İSTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

A FLIGHT SIMULATION TOOL FOR END-GAME SIMULATIONS

M.Sc. Thesis by
Özer ÖZAYDIN

Department : Advanced Technologies

Programme : Computer Science

JUNE 2009

İSTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

A FLIGHT SIMULATION TOOL FOR END-GAME SIMULATIONS

M.Sc. Thesis by
Özer ÖZAYDIN
(704051012)

Date of submission : **04 May 2009**
Date of defence examination: **05 June 2009**

Supervisor (Chairman) : **Assist. Prof. Dr. D. Turgay ALTILAR (ITU)**
Members of the Examining Committee : **Prof. Dr. İbrahim EKSİN (ITU)**
Assist. Prof. Dr. Mustafa E. KAMAŞAK (ITU)

JUNE 2009

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

SON AN BENZETİMLERİ İÇİN BİR UÇUŞ BENZETİM ARACI

YÜKSEK LİSANS TEZİ
Özer ÖZAYDIN
(704051012)

Tezin Enstitüye Verildiği Tarih : **04 Mayıs 2009**
Tezin Savunulduğu Tarih : **05 Haziran 2009**

Tez Danışmanı : **Yrd. Doç. Dr. D. Turgay ALTILAR (İTÜ)**
Diğer Jüri Üyeleri : **Prof. Dr. İbrahim EKSİN (İTÜ)**
Yrd. Doç. Dr. Mustafa E. KAMAŞAK (İTÜ)

HAZİRAN 2009

FOREWORD

I would like to extend my sincere appreciation to my advisor for his supervision, guidance, friendship and his encouragement in my study.

I would like to thank to my family for their kind support not only in my thesis, but in all my life.

I would also like to thank to my managers in Nortel Netas who always supported me and who were always tolerant.

I would especially thank to F. Rana Ceylandağ for her patience and lovely support.

May 2009

Özer Özaydın
Computer Science Department

TABLE OF CONTENTS

Page

ABBREVIATIONS

3D	**:** Three Dimensional
SDL	**:** Simple Direct Media Layer
OIS	**:** Object Oriented Input System
FDM	**:** Flight Dynamics Model
DoF	**:** Degree of Freedom
LGPL	**:** Lesser General Public License
OpenGL	**:** Open Graphics Library
FPS	**:** Frames Per Second
XML	**:** eXtensible Markup Language
Kts	**:** Knots
PNG	**:** Portable Network Graphics
HUD	**:** Head Up Display

LIST OF TABLES

Page

LIST OF FIGURES

Page

A FLIGHT SIMULATION TOOL FOR END-GAME SIMULATIONS

SUMMARY

In this thesis, a new discrete flight simulation framework, SahinSim is introduced and implemented. SahinSim is a part of an on going academic project on proportional navigation guided missiles and aircrafts' practical evasive manoeuvres against these missiles. In this project, a 3D proportional guided missile was designed and the behaviour of the missile was investigated when the missile was fired to a target at different angles and distances. On the other hand, an F16 model was designed as the target and the F16 tried to escape from the missile by performing different evasive manoeuvres. The performance of the different manoeuvres was investigated. An end-game simulation application, VEGAS was implemented and used as the simulation environment. SahinSim is a new simulation framework which is designed for a similar purpose to VEGAS.

SahinSim provides an easy to use and flexible 3D visual simulation environment, as well as an interface to an accurate flight dynamics model to this project. Open source projects JSBSim, SimGear, OGRE, SDL and OIS are used within SahinSim. Although SahinSim is intended to be used for end-game simulations such as air-to-air combat scenarios, it can be extended to be used for also other aerospace related issues.

This thesis deals with the implementation of the simulation environment and the components used in the project. The graphics engine OGRE is investigated and introduced. The flight dynamics model JSBSim is investigated, and its integration to SahinSim is explained. The overall architecture of the simulation environment and the integration of the open source projects are explained in details.

SON AN BENZETİMLERİ İÇİN BİR UÇUŞ BENZETİM ARACI

ÖZET

Bu tezde yeni bir ayrık uçuş simülasyonu olan SahinSim tanıtılmış ve gerçeklenmiştir. SahinSim, orantısal güdümlü füzeler ve bu füzelerden kaçmak için değişik manevraların incelendiği bir projenin parçasıdır. Proje kapsamında üç boyutlu orantısal güdümle hareket eden bir füze tasarlanmış ve değişik açı ve uzaklık değerlerinde bir hedefe doğru ateşlendiğinde füzenin davranışları incelenmiştir. Diğer yandan hedef olarak F16 modelinde bir uçak tasarlanmış ve uçak değişik manevralar yaparak füzeden kaçmaya çalışmıştır. Farklı manevra türlerinin füze karşısındaki başarısı incelenmiştir. Bu proje için VEGAS isimli bir simülasyon programı kullanılmıştır. SahinSim VEGAS yazılımıyla benzer amaca yönelik yazılmış yeni bir yazılımdır.

SahinSim bu projeye kolay kullanılır ve değiştirilebilir bir üç boyutlu simülasyon ortamıyla birlikte hassas bir uçuş dinamiği modeline arayüz sağlar. Projede açık kaynaklı olan JSBSim, SimGear, OGRE, SDL ve OIS projelerinden yararlanılmıştır. SahinSim esasında havadan havaya çarpışma senaryolarının incelendiği son an simülasyonları için tasarlanmış bir uygulamadır; ancak esnek tasarımı sayesinde SahinSim diğer havacılık konularıyla ilgili kullanılmak üzere kolaylıkla genişletilebilir.

Bu tez simülasyon ortamının gerçeklenmesi ve tezde kullanılan açık kaynak kodlu projelerle ilgilidir. Tez kapsamında grafik motoru olarak kullanılan OGRE incelenmiş ve tanıtılmıştır. Uçuş dinamiğini sağlayan JSBSim incelenmiş, proje içerisinde nasıl kullanıldığı açıklanmıştır. Projenin mimarisi ve kullanılan açık kaynak kodlu yazılımların SahinSime entegrasyonu detaylı olarak anlatılmıştır.

1. INTRODUCTION

Defense issues have always been a driving force for science since it has always been one of the important concepts for countries. Depending on their strategy, countries may have very high expenses on their defense forces. Air defense is one of the key areas in homeland security. Eventually missiles and aircrafts are inevitable parts of air defense strategies. The missiles and aircrafts are produced at high costs, which make it working on real missiles and aircrafts very hard while designing them. On the other hand, in most cases human life is risked during the design and test phase, which is more important than money. This is where computer simulation takes place; while designing air defense weapons such as missiles and fighter aircrafts, computer simulations are used to risk less lives and to design at low costs.

1.1 Purpose of The Thesis

This thesis is a part of an on going academic project on proportional navigation guided missiles and aircrafts' practical evasive maneuvers against these missiles. The main purpose of the thesis is to provide a flight simulation environment that includes an input system, a visual interface and a flight dynamics model.

1.2 The Approach

A discrete flight simulation framework is designed for use in research or academic environments. Aerospace researchers can benefit from SahinSim while studying aerospace studies such as aircraft models, tracking algorithms, auto pilot-control applications. SahinSim provides users an accurate flight dynamics model, a fairly nice 3D graphical user interface and an easy to use input interface. Without dealing with lots of programming code, a user can concentrate on his own research topic while saving time and effort.

The implementation is very straightforward which makes it easy to extend the simulation regarding particular requirements. While designing the simulation environment, popular open source projects are used to strengthen the simulation.

In the next section, related work and motivation are explained. The third section has details about the simulation environment that SahinSim provides. The fourth section gives information about the implementation details of the application and the last section concludes the thesis.

2. BACKGROUND AND MOTIVATION

SahinSim has been developed as the second generation visual end-game simulator to another on going project that investigates evasive manoeuvres of an aircraft against proportional navigation missiles. The first generation simulator, named VEGAS (Visual End-Game Simulation) was implemented as a complementary work to visualize the end-game. Akdag and Altilar worked on modelling an agile aircraft capable of moving high-g manoeuvres and performing different evasive manoeuvres [1]. Moran and Altilar implemented a missile model using proportional navigation techniques to track previously implemented aircraft model [2]. They used 3-DoF flight dynamics equations for both missile and aircraft models. The models were embedded in source code so that after changing any model the code had to be recompiled. Models could not be controlled via user inputs from keyboard or joystick; VEGAS could only run for predetermined scenarios which literally indicate that the simulation could run only in batch mode. For visualization, OpenGL was used with immediate mode commands to draw missile and plane objects, which caused performance degradation while running the simulation.

Compared to VEGAS, SahinSim uses 6-DoF flight dynamics equations. The flight dynamics parameters are changed through configuration files so that no recompilation is required. SahinSim provides easy use keyboard and joystick interface to fly airborne objects. The graphics engine provides better visuals and it even performs better than VEGAS.

There are other flight simulators like FlightGear and OpenEaagles [3-4]. These are very comprehensive and complete simulation environments that can be used for the on going project but their source code are too complex so that it takes much time to be able to understand the code and implement scenarios such as VEGAS. Since SahinSim is specially designed for end-game simulations, it only provides the most important capabilities while keeping the source code simple.

The power of SahinSim's graphics interface is provided by the open source graphics engine OGRE. The other underlying structure in SahinSim is the flight dynamics model, JSBSim. In this chapter the basic elements of SahinSim, which are the graphical engine OGRE and the flight dynamics model JSBSim, are introduced.

2.1 JSBSim

JSBSim is an open source flight dynamics model under LGPL license, freely available for proprietary and public use. It's written in C++ and can be compiled by any robust C++ compiler. In 1996, JSBSim was conceived as a batch simulation tool for modelling flight dynamics and flight control. It was designed for use in aircraft design and control courses. Later, the author John S. Berndt started to work in FlightGear project, which is a comprehensive flight simulator, and JSBSim integrated with FlightGear in 1999. Today JSBSim is the default flight dynamics model in FlightGear [3].

JSBSim provides a mathematical model for rigid aircraft equations of motion. Aerodynamics of the aircraft is modelled using a component build-up method. All forces and moments on the aircraft are calculated by summing up all contributions to each force and moment axis (Figure 2.1). After calculating the forces and moments, JSBSim returns next state of the aircraft in discrete time steps [5].

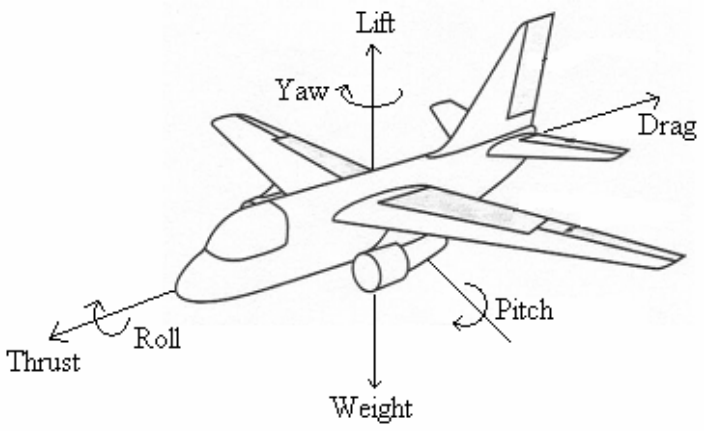

Figure 2.1 : Force and moment axises on an aircraft

Propulsion system of an aircraft is also modelled in JSBSim. In order to provide a realistic perception of the propulsion system from pilots' point of view, several engine types such as piston, turbine, rocket and electric are defined. Although the

models are not precise engineering models, they provide relatively accurate forces and moments on the aircraft.

2.1.1 Aircraft and Systems Definition in JSBSim

Aerodynamic characteristics, propulsion system, control and automatic control systems are described in configuration files which are written in XML format. Any change of the properties can be tried in the simulation without any code change and without recompiling the code in JSBSim. Obviously mathematics takes place while modeling the flight physics. JSBSim makes use of arbitrary algebraic functions to define aerodynamic and flight control characteristics. As flight dynamics characteristics are often stored in tables, JSBSim also allows usage of tables in the configuration file. In the following section, first the property concept is explained, then mathematics model of JSBSim is introduced.

2.1.1.1 Properties

Generally simulation programs manage large amount of state information which may cause some problems. There may be different interfaces used for maintaining the variables such as environment variables, specification files, command line options etc. When the state information contains lots of variables using different mechanisms to be maintained, the runtime reconfigurability becomes difficult.

The property management system provides a single interface for the state information of an instance in the program. The properties can be parsed from a configuration file and be created at run-time. More detailed information about the property system is given in later sections.

2.1.1.2 Functions

Algebraic functions can be defined in a JSBSim configuration file by using JSBSim's function specification notation. The notation is similar to MathML (Mathematical Markup Language). A function definition consists of an operation, a value, a table or a property (which refers to a value). Currently, along with basic algebraic functions, trigonometric functions and some utility functions such as max, min, average, random, etc. are supported. An example operation is defined in (Figure 2.2).

```
<sum>
  <value> 3.14159 </value>
  <property> velocities/qbar </property>
  <product>
    <value> 0.125 </value>
    <property> metrics/wingarea </property>
  </product>
</sum>
```

Figure 2.2 : An operation example

The operation is similar to (Equation 2.1). The inner items in the *sum* operation is summed up, also nested operations can be used as operation elements (product operation in that case).

$$3.14159 + qbar + (0.125 * wingarea) \quad \textbf{(2.1)}$$

A full function definition which may be used in defining the aerodynamics properties of an aircraft includes the function element and within that function element it contains the operations. The function element can just have one operation as its top-level operation which is generally a sum or product operation. The deepest level in a function is always a property or a value, which can not contain another element. An example function definition is in (Figure 2.3).

```
<function name="aero/coefficient/Clr">
  <description>Roll moment due to yaw rate</description>
  <product>
      <property>aero/qbar-area</property>
      <property>metrics/bw-ft</property>
      <property>aero/bi2vel</property>
      <property>velocities/r-aero-rad_sec</property>
      <table>
          <independentVar>aero/alpha-rad</independentVar>
          <tableData>
               0.000 0.08
               0.094 0.19
          </tableData>
      </table>
  </product>
</function>
```

Figure 2.3 : A function example

2.1.1.3 Tables

Table values are widely used while creating the aerodynamic equations,. In JSBSim, one, two and three dimensional lookup tables can be defined. An example table definition is in (Figure 2.4).

```
<table>
  <independentVar lookup="row"> aero/alpha-rad </independentVar>
  <tableData>
   -1.57 1.500
   -0.26 0.033
    0.00 0.025
    0.26 0.033
    1.57 1.500
  </tableData>
</table>
```

Figure 2.4 : A one dimensional table

The lookup key “row” indicates that the table is indexed by the property “aero/alpha-rad”. The first column has the index keys and the second column has corresponding values. For instance, the corresponding value to alpha-rad=-0.26 is 0.033. The values in the tables are interpolated linearly, however no extrapolation is done out of the limits of the index value, instead the highest or lowest value is returned if the index is out of range.

Similar to one dimensional tables, two dimensional tables can be defined using “row” and an additional “column” lookup keys. An example is show in (Figure 2.5).

```
<table name="property_name">
  <independentVar lookup="row"> property_name </independentVar>
  <independentVar lookup="column"> property_name </independentVar>
  <tableData>
                {col_1_key   col_2_key   ... col_n_key }
    {row_1_key} {col_1_data col_2_data ... col_n_data}
    {row_2_key} {...         ...         ... ...        }
    { ...     } {...         ...         ... ...        }
    {row_n_key} {...         ...         ... ...        }
  </tableData>
</table>
```

Figure 2.5 : 2D table example

In order to define a 3D table, along with “row” and “column” lookup keys, the “breakpoint” key is used (Figure 2.6).

2.1.2 Aerodynamics

JSBSim uses coefficient buildup method to model the aerodynamic forces and moments that act on an aircraft. In the coefficient buildup method, all contributions to each force and moment axis is summed and the result is used to derive the next state of the aircraft. Depending on the aircraft and the accuracy of the model, contributions differ. For instance, in order to calculate the lift force, the contributions from wing, elevator and flaps can be summed. While calculating the forces and

moments, some aerodynamic coefficients are used. These coefficients can be taken from textbooks, flight test reports or just hand calculated. JSBSim also supports specifying aerodynamic coefficients as functions.

Aerodynamics properties of an aircraft are defined in the <aerodynamics> section of the XML configuration file. This section has six subsections which are named <axis>. Each <axis> element represents one of three force and three moment axes. The basic layout of the <aerodynamics> section is in (Figure 2.7).

```
<table name="property_name">
  <independentVar lookup="row"> property_name </independentVar>
  <independentVar lookup="column"> property_name </independentVar>
  <tableData breakpoint="table_1_key">
               {col_1_key   col_2_key   ... col_n_key }
    {row_1_key} {col_1_data col_2_data ... col_n_data}
    {row_2_key} {...         ...         ... ...        }
    { ...     } {...         ...         ... ...        }
    {row_n_key} {...         ...         ... ...        }
  </tableData>
  <tableData breakpoint="table_2_key">
               {col_1_key   col_2_key   ... col_n_key }
    {row_1_key} {col_1_data col_2_data ... col_n_data}
    {row_2_key} {...         ...         ... ...        }
    { ...     } {...         ...         ... ...        }
    {row_n_key} {...         ...         ... ...        }
  </tableData>
  ...
</table>
```

Figure 2.6 : 3D table example

```
<aerodynamics>
  <axis name="DRAG">
    { force contributions …}
  </axis>
  <axis name="SIDE">
    { force contributions …}
  </axis>
  <axis name="LIFT">
    { force contributions …}
  </axis>
  <axis name="ROLL">
    { moment contributions …}
  </axis>
  <axis name="PITCH">
    { moment contributions …}
  </axis>
  <axis name="YAW">
    { moment contributions …}
  </axis>
</aerodynamics>
```

Figure 2.7 : Aerodynamics configuration

The supported grouped set of axes in JSBSim are; drag, side, lift set (wind axes), x, y, z set (body axes), and axial, side, normal set (body axes). All these axes systems accept roll, pitch and yaw axes definitions. Force and moment contributions are defined in <axis> sections by using functions. For each axis, the function outputs are summed to calculate the total moment or force. In defining a moment or force, functions provide use of tables, constants, trigonometric functions and standard mathematical C library functions as mentioned previously. Properties are used to access simulation parameters. An example contribution function is in (Figure 2.8). In this function, the force contribution of drag due to leading edge flap deflection is the product of the properties and one value determined the table indexed by "aero/alpha-rad" property.

As mentioned previously, all functions in the <axis> section are summed and applied to the aircraft in the appropriate manner. In the functions, some properties can be outputs of other functions. In JSBSim, the functions that are specified outside of <axis> section are calculated but they do not particularly contribute to a specific force or moment axis, however these function outputs can be referenced by other functions in the <axis> section. This feature allows function outputs that may be applied to several functions to be calculated once and used multiple times. The typical usage of this feature is to calculate aerodynamic coefficients in a function outside of an <axis> section and use the output of this function in the functions that are in <axis> section and define the forces and moments. This usage simplifies the aerodynamic definitions and prevents multiple calculation of the same function which saves computational time.

2.1.3 Flight Control and Autopilot Models

In modern aircrafts, either military or commercial ones, the aircraft is controlled through an electronic flight control system. Commands given by the pilot are processed in the control system (i.e., flight computer) and actual control commands to actuate the mechanical control system are produced by the flight computer. By using JSBSim, flight control and autopilot systems can be defined. Just like a real control system, the control system can be designed by connecting chains of control components each other. Some control components that are modeled in JSBSim are; filter (lag, lead-lag, second order, integrator, etc.), switch, gain and summer control

blocks. Each component runs in the order of definition and calculates the output regarding its type.

```
<function name="aero/coefficient/CLDlef">
  <description>Lift_due_to_leading_edge_flap_deflection</description>
  <product>
    <property>aero/qbar-psf</property>
    <property>metrics/Sw-sqft</property>
      <property>fcs/lef-pos-norm</property>
      <property>aero/function/kCLge</property>
      <table>
        <independentVar>aero/alpha-rad</independentVar>
          <tableData>
            -0.1750  -0.0120
            -0.0870  -0.0040
             0.0000  0.0020
             ...
             ...
             0.6980  0.0280
             0.7850  0.0250
          </tableData>
      </table>
  </product>
</function>
```

Figure 2.8 : A force contribution function

Autopilot systems can also be defined by using the same control components available for a flight computer in JSBSim[6]. The main purpose of the flight computer is to actuate necessary parts(e.g., elevator, aileron, rudder) of the aircraft in order to perform the desired move (e.g., turn, dive). On the other hand, autopilot systems are generally designed to perform specific actions such as keeping the altitude, keeping the heading or automatically heading towards a specified angle or even landing the aircraft automatically. Automatic pilots can also be implemented as a part of the main flight control system depending on the design intend.

2.1.4 Scripted Flights

JSBSim can be used as a flight dynamics library or it can be run in batch mode. When run in batch mode, JSBSim controls the aircraft in the way that it is defined in a configuration script. Scripting allows users to define actions when any defined condition occurs. In a scripted flight “action” means setting a property of the aircraft (e.g., setting wing leveler autopilot switch on/off, moving the flight stick, adjusting the throttle, etc.). Any property can be set to a fixed number or set to an output of a function which is defined in the script. Conditions can depend on any property of the aircraft. Test operations "==", "!=",">",">=", "<" and "<=" can be used in conditions,

also logical operators “AND” and “OR” can be used with nested condition checks. Simply, a scripted flight can be thought as an autonomous robot flying the aircraft by following the predefined movements in its program. The robot can start the engine at a time, advance the throttle, pull the flight stick when the aircraft reaches at a defined speed, head the aircraft to a location when the aircraft reaches a defined altitude and fly the aircraft to a location. Scripted flights provide exactly the same responses and actions repetitively are very useful in aircraft performance tests and control systems development.

2.2 OGRE

OGRE is an open source graphics rendering engine which is written in C++ language. It is runs on many platforms such as Windows, Linux, Mac OS, etc. and supports OpenGL and DirectX. It provides an easy-use and powerful interface to graphics hardware. Rather than dealing with native OpenGL or DirectX libraries, applications can be visualized by using OGRE in a convenient way. OGRE handles most of the routines and the user can concentrate on the main subject.

SceneManager, Entity and SceneNode classes are fundamental building blocks of OGRE engine. These fundamental blocks are mostly accessed by using the highest level of classes that is the Root class. The SceneManager manages and keeps track of everything that appears on the screen. Scenes may consist of models such as aircrafts, robots, or trees, static geometry such as building interiors or terrain, light sources that illuminate the scene and cameras that are used for viewing the scene. Rendering techniques of different scenes vary, so considering the environment on the scene, different SceneManagers which are optimized for the particular scene can be selected.

Entities can be considered as types of objects that are rendered on a scene, in other words an Entity can be thought as the objects represented by a 3D mesh. For instance, a monster, a chair, a tree or a robot can be Entities. OGRE separates renderable objects from their location and orientation, which means that Entities can not be placed on the screen directly, instead Entities are attached to SceneNodes and displayed.

A SceneNode keeps track of location and orientation information for all objects attached to it. An Entity is not rendered on the screen until it is attached to a SceneNode. In addition to this, a SceneNode which has no Entity is not displayed on the screen. The only way to display an Entity on the screen is creating a SceneNode and attaching the Entity to it. SceneNodes can have any number of objects attached to them. Other objects such as Lights and Cameras can be attached to SceneNodes. SceneNodes can be attached to other SceneNodes and can create hierarchies. One of the most important feature of a SceneNode is that it's position is always relative to its parent SceneNode. When the parent SceneNode is moved, all child SceneNodes attached to the parent SceneNode are automatically moved. This feature is very useful when a SceneNode has other SceneNodes that has to move with it, for instance a walking robot can be created such that, a robot Entity, a light source and a camera are attached to child SceneNodes. In that case only moving the parent SceneNode is enough to move all objects [7].

In summary OGRE is an easy to use and effective graphics engine which makes creating visual applications faster and easier. Typically the user just cares about what to do where; not how to do it. OGRE hides the details from the user, handles the routines and draws the required instances.

3. SAHINSIM SIMULATION ENVIRONMENT

This section introduces the simulation environment from a high-level perspective. SahinSim is written in C++ language and can run on any Linux distribution that includes the dependency packages needed by the FDM and the graphical engine. The FDM and the graphical engine are also written in C++ so that the FDM and graphic libraries can be natively used without requiring any conversion interface which generally causes performance degradation. Moreover, C++ is a widely used-well known programming language which achieves both flexibility and performance requirements of a flight simulation program.

In this project, five open source projects have been integrated; these are JSBSim, SimGear, OGRE, SDL and OIS. The flight dynamics engine is JSBSim [8]. SimGear is used with JSBSim to provide some geometric calculation functions and a convenient logging interface [9]. The graphics engine is OGRE and both OIS and SDL are used for keyboard and joystick input interface [7,10,11]. (Figure 3.1) is a diagram of the mentioned projects in SahinSim.

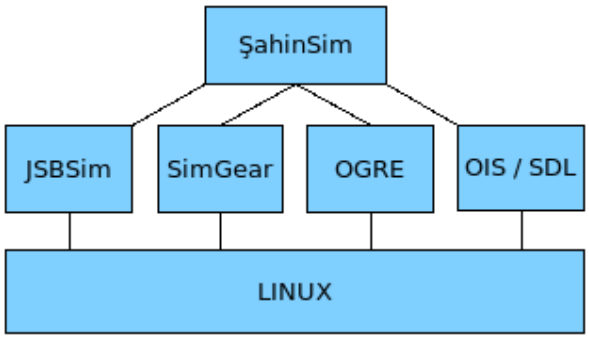

Figure 3.1 : Open source projects used in SahinSim

3.1 Coordinate System

There are several coordinate systems used internally within each component of SahinSim. However, only two coordinate systems are visible to the user. The simulation environment is centered to a point defined in the configuration file. This

point is given in geodetic coordinates which is the coordinate system used in a real world map. All objects in the environment are positioned relative to the center of the environment. Coordinate axis frame of SahinSim is local horizontal/vertical frame (Figure 3.2). x axis points North, y axis points East and z axis points the center of Earth.

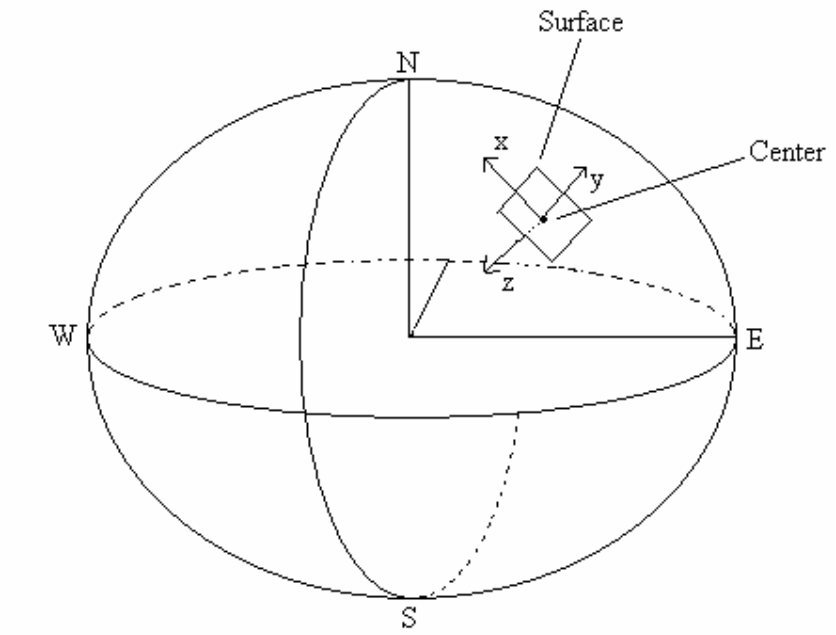

Figure 3.2 : Coordinate system

3.2 Flight Dynamics Model

JSBSim is the flight dynamics engine of SahinSim. SahinSim uses the libraries generated by JSBSim and it is compiled with the interfaces of JSBSim. Basically SahinSim sends the control inputs to JSBSim; JSBSim calculates the next state of the planes and sends back the results to SahinSim. As SahinSim is an end-game simulation, it generally lasts around a minute or two. The result of the simulation is either a hit or miss of the missile. The simulation doesn't have to deal with takeoff and landing stages of a flight course. Consequently physical gear model of JSBSim is not used in SahinSim, all references and functions of the gear model are removed from JSBSim interface. Aircrafts start in the air and never expected to land with gears.

3.3 Graphics

The graphics engine OGRE provides a flexible, easy to use and fast graphical interface which runs on top of OpenGL graphics library [7,12]. Every texture and 3D model can be changed in the simulation without compiling the simulation code. In order to visually observe the details of the simulation SahinSim provides a fairly nice graphical interface (Figure 3.3).

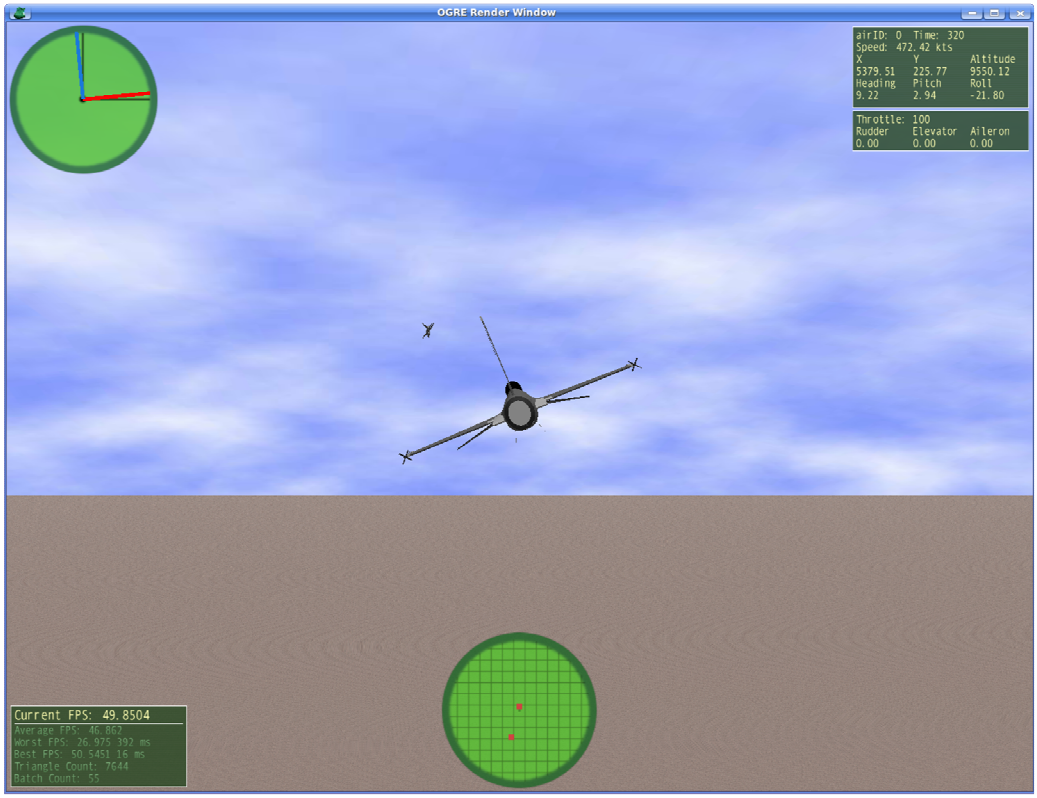

Figure 3.3 : SahinSim screenshot

The active plane is centered in the screen and the default camera is just at the back of the plane. At top-left of the screen, there is the tracking radar, at left-bottom there is the FPS information panel. At the bottom of the screen there is the overall radar and at top-right of the screen there is the flight information panel. Graphical interface can be easily modified using OGRE's configuration files.

3.3.1 Panels

Panels are just for basic usage and can be extended easily. Panel positions, panel backgrounds and even the text font in the panels can be modified without any need to compile the application. Currently there are four panels used.

3.3.1.1 Tracking Radar

As its name indicates this radar is designed for tracking a specified target. It can be used in dog-fight scenarios as well as in regular escorting missions and end-game simulations. Without need to actually see the target, pilot can estimate the position of the target in three dimensions by just looking at this two dimensional radar (Figure 3.4).

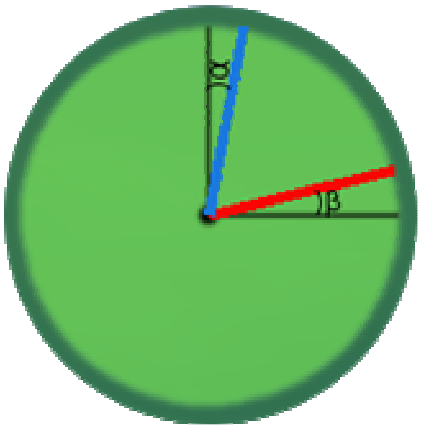

Figure 3.4 : Tracking radar

There are two needles in the radar. In order to track the target from behind, the pilot should keep the needles on their fixed origins, red needle on the vertical fixed black line and blue needle on the horizontal fixed black line. The red needle indicates the vertical position of the target. The length of the needle varies regarding the altitude difference between the plane and its target. The blue needle indicates the horizontal position of the target. Its length varies regarding the distance between the plane and the target on x-y plane. α and β angles are demonstrated in (Figure 3.5).

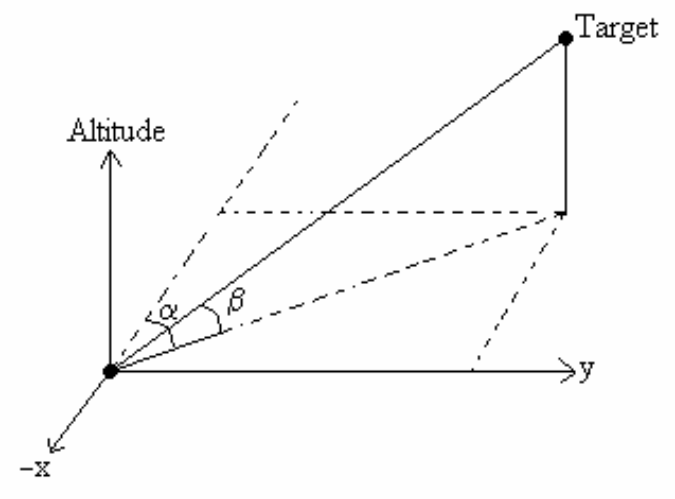

Figure 3.5 : α and β angles

3.3.1.2 Overall Radar

This radar shows all airplane instances in range of the active plane. The radar orientation is fixed and top of the radar is always aligned with the heading of the plane. All instances on the radar panel orientates around the plane as shown in (Figure 3.6)

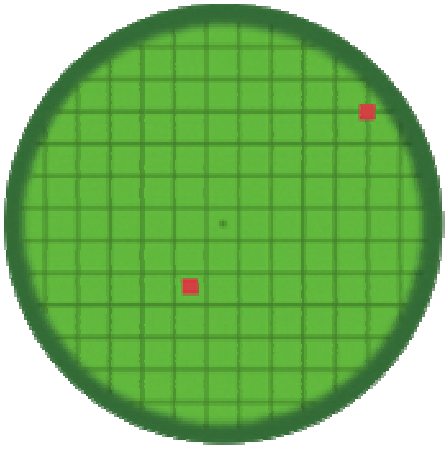

Figure 3.6 : Overall radar

The range is calculated in three dimensions by using Euclidean distance.

3.3.1.3 Flight Information Panel

The panel shows basic information about the simulation and the plane (Figure 3.7).

Figure 3.7 : Flight information panel

The variables presented in the panel and their meanings are expressed in (Table 3.1).

3.3.2 Cameras

There are two camera types implemented in SahinSim. By default, camera is attached to the first plane defined in the configuration file. User can cycle through the cameras by pressing the ‘c’ key on the keyboard, and can cycle through the planes by pressing the ‘n’ key. Camera positions are saved for each camera and each

plane so that when user selects the previous camera while cycling through the cameras, the camera doesn't reset its orientation, instead previous orientation is restored. Every camera is positioned at the back of its plane and parallel to surface of the simulation environment by default.

Table 3.1 : Flight information panel

Name	Description
airID	This is the unique number given to each object in SahinSim
Time	Simulation time in number of iterations
X, Y, Altitude	Position information relative to the center of the simulation
Heading, Pitch, Roll	Orientation information of the aircraft
Throttle	Engine power percentage. 100% is maximum
Rudder, Elevator, Aileron	Normalized control commands

The first camera is a free-cam that can be positioned around the plane. The camera always "look at" to the center of the plane model and always parallel to the surface. Movement axis of the camera can be thought as a sphere which has the plane in its center. In order to move the camera, joystick hat can be used. The second camera is a fixed-yaw camera which is always at the back of the plane. The camera's yaw angle always follows the plane's yaw angle. This camera can not be controlled; its orientation is automatically updated relative to plane's orientation.

3.3.3 Replay Engine

Another feature of SahinSim is its replay engine. The replay engine can be used to examine every movement details of the planes for each simulation tick. If replay is enabled in the configuration file, the simulation can be paused by pressing the 'space bar' on the keyboard at any time. After that, simulation enters into replay mode.

In replay mode, simulation time can be controlled by using the left and right arrow keys on the keyboard. If left arrow is pressed, simulation time decreases, if right arrow is pressed, simulation time increases until the time the simulation is paused. Speed of the replay mode can be adjusted by using up and down arrow keys. For fast

forward or fast backward replay speed, up arrow can be used. For a slow motion replay mode, the speed of the replay can be decreased by pressing the down arrow key.

For every simulation tick, every object's location, orientation, radar information, tracking radar information (if enabled) and flight panel information are saved; these information can be investigated in replay mode. 'Space bar' can be used to exit the replay mode. When user quits the replay mode, the simulation time will advance to the time the simulation is paused.

3.4 Simulation Configuration

Simulation environment and planes can be configured using a configuration file in XML format. Without losing time to configure the simulation from source code, SahinSim can easily be configured through its configuration file, sahin_conf.xml. The simulation configuration and plane configurations must be in context of main XML element named "sahin_conf". Currently there are two main configuration elements within this context; those XML elements are named environment and plane. Environment element must be unique but multiple plane instances can be defined (Figure 3.8).

```
<sahin_conf>
        <environment>
        ...
        </environment>
        <plane name="Plane1">
        ...
        </plane>
        <plane name="Plane2">
        ...
        </plane>
        ...
        ...
</sahin_conf>
```

Figure 3.8 : Simulation configuration file

3.4.1 Environment Configuration Options

This XML element can be used for configuring simulation environment options. More options can be added with a few code changes. An example environment element is in (Figure 3.9).

```
<environment>
        <latitude> 10 </latitude>
        <longitude> 50 </longitude>
        <sim_time> 10000 </sim_time>
        <fps> 50 </fps>
        <replay> on </replay>
</environment>
```

Figure 3.9 : Environment configuration element

Element parameters and their descriptions are listed below in (Table 3.2).

Table 3.2 : Environment configuration parameters

Tag	Description
<latitude> <longitude>	These options specify the center of the simulation environment in geodetic coordinates.
<sim_time>	Simulation stops at this tick unless any other exit event occurs before this simulation tick. In other words, it's the maximum simulation time.
<fps>	This option specifies the desired frame rate for the simulation. SahinSim can run on different machines with different hardware configurations or operating systems, hence the speed of the simulation may vary. In order to keep simulation running always in a fixed frame rate, the frame rate is regulated to a given FPS value. After each iteration, SahinSim simply waits for sometime to meet the given frame rate. This allows users to control the planes in real-time, otherwise on a fast computer the simulation would run faster than normal, on a slow computer it would run slower than normal annoying the users in both conditions.
<replay>	This option turns off or on the replay mode. It's not mandatory to specify this option, the default value is replay-off. If replay is turned on, all states of panels and planes are recorded. This may slightly decrease the performance depending on the hardware and frame rate chosen.

3.4.2 Plane Configuration Options

Each plane instance is configured by using "plane" xml element. SahinSim parses the simulation configuration xml file and creates a plane instance for every plane element. Each plane is given a unique number starting from zero in the order they are parsed. An example plane configuration is in (Figure 3.10).

```
<plane name="Sahin1">
        <location x="0" y="0" z="10000"/>
        <orientation roll="0" pitch="0" heading="0"/>
        <speed> 400 </speed>
        <tick_hz> 50 </tick_hz>
        <fdm_root> /home/ozer/tez/JSB-Models/ </fdm_root>
        <fdm> f16 </fdm>
        <model> f16.mesh </model>
        <targetID> 1 </targetID>
</plane>
```

Figure 3.10 : Plane configuration element

Initial location, orientation, and speed of the plane can be changed by just editing a few lines. Without losing time on compiling, the simulation can be run with different initial configurations in a few seconds. Moreover, by just changing the name of the aircraft, any desired aircraft can be used in same initial conditions. The use of plane

Table 3.3 : Aircraft configuration parameters

Tag	Description
<location>	This option specifies the initial location of the plane in SahinSim coordinate system regarding the defined center of simulation point. Metrics are in feet.
<orientation>	The initial orientation of the plane is defined by this option. Metrics are in degrees.
<speed>	Initial speed of the plane in the direction of plane's flight path in Kts.
<tick_hz>	This option is the number of times the simulation is iterated in one second. In other words, each simulation tick, the plane moves for 1/tick_hz seconds.
<fdm_root>	Root directory of the flight dynamics models is defined here. JSBSim looks for engine, system and aircraft definition files in "fdm_root/engine", "fdm_root/system" and "fdm_root/aircraft" directories respectively.
<fdm>	This option defines the flight dynamics model to be used by JSBSim for the plane. JSBSim loads the aircraft model in file "fdm_root/aircraft/name.xml".
<model>	The 3D mesh file defined in this option is loaded by OGRE for this plane. OGRE looks for this file in its search directories.

configuration is very handy and makes running the simulation with different models or different conditions very easy. Available options for plane configurations are listed above in (Table 3.3).

4. IMPLEMENTATION DETAILS OF SAHINSIM

From a high level view, SahinSim consists of five components (i.e., set of classes) (Figure 4.1). SimMan (simulation manager) is the main component which creates, initializes and controls all other components. Input component is an interface to keyboard and joystick. It polls keyboard and joystick states at each iteration and keeps the state information internally. Input component has both SDL and OIS class instances which are used for polling the actual hardware. Graphics component creates the visual interface, loads the 3D environment, information-radar panels and 3D models. It is the interface to the graphical engine, OGRE. Radar component has position information of all objects in the environment. It is actually an internal class but represented separately to differentiate the component roles.

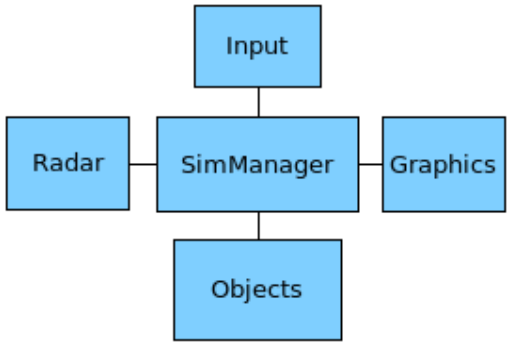

Figure 4.1 : SahinSim components

4.1 Library Dependencies

SahinSim needs some libraries from OGRE, JSBSim, SimGear, OIS and SDL projects. Depending on the Linux distribution, these libraries should be either downloaded from a repository or compiled from source. The Linux platform that is used to develop SahinSim is Kubuntu version 8.04.2. All packages except JSBSim and OIS are downloaded and installed from the universal repository of Kubuntu. Since some compatibility problems with JSBSim and OIS versions that are in the

repository had been experienced, they were both compiled from source. Open source projects used in SahinSim and their versions are given in (Table 4.1).

Table 4.1 : Versions of the projects used in SahinSim

Project	Version
JSBSim	1.0.0
OGRE	1.6.0
SimGear	1.0.0
SDL	1.2.13-1ubuntu1
OIS	1.2.0

4.2 Initialization

Initialization steps of SahinSim are illustrated in (Figure 4.2). SimMan class first parses the SahinSim configuration file and fills an internal configuration structure. Having the configuration file read and considering the relevant simulation resource requirements, SimMan creates the classes related to resource interfaces. The first resource is the SDL library which is used for the joystick device. This initialization is mandatory before actually creating the joystick class and before initializing the graphical engine OGRE. If SDL is initialized successfully, next step is initializing the graphical engine. A void user interface without the AirObjects is created at this step. The sky and the terrain are loaded and lighting of the environment is configured. The last resource is the input interface. At this stage SDL joystick and OIS keyboard classes are created and initialized. After initializing the resources, all AirObjects are created and initialized regarding the configuration elements defined in the XML file. SimMan actually creates the AirObject instances and calls the init functions of the objects by passing the related XML element to each instance. AirObjects are responsible for initializing themselves.

4.3 Objects in SahinSim

In C++ terms, all object instances are derived from an abstract class named AirObject. SimMan calls all object instances' methods by using their down casted pointers to AirObject class type. AirObject can be regarded as the interface to

SahinSim. Any kind of simulation objects can be created, initialized and run through this interface.

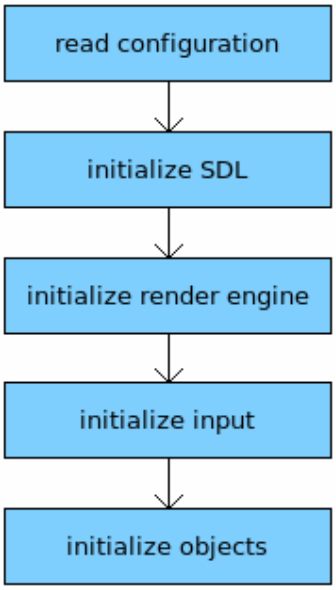

Figure 4.2 : SahinSim initialization steps

Currently all aircraft instances are derived objects from AirObject and aircraft class name is JSBSimPlane (Figure 4.3). JSBSimPlane object also uses other interfaces and other class instances to fully use the SahinSim simulation environment. Additionally, JSBSimPlane is derived from HUDUser and FGXMLFileReader classes and it has Model and FGJSBsim class instances. HUDUser class is used for updating the head-up displays(HUD) of the aircraft. FGXMLFileReader class is used for decoding the XML configuration options. Model class is an interface to the graphical engine and hold 3D model information about the object. Finally, JSBSimPlane has an FGJSBSim class instance to use the JSBSim flight dynamics model.

Different aircraft or missile objects can be implemented by using the model defined above. Regarding the requirements of a particular simulation scenario, SahinSim objects can be derived from other additional classes or may contain new classes. Main SahinSim structure will not be affected by these particular changes.

4.4 Main Simulation Loop

Following the initialization stage, simulation runs an infinite while loop and breaks when the stop time is reached or any escape condition defined in the code(e.g., user pressed 'escape' key, missile hit the target, etc..) occurs. Functions that are called in main simulation loop are in shown in (Figure 4.4).

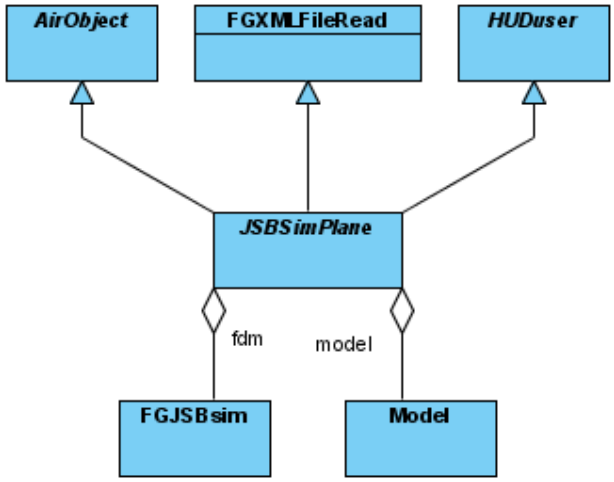

Figure 4.3 : JSBSimPlane class diagram

In each iteration, at the beginning of the loop, a timer to be used for frame regulation is reset. This timer is checked at the end of the loop and if the loop is finished before the expected time, simulation sleeps to synchronize to the required frame rate. After resetting the time, keyboard and joystick states are polled by the input class. Just after polling the input interfaces, simulation checks if any escape condition is occurred. Upon the occurrence of any escape condition, the cleanup function is called to terminate the simulation safely.

The simulation has two running modes; one is active mode and the other is paused mode. If the simulation is in the active mode, all AirObject instances' *run* and *move* methods are called. Basically run methods are used for calculating the next state of the object (i.e., orientation, location, speed, etc.) and move methods are used for moving the objects in the 3D environment (at this stage objects are not drawn, only state information in OGRE engine is updated) . After calling these methods, simulation tick counter is incremented by one.

If the simulation is running in the paused mode, ReplayEngine's *run* method is used to view the state of the simulation at a time frame. At each iteration, if replay mode is

enabled, each AirObjects' state information is recorded in ReplayEngine class. The main simulation loop ends with the frame regulating code.

4.5 SahinSim Components

In this section, basic components of SahinSim, which are input, radar, graphics, replay engine and JSBSimPlane, are explained.

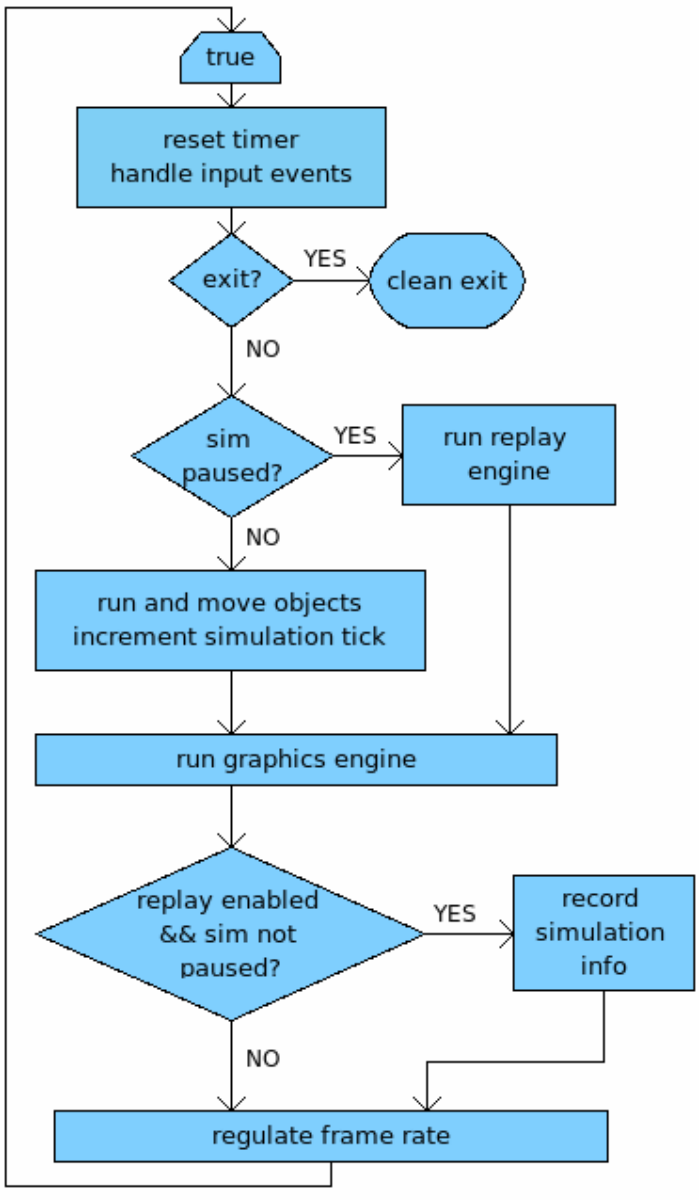

Figure 4.4 : Main simulation loop

4.5.1 Input

This component has one source file which is Input.cpp defining the Input class. Input class is responsible for capturing keyboard and joystick states when required. It uses OIS library for polling keyboard state and SDL library for polling joystick state. Input class has internal structures to keep latest state information about the keyboard and the joystick. In every iteration, input states are polled and saved in the structures once and every object in the simulation environment can get the same state information via Input class.

OIS and SDL interfaces handle input events in different ways. OIS uses callback functions to inform any event occurrence; SahinSim uses OIS for only keyboard events so the event is either a key press or release. In SahinSim, Input class is derived from OIS::KeyListener class and when OIS keyboard object is created in Input class, Input class registers itself as a key listener object. KeyListener objects has to overwrite *keyPressed* and *keyReleased* virtual functions of KeyListener class. Those two functions are called when OIS keyboard object's capture function is called. Capture function polls the state of the keyboard and calls registered key listeners' *keyPressed* and *keyReleased* functions. Input class's *keyPressed* and *keyReleased* functions overwrite the functions in KeyListener class and update the internal keyboard structure regarding the key pressed or released.

Unlike OIS, SDL does not use callback functions, instead events are hold in an event queue and polled by *SDL_PollEvent* SDL library function. *SDL_PollEvent* function takes a pointer to an SDL_Event structure which has SDL event information such as the type of the event, type of button pressed or change on joystick axis position, etc. and fills in this information to the SDL_Event structure. This function returns 0 if an event is polled from the queue or returns 1 if no event is polled from the event queue. After capturing keyboard information via OIS, Input class polls joystick events until all events are processed and internal joystick structure us updated. Objects in SahinSim can simply get access to keyboard and joystick structures via *getKeys* and *getJoystick* functions.

4.5.2 Radar

Radar component is responsible for keeping coordinate information of the objects in the environment while providing some radar utility functions. It has one source file

named Radar.cpp which defines the Radar class. The Radar class is created by the SimMan class and there is only one instance in the environment. Every AirObject in the simulation is registered to the Radar class by SimMan just after creation. In each iteration, every object updates its position in the radar. Radar class keeps pointers to each AirObject in airObjects array and keeps coordinate information in a two dimensional Coordinate array for each simulation tick and object.

Radar class provides functions that return pointers or coordinates of near objects to the calling object (Figure 4.5). *getNearObjects* function updates either the AirObjects array pointer or the Coordinate array pointer taken as an argument regarding the calling object's airID, range and simulation tick. The return value is the number of AirObjects in range. In other words, AirObjects can get Coordinates or pointers of the objects within range (in Euclidean distance) at each simulation tick. *updateCoordinate* function is used to update AirObjects position in the Radar for the given simulation tick. *getCoordinate* function simply updates the Coordinate argument with the coordinate of the AirObject which has airID at a given tick.

```
class Radar
{
public:

...

  int getNearObjects(const int airID, const int tick,
                     const double range, AirObject** nearObjects);
  int getNearObjects(const int airID, const int tick,
                     const double range, Coordinate* nearObjectCoords);

  bool getCoordinate(const int airID, const int tick, Coordinate& coord);
  bool getCoordinate(const int airID_me, const int airID_target,
                     const double range, const int tick, Coordinate& coord);

  void updateCoordinate (Coordinate coordination,
                         const int airID, const int tick );

  int registerAirObject(AirObject* pAirObj);

  inline const Coordinate(*(getAllCoordinates)())[MAX_SIM_TIME]
                         { return all_coords; };
...

}
```

Figure 4.5 : Radar class

4.5.3 Graphics

Graphics component can be divided into three sub components (Figure 4.6). RenderEngine is the main component which initializes and controls other components, runs OGRE engine and controls cameras. The Overlay component is responsible for controlling the panels and HUDs in the screen. The last component Model is the interface between AirObject instances and RenderEngine, which handles OGRE specific requirements.

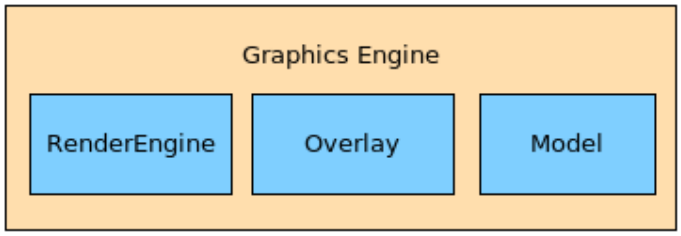

Figure 4.6 : Graphics components

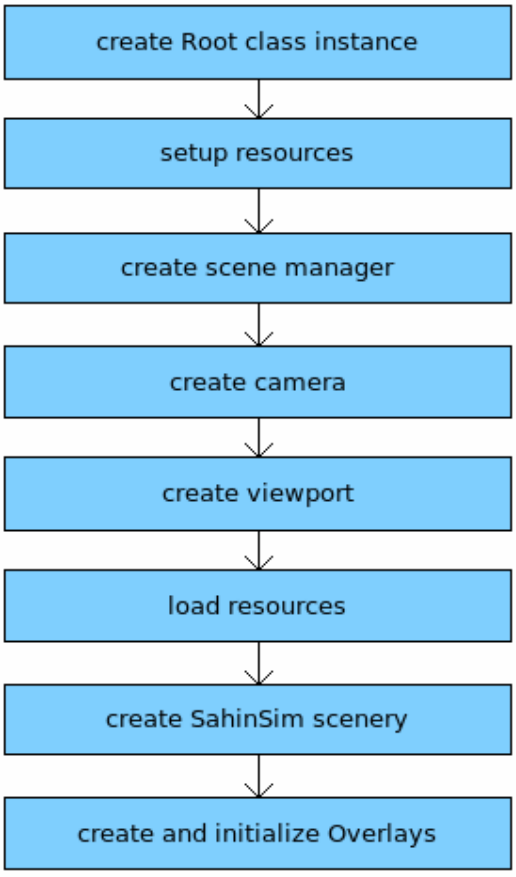

Figure 4.7 : Graphics engine initialization steps

4.5.3.1 Initialization

SimMan starts the initialization process of the graphics engine by calling the *init* method of the RenderEngine class. The initialization steps of the graphics engine is illustrated in (Figure 4.7).

The first step is creating the Root class instance of OGRE with plugins.cfg, ogre.cfg and Ogre.log arguments. Plugins.cfg file defines the OGRE plugins to be loaded. After creating the Root instance, *setupResources* function sets up the resource search directories and resource files from resources.cfg configuration file. OGRE scripts, specific settings, textures, models, all OGRE related materials are searched in the paths defined in this file. Next step is configuring the render window screen and graphics driver. The render Ogre.cfg is the display settings file; if it is found, previously saved display settings will be used, if not, settings screen will pop-up and user can set display settings such as resolution, anti-aliasing options, etc. (Figure 4.8).

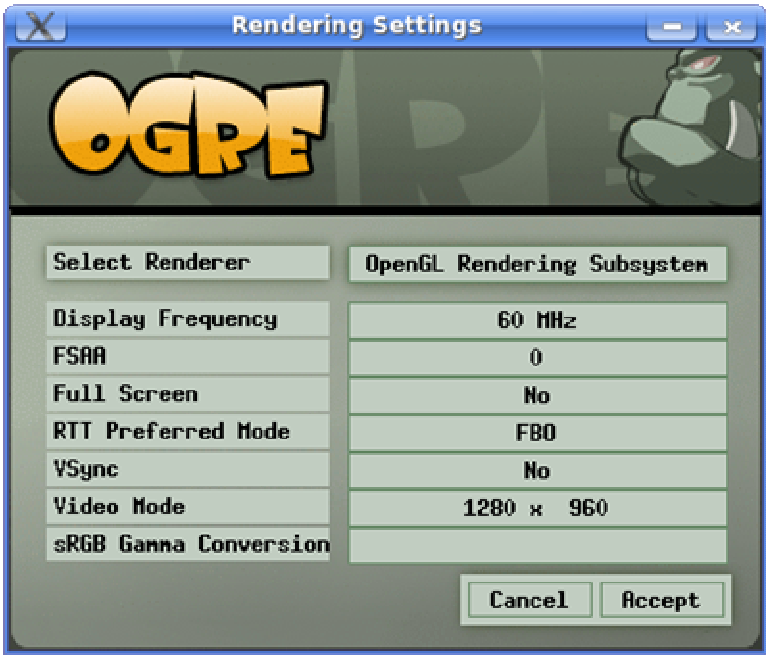

Figure 4.8 : Graphics engine settings window

A generic type scene manager is created in the render window. After loading the scene manager, the default camera is created and its position and looking direction is set in *createCamera* function. Just after creating the camera, the default viewport is created for the entire render window and the camera is attached to the viewport. Until now, only a blank screen with some options is created, there is no visible objects yet.

Before loading the terrain, sky, etc., resource directories are parsed and initialized in *loadResources* function.

CreateScene function loads the scenery of SahinSim. The ambient light is set and a directional light source (i.e., Sun) is created for a realistic look for the 3D objects. The directional light source is the only light source in the scenery which behaves like the Sun. After setting the light, a skydome type sky model is created with could textures. Posterior to sky creation, the surface of the simulation environment is created. In current implementation, the surface is just a plane. The texture of the plane is sourced from a PNG file which is repeated a number of times to fill the plane. At the end of the scenery creation, some fog effect is added close to the surface. As the last step, debug and HUD overlays are created and added as framelisteners to the Root class. When initialization completes, the basic scenery with terrain, sky and panels are loaded.

4.5.3.2 Panels and HUDs

Panels in SahinSim are implemented with OGRE overlays. There are three overlays defined in SahinSim; Debug, HUD and Needle overlays (Figure 4.9). The Debug overlay defines the debug panel in the left bottom of the screen. This panel shows frame statistics such as average FPS, best FPS, number of triangles rendered, etc.

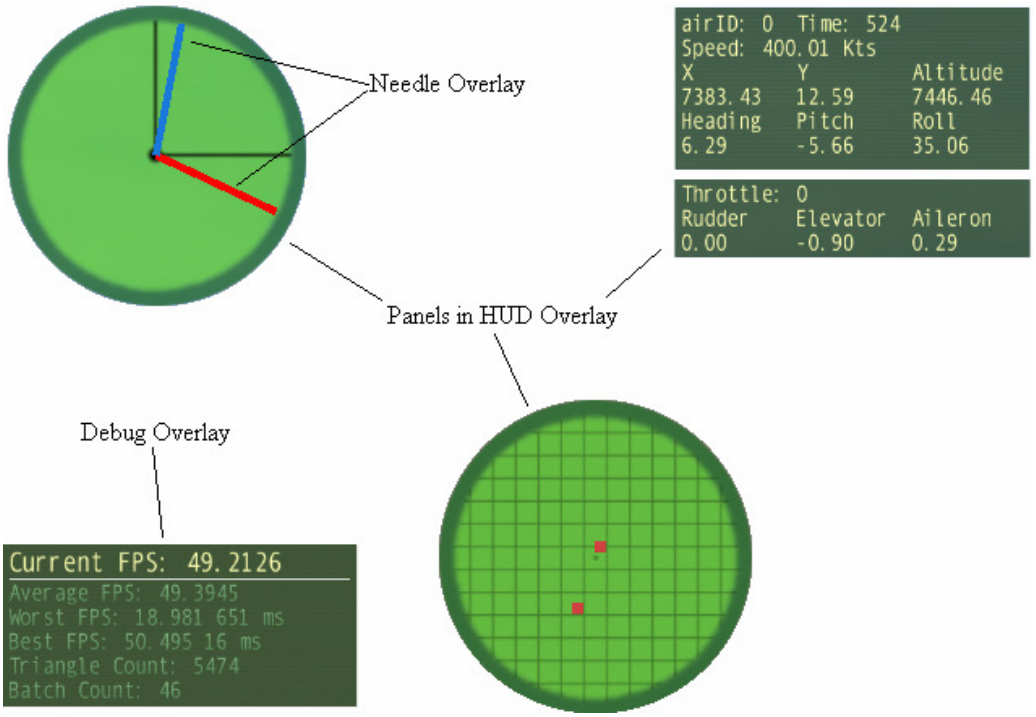

Figure 4.9 : HUD, Debug and Needle overlays

Debug overlay is an internal overlay which is not used by AirObject instances. The DebugOverlay class, which is implemented in the source file DebugOverlay.cpp, is a FrameListener of the OGRE Root object and it is added to the frame listeners list in Root object by RenderEngine. DebugOverlay overwrites the virtual function *frameEnded* (inherited from FrameListener class) which is called by Root object after each frame is rendered on the screen. In other words, *frameEnded* function is a callback function which is called just after each frame ends. DebugOverlay updates the panel information in this function.

The HUD overlay contains all panels that are used by objects in the environment. Overlay is implemented in the source file HUDOverlay.cpp. It has only one public function named *registerHUDuser* which is used to register the panel users to the HUD overlay. *registerHUDuser* function takes two arguments; a pointer of

```
typedef struct HUDinfoFlight {
  int time;
  int airID;
  double speed;
...
  bool valid;
}sHUDinfoFlight;
...
...
typedef struct HUDinfoInput {
...
...
}sHUDinfoInput;

class HUDuser {

  public:
...
  virtual sHUDinfoTrack  getHUDinfoTrack();
  virtual sHUDinfoRadar  getHUDinfoRadar()=0;
  virtual sHUDinfoFlight getHUDinfoFlight()=0;
  virtual sHUDinfoInput  getHUDinfoInput()=0;
  protected:
...
  sHUDinfoTrack  HUDTrack;
  sHUDinfoRadar  HUDRadar;
  sHUDinfoFlight HUDFlight;
  sHUDinfoInput  HUDInput;
...
  void registerHUD();
...
};
```

Figure 4.10 : HUDuser class and panel information structures

HUDUser class and integer airID. HUDUser class is an abstract class which contains virtual functions to update panel information and structures for the panels (Figure 4.10). When an object registers itself as a HUDUser (the object must be derived from HUDUser), HUDOverlay adds the HUDUser pointer to its internal structure array named display. The display structure array is indexed by airID and each array element contains HUD switches and a pointer to HUDUser registered. Any object that is derived from HUDUser class can use the panels and update the information.

Every HUD structure contains information of a specific HUD type (i.e., tracking radar, overall radar, etc.). Every object that uses HUDs has its own HUD information structures (inherited from HUDUser class). The HUD user objects overwrite the pure virtual functions of HUDUser class which are used to retrieve HUD information.

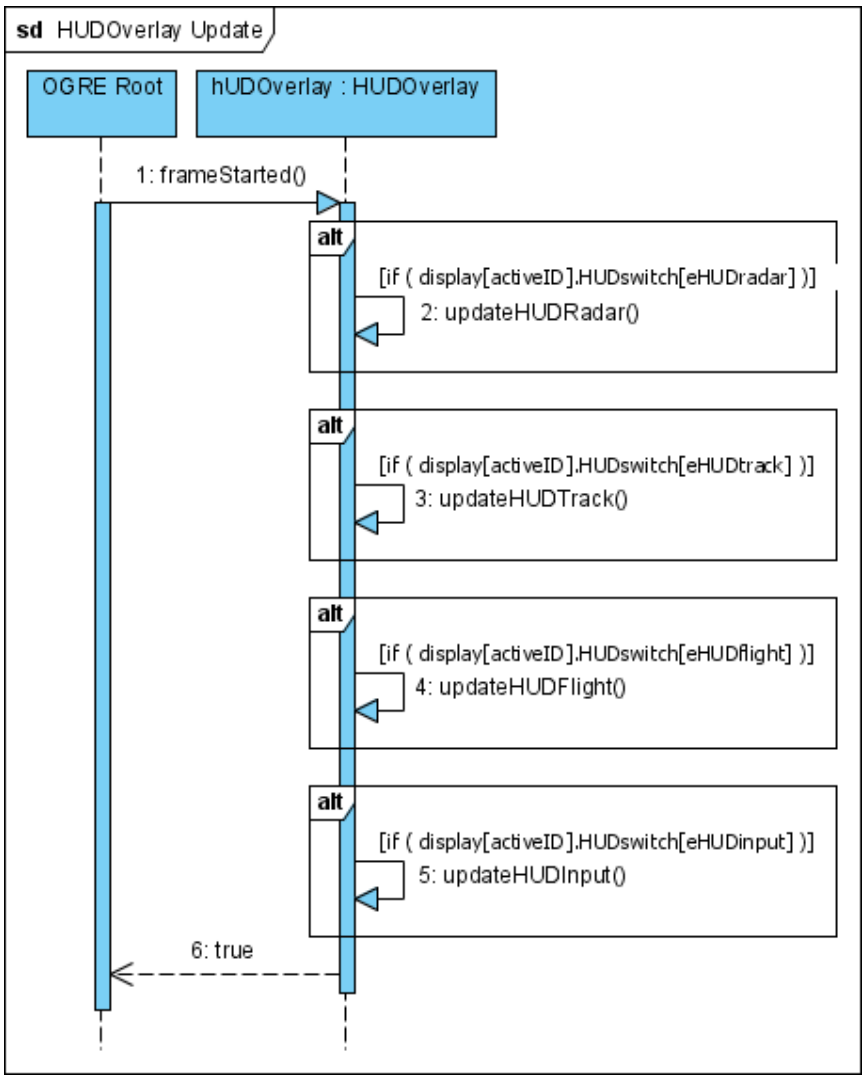

Figure 4.11 : HUDOverlay update sequence diagram

When related update function is called by the HUDOverlay object, the HUD user object fills in related HUD information and returns to the structure. Similar to DebugOverlay, HUDOverlay is a FrameListener and overwrites *frameStarted* function which is called by OGRE Root class just prior to rendering the visuals for each frame. The *frameStarted* function of the HUDOverlay object calls internal private HUD update functions for each HUD type if the HUD switch of the active object's HUD is on (Figure 4.11).

A HUD update function first retrieves its HUD information. If the simulation is running in paused mode, the information is retrieved from ReplayEngine, if not, the information is retrieved by calling the HUD user object's related update function (Figure 4.12). Posterior to retrieving the HUD information, if the data is valid, the panel is drawn regarding the information.

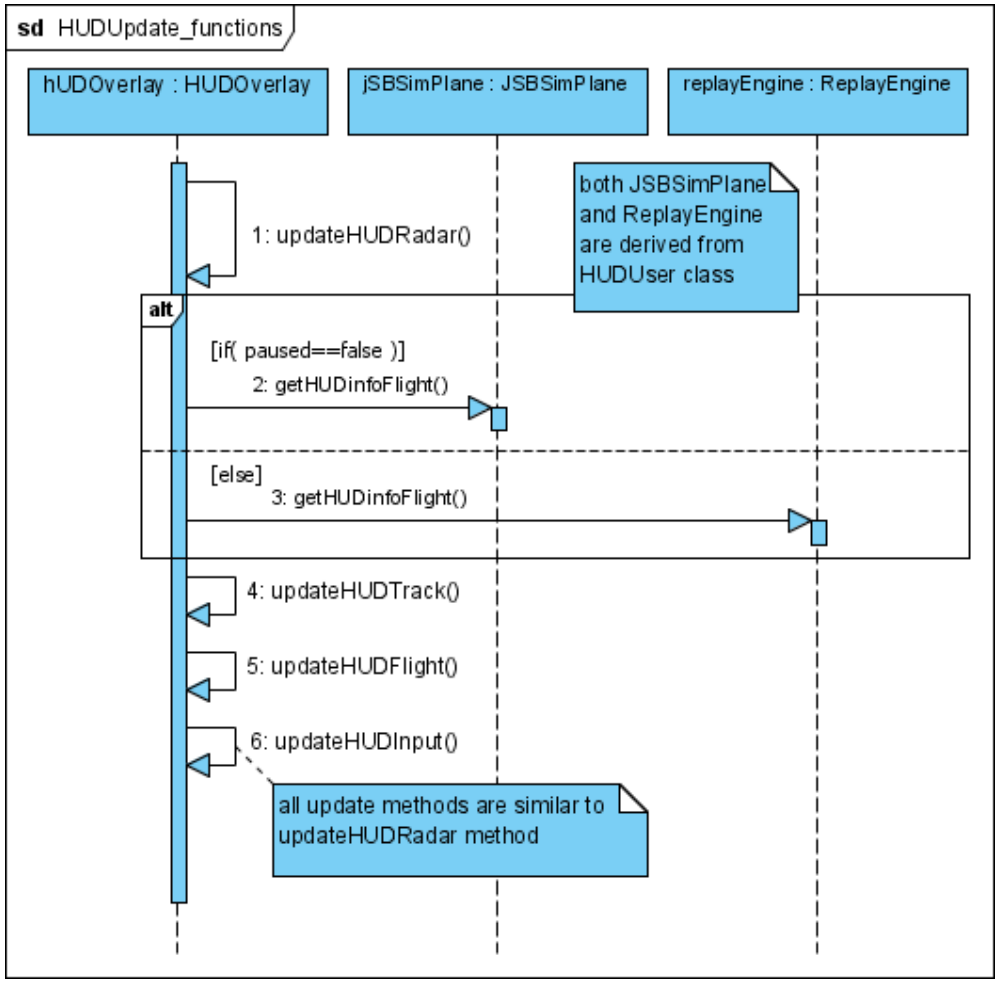

Figure 4.12 : HUDUpdate functions

This design allows adding new panels easily. Every HUD user can update its HUD in the way it wants; just derive the HUD user object from HUDUser class, implement related update functions and register to the HUDOverlay object. Also only active users panel information is updated so that unnecessary function calls are avoided.

4.5.3.3 3D Models and Cameras

3D OGRE models can be easily loaded and controlled by using the Model class. Every AirObject in the environment which requires a visible 3D object in the simulation environment must have a Model class instance. *LoadModel* function is used to load the 3D OGRE mesh from the file name given as the function parameter. 3D model's position and orientation can be set by using *setModelPosition* function. Model class can be regarded as a simple interface class between the user and the OGRE.

LoadModel function creates OGRE scene nodes and a 3D entity (i.e., the mesh model). After creating the scene nodes, the Model instance is registered to the RenderEngine by a call to *registerModel* method of the RenderEngine object. The scene node hierarchy of one Model class is illustrated in (Figure 4.13)

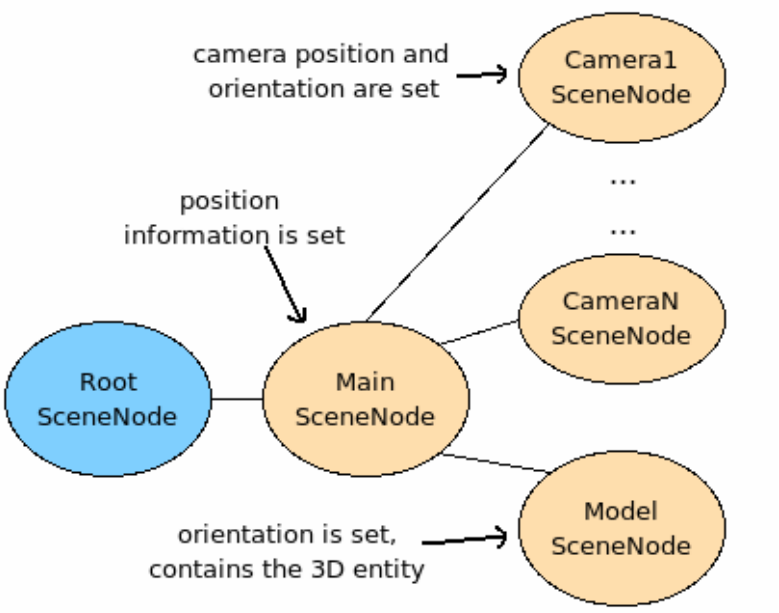

Figure 4.13 : OGRE Scene Nodes of one Model class instance

The main scene node is created as a child of the root scene node of the OGRE scene. The main scene node is the parent of the model node of the Model class and also is parent of all camera nodes. Child nodes' positions and orientations are set relative to

the parent node in OGRE. The *setModelPosition* function only sets the position of the main scene node; by that way it automatically moves other child nodes as well. However it doesn't set the orientation of the main node, instead orientation is only applied to the model node which has the 3D entity. In summary, Model class provides functions to load and control a 3D model.

SahinSim cameras are handled in RenderEngine class. There is only one OGRE camera entity in the environment which is attached to the active camera node. When the Model class is registered to the RenderEngine via *registerModel* function, cameras of the model are created as child nodes from the main scene node of the Model. If the registered Model is the first registrant, then the camera is attached to its first camera node. RenderEngine keeps all Model instance's model information in a Model structure array and saves the registrant information in the array node indexed by modelID (currently same as airID) (Figure 4.14).

```
typedef struct model{
    Model* pModel;
    SceneNode* node;
    SceneNode* modelNode;
    SceneNode* camNodes[MAX_CAMS];
    eCamType eActiveCam;
}sModel;
class RenderEngine
{
public:
...
    bool registerModel( Model*, SceneNode*, SceneNode*,
                        const int,const bool );
...
private:
...
    sModel models[MAX_MODELS];
...
}
```

Figure 4.14 : Model structure and array

Active owner of the camera and camera movements are handled in RenderEngine class's *handleInputs* function. If user presses “n” key, the active model is changed to the next model. In that case, RenderEngine sets the internal active model ID (i.e., activeID) to the next model, as well as Debug and HUDOverlay's activeIDs. RenderEngine simply detaches the camera from current active camera node and attaches it to the next active model's active camera node.

4.5.4 Replay Engine

As its name indicates, replay engine component provides replay functions of SahinSim. The ReplayEngine class is defined in source file RenderEngine.cpp. If replay mode is enabled, replay engine records all AirObjects' panel information in its internal HUD information structure arrays (Figure 4.15) . The panel information is recorded for each AirObject and for each time frame.

SimMan pauses the simulation when the user presses the 'space bar' key. *SetReplay* method of the ReplayEngine is called and replay mode starts. The paused simulation

```
class ReplayEngine : public HUDuser
{
public:
...
    void record(int tick);
    void run();
...
private:
...
    sModel* all_models;
    const struct HUDdisplay* display;
    const Coordinate (*all_coords)[MAX_SIM_TIME];
    HUDinfoRadar  radarInfo[MAX_AIR_OBJECTS][MAX_SIM_TIME];
    HUDinfoTrack  trackInfo[MAX_AIR_OBJECTS][MAX_SIM_TIME];
    HUDinfoFlight flightInfo[MAX_AIR_OBJECTS][MAX_SIM_TIME];
    HUDinfoInput  inputInfo[MAX_AIR_OBJECTS][MAX_SIM_TIME];
...
}
```

Figure 4.15 : ReplayEngine class

tick is saved and replay mode's simulation time can not advance further than thepaused tick. SimMan calls ReplayEngine's *run* method to update the positions of the AirObjects and panel information in the paused mode. In this method, the snapshot of the simulation at a simulation tick is displayed and the time tick of the snapshot is controlled via user inputs. ReplayEngine always shows the current tick's snapshot. 'Right arrow' and 'left arrow' keys controls the current tick variable whereas 'up arrow' and 'down arrow' keys control the speed of the replay.

ReplayEngine has all AirObjects coordinate information via a pointer to all_coords array in Radar class. ReplayEngine also has all Model instance's pointers and moves all AirObjects in the environment by calling *setModelPos* method of each Model with the related coordinate information. ReplayEngine is derived from HUDuser

class; just like an AirObject HUD user, HUDOverlay calls related *getHUDinfo* methods to update the HUD panel contents.

4.5.5 JSBSimPlane Class

JSBSimPlane class implements JSBSim aircrafts in SahinSim. For each plane element in the XML configuration file, a JSBSimPlane instance is created. SimMan object initializes the JSBSimPlane instances by calling *init* method of each instance and passes the related XML element to *init* function. JSBSimPlane initialization steps are illustrated in (Figure 4.16).

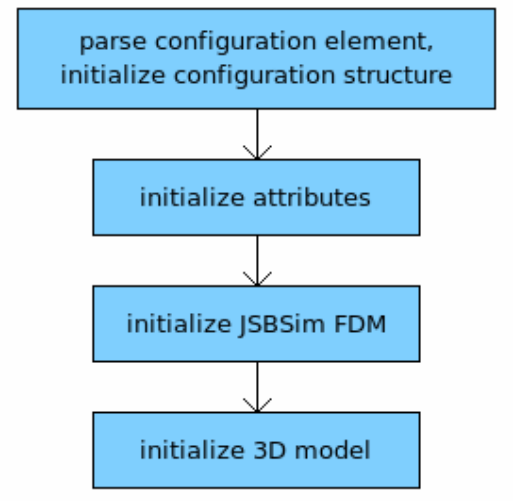

Figure 4.16 : JSBSimPlane class initialization

First step is reading the XML element. JSBSimPlane parses its configuration element and fills in the internal configuration structure named PlaneConf. This structure has all configuration variables that can be used within the class. Next step is initializing some internal attributes via *initAttributes* function. After that, JSBSim FDM is initialized with *initJSBSim* function. *InitJSBSim* function creates the objects related to JSBSim and initializes them. These class instances are explained in the next section. The last step is creating the 3D model for the plane, *initModel* method creates a Model instance and loads the 3D model defined in the configuration structure.

4.5.5.1 JSBSim FDM Integration

JSBSim library is integrated into applications by using FGFDMExec class. FGFDMExec encapsulates the JSBSim simulation executive. All JSBSim classes are

instantiated, initialized and run through this class. Some public methods of FGFDMExec class is shown in (Figure 4.17).

FGFDMExec is created for each plane that uses JSBSim. All objects related to JSBSim are created within FGFDMExec class's constructor method. The *LoadModel* method is used for loading the JSBSim model. When the JSBSim model is loaded, related XML configuration file is parsed and related class instances are created with corresponding configuration elements (autopilot, aerodynamics, propulsion, etc.). After loading the aircraft, initialization is performed. Initialization starts with copying control inputs into appropriate JSBSim data structures. Then the initial

```
class FGFDMExec : public FGJSBBase, public FGXMLFileRead
{
public:
...
  bool Run(void);
  bool RunIC(void);
  bool LoadModel(string AircraftPath, string EnginePath,
                 string SystemsPath, string model,
                 bool addModelToPath = true);
...
private:
...
}
```

Figure 4.17 : FGFDMExec class

conditions of the aircraft (i.e., initial location, speed, etc.) are set. Subsequent to setting initial conditions and copying control inputs into JSBSim structures, JSBSim is run once by calling the *runIC* method to initialize the JSBSim components without actually integrating the aircraft (i.e., dt=0). The typical usage of JSBSim within an application is illustrated in (Figure 4.18). An interface class handles routines to create and run FGFDMExec class. After initializing the model, at each iteration, the interface class copies control inputs into JSBSim, runs JSBSim, and copies state variables of the JSBSim aircraft into its structures. These state variables can be used to drive instrument displays or place the vehicle model into simulation environment for visual rendering.

JSBSim is integrated into FlightGear by using a similar approach. FlightGear supports several flight dynamics models and all models have to be derived from an interface class named FGInterface. FGInterface has internal variables and public methods which are used for defining a generic FDM. A particular FDM which uses

the interface, has to implement some virtual functions that are used for getting and setting FDM variables and an update function. FGJSBSim is the class that integrates JSBSim into FlightGear. It is derived from FGInterface class. Basically, at each simulation tick, FlightGear just calls update method of the FGJSBSim class. FGJSBSim copies FlightGear variables into JSBSim variables, runs the FDM and copies updated state information to FlightGear structures. This usage is similar to the approach shown in (Figure 4.18). SahinSim uses the classes for JSBSim in FlightGear with some modifications. Along with FGJSBSim and FGInterface classes, three classes from FlightGear are used as well. These are FGProperties, FGControls and FGGlobals classes.

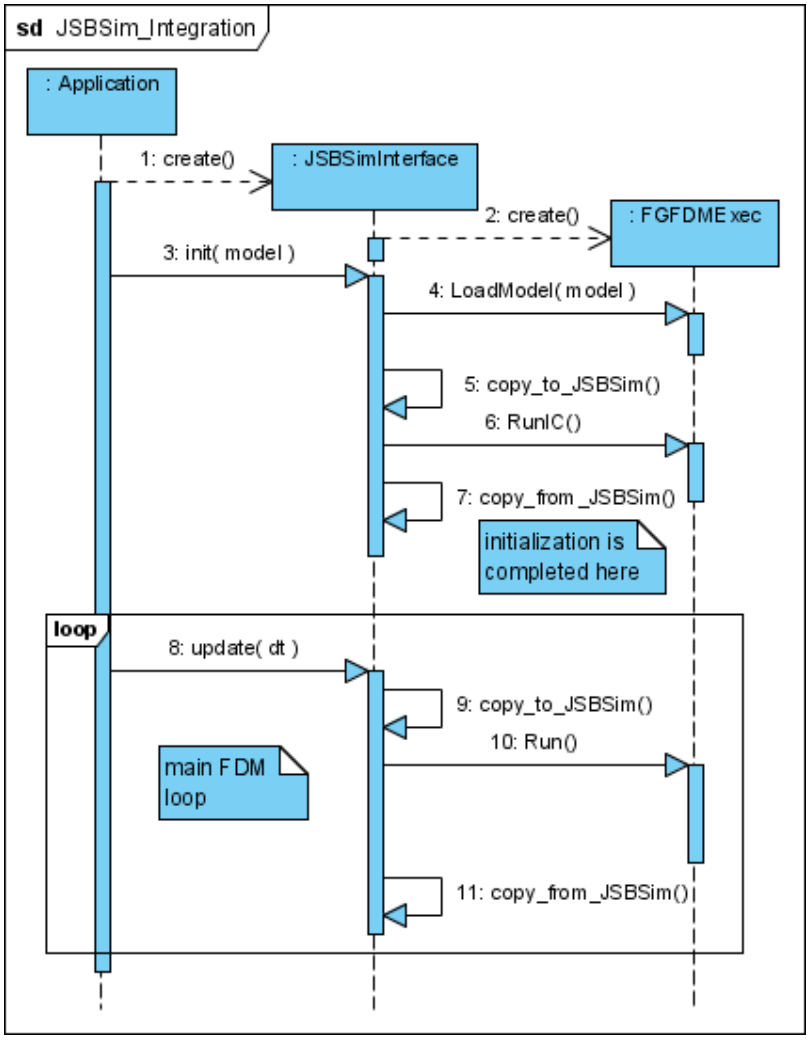

Figure 4.18 : JSBSim integration to an application

FGProperties class is a property management system which provides an interface to the state information of the aircraft. The properties can be regarded as access

controlled global variables such as read/write or only read. The properties are kept in a tree-like structure that is similar to the structure of a Unix file system. The property tree has one root node, sub nodes attached to the root node and other sub nodes (like directories), and at the end of sub-nodes the end-nodes (Figure 4.19).

The properties are associated with a string such as "/controls/flight/left-aileron" and also can be associated with get and set functions by tie method of the property manager. The properties can be accessed with either directly using property manager's methods or using get and set methods associated with the property. In SahinSim, properties can be accessed by using functions defined in fg_props.cxx.

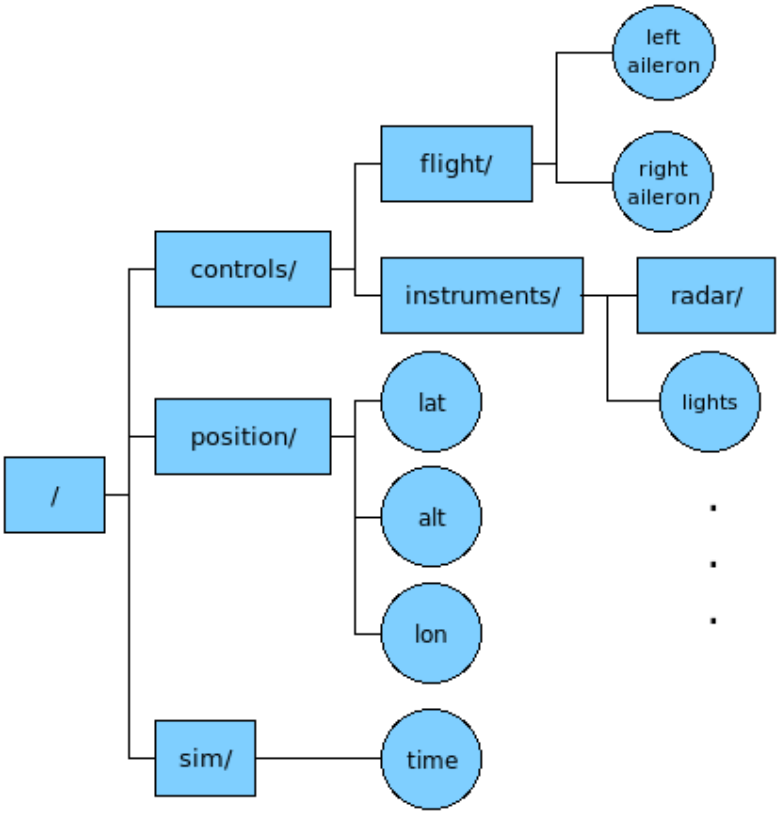

Figure 4.19 : Property tree example

FGControls class contains an extensive set of control variables for the aircraft from elevator, steering to electric system controls and even the tail hook. When initialized, FGControls creates related property tree node for each variable and assigns get and set functions that can be used for accessing the properties. FGGlobals class has pointers to an aircraft's state information modules that is required to be shared among

FlightGear simulation modules. All other modules access to the aircraft's modules by calling get methods of global FGGlobals pointer which is named globals.

The six classes that are involved while running the JSBSim FDM in SahinSim are shown in (Figure 4.20). Each JSBSimPlane has one FGGlobals, one FGControls and one FGJSBSim class instances.

4.5.5.2 FGJSBSim Initialization

JSBSim interface is initialized with *initJSBSim* method. *InitJSBSim* first creates an instance of FGGlobals class named global. After creating the FGGlobals instance, it sets the global FGGlobals pointer named globals. This variable is global and used by other modules so that each FGJSBSim instance has to set the globals pointer to its

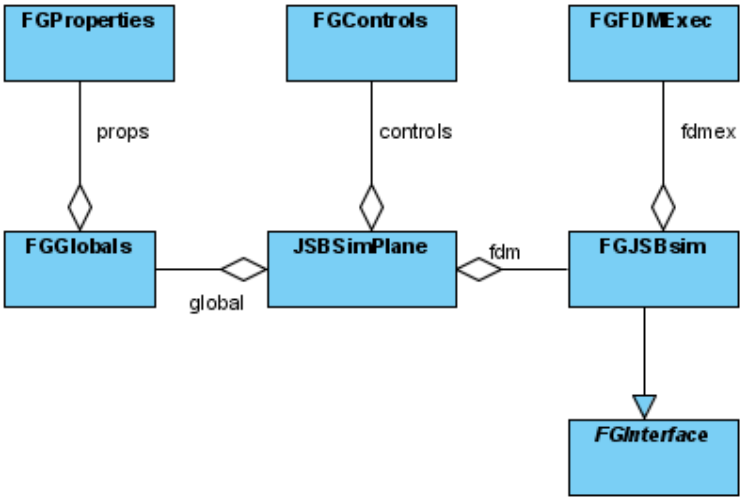

Figure 4.20 : JSBSimPlane class relations

own FGGlobal instance in order to not use previously assigned globals pointer. After creating FGGlobals instance, initial state of the aircraft is set according to the configuration through properties. Then FGControls instance is created. FGControls assigns default variables in its init method and ties to the properties in its bind method. These two methods are called just after creating FGControls instance. The last created instance is FGJSBSim. It is created and its init and bind methods are called. The *init* method initializes internal JSBSim objects and the *bind* method associates some set and get methods of the FGJSBSim instance's properties.

Basically, after initializing JSBSim FDM, JSBSimPlane copies control inputs from SahinSim to JSBSim and runs the FDM. FDM calculates the next state of the aircraft and updates its property tree.

5. CONCLUSION AND FUTUREWORK

SahinSim is an end-game simulation environment providing a 3D graphical interface, an input interface and an interface to a popular flight dynamics model. Multiple instances of manual or script controlled, JSBSim or user-created airborne objects can run in the same environment. It is easy to understand simple and straightforward architecture of the code, and the generic design of SahinSim allows it to be extended while considering specific requirements.

SahinSim application presented in this paper can be used in the missile-aircraft engagement scenarios which were simulated using VEGAS application [1,2]. However, there are some areas that will be improved to provide a more generic simulation environment. Network support is one of these areas. In a user controlled combat scenario, every user can control objects with a number of joysticks but it won't be feasible because only one camera will be active at a time in this version of SahinSim. In that case only one user can actually see his object and control it on the screen. A network interface will be defined to allow multiple users share the same simulation environment. Another improvement area is collision detection. Current implementation leaves collision awareness to objects in the environment. Every object must check that if it collides with another object itself. An overall collision detection technique, which checks collisions regardless of object implementations will be introduced.

REFERENCES

[1] **Akdag, R., Altilar, D. T.,** 2005. A Comparative Study on Practical Evasive Maneuvers against Proportional Navigation Missiles", *AIAA Guidance, Navigation and Control Conference*, San Francisco, California, USA.

[2] **Moran, I., Altilar, D. T.,** 2005. Three Plane Approach for 3D True Proportional Navigation, *AIAA Guidance, Navigation, and Control Conference*, San Francisco, California, USA.

[3] **FlightGear,** <*http://www.flightgear.org/*> (as of 25/03/2009)

[4] **OpenEaagles,** <*http://www.openeaagles.org/*> (as of 25/03/2009)

[5] **Berndt, J. S.,** 2004. JSBSim: An Open Source Flight Dynamics Model in C++, *AIAA Guidance, Navigation and Control Conference*, Providence, Rhode Island, USA.

[6] **JSBSim Reference Manual,** <*http://jsbsim.sourceforge.net/JSBSimReferenceManual.pdf*> (as of 25/03/2009)

[7] **OGRE,** <*http://www.ogre3d.org/*> (as of 25/03/2009)

[8] **JSBSim,** <*http://jsbsim.sourceforge.net/*> (as of 25/03/2009)

[9] **SimGear,** <*http://www.simgear.org/*> (as of 25/03/2009)

[10] **OIS,** <*http://sourceforge.net/projects/wgois*> (as of 25/03/2009)

[11] **SDL,** <*http://www.libsdl.org/*> (as of 25/03/2009)

[12] **OpenGL,** <*http://www.opengl.org/*> (as of 25/03/2009)

CURRICULUM VITA

Candidate's full name: Özer ÖZAYDIN

Place and date of birth: ALANYA/1983

Permanent Address: Divan C. Çamyuva Sitesi No:33 A-3 Blok D:19
Y.Dudullu/Ümraniye/İSTANBUL

Education:

2005 –	Istanbul Technical University, Computer Science, MSc
2001 – 2005	Istanbul Technical University, Telecommunication Engineering., BSc
1994 – 2001	Antalya Anatolian High School

Publications: **Ozaydin, O., Altilar, D. T.**, 2009. SahinSim: A Flight Simulator for End-Game Simulations, *SCSC' 09,* July 13-16, Istanbul, TURKEY (to be published)