantly.

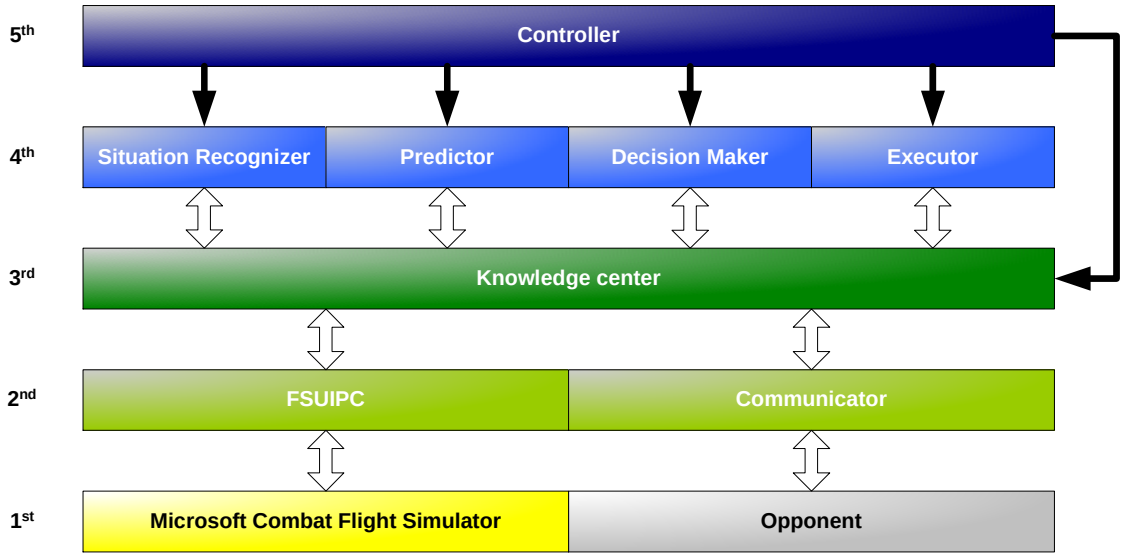

Figure 9: Object architecture, information flow and control flow

3.3.2. Information and control flow

Figure 9 also shows the information and control flows between the different objects. The thick white arrows represent information flow. The thin black arrows represent control flow. Notice that the information always flows in both directions. The control always flows in one direction.

3.4. Objects

3.4.1. Introduction

In this chapter the design of all the objects will be described. The two external objects and the FSUIPC object will not be described, for the simple reason that there will not be a design for these objects in the agent's architecture. These objects just are what they are. All the active objects will be described in an activity diagram. The four objects in the Intelligence layer will be described shortly in how they cooperate with other objects. Their intelligence will be described in more detail in chapter 4.

3.4.2. Communicator

The Communicator will be a separate thread. The activity diagram for the communicator is presented in the figure below. The Communicator sends and receives airplane data with the opponent. The data is retrieved from the Knowledge centre and the opponent data is written into the Knowledge centre. All this is done five times a second, since the agent works best with up-to-date data.

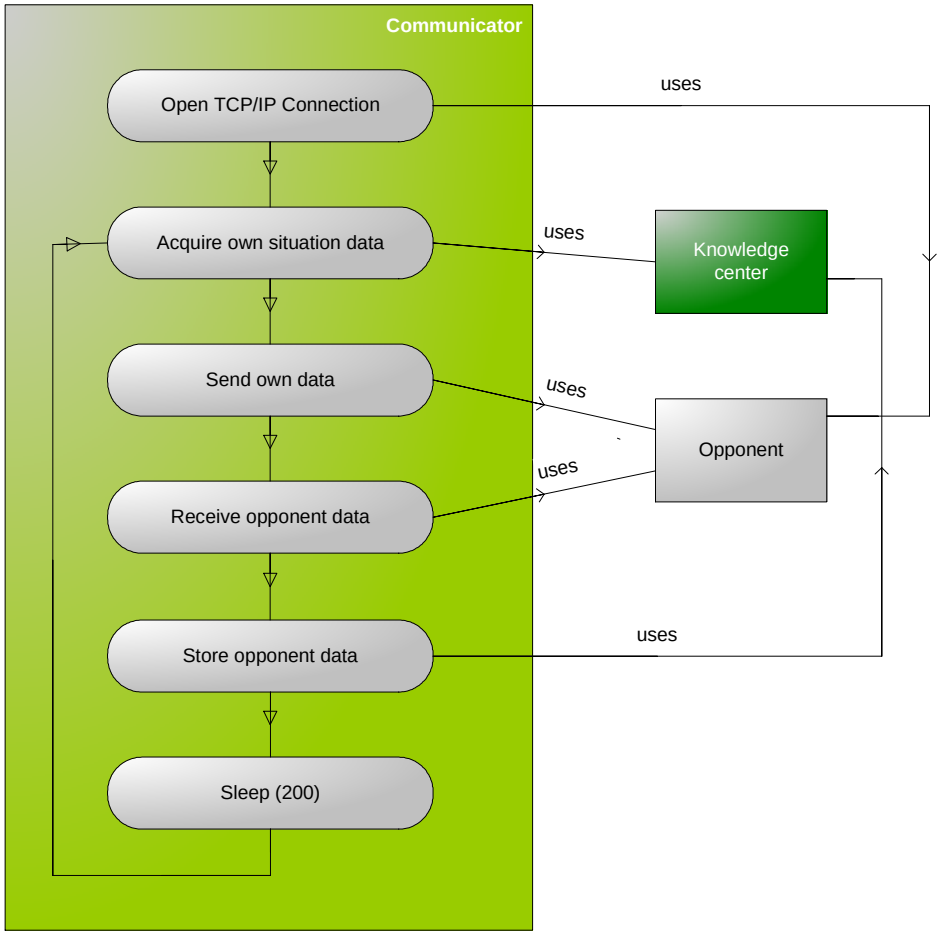

Figure 10: Activity diagram for the Communicator

3.4.3. Knowledge centre

Also the Knowledge centre will be a separate thread. The knowledge centre retrieves data about the airplane from the MSCFS via the FSUIPC object. The data will be converted into easy to use values. This all will be done ten times a second. Apart from that, the Knowledge centre can get a signal from the Controller to write some data into the MSCFS via the FSUIPC.

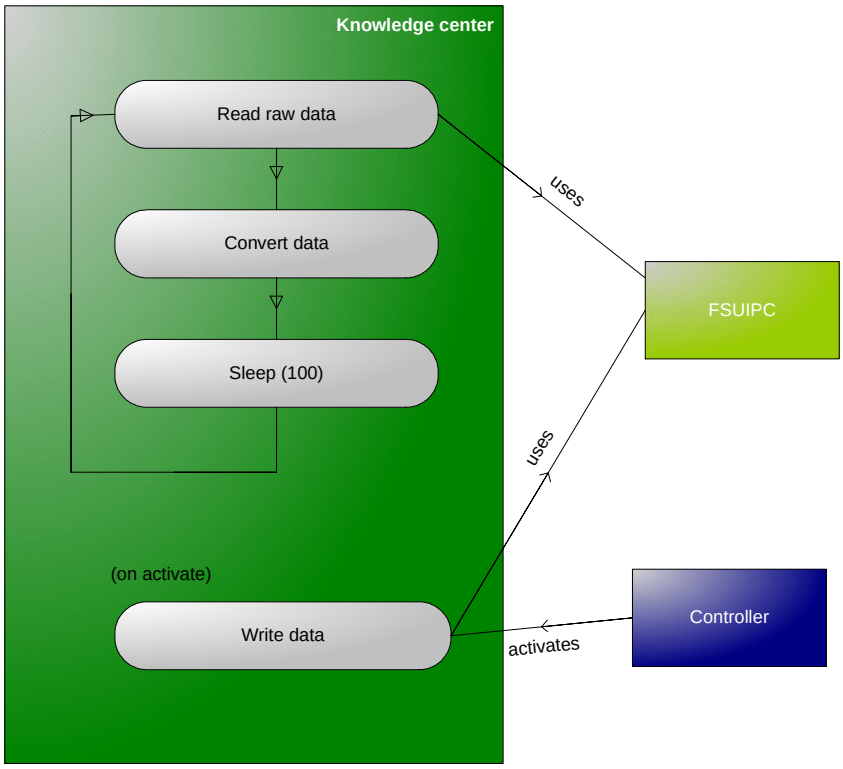

Figure 11: Activity diagram for the Knowledge centre

All the essential data is available in the Knowledge centre. The other objects can retrieve data from and write data into the Knowledge centre. There are three data levels: lower-level, middle-level and higher-level data. Figure 12 shows these three levels and the most important variables in each of the levels.

All lower-level data is available for both airplanes, the bot its own airplane and the opponent's airplane. The middle-level data is computed by the Knowledge center itself from the lower-level data of the two airplanes. Objects in the Intelligence layer create the higher-level data.

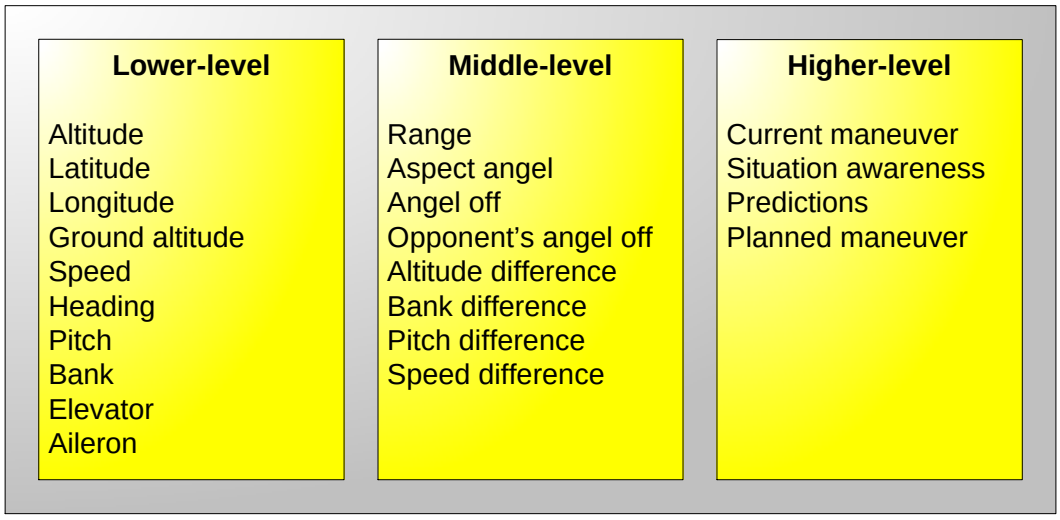

Figure 12: Data in three levels

3.4.4. Situation recognizer

This object is responsible for the situation recognition of the agent. The situation recognition itself may be very complex or remain very simple. From the point of view of software design it is only important that the object exists and that there is one single method to activate situation recognition for the current situation. Furthermore it is of course important that situation recognition does not consume a lot of time, but this holds for all tasks of the agent.

3.4.5. Predictor

The predictor object must predict future values. Based on past and current values this object must predict these values for the next seconds. Again from a software design point of view the complexity of the AI algorithm used in the predictor is not very important as long as it can be activated.

3.4.6. Decision-maker

This object must decide which maneuver is best to execute. Each time step the Decision Maker is activated by the controller. Based on the current situation and eventually the predicted situation the Decision Maker must choose one of the available maneuvers. How this is done is described in section 4.3.

3.4.7. Executor

Executes the currently planned maneuver. Based on the current situation of the airplane and the planned maneuver, the control outputs (elevator and aileron) are computed that are needed to execute the planned maneuver.

3.4.7. Controller

The Controller runs in a separate thread. The Controller itself does almost nothing. It only tells some other objects to do their work. The five other objects will be activated one-by-one. These objects are, in order, the Situation Recognizer, the Predictor, the Decision Maker, the Executor and the Knowledge centre. All these steps will be done once every 300 milliseconds.

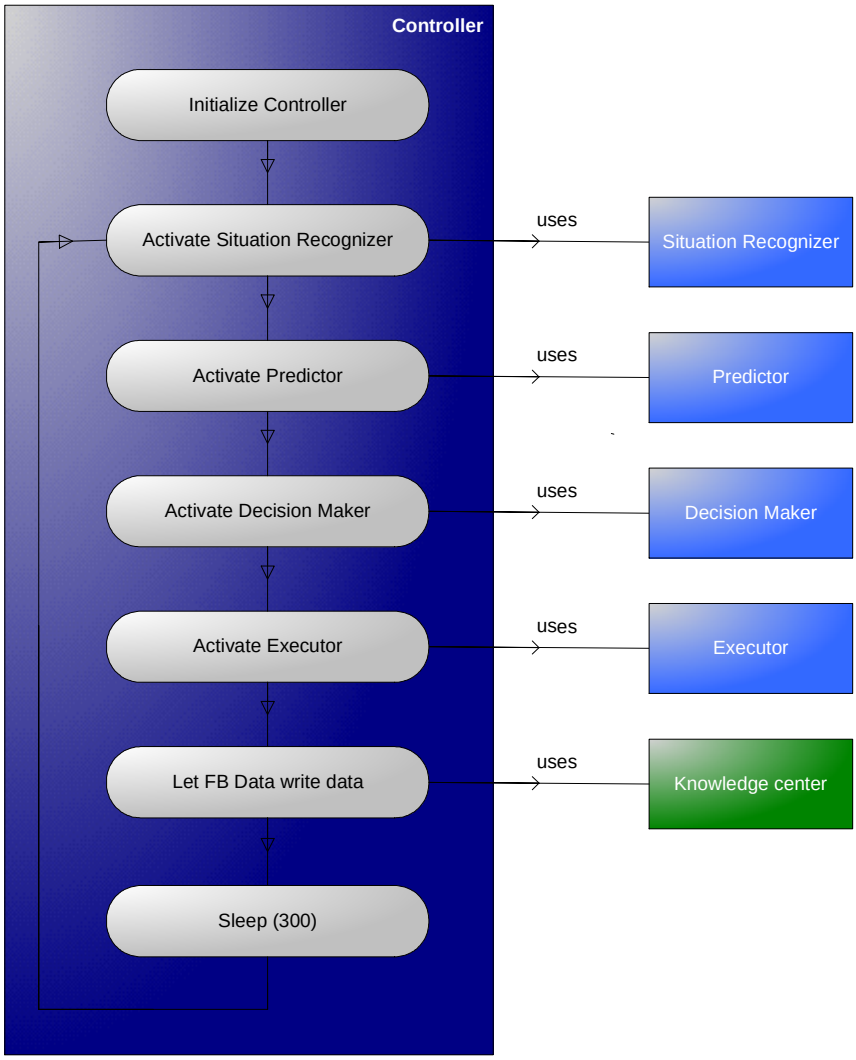

Figure 13: Activity diagram for the Controller

3.5. UML class diagram

The presented design can be put into a UML class diagram. Figure 14 shows a simplified UML class diagram of this project. It is simplified in the way that only properties and methods are included that are relevant for a good overview of the design. Most methods and properties that are only necessary in the implementation phase are left out. This includes class constructors, methods like 'Initialize()' and button variables in the Dialog class. Before private properties and methods is a '-', the publics have a '+'. Relations are presented by arrows. The 'Dialog' class has for example a property 'controller' of the 'Controller' class. This is shown by the light green arrow. Each relation is a one-on-one relation, therefore these numbers are not included in the figure.

The class diagram can be compared to Figure 9. It is clear that there are not many differences. The Predictor is not available in the UML class diagram, because this object is not worked out any further in the design and implementation. Also the two external objects do not appear in the class diagram. This UML class diagram is not very new. All objects and there relations are already described in the previous sections. But this diagram does show that the design is translated to a class diagram without many changes. The implementation, which is described in chapter 5, will follow this diagram closely.

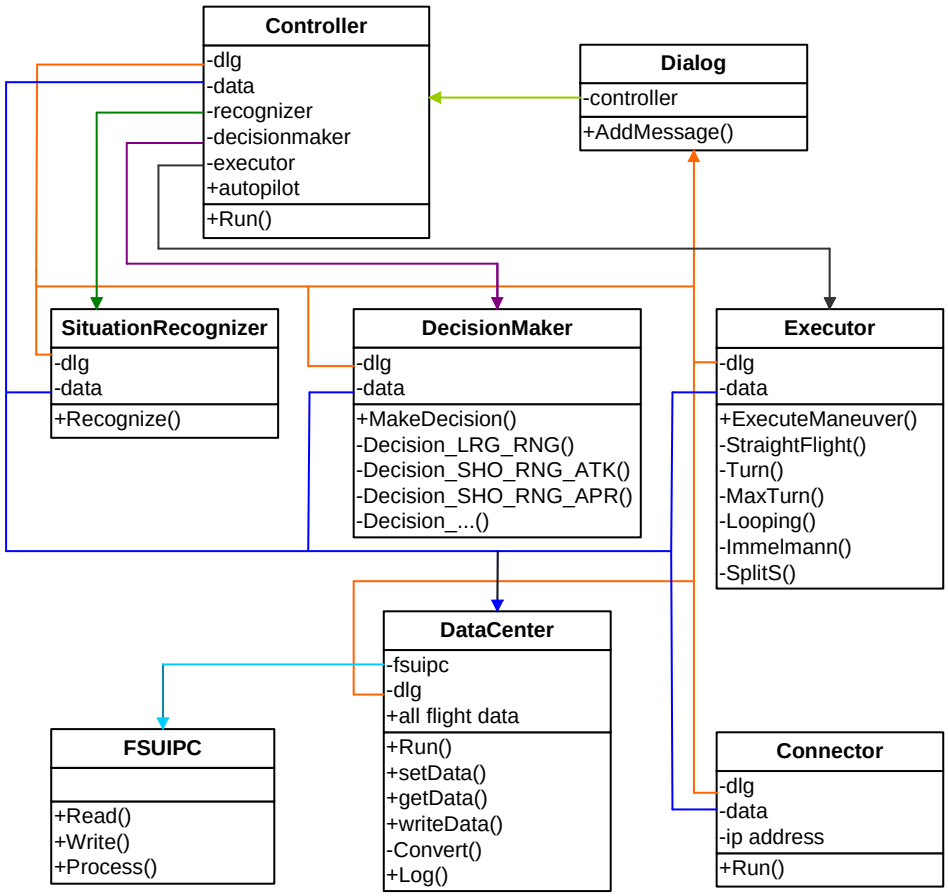

Figure 14: Simplified UML class diagram

4. AI design

4.1. Situation recognition

Situation recognition is actually the first task of an intelligent agent. Based on the situation the agent can decide to do something, or nothing.

4.1.2. Lower- and middle-level data

The situation of the agent can be defined by the values of all available data. The best awareness of the situation can only be achieved when for each data field the current value is known. This includes all three levels of data: lower-, middle- and higher-level data. The lower-level data is perceived directly from the environment. The middle-level data can be computed directly from the lower-level data each time the lower-level data is perceived. This work is already done by another part of the agent: the Knowledge Center.

Is does not require an AI technique to retrieve the lower- and middle-level data. Because the situation recognition is done in the intelligence layer, the responsibility for the lower- and middle-level data is for the Knowledge Center. The design and implementation of the situation recognition can be focused only on necessary AI techniques.

4.1.3. Maneuver recognition

To be fully aware of the situation it is necessary to be aware of the current maneuver and of the opponents maneuver. The maneuver that the bot makes itself is of course always known, but for training purposing it is necessary to be able to recognize the maneuver the aircraft is in. The maneuver that the opponent is making is of course always a subject for maneuver recognition. The recognition of these maneuvers is a part of the whole situation recognition. The recognition of the current maneuver is of course not so difficult when the agent decides itself which maneuver to execute. But recognizing the current maneuver of the opponent is of course more difficult. In [Mouthaan, 2003] situation recognition is done using Bayesian belief networks. This work however deals with situations like: 'taking off', 'normal flight' and 'dogfight', and not with specific maneuvers. In [Capkova, 2002] neural networks are used to recognize the maneuvers: 'going up', 'regular flight', 'turning right', 'turning left', 'going down', 'parked' and 'taxiing'. The maneuver recognition in this project must be able to recognize all maneuvers available:

- Straight flight
- Straight flight upwards
- Straight flight downwards
- Going down
- Turn left
- Turn right
- Extreme turn left
- Extreme turn right
- Looping
- Split-S
- Immelmann turn

The maneuvers are characterized by a set of features, such as the pitch, the bank, and the speed of the airplane. The distances between the features are rather big for most maneuvers. Therefore it is not really necessary to use more complex classifiers like a neural network or a support vector machine. A simple decision tree, as it is shown in Figure 15, will be sufficient.

The only problem arises with the difference between a loop and a split-s maneuver and between a loop and an Immelmann turn. Big parts are exactly the same in these maneuvers. Therefore the loop maneuver is simply left out of the decision tree. This seems like a mayor flaw of the agent, but it isn't. The maneuver recognition is not meant to be 100% perfect. It is only used, so the decision-making system can use the information about the opponent's current maneuver. In fact this goes a little bit into the direction of prediction. Because when the current maneuver the opponent is executing is known, the decision-making system can anticipate not only on the current position of the opponent, but also on the position it will be in, in the next time step(s). But for this reason it is not very important whether the opponent is execution a split-s maneuver or the first part of a looping. In both cases the position of the enemy in the next time step will be the same.

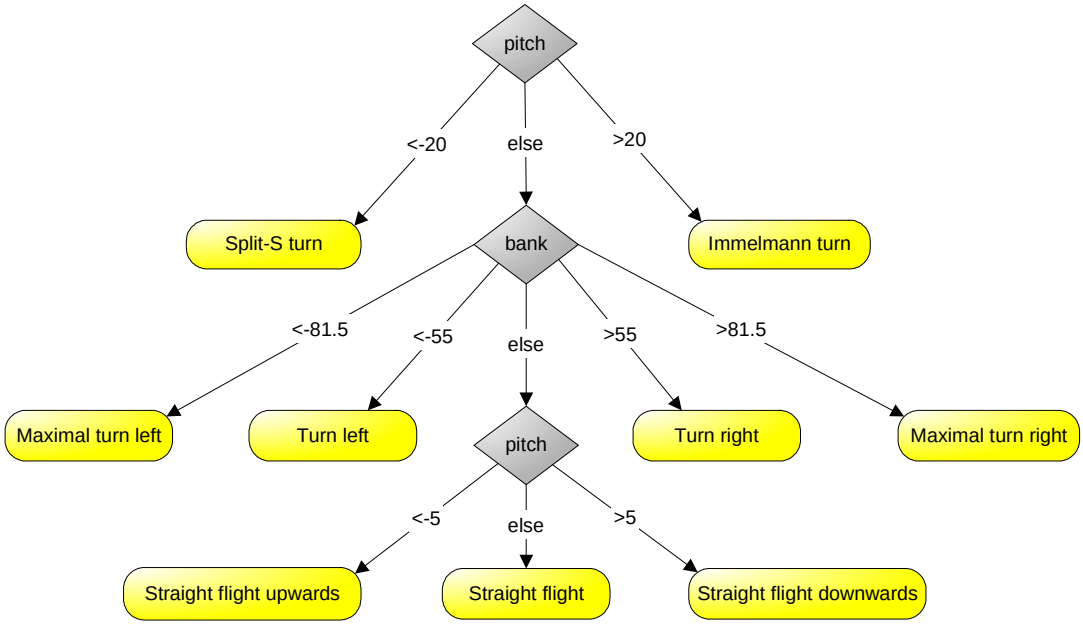

Figure 15: Decision-tree for maneuver recognition

4.1.4. Position recognition

Besides recognizing the maneuver of an individual airplane, the situation recognition includes recognizing the relation between the positions of the two airplanes. Figure 16 shows all the possible position relations. Again a decision tree is a natural choice to model the recognition of these position relations.

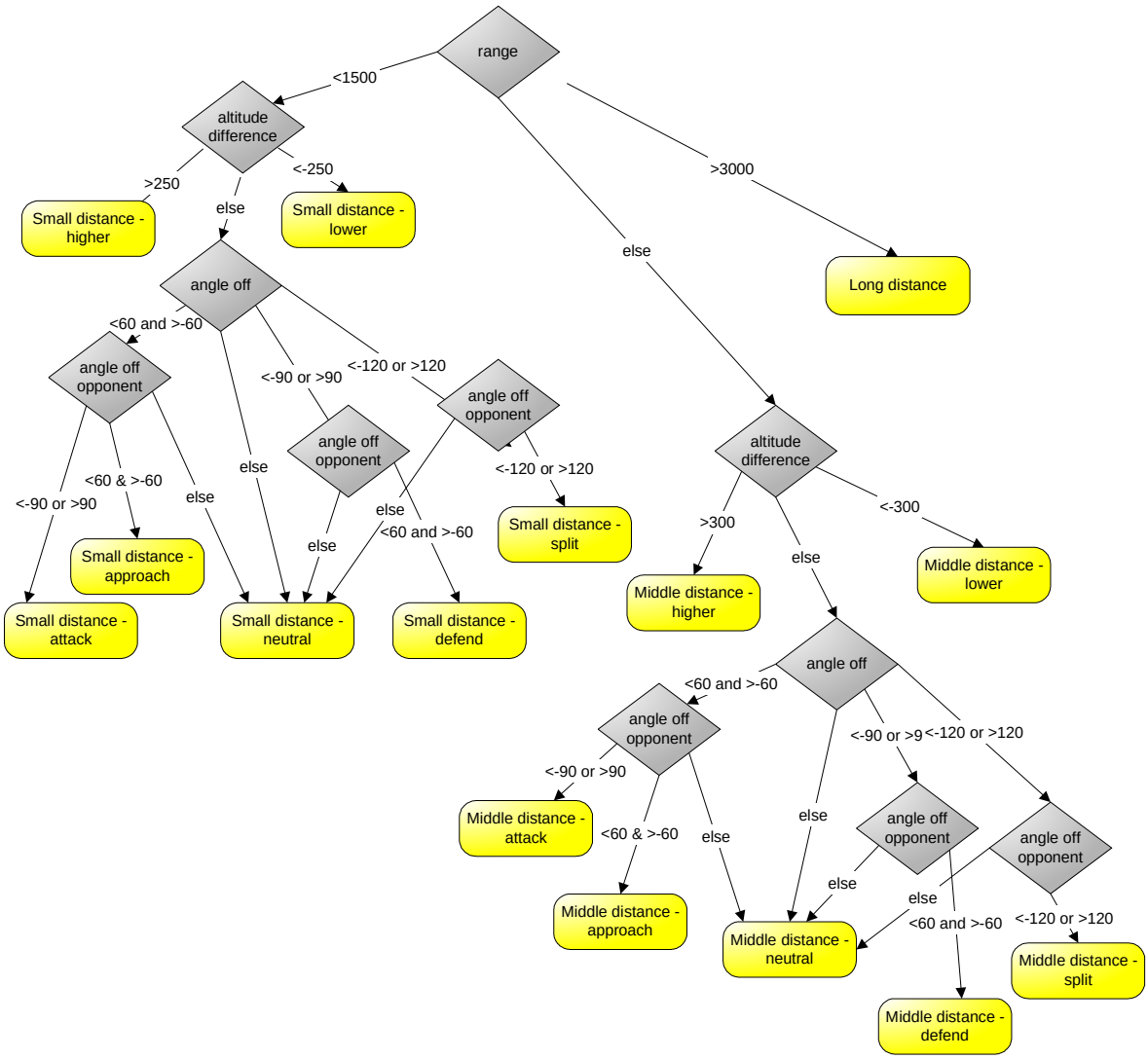

Figure 16: Decision-tree for position recognition

4.2. Prediction

Prediction is not included in the AI design and also not in the implementation. It is not essential for a basic working of the bot, however it can improve the results of the bot. It may be useful to predict the location of the two airplanes for future time steps or to compute the probabilities that the opponent will start some other maneuver.

4.3. Decision-making

4.3.1. Decision-making mechanism

Decision-making is maybe the most intelligent process inside of the bot. Based on the current situation the bot needs to decide the next maneuver to execute. The decision-making will be done by a set of logical rules. The big advantage of rule-based decision-making is that humans can understand it quit easily. This is especially the case compared to numerical or connectionist techniques. Rules can be created by humans, but rules can also be generated by computer algorithms en then still be understood by humans. All this makes a rule-based system very suitable for a first prototype.

The rule-based decision system can be viewed in the Figure 17. The current situation is represented by a list of parameters en their current values. Based on the values of these parameters, the rule-based system decides which maneuver will be executed in the current situation. This mechanism is repeated every time step.

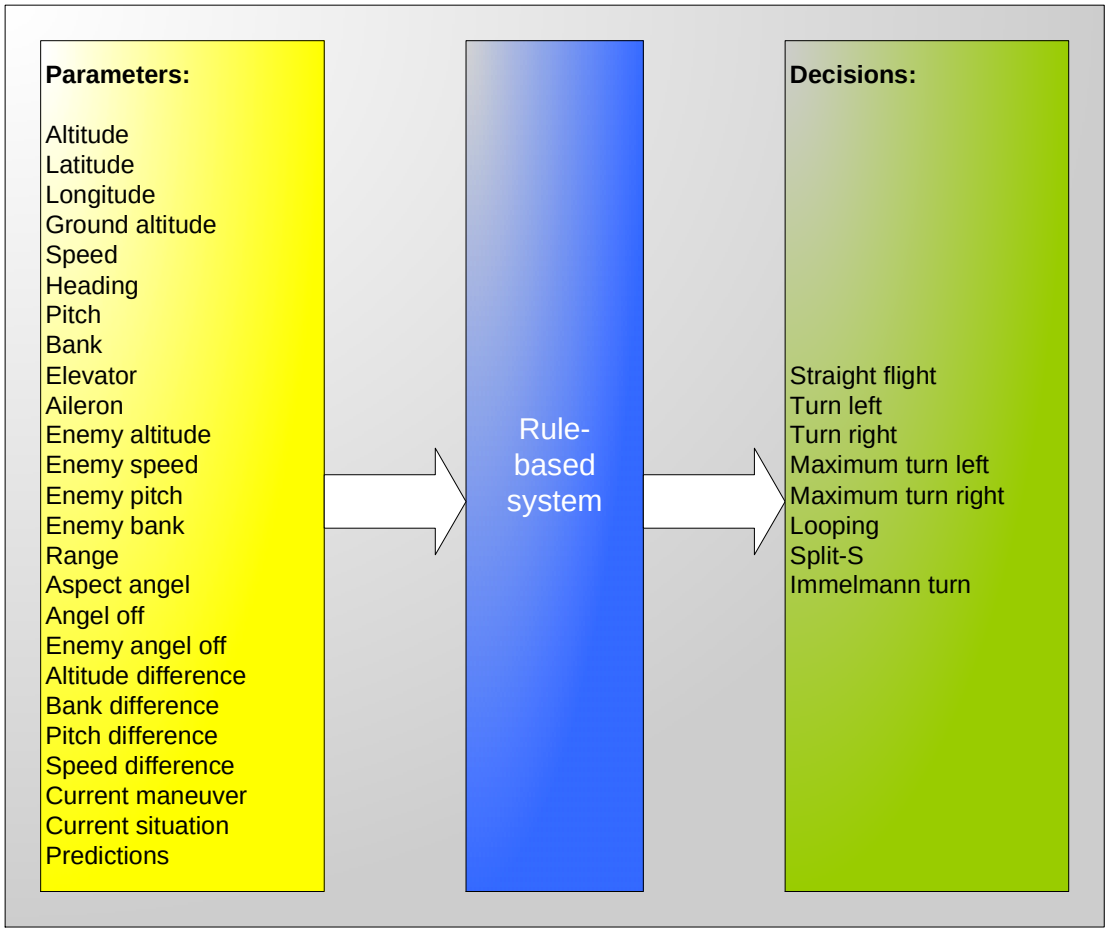

Figure 17: Rule-based decision mechanism

4.3.2. Acquisition of rules

The most difficult task in creating a rule-based system is the acquisition of the rules. This knowledge can be acquired in different ways:

- Entered by domain experts
- Learned by observing experts
- Learned by experience

In case of entering rules by a domain expert the expert must be capable of extracting well-defined rules out of his domain knowledge. For example:

> "IF angle off > 0 THEN turn right"

This method requires extensive domain knowledge. In [Andriambololona, 2003] a decision tree is presented that contains some basic decisions for a dogfight situation. The biggest advantage of this method is that it is not difficult to implement.

Another method is to learn decision rules by observing experts. In this case it is necessary to log data about the current situation of an airplane flown by an expert and of the maneuver the expert decides to make in that situation. In this method it is necessary to use software like Matlab to analyze the logged data and to extract decision rules out of it. When these rules are extracted, they are, just as in the previous method, not difficult to implement. Learning decision rules by observing experts is for example used in [Lent, 1998].

The third method is a bit more complex. In this case the decision-making system should have enough experience to make the right decision. From each experience the system should learn whether it made the right or the wrong decision. When a human expert tells the system whether a decision was right or wrong, this is called supervised learning. Unsupervised learning means the system knows by itself whether a decision was right or wrong. An AI technique to make an agent learn unsupervised is genetic selection [Spronk, 2003]. Another possibility is to use a function that evaluates the situation each time step and can learn that way whether a previous decision was good or bad. This function is called a fitness function. A fitness function in a dogfight agent could be based on the position recognition. A 'small distance attack' situation for example is obviously better than a 'small distance neutral' situation, which is on its turn better than a 'short distance defend' situation.

Learning by experience in a dogfight agent however is a complete research subject by itself. For a successful prototype of a dogfight agent it is necessary that each part of it works at least at a basic level. Learning by experience is complex en it is difficult to get at least a reasonable result. It is therefore not a good choice for the first prototype, because it is not sure that it is possible to get it work within the time limits of this project.

4.3.3. State-based decisions

To make the decision mechanism and specifically the acquisition of the decision rules easier, the decision mechanism will be divided into different states. Each state will correspond to one of the positions described in section 4.1.4. For each state there will be a different set of decision rules.

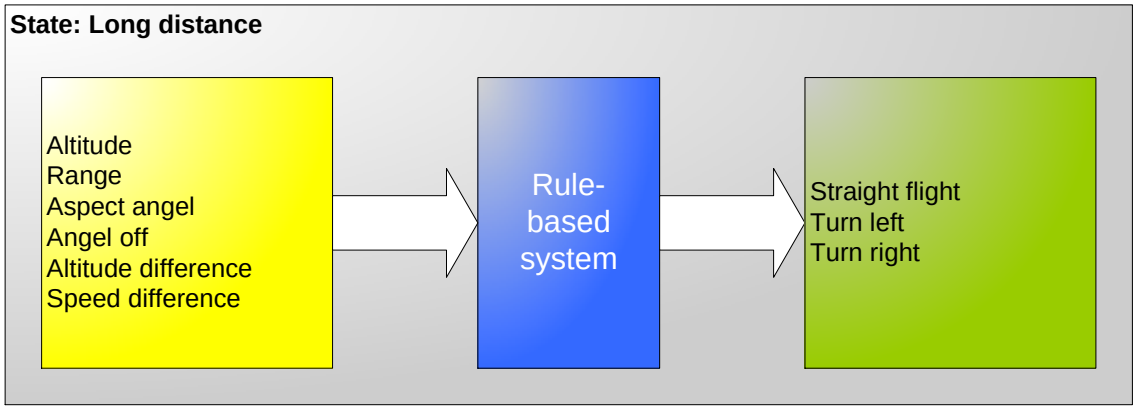

Figure 18: Rule-based system for long distance state

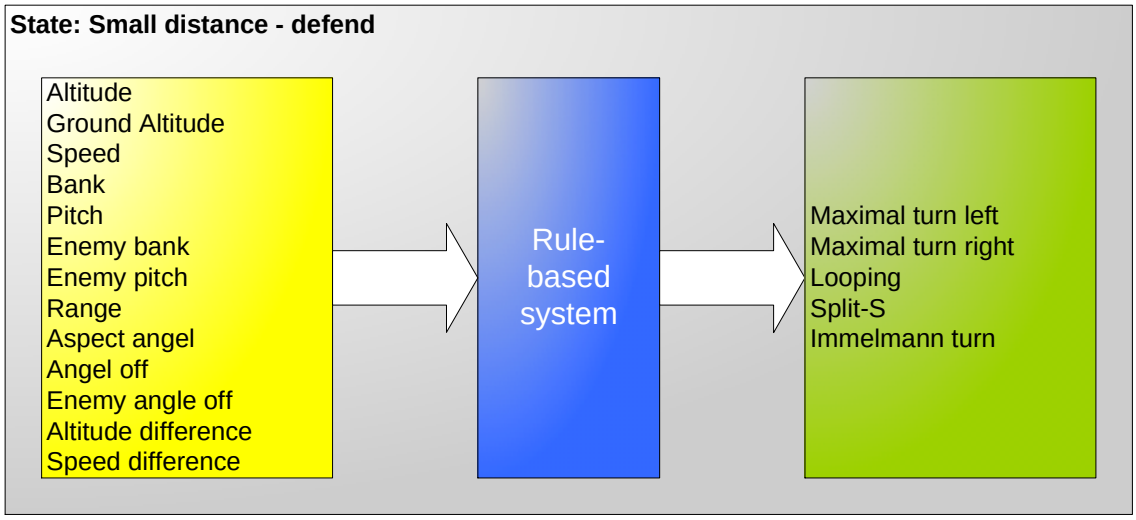

Figure 19: Rule-based system for small distance - defend state

For example a rule in the 'small distance attack' situation might be:

IF angle off < -5 and
  aspect angle < 0 and
  bank of the enemy < 30
THEN make a maximal turn left

4.4. Execution

4.4.1. Execution of a maneuver

The decision-making system decides which maneuver will be executed. After this the selected maneuver must be executed by an execution system. The agent cannot order the aircraft directly to make, for example, a turn left. The agent can only use the elevator and aileron parameters to perform the desired maneuver (see section 2.3.2). The agent must be able to execute all maneuvers described in section 2.3.3. Most maneuvers are different from each other, not only in their result, but also in the approach of the pilot. A looping, for example, requires a good starting situation. Thereafter, the only action needed is to pull the elevator and wait until the maneuver is completed. A turn, on the contrary, needs constantly small adjustments to control the bank of the aircraft properly. For these reasons the execution of each maneuver will be separately designed and implemented.

4.4.2. Learning maneuvers by observation

To get the agent flying the airplane, with the desired maneuver, using only the elevator and aileron is not a straightforward task. It is possible to change the elevator or aileron a little bit each time step until the aircraft flies in the desired position. This method is used by [Tamerius, 2003]. With this method it will be difficult to find suitable values for the changes in elevator and aileron. It will also be very difficult to get a very quick reacting agent. Another option is to use a neural controller as described in [Liang, 2004]. With a neural controller it is not necessary to find suitable values for changes anymore. But the results of [Liang, 2004] show that it is still not very easy to create a highly reactive agent that is able to react fast enough to survive in a dogfight situation. Therefore in this project the execution of maneuvers is created by learning from experts, by observing how an expert executes maneuvers. This will result in a direct mapping between the control parameters, elevator and aileron, on one side and both the current maneuver and desired maneuver on the other side.

For this purpose a log file is created from a flight by a human expert. The logfile contains the values of relevant flight parameters for a large number of consecutive time steps. The expert executed maneuvers like nice steep turns, straight flights, but also complete random flights.

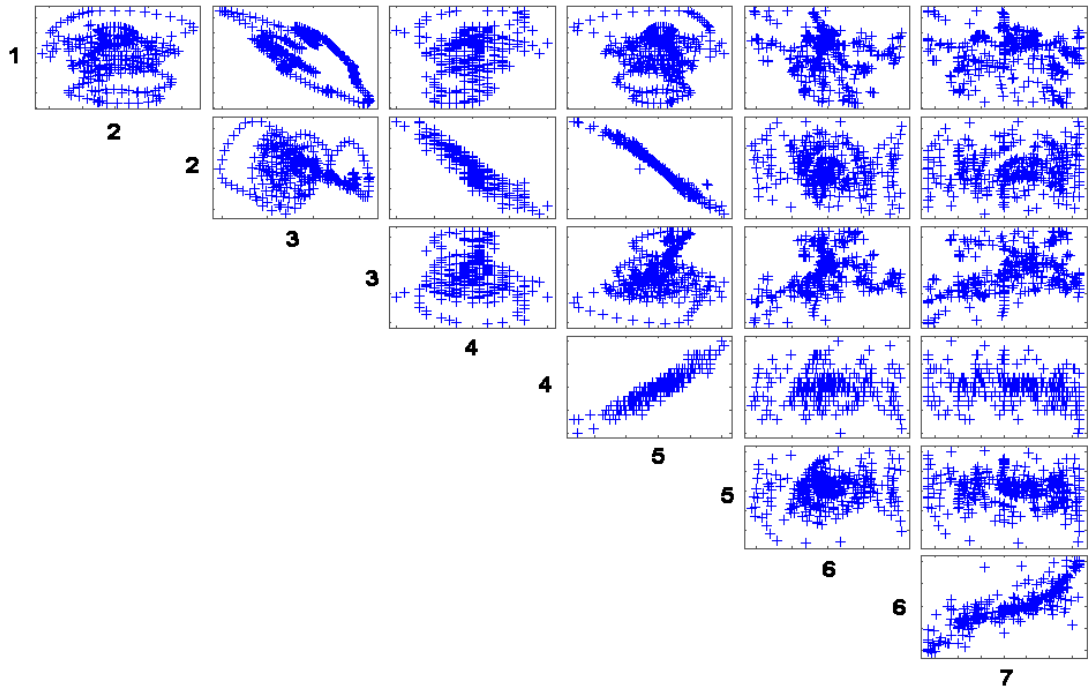

Figure 20: Relations between altitude (1), altitude change (2), speed (3), speed change (4), pitch (5), pitch change (6) and elevator (7) in a random flight.

From Figure 20 it is clear that some parameters are related to each other, even in a random flight. It is possible to find relations between the input parameter elevator and flight parameters that are necessary to fly specific maneuvers. For example from the figure above it is clear there is a relation between altitude change (feature 2) and the pitch (feature 5). There is also a relation between pitch change (feature 6) and elevator (feature 7). Using these two relations it is possible to get the airplane to a specific altitude, fast and smooth, with only updating the elevator value constantly according to a value found by the mapping.

4.4.3. Mappings of input controls on desired output parameters

The agent should be able to know the value of the input control that will result in a desired value of an output flight parameter. This can of course be done by training a neural network on the logged data. For the prototype we will use a less complex technique. With polynomial regression it is possible to find a polynomial function that expresses the relation between parameters. This method is not difficult to implement and will result in a fast and determined reaction of the prototype agent.

For example during a flight the agent could desire an altitude change. To change the altitude, a certain pitch is necessary. The relation between altitude change and pitch is also visible in Figure 20 (features 2 and 5). A positive altitude change needs a negative pitch value (the airplane will go up with a negative pitch value). But this is not the end. It is not possible to set the pitch value, but a desired pitch value can be achieved by setting other parameters. Figure 20 also shows that there is a relation between pitch change and elevator. This relation can be useful, because the elevator finally is a parameter that can be controlled directly by the agent.

To find suitable relations and functions it is possible use regression. Table 1 shows de mean squared error (MSE) and de R2 statistic for functions that map a variable onto the pitch change parameter. The errors are found using a 10-fold cross validation. The functions are polynomials with dimensions 1 till 4 that are found using polynomial regression.

Table 1: MSE en R2 statistic for mapping onto pitch change with n polynomial regression

	altitude		altitude change		speed		speed change		pitch	
	MSE	R2	MSE	R2	MSE	R2	MSE	R2	MSE	R2
1	2.31E+04	84.12	2.33E+04	3858.30	2.34E+04	1576.70	2.14E+04	11.85	2.34E+04	220.26
3	2.28E+04	34.78	2.33E+04	471.60	2.34E+04	630.02	2.15E+04	11.79	2.34E+04	204.37
3	2.08E+04	8.54	2.34E+04	381.75	2.31E+04	50.20	2.11E+04	9.76	2.35E+04	186.07
4	2.07E+04	8.23	2.34E+04	283.61	2.31E+04	45.01	2.11E+04	9.78	2.35E+04	165.28

	pitch change		bank		bank change		elevator		aileron	
	MSE	R2	MSE	R2	MSE	R2	MSE	R2	MSE	R2
1	3.18E-26	1.00	2.34E+04	565.68	2.30E+04	52.43	2.05E+04	7.80	2.30E+04	44.63
3	8.41E-26	1.00	1.70E+04	3.68	2.30E+04	46.80	2.03E+04	7.40	2.30E+04	43.89
3	3.81E-26	1.00	1.68E+04	3.53	2.30E+04	45.64	2.02E+04	6.87	2.30E+04	41.59
4	1.24E-25	1.00	1.68E+04	3.52	2.30E+04	45.18	2.03E+04	6.83	2.30E+04	40.91

The MSE for the function that maps the pitch change onto the pitch change itself is of course almost 0 (it is not exactly 0, due to rounding errors in the learning algorithm), and the R2 is 1, which is the best value for the R2 statistic. A lower MSE means a larger relation between the two parameters. A surprise is the strong relation between the bank and the pitch change. The most important conclusion of this table is that there is a reasonable 2 or more dimensional function between elevator and pitch change in the random flight logged data. Always when this method results in a low MSE and a R2 near to 1 there is a relation between the two parameters that is possibly useful.

A little bit more advanced possibility is to use 2 flight parameters as input for a function to find a good value for another parameter. Table 2 shows that the errors become lower when we use the elevator in combination with the current pitch or in combination with the current bank to find create a function for the pitch change.

Table 2: MSE and R2 for polynomial regression with two variables

	pitch & elevator		bank & elevator	
	MSE	R2	MSE	R2
1	6.02E+07	6.91	6.15E+07	7.59
3	5.62E+07	4.72	4.63E+07	2.95

The results look better, specifically when the bank is used in combination with the pitch change, but unfortunately the use of functions with two input parameters resulted sometimes in strange behavior of the airplane. But nevertheless it seems there is a big influence from the current pitch and even more from the current bank on the pitch change.

Another way to model the influence of the bank on the pitch change and the required elevator control is to find different functions for situations with a different bank. Therefore a separation is made between situations when the bank is not very high or low, but closer to 0 (this occurs when the airplane is flying relatively straight) and when the bank is very high or low (this occurs when the airplane is in a turn). For this purpose we could use the data that is logged only during specific maneuvers. For example, when the airplane is already in a turn and thus has a bank that is far from 0, the following function can be used:

$$needed\_aileron = f(x = desired\_bank\_change) = 1.115x^3 + 6.236x^2 - 1153x - 2745$$

When the airplane is currently flying straight the next function is better:

$$needed\_aileron = f(x = desired\_bank\_change) = 2.000x^3 - 3.300x^2 - 1466x - 341$$

4.4.4. Different functions for different maneuvers

In section 4.4.1. it is described that it is better to create different maneuvers separately. Therefore also different functions are created for different maneuvers. For example: There will be a separate part of the implementation that can execute a 'turn left' maneuver. When a 'turn left' is desired in the next time step, the first interesting thing is whether the airplane is already in a 'turn left' situation. When this is not the case, the agent must try to get the airplane into a 'turn' position by changing the bank of the airplane. A function that was found with the regression described above is used to find the appropriate aileron value. Furthermore the agent raises the elevator value already a little bit. Normally the agent will decide to continue the 'turn left' maneuver some consecutive time steps. When the agent already is in a 'turn left' position another function is used that finds a good value for the aileron just to keep the airplane in the turn. Notice that appropriate action is always needed to adjust the aileron value and keep the airplane in a nice turn also when it already is in a turn position for a long number of consecutive time steps. In case the airplane already is in a turn left position, the elevator can then be set to maximum to make the turn very steep.

In most cases the strategy to execute a maneuver will be to get the airplane in the appropriate position first and thereafter use other functions found by regression, specifically to optimize the maneuver in case the airplane already is in the desired position for that maneuver.

4.5. Example

In the previous sections the reasoning process and all aspects of it are described. An example might give a clearer view on how this reasoning process actually works. This example is meant to give a clear overview of the reasoning process. It is not meant to explain specific decisions or maneuvers. The example will handle one whole 'reasoning cycle' as it is described in section 3.1.2. Suppose both airplanes are in a situation as it is shown in Figure 21. In this situation the agent is close behind its opponent.

Figure 21: Top view of the starting situation in the example

Step 1: data retrieval

Up-to-date data is very important in a highly dynamic environment. The Knowledge center retrieves the last data from the MSCFS continuously in a separate thread especially for this purpose. Another thread, the Communicator, communicates with the opponent, so also that part of the data is always up-to-date. Keeping the data up-to-date is a continuous process and this process is in fact independent of the rest of the agent. Whether the agent is reasoning or not, data retrieval will always take place. So actually this data retrieval is not really step 1 in the reasoning process, because data retrieval happens all the time, and even more often than the reasoning itself. Instead of data retrieval, step 1 could be: 'having up-to-date data available'.

Step 2: situation recognition

The first intelligent task for the agent is situation recognition. The Controller tells the Situation Recognizer to start. All lower-level data (see Figure 12) is already available. Also the middle-level data is already computed out of the lower-level data. The Situation Recognizer must recognize the maneuver the opponent is making and the position of the agent related to the opponent. Suppose the decision-tree for maneuver recognition finds out the opponent is making a 'turn right'. The decision-tree for position recognition will recognize a 'small distance attack' situation. This position is quite obvious, because the agent is close behind its opponent. Both, the opponent's maneuver and the agent's position, are stored in the Knowledge center.

Step 3: decision making

When the situation recognition is completed, the Controller will start the Decision Maker. Because the agent is in a 'small distance attack' situation, the corresponding decision system will be used. In the case of a 'small distance attack' situation some important parameters are:

- The 'angle off' (about -12 in this example)
- The 'aspect angle' (about +15)
- The opponent maneuver (turn right)

In this case the following rule will apply:

IF angle off < -5 AND
aspect angle > 0 AND
opponent maneuver == turn right
THEN start a turn right

The reason for this decision is as follows. Although the opponent is yet on the left side in front of the agent, the opponent is at this moment making a turn right. The opponent will be soon right before and thereafter on the right side of the agent's airplane when the agent would fly straight ahead. A turn left would lead to the same, but then a bit faster. When the agent starts a normal turn right, not a steep one, the agent will end up straight behind the opponent, which is the best position. Of course, things can end up different due to unexpected events, like an unexpected change of maneuver from the opponent, but in that case the agent will get another chance to recognize this a few milliseconds later. In most cases the decision in this example will be a good decision.

Step 4: maneuver execution

It is again the Controller that starts the Executor. The Executor must find the right values for the input control parameters, so the airplane in the game will execute the maneuver that is chosen by the Decision Maker. In this example the chosen maneuver is a turn right. The Executor has a specific function for each maneuver, so in the example the function for the 'normal' turn right is used. In a turn right maneuver the desired bank of the airplane is 78. With this bank the airplane makes a nice turn without losing height. In the function the difference between the current and the desired bank is used to calculate the best aileron. Suppose the current bank of the airplane of the agent is 0, because the airplane flies straight, the difference between the current and the desired bank is 78, which is a big difference. The aileron that is returned by the function will also be very large, but the maximum aileron to use in MSCFS is 16000. Therefore this maximum value is used. The other input control parameter, elevator, is not calculated in the turn right maneuver. A human pilot will normally use the maximum elevator in a turn. This makes the turn nice and steep. But when the airplane is not yet in the turn position, an average elevator is used. An average value for the elevator in MSCFS is 12000. Altogether the Executor found the values 16000 and 12000 for the aileron and the elevator in this case.

Step 5: sending data

The final step is sending the data to the MSCFS. The Controller signals the Knowledge center to send the data after the Executor has finished. The Knowledge center uses the FSUIPC object to send the values 16000 for the aileron and 12000 for the elevator to the MSCFS.

Further on

After the data is send to the MSCFS the agent waits a few milliseconds. In this time some small changes will occur in the situations of both airplanes. This is because of the speed both airplanes already have and also because of the actions the agent and maybe its opponent took. After this the agent starts again from step 1: retrieving data. In most cases the situation will not be changed so much within the few milliseconds. In that case the agent will still be in a 'small distance attack' situation and it will again decide to make a turn right. But of course it could be that the situation changed a bit more. It could be for example that the agent is still in a 'small distance attack' situation, but meanwhile the opponent changed its maneuver from a turn right into a turn left. This might cause the agent to decide to start another maneuver. It will never be very clear for the agent how the situation will be in the future, but it gets to recognize the situation and decide for a maneuver each few milliseconds again and again.

5. Implementation

5.1. Used software

5.1.1. Microsoft Combat Flight Simulator

Microsoft Combat Flight Simulator (MSCFS) is a computer game for the Microsoft Windows platform. The game is a flight simulator in the line of all the Microsoft flight simulators. These simulators are rather realistic in graphics, simulated situations and, also very important, flight characteristics. This makes these flight simulators very well suited to use for training of pilots and also for scientific research, as it is done in the ICE project.

The MSCFS differs from the normal flight simulator in that it is a sort of First Person Shooter game. The first person is the pilot of an airplane and his mission is to shoot down the enemy airplanes, with or without the help of friendly airplanes. It is also possible to play in a multiplayer game. This option is used in this project to let a computer agent fight against a human person or against another computer agent.

5.1.2. FSUIPC

The Flight Simulator Universal Inter-Process Communication (FSUIPC) module is a DLL-file that can be loaded together with a Microsoft Flight Simulator. When loaded it makes it possible for another software program to communicate with the Flight Simulator about a large number of variables. It is possible for example to retrieve the current values of the weather, the location of the airplane or of flight-related variables like the current speed, direction, or bank of the airplane. This makes the FSUIPC module ideal for logging of flights.

It is also possible to set certain variables. Not all variables can be set. Generally something can only be set, when it would be possible in a normal airplane. For example is it not possible to change the altitude of an airplane, but it is possible to set the lights turned on or to set the elevator control. A lot of third party software is written for Microsoft Flight Simulators that uses the FSUIPC module. An example is ALERT !! [Alert] that can randomly generate failures and unexpected events to make a normal flight a bit more 'interesting'. In 'The Real Cockpit' [The Real Cockpit] the FSUIPC module is used to create a real cockpit for Microsoft Flight Simulators instead of an interface with a keyboard and a computer screen. In this project the FSUIPC module is used to let a computer agent control and airplane.

5.1.3. Microsoft Visual C++

Microsoft Visual C++ is a development tool to create computer programs in C++. It provides all the features a modern development tool should provide. For the development of the agent in this project only basic features where needed. Microsoft Visual C++ gives nice overviews of all classes, methods and variables, but the existing computer code can also be used in a normal text editor together with a compiler.

5.1.4. Matlab

Matlab is a software program for technical computing. It is very suitable for programming complex computation, as they are for example necessary in data analysis. Furthermore it has extensive probabilities for data visualization. Matlab is based on a large number of commands, with which scripts or functions can be created. So actually Matlab is a programming language with a high abstraction level.

The latest versions of Matlab include large libraries for all sorts of application domains. In this project Matlab is mainly used for data visualization and data analysis in research needed to create the AI components. For example data visualization is used to get a proper sight on the flight dynamics of the used airplane. Data analysis is used to create the proper functions that form the flight control.

5.2. Implementation limitations

The limitations of the bot will be discussed here to give a fair view on the capabilities of the implementation. Some limitations are already mentioned before, but they are mentioned here again to give a good overview.

5.2.1. Only one type of airplane possible

The implementation works only with one type of airplane. The flight behavior is based on data from earlier flights with that type of airplane, so it only works well when exactly the same type of airplane is used.

5.2.2. No throttle settings possible

In this project it was not possible to adjust the throttle. The throttle is one of the three most important control parameters together with the elevator and the aileron. It can be compared to the gas pedal in a car. To get around this problem the throttle remains always in the same state. Fortunately the throttle is automatically set to full when starting a multiplayer game. This is a good state to stay in during a dogfight.

5.2.3. No flight possible after hit

When an airplane is hit in the MSCFS game, its flight behavior changes immediately. This is of course very realistic, but it means that the bot cannot control the airplane anymore after one single bullet hit. For this reason shooting is not included in this project, so the airplanes will stay in the air much longer.

5.2.4. No prediction implemented

Prediction is included in the object design in section 4.3.1. However, due to time limitations prediction is not further considered in the implementation of this project.

5.3. Source code

In this section the implementation of some crucial parts of the agent is described. Also some examples of the C++ code are given. After the whole design is described in chapter 4, these descriptions and examples give a deeper insight in how the application really works. Furthermore one can see that the implementation follows the design quit closely.

5.3.1. Data retrieval

Data retrieval is done in a separate thread. Every 100 milliseconds all necessary values are retrieved out of the MSCFS via the FSUIPC module. Because many values are not in the best usable format it is necessary to convert those values to a more usable format.

```
{
    while (true)
    {
        // Prepare data read actions
        fsuipc->Read(0x0574, 4, &rd_alt);
        fsuipc->Read(0x0B4C, 2, &rd_gnd);
        fsuipc->Read(0x0568, 8, &rd_lon);
        fsuipc->Read(0x0560, 8, &rd_lat);
        fsuipc->Read(0x0580, 4, &rd_hdg);
        fsuipc->Read(0x0578, 4, &rd_ptc);
        fsuipc->Read(0x057C, 4, &rd_bnk);
        fsuipc->Read(0x02BC, 4, &rd_spd);
        fsuipc->Read(0x0BB2, 2, &rd_elv);
        fsuipc->Read(0x0BB6, 2, &rd_ail);
        fsuipc->Read(0x088C, 2, &rd_thr);

        // Process data read
        fsuipc->Process();

        // Convert everything to usable standards
        this->convert();

        Sleep(100);
    }
```

5.3.2. Data exchange

The 'Communicator' module of the bot exchanges data with its opponent every 200 milliseconds. This data exchange is done with one single string that includes all necessary values separated by spaces. After socket initialization is done this code can continue sending and receiving data. Notice that one of both sides needs to receive data first, before the lines of code below can be executed. This is done in the initialization of the connection.

```
    while (true)
    {
        // Retrieve a string with all necessary values
        this->data->getData(str);

        // Send data to opponent
        if (!socket.Send(str,length))
            dlg->AddMessage("bummer-send-data");

        // Receive data from opponent
        if (!socket.Receive(str,length))
            dlg->AddMessage("bummer-receive-data");
        else
            this->data->setData(str);

        Sleep(200);
    }
```

5.3.3. Control

The 'Controller' is not a very complex thread. After all necessary initialization is done it only needs to let the AI components do their work in the appropriate order.

```
    while (true)
    {
        // Only act when the mode is 'autopilot' and not 'manual'
        if (autopilot > 0) {
            // Recognize current situation
            recognizer.Recognize();

            // Decide next maneuver
            decisionmaker.MakeDecision();

            // Execute the maneuver
            executor.ExecuteManeuver();
        }

        Sleep(300); // All functions for flight dynamics are based on 300 milliseconds,
    }
```

5.3.4. Decision-making

As it is prescribed in the design, there is a separate decision-making function for each possible situation. This makes the decision-making reasoning easier to design and to implement.

```
/*************************************************************************
 Chooses a maneuvre based on the current situation
*************************************************************************/
void FBDecisionMaker::MakeDecision()
{
    // Check the current siuation, for each situation there is a specific function
    switch (data->situation)
    {
        case LRG_RNG :
            MakeDecision_LRG_RNG(); break;

        case SHO_RNG_ATK :
            MakeDecision_SHO_RNG_ATK(); break;

        case SHO_RNG_APR :
            MakeDecision_SHO_RNG_APR(); break;

        case SHO_RNG_DEF :
            MakeDecision_SHO_RNG_DEF(); break;

        case SHO_RNG_NEU :
            MakeDecision_SHO_RNG_NEU(); break;

        case SHO_RNG_SPL :
            MakeDecision_SHO_RNG_SPL(); break;
```

At this moment all functions are created by a decision-tree, which is simply implemented by a lot of 'it-then-else'-statements. In the code one can see that, besides the maneuver to execute, a desired heading (variable data->deshdg) is set. This can be done to optimize the maneuver to execute and keep track of the opponent more precisely. Besides a desired heading also a desired altitude can be set.

```
/*************************************************************************
 Chooses best maneuver in short range attack situation
*************************************************************************/
void FBDecisionMaker::MakeDecision_SHO_RNG_ATK()
{
    if (data->ao > 5)
    {
        if (data->aa > 0)
        {
            if (data->e_bnk > 30)
            {
                data->maneuver = maxturnright;
            }
            else
            {
                data->maneuver = straight;
                data->deshdg = data->hdg + data->ao;
            }
        }
        else
        {
            if (data->e_bnk > 30)
            {
                data->maneuver = turnright;
            }
            else
            {
                data->maneuver = straight;
                data->deshdg = data->hdg + data->ao;
            }
        }
    }
    else if (data->ao < -5)
    {
        if (data->aa < 0)
        {
            if (data->e_bnk < 30)
            {
                data->maneuver = maxturnleft;
            }
            else
            {
                data->maneuver = straight;
                data->deshdg = data->hdg + data->ao;
            }
        }
        else
        {
            if (data->e_bnk < 30)
            {
                data->maneuver = turnleft;
            }
            else
```

5.3.5. Maneuver executing

The execution of a maneuver is done with the help of a number of polynomial functions. In the figure below, some of the code of the maximal turn maneuver is presented. Two lines of code are not completely presented here, because that just doesn't fit nicely in this report. In this piece of code some extra adaptations are made to get a nice flying airplane. Most of these adaptations are found more or less by try out some different possibilities.

In the piece of code below, first the difference between the desired and the current bank is computed. In a maximal turn maneuver the desired bank is 85 degrees. Because a desired altitude can be set together with the decision-making for the maneuver to execute, it is possible that the airplane needs to go a little bit up or down in the turn. A bank lower then 85 will cause the plane to go up, whereas a higher bank will cause the plane to go a little bit down. To keep the turn as close to a maximal turn as possible, the desired bank will be responsible only for 1/3 of the bank, 2/3 will be caused by the best bank value for a maximal turn: 85.

Another surprising point is that there are two different functions for the two different directions in which a maximal turn can be taken, left or right, represented in the code by the variable 'dir'. One would expect that the actual function will be the same for both directions and that only a minus sign must be placed in case of one of the directions. But it appeared that from the polynomial regression, different functions showed up for going left or right. It also appeared that these two different functions performed better in practice compared to one single function with an eventual minus sign.

```
        // Compute difference between current bnk and desired bnk
        // Desired bank is 2/3 * 85 degrees + 1/3 * desired bank
        diff = max(-10,min(10,(((desbnk+dir*2*85)/3) - data->bnk)));

        // Compute the ail from the difference in desired bnk and current bnk
        if (dir > 0)
            data->ail = (short)(max(-16000,min(16000,(1.115 * pow(diff,3) + 6.236 * pow
        else
            data->ail = (short)(max(-16000,min(16000,(1.748 * pow(diff,3) -12.269 * pow
    }
```

6. Experiments and results

6.1. Introduction

This project resulted in an implementation of the dogfight agent. The implementation can really play in a Microsoft Combat Flight Simulator multiplayer game. The agent is tested in a few different scenarios. These test scenarios and the results of the agent are presented in this chapter.

6.2. Test scenario 1: Straight flight

6.2.1. Scenario

In the first test scenario the agent will have to take it up against a human player. The human player will only fly a straight flight, so it should not be too difficult for the agent to get in a good position behind the human player.

6.2.2. Results

Figure 22 shows consecutive top views of the paths of both aircrafts. Each view covers 12 seconds. The path of the human player is purple, that of the agent is blue. The agent first takes a small turn right, to head awards the enemy (see Figure 22: view 1). Thereafter it takes a turn left, and finally the agent ends up in the best possible position: close behind the enemy.

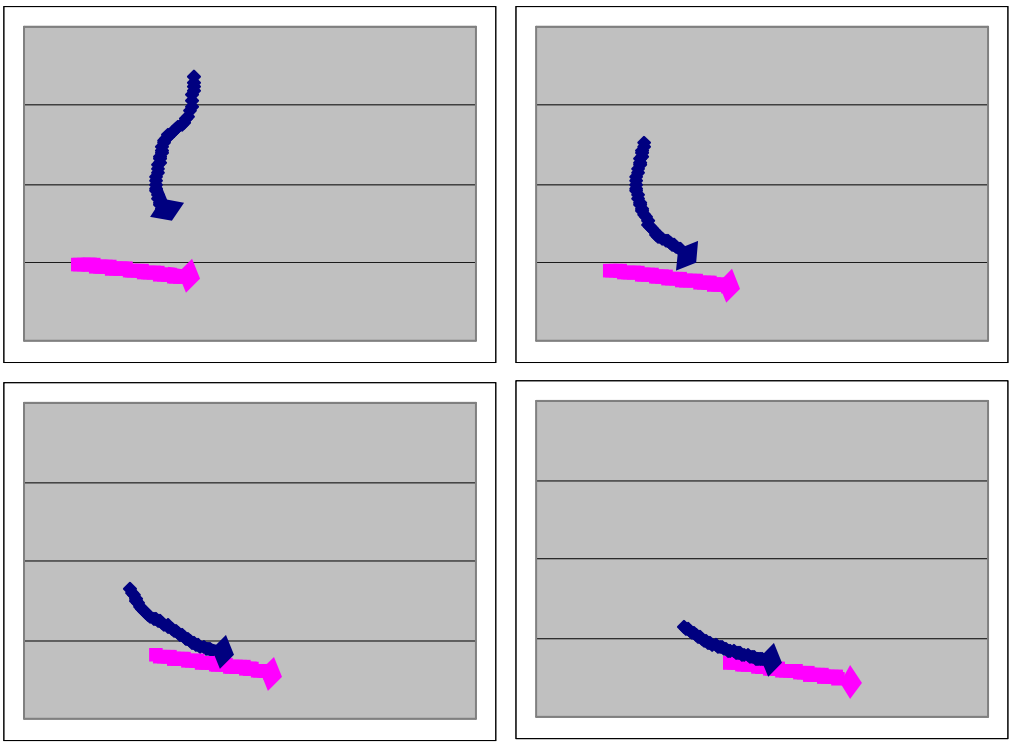

Figure 22: Straight flight / top view / 0-12s, 6-18s, 12-24s, 18-30s

6.2.3. Conclusion

This test scenario does not seem to be a problem for the agent. Within only a few seconds the agent gets in a position right behind the human player and it stays there. This result does not show a lot about the agent's decision-making skills, but it does prove that the agent is able to intercept an enemy and that its situation recognition and also its maneuver execution work, otherwise it would not be possible for the agent to succeed in even this simple test case.

6.3. Test scenario 2: Turns

6.3.1. Scenario

This test scenario will be a little bit more difficult than the previous one. The human player will make some turns left and right. The human player will not make very much or very steep turns, but anyway it will become a little bit more difficult for the agent to get and stay in a position close behind the human player.

6.3.2. Results

Figure 23 shows the consecutive flight path views. Each view covers 18 seconds. Again the human players path is purple, that of the agent is blue. The agent first makes a small turn left and thereafter needs one big turn right to get behind the enemy. It is a little bit far behind, but it rapidly gets closer by making shorter turns.

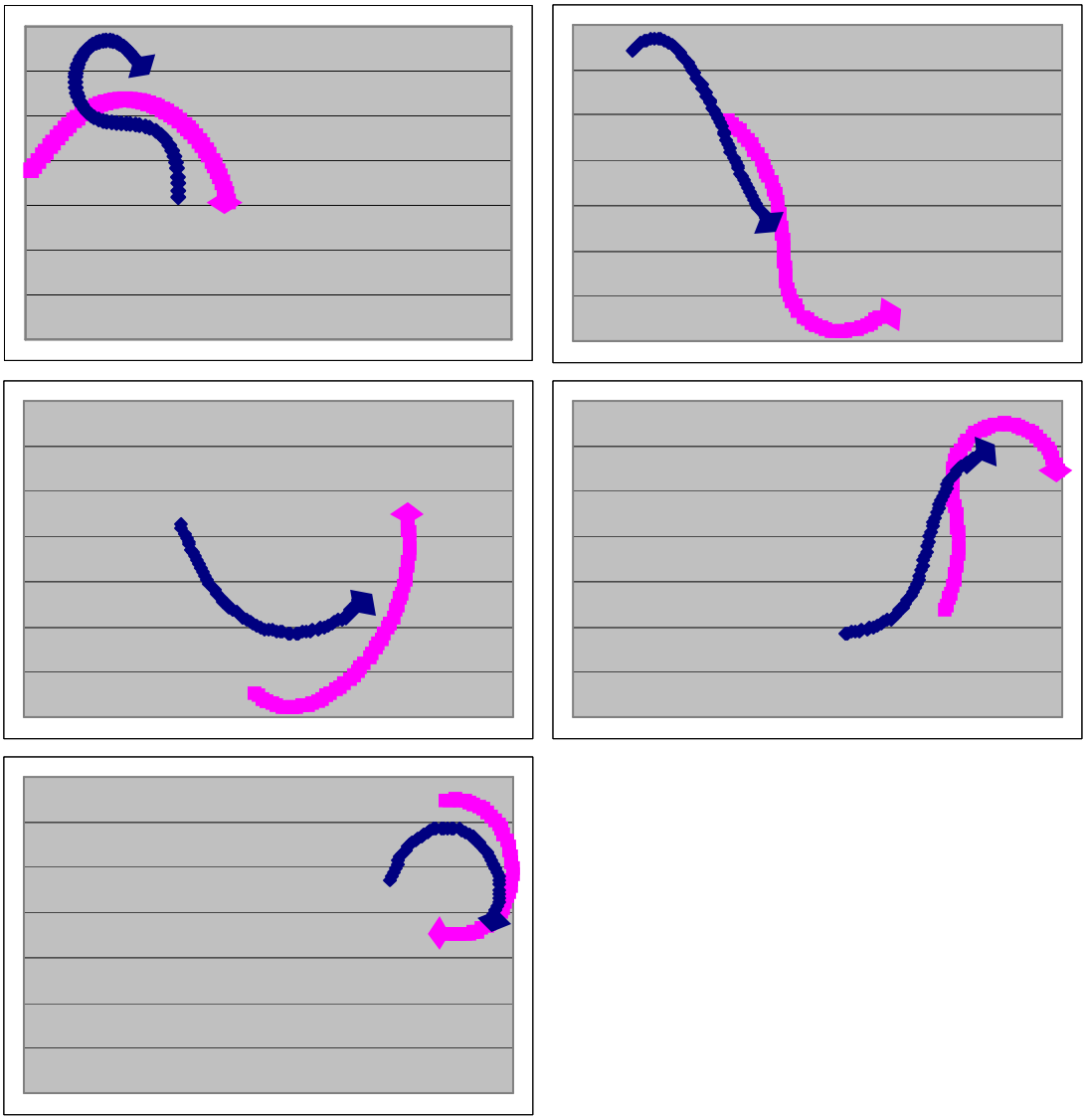

Figure 23: Flight with turns / top view / 0-18s, 12-30s, 24-42s, 36-54s, 48-64s

6.3.3. Conclusion

Equally to test scenario 1, test scenario 2 is perfectly handled by the agent. Test scenario 1 showed that the agent could intercept an enemy. In this test scenario the agent is not tested in extreme situations, such as many steep turns. However this test scenario shows that the agent is able to make the right decisions in a situation with an active opponent.

6.4. Test Scenario 3: Heavy dogfight

6.4.1. Scenario

The third test scenario is a real dogfight. The human player will try to avoid attacks from the agent and will try to get close behind the agent himself. So this time the maneuvers of the human player will be as many and as steep as possible or wanted. This is the most difficult scenario that is possible. Only a better human player could make the scenario a little bit more difficult for the agent.

6.4.2. Results

The results of this test will again be described with a number of consecutive views. The views each cover 12 seconds.

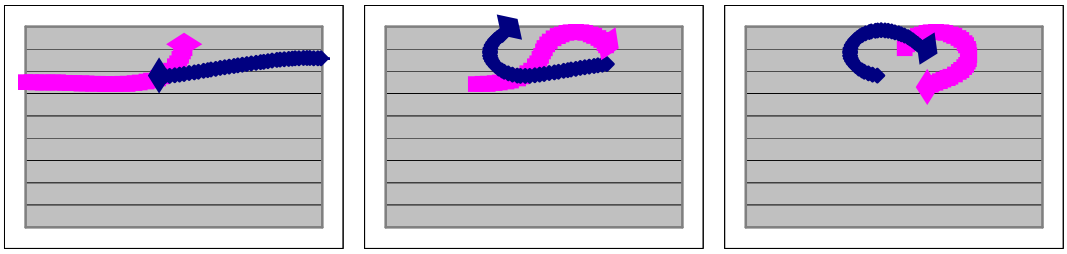

Figure 24: Dogfight / top view / 0-12s, 6-18s, 12-24s

Both players are quit offensive and start by flying towards each other. After they met, both players start to turn around each other.

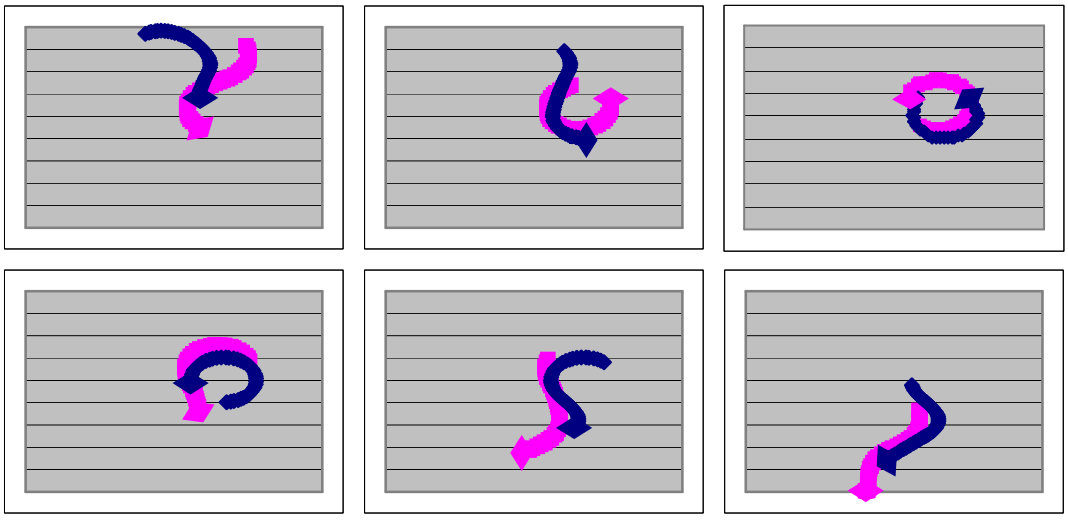

Figure 25: Dogfight / top view / 18-30s, 24-36s, 30-42s, 36-48s, 42-54s, 48-60s

The first turns of the agent are a little bit better then the turns by the human player, so the agent comes into an attacking position. At this stage the agent really is in a short-range-attack situation. The human player is in a defensive situation and tries to shake off the agent by making sharp turns.

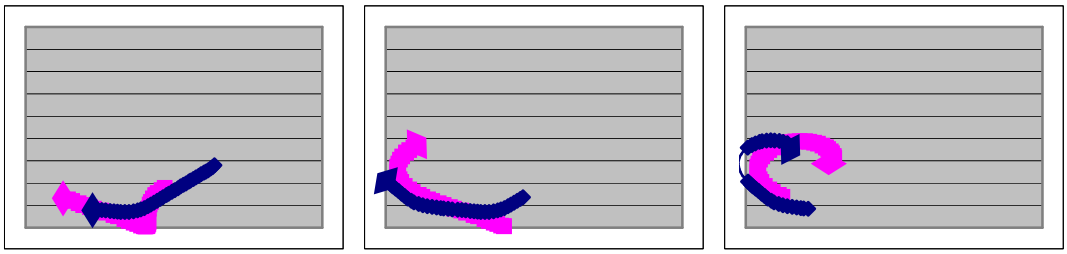

Figure 26: Dogfight / top view / 54-66s, 60-72s, 66-78s

The human player tries a split-s maneuver to escape from the agent. This maneuver results in the sharp edge in view 1 of Figure 26. At this point the human player's aircraft is moving mainly vertically, but this cannot be seen in these views. The agent is not misled by this all and is still able to follow the human player.

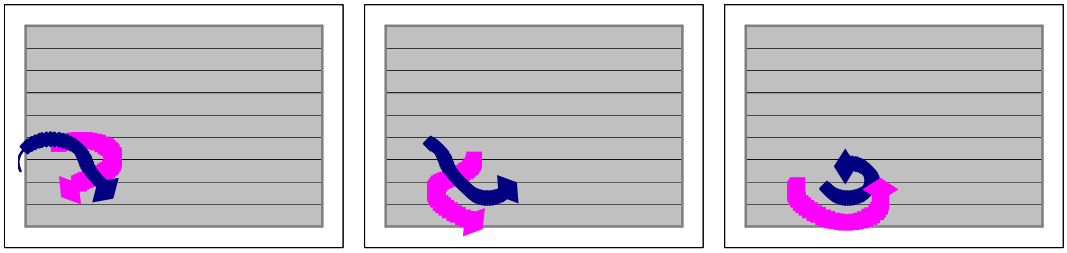

Figure 27: Dogfight / top view / 72-84s, 78-90s, 84-96s

Then the human player makes a maneuver unknown to the agent in which his aircraft looses height and makes a turn. View 1 of Figure 27 shows that the agent does not have an answer to this. The result is that the agent becomes in front of the human player. And the human player can start to follow the agent. In the views it looks like the agent ends up in a short-range-defend situation, but because the human player is flying on a much lower altitude, the actual situation is short-range-higher.

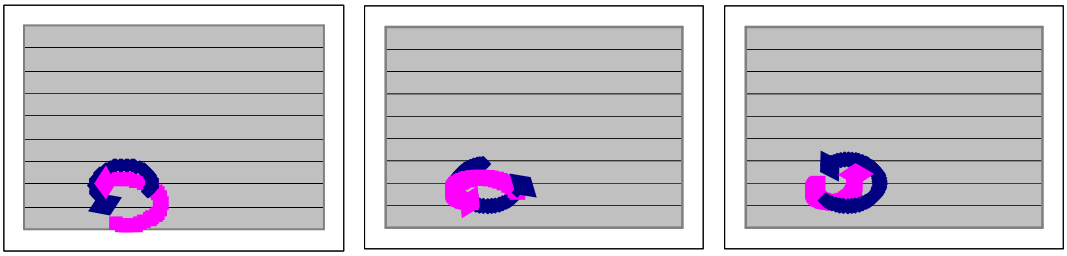

Figure 28: Dogfight / top view / 90-102s, 96-108s, 102-114s

In this stage the human player goes to a higher altitude and so finally comes into a short-range-attack situation. The agent is in a short-range-defend situation and tries to escape from this situation by making very steep turns.

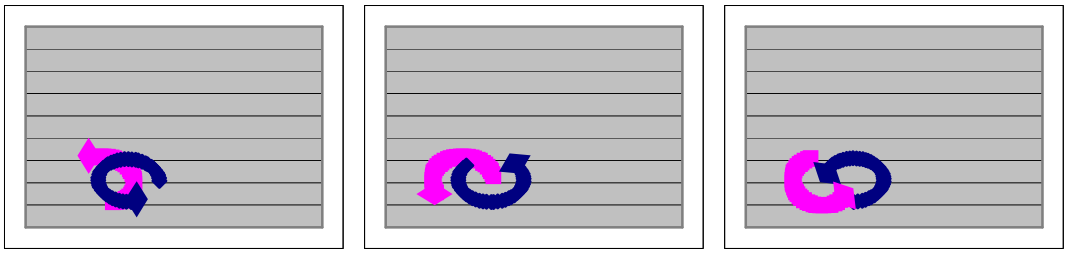

Figure 29: Dogfight / top view / 108-120s, 114-126s, 120-132s

The agent is better in making short turns and in this stage it already gets rid of it's short-range-defend situation and changes it into a short-range-neutral situation.

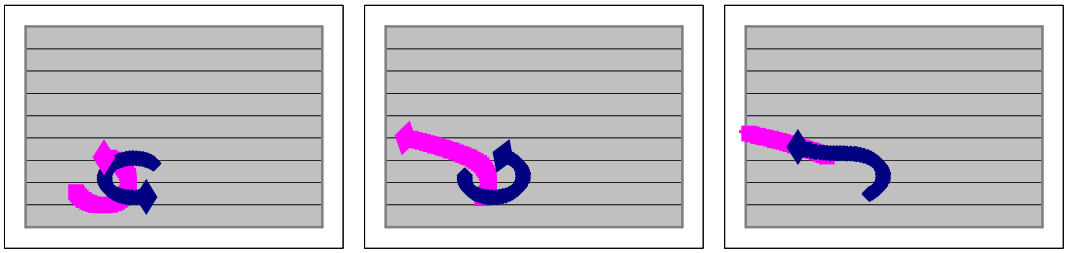

Figure 30: Dogfight / top view / 126-138s, 132-144s, 138-150s

At the end of this test scenario, the human player decides to fly away. The agent follows immediately. The whole test scenario took 2 minutes and 30 seconds.

It is also interesting to see all the consecutive situations in which the agent was during the dogfight test scenario. Table 3 shows all these consecutive situations. For each situation the number of time steps in the agent reasoning process and also the number of real-time seconds is given. Each time step takes 300 milliseconds. The table shows that there were no intervals in which the situation constantly switched between certain states. Often the agent stayed in a situation for a number of time steps. Therefore we can call this a stable process. When this process would be very instable it could cause the agent to constantly change its plan. This would result in a not very effective flight.

Table 3: Consecutive situations during the dogfight test scenario

situation	time steps	seconds
middle-range-approach	39	11.7
short-range-approach	13	3.9
short-range-neutral	21	6.3
short-range-attack	15	4.5
short-range-neutral	4	1.2
short-range-attack	35	10.5
short-range-neutral	8	2.4
short-range-attack	9	2.7
short-range-neutral	2	0.6
short-range-attack	67	20.1
short-range-higher	60	18
short-range-attack	9	2.7
short-range-neutral	14	4.2
short-range-split	1	0.3
short-range-neutral	3	0.9
short-range-higher	61	18.3
short-range-neutral	2	0.6
short-range-defend	27	8.1
short-range-neutral	55	16.5
short-range-approach	1	0.3
short-range-neutral	31	9.3
short-range-attack	46	13.8

Figure 31 shows a graph of Table 3. Each situation has its own color. A wider block means a longer duration of that situation. This figure also shows that situation switches do not occur all the time. Only in the beginning there was some switching between the attack and neutral states.

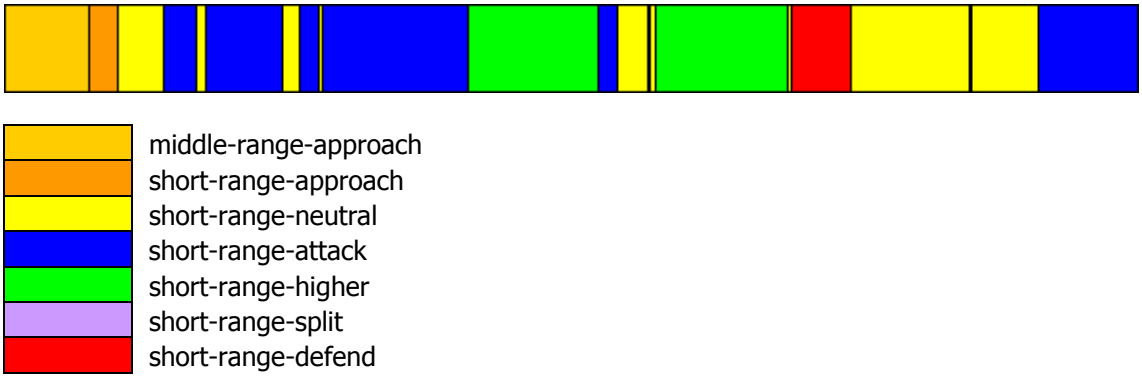

Figure 31: Graph of consecutive situations during the dogfight test scenario

Another conclusion is that the agent spend much more time in an attack situation then in a defend situation.

6.4.3. Conclusion

This test scenario shows that the agent is able to recognize situations, make decisions and execute maneuvers in a highly reactive dogfight situation. The agent really is a competitive player in a one-to-one dogfight. In this test the agent seemed to be even a little bit better than the human player. At about second 10 and also at about second 110 it is clear that the agent improved its situation just by a better and faster reaction compared to the human player. The human player on the other hand performed better at about second 80. In this case the human player was more creative than the agent, which resulted in an advantage for the human player.

The overview of situations the agent was in during the test scenario shows that the position recognition works fine together with the state-based decision-making. The position recognition decision-tree was build up by human choices. But the positions are divided in such a way that the agent is able to stay in a position for a certain amount of time. This seems quite obvious, but it could be that the agent would switch constantly between for example the 'short-range-neutral' and the 'short-range-attack' situation. Such an unstable behavior is not effective and therefore not desired.

6.5. Test scenario 4: Agent against agent fight

6.5.1. Scenario

In this test scenario the agent will fly against itself. This means that both players in the MSCFS game will be controlled by an agent of the same type.

6.5.2. Results

In Figure 32, view 1 shows that both agents fly towards each other in almost the same way. They start turning around each other in view 2.

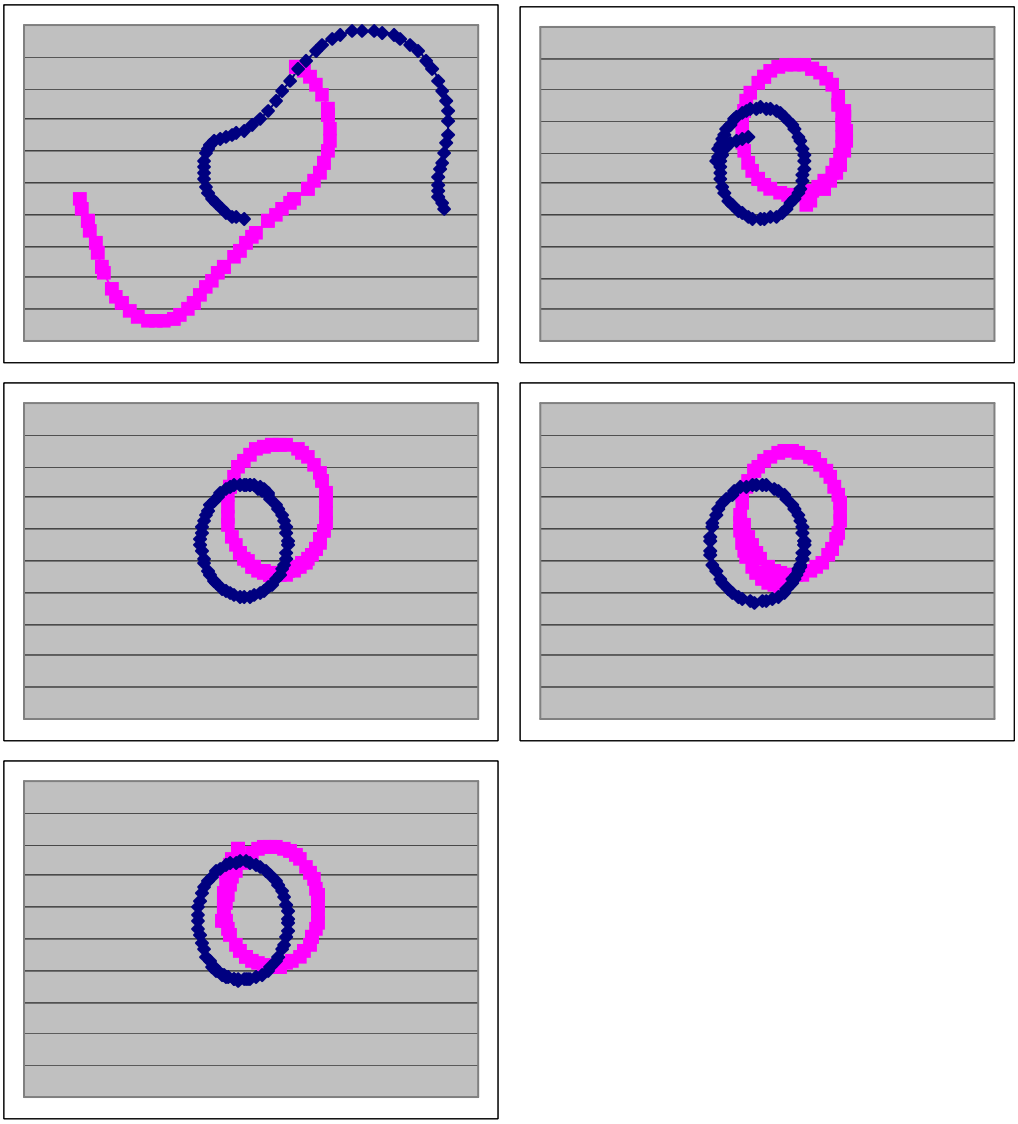

Figure 32: Agent against agent fight / top view

6.5.3. Conclusion

Because they start the fight in the same manner and they both control their aircraft in the same way, the two agents end up flying circles around each other. Probably they will keep flying this way until one of both runs out of fuel. Probably, in such a scenario one of the agents can only end up in a winning position, when it had an advantage in the situation at the start.

7. Conclusions

7.1. Rememorize the goals

Before we draw conclusions about this project and its results, let us take a step back and rememorize the goals of this project:

The goal of this project was, to create an agent that can be a competitive player in a one-to-one dogfight situation in MSCFS. This goal consisted of three assignments:

- A literature study about related projects
- A model and a design of the agent
- A prototype of the agent in the form of a C++ implementation

7.2. Literature study

Section 3.1 of this paper describes the literature study that was done. A more detailed description of this literature study can be found in [Solinger, 2004]. The literature study presents some projects in different areas that are related to this project. The literature study made clear that successful projects exist in all these different areas. However, most projects focus on one single aspect, whereas for a successful agent many aspects are important. This was already suggested in the project description in section 2.1 where the idea is proposed for a project in which some different aspects are combined in one single agent.

The projects in the area of intelligence in computer games often focus on some more different aspects. The way an intelligent agent was created in these projects has been a source of inspiration for this project. For all different flight aspects of the agent more specific flight projects were inspiring.

7.3. Model and design

In the chapters 3 and 4 a model and a design are presented. The design is set up in such a way that it meets all requirements. First of all the agent will be able to 'play' the MSCFS without any human help. Another requirement on the behavior of the agent is that it will play 'like a human player'. Therefore the agent can only act with the same set of parameters as a human player can. In other words: The agent cannot cheat. It cannot reach a higher speed or make a steeper turn or whatsoever compared to an expert human player.

A modular design was required so that the current agent can be altered or extended in the future. This requirement is very well met in the current design. Each separate part of the agent is designed as a separate object. Also each single intelligent part is put into a separate object. With this design it will be quit easy to improve for example only the decision-making. Even a complete other way of decision-making will be possible. The current rule-based decision-making object can be replaced by for example a neural network decision system without the need for changing one of the other objects. Furthermore a whole new object in the intelligent layer will be possible without any changes in the other objects in the intelligent layer. In such a case only the Controller must be altered so that it activates the new object when necessary.

7.4. Implementation of a prototype

A prototype is implemented according to the presented design. The prototype is fully functional. The performance of the implementation is actually quite good (see chapter 6). The prototype has become a 'competitive player' in a dogfight situation. Furthermore the implementation works real-time, as it was required, and does not take too much CPU time. This means that the implementation is really usable in practice.

7.5. Total project

The performance of the complete agent is dependent of the performance of each of the individual aspects of the flight automation process. This conclusion underlines the relevance of this project. By splitting the whole automation process into multiple elements it is possible to improve each of these individual elements. During the development of the prototype the performance of the bot increased significantly. Sometimes improvements in only one of these objects resulted in a big improvement in the performance of the bot. For example a small improvement in the 'turn' maneuvers once resulted in a competitive bot instead of a strange behaving airplane. It is also important to combine the elements in the right way to fully use the power of each of the individual elements.

The overall goal of creating a competitive player for a one-to-one dogfight in MSCFS has been reached. The third test scenario, which is described in section 6.4, shows that the agent is capable of dealing with a heavy dogfight situation. In many situations the agent performed even better compared to its human opponent. The agent reacted faster and it executed the maneuvers better. On the other hand a human player can have an advantage because he can be more creative and execute yet undefined or unexpected maneuvers. Altogether the current state of the agent can be compared to the first check computer programs. The agent always reacts properly and fast in known situations, this can be a large advantage when a human opponent makes a small mistake or looses concentration for a few seconds. The only way for a human player to defeat the agent is to use its ability to be more creative than the agent.

8. Future work

8.1. Advanced situation-based decisions

In the current design and prototype the decisions are situation based. This means that for each different situation there is a specific rule-based system (section 4.3.3). This works well, because it is easier to create a number of small rule-bases, than one large rule-base. The other advantage is that one rule-based system can be fully focused on one specific situation.

The fact that the decision-making process can be focused on one specific situation can be used even more. For example the decision-making process in a 'short-range-attack' situation is very different from that in a 'middle-range-defend' situation. In the 'middle-range-defend' situation decision-making is a rather creative process. One should think of an original way to escape by executing extreme and unexpected maneuvers. In the 'short-range-attack' situation decision-making is more focused on a very detailed perception of the enemy's movements and a very precise following of the enemy's airplane. It is possible to use a different method for decision-making in each different situation. One could think, for example, of a neural network approach for the 'short-range-attack' situation. The network could be trained specifically to follow the enemy as close as possible by quick and appropriate reactions on the enemy's movements.

It would be interesting to find out whether different decision-making methods are more suitable in specific situations. As described different situations require different approaches and so, maybe also different decision-making methods. Such a change could improve the performance of the agent. In the current design and implementation these changes can be made without any problems.

8.2. Prediction

In the object architecture (section 3.3.1) an object is presented in the intelligence layer for prediction. This however, is not further included in the design and implementation in this project. Of course it is interesting to investigate the effect of prediction on the overall performance of the bot. For example, future positions of both airplanes could be predicted, but also things like the chance the enemy changes its maneuver. So, first a list is necessary of useful future values that can be predicted.

To include prediction is this project a prediction object needs to be designed and implemented. Also the decision-making should be altered, so that it makes use of the predicted values.

8.3. Adaptive flight behavior

The flight behavior of the prototype is only suitable for one type of airplane. The reason of course is that this flight behavior is based on data from other flights with the same type of airplane. It will not be very difficult to change the execution object, so that it can be used for another type or airplane. This is simply a way of using data for other flights with the proper type of airplane.

A more interesting problem is that the current flight behavior is also not usable for an airplane that is hit by only one single bullet. In such a situation an airplane can often still fly very well, but not with exactly the same flight dynamics. In fact this happens in the MSCFS all the time. A human player can try to adapt to a slightly different flight behavior of his airplane. That is what the bot should do too. In [Liang, 2004] a method is presented using a neural network that can adapt during the actual flight. The prototype however is far from usable in a dogfight situation. Including adaptive flight behavior in this project will be a challenging problem.

8.4. Cooperating bots

Instead of looking into the details of the bot as it is presented in this project it is also possible to look a step further and to investigate the possibilities of multiple cooperating bots. The basis for this is available. There is a bot that can really fly in a multiplayer game in the MSCFS and that can also communicate via a TCP/IP network with another bot.

A group of agents should define some global goals and/or a global strategy and then each single agent should reach its own local goal or perform its own tasks. The field of cooperating agents is an area where there is still much to explore.

8.5. Probabilistic approach

All data used in this project is concrete. There is no missing data and there is no uncertainty. This is, because the environment of the agent is a computer game in which all necessary data is available at all time. In a real life environment this would not be the case. In that case a probabilistic approach might be necessary. Such a real life situation could be emulated by introducing noise in the data that is retrieved from the MSCFS game. If one sees it as an end goal to create an agent that can deal with a computer game, a probabilistic approach will not be necessary. But when one wants to take a step further and extend an agent like the one described in this report to be able to deal with a real-life situation, it might be interesting to think of methods that can handle errors in the data.

Bibliography

[Andriambololona, 2003]
Andriambololona, M. and Lefeuvre, P. (2003). "*Implementing a dogfight artificial pilot*", Research report DKS03-07/ICE07, Knowledge Based Systems group, Delft University of Technology, The Netherlands.

[Boogaard, 2002]
Boogaard, J. and Otte, L. (2002). "*Flightgear multiplayer engine*", Technical Report DKS02-05/ICE02, Knowledge Based Systems group, Delft University of Technology, The Netherlands.

[Brookes, 1989]
Brookes, R.A. (1989). "Fast, Cheap, and out of Control: A Robot invation of the Solar System", in *Journal of the British Interplanetary Society*, Vol.42, pp.478-485

[Capkova, 2002]
Capkova, I. Juza, M. and Zimmermann, K. (2002). "*Explorative data analysis of flight behavior with neural networks*", Research report DKS02-04/ACE02, Knowledge Based Systems group, Delft University of Technology, The Netherlands.

[Coradeschi, 1996]
Coradeschi, S., Karlsson, L., and Törne, A. (1996). "Intelligent agents for aircraft combat simulation", in *Proceedings of the 6th Computer Generated Forces and Behavioral Representation Conference*, Orlando, Florida, USA.

[Ehlert, 2003]
Ehlert, P.A.M. (2003). "The Intelligent Cockpit Environment Project", Research report DKS03-04/ICE04, Knowledge Based Systems group, Delft University of Technology, The Netherlands.

[Endsley, 1997]
Endsley, M.R. Mogford, R.H. Allendoerfer, K.R. Snyder, M.D. and Stein, E.S. (1997). "*Effect of Free Flight Conditions on Controller Performance, Workload, and Situation Awareness*", Techical note DOT/FAA/CT-TN 97/12, Federal Aviation Administration, William J. Hughes Techical Center, Atlantic City NJ, USA.

[Evans, 2002]
Evans, R. (2002). "Varieties of Learning", in *AI Game Programming Wisdom*, Rabin, S. (editor) Charles River Media, Inc., pp 567-578.

[Fyfe, 2002]
Fyfe, C. (2002). "Dynamic Strategy Creation and Selection Using Artificial Immune Systems", in *International Journal of Intelligent Games & Simulation*, Vol.3 No1, March/April 2004

[Harreman, 2003]
Harreman, R. and Roest, M. van der (2003). *"SAM: Project thesis on Situation Awareness Module*", Technical Report DKS-03-04/ICE02, Mediamatics / Data and Knowledge Systems group, Departement of Information Technology and Systems, Delft University of Technology, The Netherlands.

[Jones, 1994]
Jones, R. M. Laird, J. E., Tambe, M., & Rosenbloom, P. S. (1994) "Generating behavior in response to interacting goals", in *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*, pp 317–324. Orlando, Florida, USE.

[Laird, 2001]
Laird, J.E. (2001). "It Knows What You're Going To Do: Adding Anticipation to a Quakebot", in *Proceedings of the fifth international conference on Autonomous agents*, pp 385-392.

[Laird, 1998]
Laird, J.E. and Jones, R.M. (1998). "Building Advanced Autonomous AI Systems for Large Scale Real Time Simulations", in *Proceedings of the 1998 Computer Game Developers' Conference*, pp 365-378. Long Beach, California, USA.

[Lent, 1998]
Lent, M. van and Laird, J.E.(1998). "Learning by Observation in a Complex Domain", in *Proceedings of the Workshop on Knowledge Acquisition, Modeling, and Management*. Banff, Alberta, Canada.

[Liang, 2004]
Liang, Q. (2004) *"Neural flight control autopilot system",* Research Report DKS04-04 / ICE 09, Knowledge Based Systems group, Delft University of Technology, The Netherlands.

[Machado, 2002]
Machado, R. (2002). "Rod Machado's Ground School*" Microsoft Flight Simulator 2002.*

[Manslow, 2002]
Manslow, J.. (2002). "Learning and Adaptation", in *AI Game Programming Wisdom*, Rabin, S. (editor) Charles River Media, Inc., pp 557-566.

[Mouthaan, 2003]
Mouthaan, Q.M. (2003). "Towards an intelligent cockpit environment: a probabilistic *approach to situation recognition in an F-16*", MSc. thesis, Knowledge Based Systems group, Delft University of Technology, The Netherlands.

[Pottinger, 2000]
Pottinger, D.C. (2000). "Game AI: The State of the Industry, Part Two", in *Gamasutra*, http://www.gamasutra.com/.

[Russell, 1995]
Russell, S.J. Norvig, P. (1995). "*Artificial Intelligence, a Modern Approach*" Prentice Hall, Inc.

[Solinger, 2004]
Solinger, D. (2004). "*Creating a dogfight agent: a literature study"* , Research Report DKS04-03 / ICE 08, , Knowledge Based Systems group, Delft University of Technology, The Netherlands.

[Spronk, 2003]
Spronck, P. Sprinkhuizen-Kuyper, I. and Postma, E. (2003). "Improving Opponent Intelligence Through Offline Evolutionary Learning", in *Proceedings of the Fourteenth Belgium-Netherlands Conference on Artificial Intelligence*, pp 299-306.

[Spronk, 2004]
Spronck, P. Sprinkhuizen-Kuyper, I. and Postma, E. (2004). "Online adaptation of game opponent AI with dynamic scripting", in *International Journal of Intelligent Games and Simulation, Vol 3 No 1*, pp 45-53.

[Tamerius, 2003]
Tamerius, M.S. (2003). "Automating the cockpit: constructing an autonomous *human-like flight bot in a simulated environment*", Technical report DKS-03-05/ICE05, Knowledge Based Systems group, Delft University of Technology, The Netherlands.

[Tozour, 2002]
Tozour, P.. (2002). "The Evolution of Game AI", in *AI Game Programming Wisdom*, Rabin, S. (editor) Charles River Media, Inc., pp 3-15.

[Vidal, 1998]
Vidal, T. and Coradeschi, S. (1998). "A temporal, contextual and game-based approach for highly reactive monitoring systems", in *ECAI-98 workshop on Monitoring and Control of Real time Intelligent Systems,* Brighton, UK, August 1998.

[Virtanen, 2001]
Virtanen, K. Raivio, T. and Hämäläinen, R.P. (2001). "Modeling Pilot's Sequential Maneuvering Decisions by a Multistage Influence Diagram", in *Proceedings of the AIAA Guidance, Control, and Navigation Conference,* Montreal, Canada, 2001.

[Waveren, 2001]
Waveren, J.M.P.van. (2001). "*The Quake III Arena Bot*", MSc. thesis, Knowledge Based Systems group, Delft University of Technology, The Netherlands.

[Woodcock, 2000]
Woodcock, S. (2000). "Game AI: The State of the Industry, Part One", in *Gamasutra*, http://www.gamasutra.com/.

[Alert]
http://www.flytechsoft.com/.

[The Real Cockpit]
http://www.therealcockpit.com/.