Aerial Guardian

Iem Heng
DeVry Institute of Technology
Long Island City, NY
iheng@ny.devry.edu

Jernone James
ESpeed/Cantor Fitzgerald
NY, NY
jerjames@optonline.net

George Rainone
DeVry Institute of Tech.
Long Island City, NY
g3rain1@optonline.net

Abstract

On September 11th 2001 terrorists broke into the cockpits of four commercial airplanes, killed the pilots and took control of the aircraft. They then flew the planes many miles off their intended flight paths heading for New York and Washington D.C. They then crashed them into the Pentagon and World Trade Center, killing thousands. They accomplished this with little more than box cutters as weapons. This shows how loose our security really was. What if there was a way to prevent them from crashing the planes into buildings, what if there were countermeasures to prevent them from flying so far off their flight path, what if there was a security system that could have prevented them from taking control of the aircraft at all, even if they managed to incapacitate the pilots?

The Aerial Guardian is that system; it is intended to prevent such disasters by preventing terrorist from being able to take control of the airplanes, and flying them into populated areas. The Aerial Guardian does this by using biometrics to identify the pilots of the aircraft and will only allow authorized personal to operate it. Biometrics is the method of scanning and recognizing unique physical features of a person's body. This includes facial recognition, voice pattern identification, and fingerprint identification. The Aerial Guardian stores the biometric information of the pilots in a computer onboard the aircraft. While the aircraft is in flight, the system scans the person who is behind the controls of the airplane and compares it with the stored information. If the person is not recognized, it will take the appropriate actions to ensure the safety of the passengers and people on the ground.

In times of crisis, the airplane is controlled by the Arial Guardian. It is programmed to follow the aircrafts predetermined flight path, avoid populated areas and recover from instabilities such as nosedives and excessive roll; as such a situation might occur if there is a struggle in the cockpit between the pilots and hijackers.

2. SYSTEM OVERVIEW

2.1 Purpose of Aerial Guardian System (Hardware and Software)

The Aerial Guardian (A.G) System was developed in order to decrease the threat of terrorists taking control of commercial airplanes. The main system of Aerial Guardian is called the Aerial Guardian Protector (A.G.P). The A.G.P is responsible for preventing unauthorized pilots from flying the airplane. Its first line of defense relies on biometric identification device. The device allows the A.G.P system to prevent unauthorized pilots from misusing commercial airplanes.

The A.G.P system will also prevent unauthorized pilots from endangering civilians on the airplane and ground. It will do this by preventing unauthorized pilots from deviating from the flight path, entering into restricted areas and intentionally causing an upset condition such as diving toward the ground. When any of these situations are detected, the A.G.P will take control of the airplane.

The A.G.P is a stand-alone system that is designed to monitor the airplane and takes control if a problem arises. It makes the use of biometric device to identify pilots and the planes instrumentation to determine if the airplane is within a restricted area, outside of its flight path or is in an upset situation.

The system handles problems with the use of rules. If a rule is violated, the A.G.P will take control of the airplane in order to correct the situation. For example, if the rules [Figure 3] are set to their default values, the identified pilots may fly any where except for restricted areas. However, non-identified pilots will only able to fly within the flight path. The rules are pre-defined within the system; however, they may only be modified during maintenance operations.

The Aerial Guardian Trainer (A.G.T) is a 3D flight simulator that aids in the training of pilots, in demonstration of the system. As seen in [Figure 1] below a flight stick, which has a fingerprint scanner mounted unto it [figure 11], and the throttle are used to simulate the airplanes control inputs. The fingerprint scanner is used to identify the pilot's identity. The only task left is to mold a case around the flight stick, so that the pilot must curve his/her finger in order to activate scanner. This addition will prevent terrorist using the pilot's hand to fly the airplane. The A.G.T communicates with the A.G through a parallel port interface.

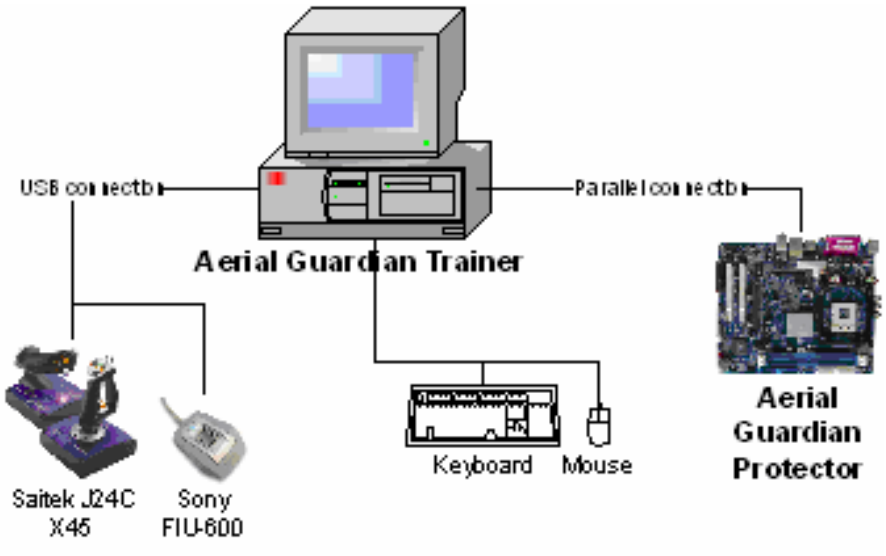


Figure 1: (Hardware block diagram)

2.2 Performance Requirements

The A.G.P/A.G.T systems have a number of modules that must perform adequately to ensure the proper operability of the system. The main control loop of A.G.P must cycle at least eight times per second on a 500 MHz Pentium II or higher. It must be able to handle corrupt input data from the plane. Finally, the communications interface must be able to transfer data at 9,600 bits per second.

The A.G.T must be able to run the simulation with frame rates of over 30 fps (frames per second). The fingerprint scanner must have a suitable false rejection rate, so that the legitimate pilot can be identified without failure. The fingerprint scanner must have suitable and false acceptance rate, so that the intruders do not gain access to the airplane. Lastly, the fingerprint scanner and the simulation must run simultaneously.

2.3 Design Methodology

We choose a modified waterfall model [1] as the development model of the Aerial Guardian. Waterfall model defines as a software process model with discrete development stages: specification, design, implementation, testing and maintenance. In principle, one stage must be completed before progress to the next stage is possible. The reason we decided to choose a modified model is because it is rigid, allowing little room for changes after a development phase. The nature of our project required us to blend stages together. This is especially when we designed and implemented modules that utilize innovative techniques/technologies like artificial intelligence. Thus, the waterfall model would fit into the criteria of blending stages together.

We also employed the ETVXM (Entry, Task, Verify, Exit and Measure) architecture throughout the development of our project. This technique is a feedback system efficiently assisted us in the creation of code and documentation. We used flow charts and UML diagrams to effectively communicate our ideas amongst each other. UML is a Unified Modeling Language that is used in object-oriented development, including several types of system model that provide different views of a system. After the specification, we created a Gantt chart in

order to identify the time it would take to complete each module. Once this was completed, tasks were assigned according to each individual's expertise.

The major steps in the development were:

1. The Analysis Of The Project*:* Documented the main objectives of the project
2. The Requirements Of The Project: Documented the functionality, performance and constraints of the project.
3. Software Architecture Design: Described the three basic elements of our software namely the architecture; processing, data and connection elements.
4. Implementation: Actual coding of each module.
5. Testing Of Individual modules: Local testing of each module.
6. Hardware and Software Interface: Interfacing local elements within module.
7. Complete System Testing: Entire system testing (all modules interfaced).

2.4 Innovative Ideas

Our project uses a number of new and innovative ideas to improve the security of commercial airlines. One method is using a biometric identifier(s). Biometric identifiers will allow the A.G system to identify any unauthorized pilot; if an unauthorized pilot is found, the system will restrict his/her ability to control the airplane. During the implementation stage, we encountered a project that was similar to one section of our project. A project named Soft Walls is being developed to guide planes away from restricted areas. This makes our related section of the project not innovative.

However, an innovative feature of the A.G is that it corrects upset conditions and restricts the airplane to its flight path, thus guiding it to an airport for landing. The A.G.P employs Artificial Intelligence techniques that allow the system to account for degrees of change in the airplanes orientation and velocity. This enables the A.G to learn how to pilot the airplane under a wide range of situations.

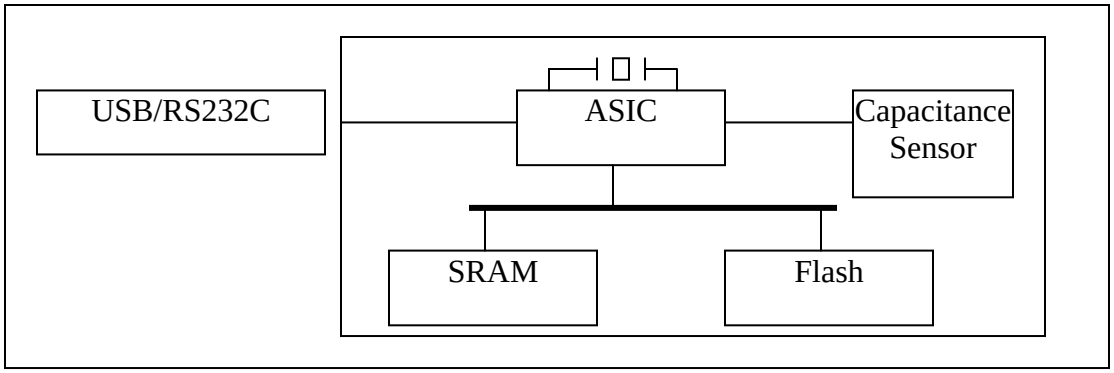
2.5 Hardware Specification and Fingerprint Identification for FIU-600

As we mention in section 2.4 on the biometric identifier(s), it is used to identify the proper pilot in control of the airplane. If any unauthorized pilot at the control, it will take the appropriate action. To understand how the FIU-600 works, we produce a list of hardware specification in the following table.

Dimensions		52(W) x 69(L) x 25(H) mm
Weight		55g
Power	Power Supply	D.C 5V ± 10%
	Power Consumption	1.0 W (Peak for verification)
Temperature Range		5°C to 35°C
Sensor Block	Number of Pixels	128 x 192
	Pixel Size	80µm x 80µm
	Sensor Area	10.2mm x 15.4mm

Memory Block	User Area	48Byte x 1000
	Template Data Size	512Byte
	Quantity of Enrollment (internally)	1000
Verification Block	Verification Method	Pattern Matching
	Verification Time	< 0.1 sec
	Registration Time	< 1.0 sec
	False Acceptance Rate	<0.01% (Measured by Sony)
	False Rejection Rate	<1.9% (Measured by Sony)
Interface and Transfer Rates	USB Bulk Data Transfer	12Mbps (Max)
	(RS232C)	1200bps to 115.2Kbps

In addition, we are illustrated how the fingerprint identification works in the following figure.



From the above figure, a fingerprint is scanned into the buffer where it is stored in raw (grey scale) format. The buffer actually stores the print in three formats. The print is scanned in 8-bit grey scale, then converted into a 1-bit monochrome image, and then to a template (template data). It is the template data that is then stored in the non-volatile flash RAM. A person's identity is verified by scanning a print, creating the monochrome image, and verifying the monochrome data against the template previously stored at a specific index. A custom ASIC (see above figure) is used for the fingerprint analysis. More details about FIU-600 can be found in [16].

***Note**:* The user may want to clear the buffer area after each logical operation to prevent using data erroneously. Data in the buffer is only deleted when it is overwritten or deleted by an API function. Some times data is not written to the buffer because a finger was not detected, but the old fingerprint data would still exist in the buffer [16].

3. Implementation and Engineering Considerations

3.1 Requirements:

We interviewed and surveyed pilots in order to take their concerns into account. Their knowledge and experience helped us to form the design and requirements of the Arial Guardian System. For instance, Michael Jacobson [15] gave specific insight on how to define the default rules. In addition, he suggested that pilots should receive warnings about possible takeovers. In summary, he was please with the demonstration of Arial Guardian System because it is possible

that future commercial airplane needs an additional safety system to encounter the terrorists. It may not solve all the problems, but it helps to have an additional safety system, Arial Guardian, in the airplane.

3.1.1 Major Requirements - Aerial Guardian Protector (A.G.P)

- Stop airplane from entering restricted areas
- Maintains comfort level for passengers
- Adapts to different models of planes
- Alert the pilot of any impending takeover
- Compatible with current autopilot systems
- System must be stable and reliable
- Convert input to vector based units
- Main Loop cycles at over 8Hz (PII 500Mhz)

3.1.2 Major Requirements - Aerial Guardian Trainer (A.G.T)

- Realistically simulate a commercial airplane
- Simulate different models of airplanes
- Friendly/Intuitive user interface
- Frame rates over 30 fps (P4 2.4GHz)
- Output instrumentation in format of an airplane

3.1.3 Requirements - Communication

- Transfer information at 10K bits per second continuously and 20K bits per second peak.
- Differentiate between data types i.e. thrust, bearing, etc

3.1.4 Deciding On Development Tools - Aerial Guardian Protector (A.G.P)

Dev. Tool Type	Name	Reason for selection
Programming Language	C++	Object-Oriented/Familiarity/Speed vs. most others
Operating System	Linux	POSIX compatibility and reliability

3.1.5 Deciding On Development Tools - Aerial Guardian Trainer (A.G.T)

Dev. Tool Type	Name	Reason for selection
Programming Language	C++	Object-Oriented/Familiarity/ Speed vs. most others
Graphics API	DirectX 9.0	Versatility(i.e. direct input)/Speed
Operating System	Windows	Widespread usage and DirectX compatibility
Operating System API	Win32	Faster than MFC

3.2 Aerial Guardian Protector (A.G.P)

The following section will provide a detailed description of the design and implementation of the A.G.P. The A.G.P is the heart of the A.G System. The A.G.P is responsible for detecting problems with the airplane and controlling the airplane, if a rule is broken. The system is designed to be modular, efficient, and most importantly reliable.

3.2.1 Operating System - Linux

We are implemented the A.G.P on Linux, which is a fairly reliable and secure operating system. Most importantly, Linux is POSIX (Portable Operating System Interface) compatible; therefore, programs written on Linux can be ported to other POSIX compatible operating systems without the need to modify the program.

Many premier RTOS (real-time operating systems) are POSIX compatible. RTOS are highly reliable operating system, which designed to execute time critical tasks in a fixed amount of time. One of these Operating systems is Lynx OS 4.0. Lynx OS 4.0 is a FAA approved

POSIX compatible real-time operating system. This operating system is used in the United States Air Force, United States Army, NASA and Boeing [21].

Our program can be ported to non-POSIX compatible operating systems because only two of the A.G.P's classes depend on the operating system, namely, the AGTimer and Communication classes. In fact during the early stages of development, the A.G.P ran on the windows operating system with no faults.

3.2.2 Design/Implementation - Aerial Guardian Protector (A.G.P)

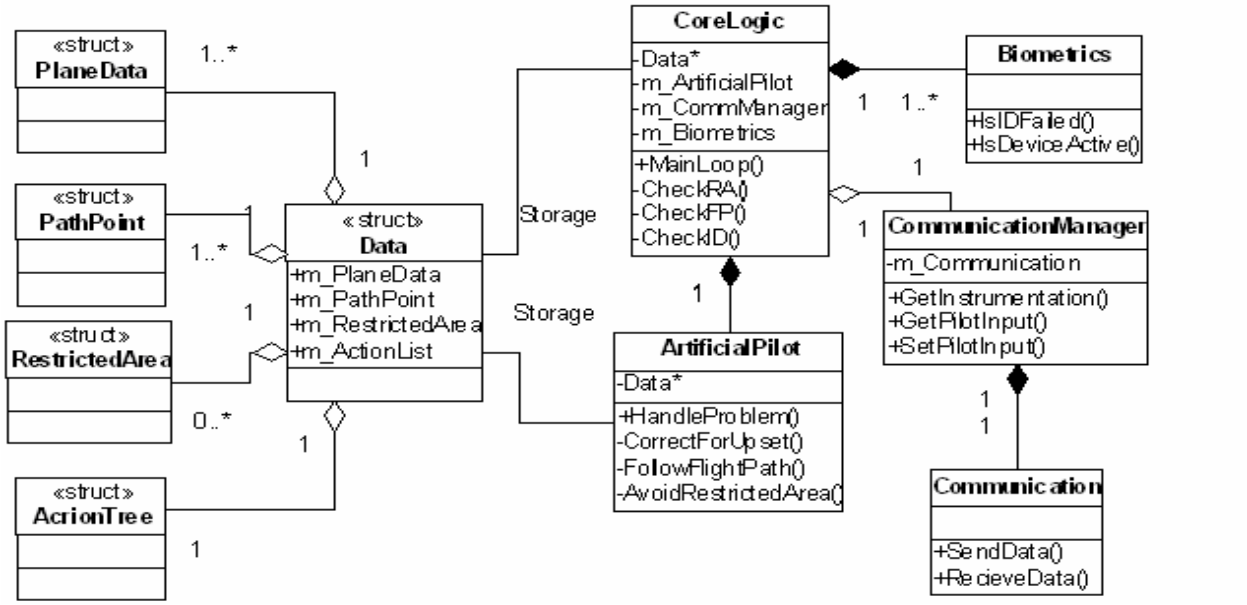


Figure 2: (UML class diagram)

I. CoreLogic::MainLoop

The A.G.P was designed from a functional perspective; each class in the A.G. serves a distinct purpose. The top level of the A.G is the CoreLogic class. The main loop of CoreLogic coordinates the efforts of all other objects in the A.G.P. The CoreLogic is also aided by a multitude of major classes. For example, the ArtificialPilot class handles problems by controlling the airplanes when problems arise. CommunicationManger class transmits information between the A.G.P and the airplane or A.G.T. Finally, the Biometric class is used to identifying pilots, thus increasing overall safety.

II. Rules

The chart to the right gives a visual description of how the A.G.P uses rules handles various problems. These problems include Upset (Is an Upset Condition Detected?), R.A (Is the airplane inside a Restricted Area?), F.P (Is the airplane outside of its flight path?) and I.D (has the biometric identification of the pilot failed?). A value of one indicates the particular problem is detected, zero equals not detected.

A one in the Default Rules column indicates that the system should take control of the airplane if the sequence of problems to the left detected; zero equals does not control the airplane. When a rule sequence is activated, control of the plane is given to the ArtificialPilot.

Problem				**Default Rules**
Upset	R.A	F.P	I.D	
1	1	1	1	1
1	1	1	0	1
1	1	0	1	1
1	1	0	0	1
1	0	1	1	1
1	0	1	0	0
1	0	0	1	1
1	0	0	0	0
0	1	1	1	1
0	1	1	0	1
0	1	0	1	1
0	1	0	0	1
0	0	1	1	1
0	0	1	0	0
0	0	0	1	0
0	0	0	0	0

Figure 3: (Rules Diagram)

Default Rules: With the default rules in place, the only way an unidentified pilot will be able to fly the plane is if there are no problems other than a failed identification. This means the airplane must not be in an upset condition, within a restricted area or outside of its flight path.

3.2.2.1 Algorithms

I. Check Biometrics

The CoreLogic class interfaces with biometric device(s) through objects of the Biometric class. The Biometric class is an abstract class; it serves as the base class for all biometric devices. It provides a general interface for the CoreLogic. Presently, the only biometric device is the fingerprint scanner. Therefore, the only class that inherits from the biometric class is the Fingerprint class.

CheckID is performed on every cycle of CoreLogic's main loop. The CheckID function inside the CoreLogic uses the interface of the Biometric class in order to determine the identity of the pilot. This is done of granting the pilot a window of time (currently 15 seconds) to identify his/her identity. If the time limit expires, the function will return true indicating the identification has failed. Currently, the function only returns if the fingerprint scanner has failed or not, but future implementations of the system may take into account other devices. As a safety precaution, the malfunctioning biometric devices are treated as a failed identification.

II. Check Upset

Upset conditions are checked on every cycle of CoreLogic's main loop. This function checks two possible upset situations. One is an airplane's nose. Is it too high or too low (pitch upset)? The other is a bank angle. Is it too great? To check for a pitch upset, we find the angle between the geometric plane parallel to the earth and the vector that is forward relative to the airplane.

Similarly, we can check for a bank upset by testing the angle between a geometric plane perpendicular to the earth/parallel to the airplane and the vector that is pointing straight up relative to the airplane. If either of the angles is not within its upset limits, the function returns true else it returns false.

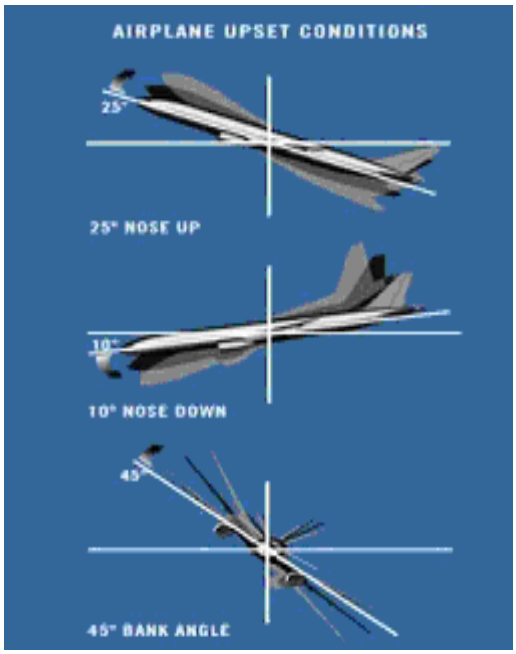


Figure 4: (Upset Conditions)

http://www.boeing.com/commercial/aeromagazine/aero_03/textonly/fo01txt.html

III. Check Restricted Area

The purpose of the RestrictedArea class is to prevent pilots from flying into certain areas. These areas may be highly populated regions, hazardous chemical/nuclear facilities or any number of areas that are deemed inappropriate for air travel.

RestrictedAreas takes the form of spherical enclosures that are usually centered on the ground at middle of the protected area. However, they may be centered under ground in order to change the slope of the sphere's surface. We also choose spheres because their symmetrical nature makes calculations extremely efficient.

RestrictedArea::TimeUntilCollision

The RestrictedArea::TimeUntilCollision function attains the estimated time it will take for the airplane to collide with a sphere (constant velocity assumed). It does this by shooting the planes velocity through the sphere and finding the intercept time(s). We check the states of the time(s) (i.e. negative/positive real or imaginary) to get the estimated time of collision (or non-collision in the case of imaginary) and test if the airplane is inside the sphere. The estimated collision time is used to warn the pilot of the time until a takeover.

CheckRA

The Core logic's CheckRA function cycles through every local Restricted Area and uses the RestrictedArea::TimeUntilCollision to get the time until collision for each Restricted Area. If a call to any RestrictedArea's TimeUntilCollision returns zero, then CheckRA returns zero indicating the airplane has entered a Restricted Area. If the airplane is not inside of any Restricted Area, a CheckRA returns the lowest time of all RestrictedArea::TimeUntilCollision. This value is used to alert the pilot of the estimated time until a takeover by the A.G.P [see section 3.2.2.1 V]. Like every other problem, the pilot will not receive a warning unless the time

until collision is below its threshold. Presently, it is set to 240 seconds for Restricted Area warnings. This time period was selected according to the pilot survey.

IV. Flight Path

The purpose of a flight path is to provide a course between two or more cities. A flight path is a list of PathPoints. PathPoints are used to form a flight path in the form of cylinders. We choose cylinders because they form a symmetrical flight path. Also, the estimated time until collision into a cylinder can be calculated efficiently. A PathPoint is a class that contains a three dimensional point and a sequential identification number. For instance, first PathPoint is numbered one. The next PathPoint is numbered two, and etc. Each flight path contains a radius that is common to all PathPoints within.

PathPoint::TimeUntilCollision

The PathPoint::TimeUntilCollision function attains the estimated time it will take for the airplane to collide with a cylinder. This is done of reorienting the position and velocity of the plane into the cylinder's local space (i.e. the current cylinder is along the Z-axis (forward)). Then we find the time it will take for the XY component of the velocity to intercept the lateral surface (side) of the cylinder and also for the Z component of velocity to intercept ends of the cylinder. Like the RestrictedArea::TimeUntilCollision function, we have to test the states of the times to get the estimated time of collision; however, in this case, we take into account the collision times of the circular ends of the cylinder.

CheckFlightPath

The CheckFlightPath checks if the airplane is within the flight path by using PathPoint::TimeUntilCollision to determine the time until a collision for each cylinder (Cylinder = (PathPoint[current], PathPoint[current +1])). If a cylinder is found to contain the airplane, its time until collision is returned. Otherwise, the airplane is not within any cylinder of the flight path. Then, a time of zero is returned. For efficiency, this function is called every second. Like the CheckRA, the pilot will not receive a warning unless the time until collision is below its threshold that is presently set to 120 seconds.

V. Messages to Pilot

The pilot receives messages of conditions that might cause the airplane to be taken over by the Aerial Guardian. This is done of looking at the rules array. For example, if the current situation is such that the problems that may lead to a takeover are currently (1 0 1 0) [figure 3]. Biometrics and Flight path messages will be added to the message list if this combination will cause a takeover. This information allows the pilot to make an informed decision and avoid problems. Most commercial airplanes have GPS (Global Positioning System) displays that inform the pilot of distances to areas. Future implementation of the system may take advantage of the GPS display or another pilot displays. A picture of the display can be seen on the bottom left section of the planes cockpit in [figure 8, figure 9].

3.2.3 Artificial Pilot

The Artificial Pilot is responsible for physically controlling the plane by manipulating the controls. Two of its main activities are to follow specified points on the flight path and to correct the plane if it is in an upset condition. The Artificial Pilot is instructed by the CoreLogic class.

Artificial Pilot Algorithms

I. ArtificialPilot::HandleProblem

HandleProblem acts as an interface to the ArtificialPilot class. The function receives a set of problems then outputs the control inputs that will correct the problem. It decides what problem to handle by prioritizing each problem. On each call of the HandleProblem function the problem (i.e. upset, restricted area, off flight path) is detected and has the highest priority be serviced; all other problems are ignored.

HandleProblem gives the highest priority to CorrectForUpset because this has to be fixed before the plane can correct any other problems. This is due to the fact that an airplane will not fly correctly if it is in an upset condition. The second highest priority is given to AvoidingRestrictedArea. The lowest priority is given to FollowFlightPath. FollowFlightPath is an exception because it does not have to be activated (i.e. the pilot does not have to be outside the flight path) for it to be triggered into action. This is due to the fact that HandleProblem is called when a rule set [figure 3] has been triggered. Therefore, if a rule set says to take over the airplane, the ArtificailPilot must still do it even if there are no perceivable problems. In this case, FollowFlightPath will take control of the airplane and stay within the flight path.

II. ArtificialPilot::CorrectForUpset

CorrectForUpset corrects the airplane if it is banked and/or pitched too much. When CorrectForUpset is called it uses the planes orientation vectors to generate a rotation matrix. However, yaw is ignored since no amount of yaw is considered an upset condition. The airplane's axes are rotated around the worlds up axis until its forward vector lies on the worlds YZ plane. The matrix is then generated, and it is by placing the three vectors Up, Right and Forward in to the columns of a 3 by 3 matrix. The world axis is then rotated by the inverse of that matrix; this has the effect of generating the opposite rotation, which is then passed to the operation function to be converted in to control outputs (flight stick movements).

III. ArtificialPilot::AvoidRestrictedArea

Avoid restricted area is called when the plane enters a restricted area. It corrects the situation by guiding the plane outside of the restricted area. This is done by checking the angle between the airplane's right vector and the vector pointing from the center of the restricted area to the airplane. Before we find the angle, each vector's y (up) components is eliminated. If the angle is less than 90 degrees, the plane turns right else it turns left.

IV. ArtificialPilot::FollowFlightPath

Following Flight path alters the planes orientation so that the airplane is pointing at the next waypoint in the flight path. This is done of subtracting the planes position from the next waypoint's position. This creates a vector pointing from the plane to the next waypoint. This vector is then passed to the ArtificialPilot::Operation function.

V. Controlling the aircraft

ActionTree

The ActionTree is a structure that is responsible for storing information that indirectly determines how the airplane should be controlled. As the name suggests, the ActionTree has the structure of a tree. Each node contains a PilotInput object (input to control stick, throttle, etc) and a Response object (how did the plane respond to the given PilotInput).

The tree is organized based on the different elements of the Response. Similar changes are grouped together; for example, all responses where the largest change is speed are grouped together onto a branch and all responses where the largest change is orientation are grouped onto another branch. Each node in a particular branch is similar to the parent node of that branch; if the parent node's response has its largest change as altitude then every node under it will also have its largest change as altitude. Primary level nodes are sorted based on the largest change; however, in the secondary level nodes are sorted based on the secondary change and so on.

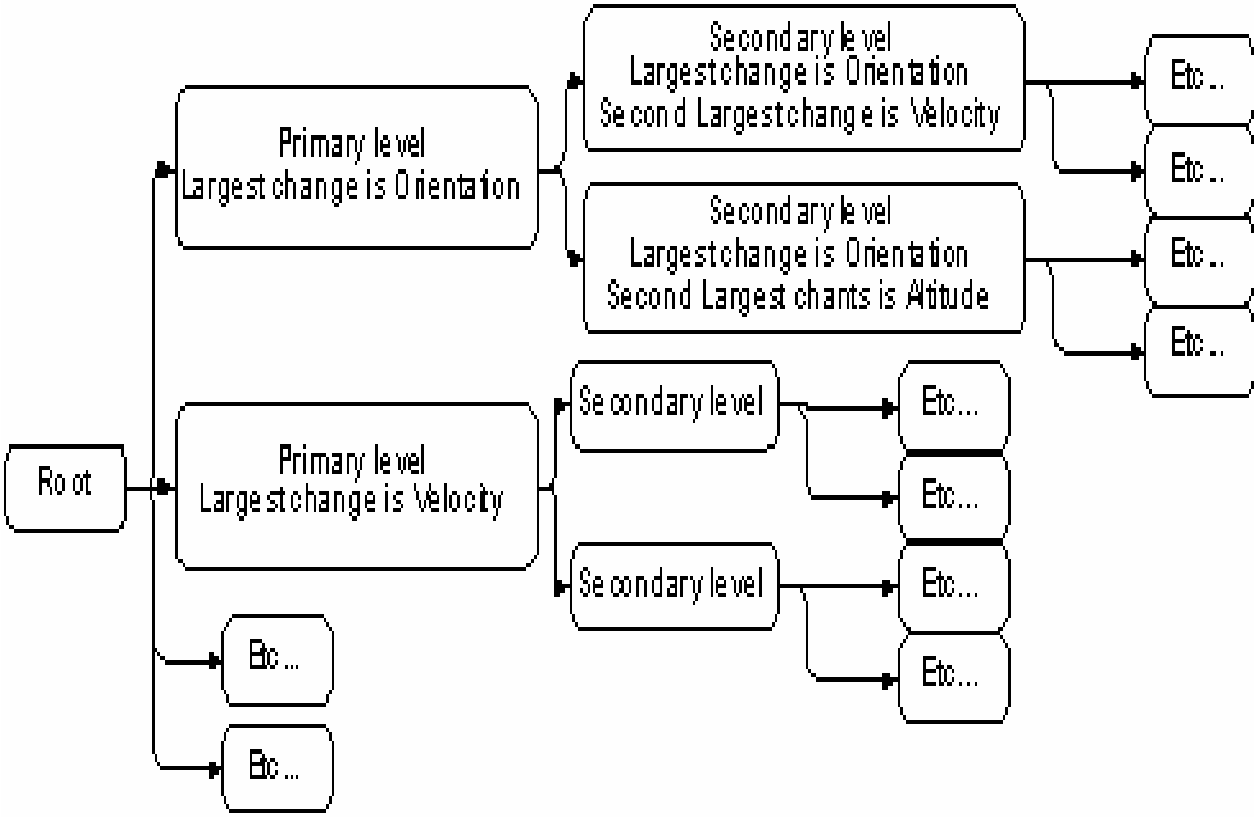


Figure 5: (Action Tree)

ArtificialPilot::Operation

The Operation is the most important function in the ArtificialPilot. It is responsible for manipulating the controls that will maneuver the aircraft. It works by searching the ActionTree for the desired response that was passed to it from the other functions (ex: ArtificialPilot::FollowFlightPath, ArtificialPilot::AvoidRestrictedArea). The ActionTree contains numerous airplane responses and the control inputs that cause them. These responses are arranged in order to facilitate a heuristic search.

When operation is called, it first analyses the values that it receives. It receives a response object, which contains the desired responses, and a Boolean mask, which indicates what elements in the response object are most important. The function then rearranges the responses in order of importance so that the most important changes are searched first. Responses with a mask value of 1 are moves to the front of the array and ones with a mask value of 0 are moved to the back. Furthermore, the responses are arranged within the 1 and 0 set, this is done according to the amount of change, larger changes are placed first. When completely sorted, the responses are prioritized so that the most important responses are first, followed by the smaller ones we care about then followed by the ones we don't care about.

The ActionTree is searched starting from the main branch in order to find a node that matches the desired response. If the response's mask (ex: change in the up vector, change in the forward vector and change in speed) is 1, the ArtificialPilot::Operation finds the best match by comparing the change in the desired response to the change in the response of nodes in the main branch of the Action Tree. However, if the response has a mask of zero, it is not compared to data in the ActionTree because 0 in a mask indicates a response that is not important.

Once a node is found, the ArtificialPilot::Operation searches the sub-branch of the previously found node for the next most important response. ArtificialPilot::Operation continues to search sub branches until it gets to the last most important response, which should also be a node on the final branch.

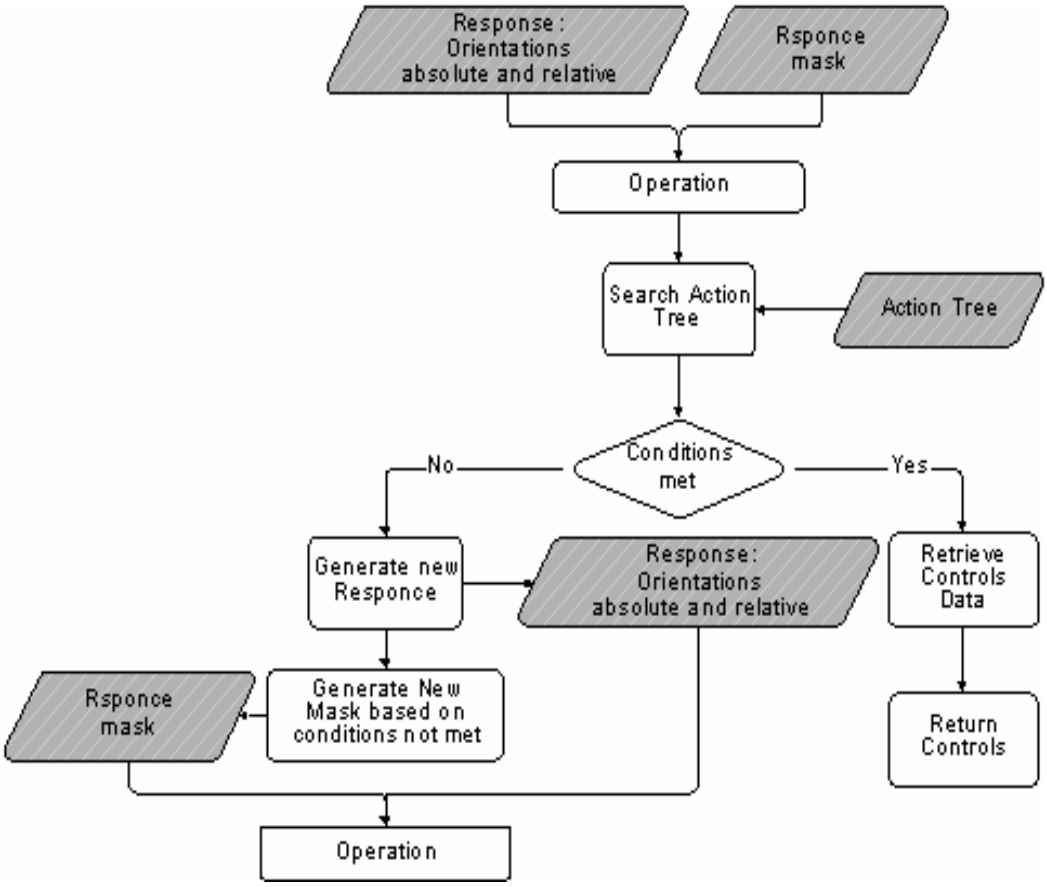


Figure 6: (Operation - Flowchart)

Once a response item is found, it is then compared to the current plane conditions. All responses with a mask of 0 are now treated as conditions. Because these conditions can change the way the plane responds to the controls, we want them to match the current plane conditions as best as possible. If they match, the control inputs stored in that particular tree node are used. If they are different, we need to find an action that will make them the same. This is done by creating a new mask that has 1's for the elements that are different from the ActionTree, and a new response object that has the differences between the current plane condition and the desired conditions that where initially found in the ActionTree. To find the node that matches this new response, a recursive call to operation is made using the new response and mask. When the conditions are satisfied, ArtificialPilot::Operation will be able to perform the desired task.

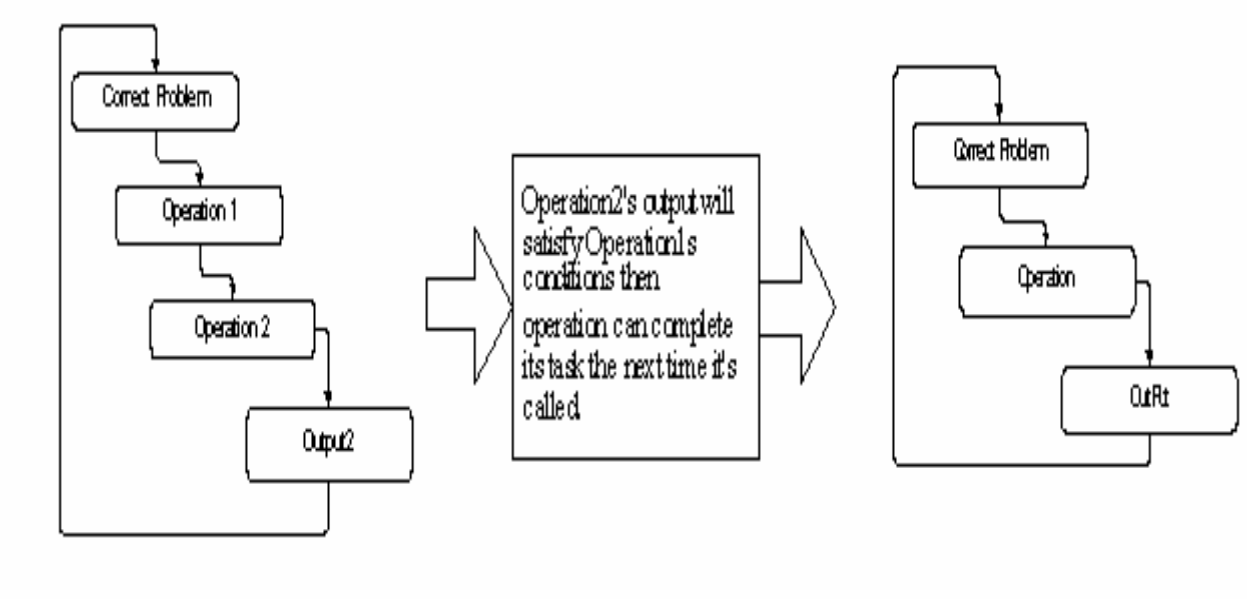


Figure 7: (Operation – State Transition)

3.2.4 Optimizations Used - Aerial Guardian Protector

Checking if the airplane is inside of each restricted area is a very lengthy process. This is why we choose to keep a local list RestrictedAreas. The local RestrictedArea information is updated every ten seconds with new data. This local data is updated with all the RestrictedAreas that are within the Distance of (DesiredRange) * (MaxVelocity) * (LargestTimeTillCollision). In our case, this is 5*MaxVelocity*240s. This means at any given time the A.G can check if the airplane will intercept a Restricted Area or that is four to five times what that the airplane can fly in 120 seconds.

Another optimization is that certain problems are test periodically as opposed to every cycle of the MainLoop. Since the checking of restricted areas and flight paths are costly processes, this action dramatically increases execution speed.

3.2.5 Design and Implementation Tradeoffs - Aerial Guardian Protector

The modularity of the system made it difficult to design. The flow of data had to be logical yet efficient. A large portion of the planning stage was spent considering the impact of various organization schemes. However, the modularity did decrease the time needed to alter the system.

3.3 Aerial Guardian Trainer (A.G.T)

In the following section we will give a detailed description of the design and implementation of the A.G.T. The A.G.T is responsible for realistically simulating all functionality of an airplane.

3.3.1 Operating System - Windows

The Windows XP operating system is an operating system developed by Microsoft. It uses an event driven (i.e. user clicks mouse or types key) GUI (Graphical User Interface) in order to send commands to various windows. It is usually programmed using the win32 API (Applications Programming Interface) and/or MFC (Microsoft Foundation Classes). We choose Windows because we were familiar with the development of win32 applications and it supports the Direct X API.

3.3.2 Graphics API Comparison - OpenGL vs. DirectX

We decided to develop the A.G.T with DirectX as opposed to OpenGL because DirectX is a feature rich 3D API. DirectX 9.0 comes with a variety of features like DirectInput that enables developers use a generic interface to get information from a variety of input devices like joysticks, flight yokes/Throttle. In addition, Microsoft constantly updates DirectX's graphical device drivers. Therefore, DirectX 9.0 is very likely to make full use of the capabilities of the graphics card, thus increasing frame rates.

On the other hand, OpenGL has the benefit that it is very portable. It is also more established 3D API than DirectX. OpenGL can also generate great frame rates; however, our lack of experience with this API and the fact that DirectX includes input support (joystick, pads, etc) made it DirectX the obvious choice.

3.3.3 Aerial Guardian Trainer Design/Implementation

I. AppWizard and Environment

The A.G.T is responsible for the simulation of an airplane. It serves as a training tool for pilots in the use of the system, and a demonstration too. The A.G.T is organized with an Object Oriented Approach. To create the basic layout of our project, we used the DirectX 9.0 VC++ Wizard, which is the starting point for many DirectX applications. We then modified the generated code by adding the Environment and its subclasses to the wizard's CMyD3DApplication class.

The Environment class is responsible for coordinating the efforts of all the non-template created classes. We used this method to implement the A.G.T because it allowed our code to be as independent as possible. Nevertheless, certain facilities of the A.G.T required us to modify the Wizard's code.

The Environment contains the Terrain class, which renders the surrounding terrain. The terrain contains the City class and Airport class that is responsible for rendering cites and airports. Initially, the City and Airport classes rendered themselves by translating the different sections (blocks) of the City or airport then moving them onto the terrain at run time. This proved timely; therefore, we decided to trade video card memory for rendering speed by moving the vertices of the terrain and city to the video card.

II. Plane

The Environment also contains the Plane class, which simulates an airplane. This class renders the airplane's cockpit, which was created in 3DStudio. 3DStudio is a 3D modeling and animation program. It can be used to create a mesh (i.e. a group of triangles that form a 3D object) that can be exported to DirectX 9.0 via an X-file. An X-file places the mesh data in a format that DirectX can understand. The final responsibility of the Plane object is to control the camera's view and transform/orient the airplane's cockpit to its position in 3D space.

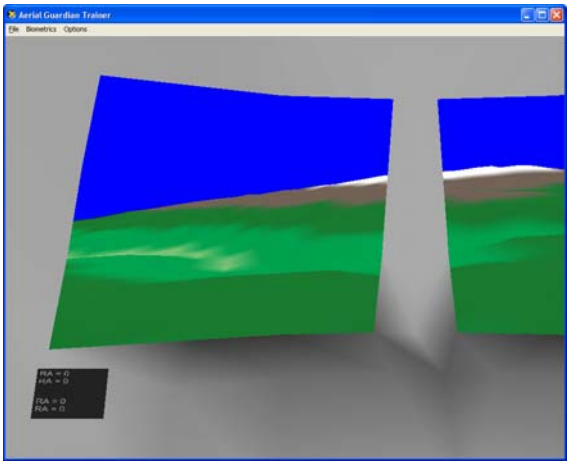


Figure 8: A.G.T (view of terrain through cockpit)

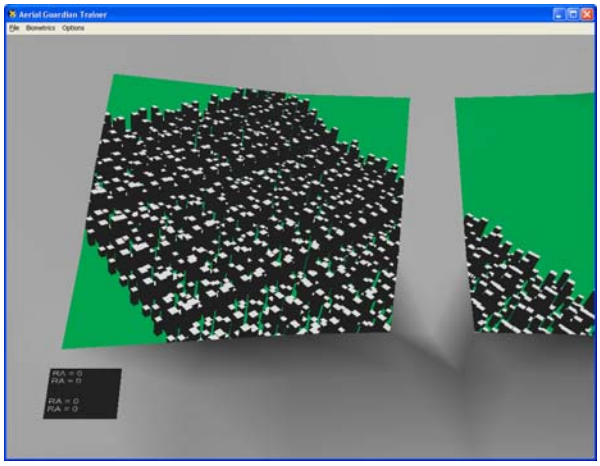


Figure 9: A.G.T (view of city through cockpit)

DlgPlane

The Plane class also contains the DlgPlane class that is mainly responsible for handling windows messages to the (Modify Plane Characteristics) dialog box. This class is also responsible for managing a database of various airplane models.

Each airplane model has various characteristics that allow the Physics class [section 3.3.3] to realistically simulate the airplane. Airplane models may be modified with the use of the View/Modify Characteristics section of the dialog box.

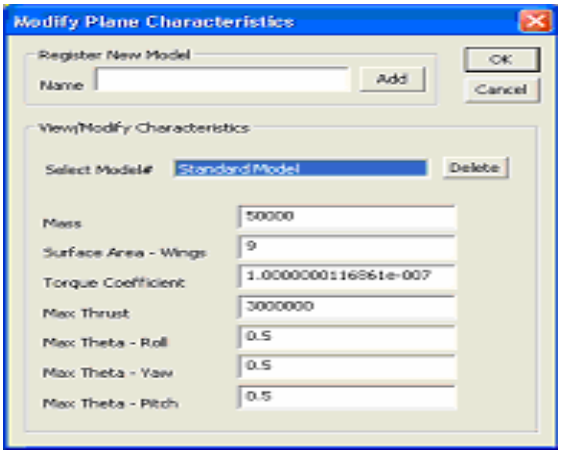


Figure 10: A.G.T (Modify Airplane Characteristics)

IOFIU

Like the Environment class, the Plane class contains a variety of objects. One object is the IOFIU object, which is responsible for the communicating with the fingerprint scanner. It provides an interface that verifies and registers fingerprints. This object also holds a list of all users that are registered on the system.

Figure 11: (Fingerprint scanner and Throttle)

DlgFIU

The DlgFIU class is responsible for handling windows messages directed to the (Modify FingerPrint Database) dialog box. The DlgFIU uses the IOFIU object to register, erase and modify users and their fingerprints in the Sony FIU 600 and PC database. Details on Sony FIU 600 can be found in section 2.5. Each user is able to register multiple fingerprints.

The fingerprint scanner's database is limited by its internal memory. It can store up to 1000 fingerprints; each fingerprint has 40 bytes allocated for additional user information. This additional space is used to store the names of the user that belong to the corresponding fingerprint.

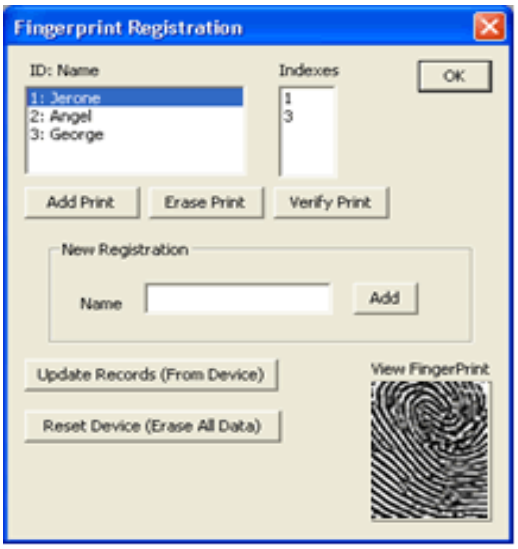


Figure 12: A.G.T (Fingerprint Registration)

ScanContinouslyFIU

The ScanContinouslyFIU class is responsible for scanning the pilot's fingerprint while the A.G.T is running. To accomplish this, an object of the ScanContinouslyFIU class runs on a worker thread. The object could be in one of the two states. State one being paused (i.e. waiting on a signal from the A.G to begin scanning). The second state is the active state in which it uses the IOFIU object to verify if the fingerprint that is currently on the unit matches a pre registered list of fingerprints. Each attempt to verify if a print matches the list takes approximately half a second. Since the Sony FIU 600 has built in fingerprint verification, this process does not hinder the A.G.T because the worker thread spends most of its time waiting on an input from the fingerprint scanner. On completion of the scan, the results of the match are sent to the A.G.

InterfaceToAGP

The Plane class also contains the InterfaceToAGP class. The InterfaceToAGP class transforms the instrumentation information into data that mimics a real airplane's instrumentations. It then sends this information to the A.G.P with the aid of the Communication

class [section 3.4]. The Communication class is also used to receive updated user input information from the A.G.

III. Physics

The Environment class contains the Physics class, which determines how an airplane responds in various situations. This class uses various physics formulas in order to model the reaction of an airplane to the surrounding environment. These formulas make use of many of the airplane's characteristics which include mass, lifting surface area and torque.

We are currently in the process of attaining modules that utilize aerospace aircraft equations of motion. These modules will be provided by NASA LARC. The models are based on actual wind tunnel data; NASA uses this data to test many of their aircrafts. The modules will require alteration to be compatible with our code, but once installed they will improve the realism of the Aerial Guardian Trainer.

3.3.4 Optimizations

As mentioned before we traded system memory (ram) and processing time for video card memory. This was done in order to improve the speed of the simulation. This action increased frame rate from 30 fps to 100 fps on a P4 2.4GHz, Gee force 2 (resolution 1280*1024).

3.3.5 Design and Implementation Tradeoffs:

Like the A.G.P, we design the A.G.T to be modular. The A.G.T incurred the same benefits and disadvantages of modularity as the A.G.P. However, we believe the benefits far out weigh the disadvantages.

3.4 Communication Design/Implementation

The Communication class is responsible for transmitting information between the A.G.P and the A.G.T or airplane. It is designed to be portable because it operates on two different operating systems. The class makes use of the parallel port in order to transmit information.

Since the parallel port inputs 5 bits at a time, we split the data into nibbles in order to facilitate data inputs. Due to the design of the parallel port, the inputs are received in the upper five bits of the status register. Therefore, the data received must be shifted three spaces to the right in order for the system to read correct information. The fifth bit of the status port (S4) is used to determine if the data on the pins is new information. The remaining lower four bits (nibble) are utilized for data.

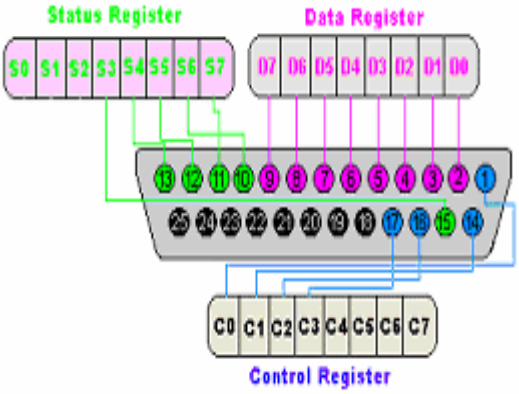


Figure 13: (Parallel Port - Pins)

Stop and Wait:

Stop and Wait is used for the communication between systems. This flow control is efficient for the reason that the packets are not relatively large. In addition, it serves the purpose of transferring data in an efficient and controlled way. The communication class serves as the foundation of the CommunicationManger class.

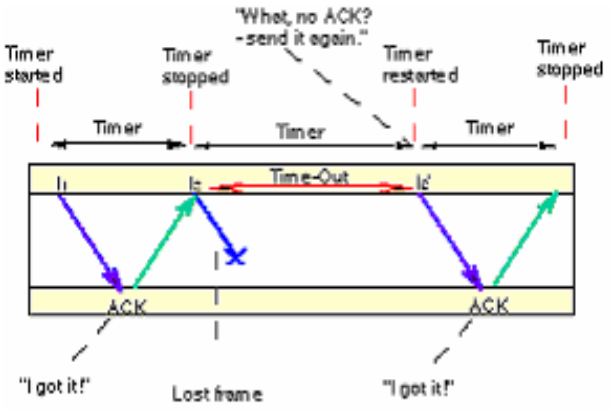


Figure 14: (Stop and Wait)

The main responsibilities of the CommunicationManger class are packaging/unpackaging of data. It utilizes the Communication class in order to transmit/receive information between the A.G.P and airplane or A.G.T. Threads are employed in order to allow the CommunicationManger and the A.G.P or A.G.T to run simultaneously. Due to the fact that the information is accessed by the other classes (i.e. A.G.P-CoreLogic, A.G.T-InterfaceToAI), mutex must be used in order to prevent deadlocks and accessing data currently used by another process. A mutex is a special variable that can be either in the locked or the unlocked state. If the mutex is locked, it has a distinguished thread that holds or owns the mutex. If no thread holds the mutex, we say the mutex is unlocked, free or available. In short, a mutex is meant to be held for short periods of time, and it is the simplest and most efficient thread synchronization mechanism.

The CommunicationManger transmits and receives information as packets that composed of two fields. The first is being the action field that specifies nature of the data (e.g. throttle, altitude, activate scan, etc). The second field is not mandatory; it will only accompany the first field if information is being sent. For example, if the first field is activate scanner, it will not be necessary to have a second field because activate scanner requires no additional information. We used a scheme such as this to increase the speed of information transfer.

The CommunicationManger and Communication classes are the only plane specific classes in the A.G.P. This was unavoidable because almost every model of airplane transfers information with a different protocol. For this reason, we created these classes to be very flexible, so that future changes will be effortless.

3.5 Testing

3.5.1 Testing – Aerial Guardian Protector

The Operation function of the ArtificialPilot was tested by tracking the time and number of oscillations it took for the plane to stabilize after a procedure (i.e. procedure = orient the plane to face a point). This was done by testing how long and how many oscillations it took for the plane to stabilize itself. The threshold or stabilization time is two seconds. This time starts when the airplane is at the desired orientation. The results were as expected. The more detailed the ActionTree the better the airplane oriented and stabilized itself.

The Pure Fuzzy Logic Based Autopilot (P.F.L.B.A) [section 3.6] was used in order to test the A.G.P system. The P.F.L.B.A was run from the A.G.T where it acted as the pilot. The

P.F.L.B.A was given the same flight path as the A.G.P. Then, P.F.L.B.A would veer off the flight path or enter a restricted area at random time intervals. Therefore, we saw if the system operated as expected and how long it took to correct situations. From these test, we were able to see what situations the system performed well and vise versa.

We also test the system's ability to compensate for errors in the input. The final test we must implement on the Aerial Guardian is its ability to operate for a prolonged period of time. This means the system must run without errors for over 150 hours or one week.

3.5.2 Testing – Aerial Guardian Trainer

The tests performed on the A.G.T were general tests. We tested the false acceptance/rejection rates of the FIU-600 fingerprint scanner. We needed false acceptance rate of at less than .5% and false rejection rates at less than 10%, because when scanning the pilot may scan his/her print for a period of 5 seconds. Therefore, he/she may attempt to a total of 10 times (.5 seconds per scan). The odds of the pilot being falsely identified all 10 times is less than $10^{-5\%}$. We scanned 400 prints their where no false acceptances and the false rejection rate ranged from 3%-8% depending on the individual. Finally, we tested the speed of the Communications class, which had to be 9600 bps or more. The class consistently produced transfer rates of 40kbps and more.

3.6 Tools Implemented - Pure Fuzzy Logic Based Autopilot

About Fuzzy Logic

Fuzzy logic is a superset of conventional Boolean logic that has been extended to handle values between completely true and completely false. Our implementation of fuzzy logic uses a variety of classes. At the lowest level of fuzzy classes is the FuzzySet group of classes.
FuzzySet(Low,Middle,High)

The FuzzySetLow, FuzzySetMiddle, FuzzySetHigh represent low, middle and high fuzzy sets. When given a non-fuzzy value the set returns a fuzzified value from zero to one that represents the degree of membership. In the picture to the right, there are 3 set, namely, Low, Middle and High.

The reason for three fuzzy sets class is that it allows the MemberFunction classes to create multiple sets in the middle of the function. For example, the diagram on the right shows one middle set, but additional middle sets can be added.

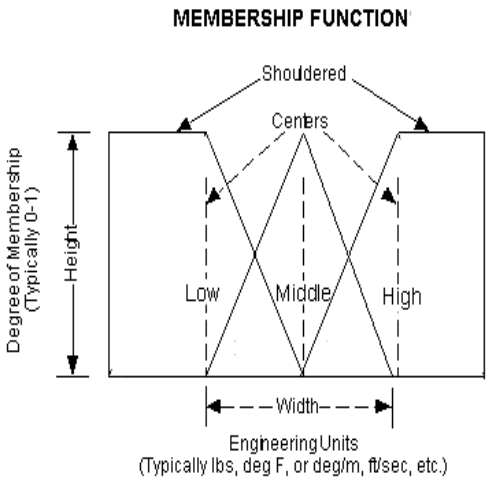


Figure 15: (Fuzzy Set- MemberShip Function)

http://www.seattlerobotics.org/encoder/mar98/fuz/fl_part4.html

FuzzyLogic (class)

The reason that this class was created is to provide a medium to test the A.G.P in its initial developmental stage. This class was used to create a fuzzy logic based autopilot system. This class was inspired by modern fuzzy logic control theory. The system is received the desired change in orientation with respect to the airplane. The FuzzyLogic then uses this in order to orientation the airplane to face the desired orientation (in the form of a vector).

The FuzzyLogic does this by using the FuzzyMemberFunction class to map the input (which is the angular change) of two fuzzy universe of discourse. The first universe of discourse is responsible for orienting the airplane to the desired bank. This is done by mapping the desired angular change in the bank to a specific output to the ailerons (airplane control surface). The second universe of discourse is responsible for orienting the airplane to the desired pitch. This is done by using a fuzzy membership function to map the desired change in angular pitch to the output of the elevators (airplane control surface).

There are two other FuzzyMemberFunctions. They take into account the angular velocity of the airplane, and they are used in order to quickly and gently stabilize the airplane onto the desired vector. Without the use of this additional step, the airplane would become unstable as it came closer to the desired input orientation. The down side to the current implementation of this system is that it is only reliable within normal orientation conditions, due to the fact that it is unable to adapt to new situations.

3.7 Cost

Category/Item	Cost
A.G.T	
1. Fingerprint Scanner	$105.00
2. Joystick/Throttle	$80.00
3. Carbon Fiber with Epoxy	$20.00
Communication	
1. Parallel Cable	$15.00
Total Price	**$220.00**
Cost not included in competition requirement **(computers does not count)**	
MicroATX based system (size 6”x 6”)	$130.00

4. Summary

4.1 Purpose of the Aerial Guardian

The inspiration for the A.G came from the need for improved commercial airplane security. The A.G.P accomplishes this by utilizing several safety features. These include biometric technologies in order to identify the current pilot, the ability to correct or prevent certain upset conditions, the ability to restrict an airplane from flying into a certain areas. Finally, the system adds an extra dimension of safety by restricting the airplane to a flight path. If the default rules are set, an un-identified pilot is not allowed to fly outside of the flight path. Lastly, Aerial Guardian may not solve all the safety problems in the air; however, it is better to have an additional safety system in the commercial airplanes.

The A.G.T system was created in order to simulate the functionality of an airplane. The

A.G.T also provides a way to train pilots, and serves as a demonstrational tool for prospective buyers. The A.G.T simulates the functionality of an airplane, and it transmits information to the A.G.P in the form that a real airplane would, which means error is placed into instrumentation data.

4.2.1 Design

The A.G was design from a functional perspective. The system has two main modules, namely, the CoreLogic and Artificial Pilot. The CoreLogic is responsible for detecting problems and warns the pilot of impending danger by way of messages. The Artificial Pilot is responsible for the control of the plane. It uses artificial intelligence techniques in order to provide a smooth output to the airplane.

The A.G.T system was designed with an object-oriented approach. The system is enclosed within codes created by the DirectX Wizard. The environment is responsible for coordinating all the other classes within it. These classes are responsible for rendering various objects and creating a realistic simulator. The A.G.T communicates with the A.G.P via the Communication class.

4.2.2 Cost

Considering that the cost of a commercial airliner is in the millions, we believe the price of around $350 (depending on the current equipment on the airplane) per unit is extremely cost effective. The installation cost of the system will vary between systems. However, we believe the Aerial Guardian has an ability to improve the safety of commercial airplanes and will greatly compensate for additional cost.

4.3 Testing

Our system has under gone considerable testing; however, almost no non-trivial software system is error free. Therefore, there are sure to be flaws in the design and implementation of this system. Due to the nature of this system, it must undergo a variety of additional test to make sure it is safe. Like every other commercial flight system, it will have to undergo extensive simulator, and in flight-testing to be considered for use in commercial airplanes. However, special care must be taken when testing of this system because it has the ability to take control of an airplane, with no override available.

4.4 Future Implementations – Aerial Guardian II

1. Allow pilot input to bypass the system in case of malfunction in the Aerial Guardian.
2. Backup for GPS data (possible inertial navigation)
3. Blending of pilot/A.G. inputs to give pilot more control during take over situations.
4. Inform the control tower of tampering with the A.G.P
5. In case of pilot and copilots for any reason are unable to perform their duties, allow passage pilot on board to assist and flying the airplane.

5. References

5.1 Books

[1] Cooling, Jim. "Software Engineering for Real-Time Systems," Addison Wesley, 1st Edition, 2002.
[2] Jones, Tim. "A.I. Applications Programming," Charles River Media, 1st Edition, 2003.
[3] Yen, John and Langari, Reza. "Fuzzy Logic Intelligence, Control, and Information," Prentice Hall, 1999.
[4] Horton, Ivor. "Beginning C++: The Complete Language," Wrox Press, 2nd Edition, 1998.
[5] Josuttis, Nicolai M. "The C++ Standard Library: A tutorial and Reference," Addison Wesley, 1st Edition, 19999.
[6] Petzold, Charles. "Programming Windows," Microsoft Press, 5th Edition, 1998.
[7] Richter, Jeffrey. "Programming Applications for Microsoft Windows," Microsoft Press, 4th Edition, 1999.
[8] Luna, Frank D. "Introduction to 3D Game Programming with DirectX 9.0," Wordware Publishing Inc., 1st Edition, 2003.
[9] Engel, Wolfgang. "Beginning Direct3D Game Programming Second Edition," Muska & Lipman/Premier – Trade, 1st Edition, 2003.
[10] Mazid, Muhammed A. and Mazidi, Janice G. "The 80x86 IBM PC and Compatible Computers: Assembly Language, Design and Interfacing (Vol. I & II)," Prentice Hall, 3rd Edition, 2000.
[11] Axelson, Jan. "Parallel Port Complete: Programming, Interfacing & Using the PC's Parallel Printer Port," Lakeview Research, 1997.
[12] Bourg, David M. "Physics for Game Developers," O'Reilly Media, 1st Edition, 2001.
[13] Giancoli, Douglas C. "Physics for Scientist and Engineers," Prentice Hall, 3rd Edition, 2000.
[14] Collinson, R.P.G. "Introduction to Avionics," Institute of Electrical and Electronics Engineering, 1998.

5.2 People

[15] Michael Jacobson - Pilot - Randolph, NJ

5.3 Electronic Media/Internet

[16] Sony FIU-300/600 SDK documentation - Sony Corp
[17] www.msdn.com – Microsoft Corp
[18] www.seattlerobotics.org/encoder/mar98/fuz/flindex.html - Steven D. Kaehler
[19] www.dedicated-systems.com/encyc/buyersguide/rtos/Evaluations/doc.pdf - RTOS valuations
[20] http://courses.ece.uiuc.edu/ece291/resources/parallel.html - Parallel port
[21] www.lynuxworks.com - LynxOS Developer