CENG491

DETAILED DESIGN REPORT

ANKA YAZILIM

January 2006

Aysun BAŞÇETİNÇELİK
C. Acar ERKEK
Çağıl ÖZTÜRK
Mennan GÜDER

PRELUDE **3**

1 INTRODUCTION **4**

1.1 PURPOSE AND SCOPE OF THE DOCUMENT 4
1.2 DESIGN CONSTRAINTS AND LIMITATIONS 4
1.3 DESIGN GOALS 5

2 OVERVIEW OF THE APPLICATION **6**

2.1 PREPARING THE ENVIRONMENT 6
2.1.1 Defining the map *6*
2.1.2 Constructing a game *6*
2.1.3 Constructing an animation *7*
2.2 PLAYING THE GAME 8
2.3 WATCHING THE ANIMATION 8

3 USER INTERFACE DESIGN **9**

3.1 PLAYER MAIN MENU DESIGN 9
3.2 EDITOR DESIGN 11
3.2.1 Adding a User Display Message *14*
3.2.2 'Yeni' tool-command *15*
3.2.3 'Aktar' tool-command *15*
3.2.4 'Aç' tool-command *15*
3.2.5 'Araç Kutusu' Sub-Window *15*
3.2.6 'Eklenmiş Nesneler' Sub-Window *16*
3.2.7 'Özellikler' Sub-Window *16*
3.3 VIEWER DESIGN 16

4 SYSTEM OVERVIEW **17**

4.1 USER INTERFACE (1.0) 19
4.1.1 Input Handling (1.1) *19*
4.1.2 Output Integration (1.2) *20*
4.2 GRAPHICS ENGINE (2.0) 20
4.3 SOUND ENGINE (3.0) 20
4.4 AI ENGINE (4.0) 21
4.5 EDITOR ENGINE (5A.0) 22
4.5.1 Program Core (5A.1) *22*
4.5.2 Data Calculations (5A.2) *22*
4.5.3 Command Consumer (5A.3) *23*
4.6 VIEWER ENGINE (5B.0) 24
4.6.1 Action Triggers (5B.3) *25*
4.7 FILE MANAGER (6.0) 25
4.7.1 Save (6.1) *25*
4.7.2 Load (6.2) *25*
4.7.3 Load Model and Sound (6.3) *25*
4.7.4 Export Animation (6.4) *26*

5 DATA DESIGN **26**

5.1 CLASSES 26
5.1.1 VisualObject Class *27*
5.1.2 Human Class *27*
5.1.3 Vehicle Class *27*
5.1.4 StaticObjects Class *28*
5.1.5 Traffic Class *28*
5.1.6 BlueArea Class *28*

5.1.7 UserMessage Class *29*
5.1.8 BaseApp Class *29*
5.1.9 Game Class *31*
5.1.10 Animation Class *32*
5.1.11 Map Class *33*
5.1.12 Road Class *34*
5.1.13 RoadSegment Class *34*
5.1.14 Path Class *34*
5.1.15 FileManager Class *35*
5.1.16 UserInterface Class *36*
5.1.17 Vector2 Class *36*
5.1.18 Vector3 Class *36*
5.2 FILE FORMATS 37
5.2.1 Model Files *37*
5.2.2 Animation Files *39*
5.2.3 Game Files *40*
5.2.4 Map Files *41*
5.2.5 Log Files *41*
5.2.6 Storing Files in ".tar" Format *41*
6 SEQUENCE DIAGRAMS 42
6.1 SEQUENCES RELATED TO USER INTERFACE 42
6.1.1 Ruler *42*
6.1.2 Load Animation / Game *43*
6.1.3 Convert Animation to .avi format *45*
6.1.4 Add Object *45*
6.1.5 Capture *46*
6.1.6 Save Animation or Game File *47*
6.1.7 Play Selected Message *48*
6.1.8 Displaying Animations *49*
6.1.9 Playing Games *50*
7 SOUND EFFECTS DESIGN 50
8 ARTIFICIAL INTELLIGENCE DESIGN 51
9 ADDITIONAL CONVENTIONS 53
9.1 VIRTUAL 3D WORLD 53
9.2 CAMERA MODES 54
9.2.1 Editor *54*
9.2.2 Viewer *54*
9.3 FRAME RATE ISSUE IN GAMES AND ANIMATIONS 55
9.4 TRAFFIC DENSITY ISSUES 56
9.5 IMPLEMENTATION OF MODULES 56
10 CONCLUSION 56
11 APPENDIX 57
11.1 APPENDIX A – CLASS HIERARCHY AND RELATIONS 57
11.2 APPENDIX B – GANTT CHART 58

PRELUDE

After the review of our initial design report, we have made some corrections and changed some parts. These can be summarized as:

- Class structure is reviewed.
    - Name of the class "Message" is changed to "UserMessage".
    - Traffic class is explained clearly.
    - Blue Area class is explained clearly.
- Use case scenarios are fixed.
- New design approach for processing user commands.
    - Produce / Consume method is applied.
    - DFD's are updated.
    - System overview section is updated.
- GUI design is updated
    - Adding a user display message GUI is made.
    - GUI for viewer is designed again for a friendlier look.
- New design approach for frame rate issue.
- Explanation for traffic density factor is added.
- How to convert to/from a ".tar" file is explained.
- Schedule is made for the second term including implementation and testing.
- We have found a solution to checking the movements of user in games.
- Detailed explanation of camera modes is added.
- Sequence diagrams of displaying of animations and playing of games are added.
- Sound effects design made more detailed.
- Artificial Intelligence usage is explained in more details.

1 INTRODUCTION

"Anka Trafik" provides an editor for educator and an application environment for the learner. It is a tool designed for traffic education for primary school students focusing on age above ten. The details about the application have been told in our requirement analysis report.

1.1 Purpose and Scope of the Document

The purpose of this report is to explain the details of our design approach to our application "Anka Trafik". These details are basically about the relations and interactions between the components of the overall structure. Parts of the system architecture which are explained in this document are:

- The data flow between different parts of the system
- The file types that we will be used
- External libraries and sources that will be used
- Detailed description and specification of the modules and the classes
- Event handlers
- Integrating Artificial Intelligence

1.2 Design Constraints and Limitations

Besides the time limitation important limitations are:

- **User environment:** The current technology used in end users' computers determines our program's ability. Our end users are teachers and students and the environment is generally schools and the students' and teachers' home computers. Thus we have to consider end user environment in order to determine the capabilities of our tool.

- **External libraries and tools:** We are using software tools and libraries like Ogre, OpenSteer, etc the limitations in these can directly affect our design. As a result of our search about the external tools we tried to select the ones which have all of the functions we will need. Thus, we will not spend much time integrating them.

- **Experience on external tools and libraries:** Many of the tools and libraries we will use are not familiar to us. However, as a result of our survey we have got familiar to them. During the prototype creation we will have a deeper understanding of these tools and environment.

1.3 Design Goals

In order to create a high performance and useful tool the following goals will be the basis of our developing manner:

- ***Performance:***

  Since graphics is the base of our software application, we have to consider the performance problems in a detailed manner. In order to avoid future performance problems, we would try to determine the best frame rate for the running application. The frame rate calculations will be carefully done. We will also determine the data structures that we will use in a way that high performance speed is as optimized as possible.

- ***Well organized code***:

  By the help of using C++, an object oriented approach; we are planning to create a well organized code. As a result of this, our tool will be easily maintainable and extendible.

- ***Usability:***

  Creating an easy to use tool is one of our main aims. Because of the fact that the end users are teachers and students, their computer knowledge is limited. Another usability condition is the complexity of learning the tool. Our aim is to develop a tool that is easy to learn how to use.

- ***Reliability***

  The tool will be tested in every part of the implementation phase, thus crashes will be tried to be avoided.

- **Satisfying user needs:**

  Most important area that our tool will be used is the school. Thus we have to consider the needs of teachers while he/she is using the tool. The general needs of a teacher are:

- *Creating all possible scenarios in traffic education of children:*

  We are enabling user for creating possible scenarios by not doing our development scenario based.

- *Evaluating the children:*

  Our tool will make this possible by supplying a file which contains the number and types of the errors that has been done by the player or watcher during the animation and game.

2 OVERVIEW OF THE APPLICATION

In order to use the tool there are two phases:

- First the teacher prepares the environment or the animation
- Second the learner watches or plays the prepared game or animation

2.1 Preparing the Environment

The person who prepares the environment must:

2.1.1 Defining the map

- Edit predefined animations or games as a starting pattern by opening.

or

- Create a completely new map by opening a new map
- Define the traffic rules which are aimed to teach and find the related traffic signs by selecting from predefined ones in the tool. For example:
  - Traffic lamps
  - Pedestrian crossing
  - Overpass, etc.
- Insert the selected objects
- Then according to the choice, game or animation creation, follow one of the following ways:

2.1.2 Constructing a game

- After creating the basic map, educator should determine the path for pedestrians and vehicles by using the mouse. Also, he/she should determine the density of them.

- Define the role of the learner and the role must be explicitly written in the mission part.
    - Mission part will be displayed at the beginning of the learner's session for learner to read before starting to play the selected game.
- Define start and end positions related to the defined aim
- Choose a character that will be controlled by the learner. The character can be one of the following
    - A person as a pedestrian
    - A person using a skateboard
    - A person using a bicycle, etc.
- Define possible mistakes that a learner can do related to that rule.
- Define the "Blue Area" that corresponds to the mistake like crossing the road in a wrong place. "Blue Area" is the coordinates of that wrong crossing place for example, and in case of entering that area we detect the mistake.
- Define the message that will be passed to the learner when the mistake occurs:
    - Message can be constructed by recording someone's own voice or displaying a written message.
- Then saves the environment created

2.1.3 Constructing an animation

- After creating the basic map with static objects, educator should add the objects he/she will use for the animation. He/she will do that by determining the frames of the animation.
- Define the frames
    - Select the object
    - Assign a task to the object
    - Choose the start frame of the task which is the current frame
    - According to the task give the parameters using the mouse or the keyboard
- Define questions which will appear during the animation

- Define the message that will be passed to the learner during the animation with the frame it will appear. Messages can be
    - One of the questions defined or
    - Information about the rule that will be stressed
- Then saves the environment created

2.2 Playing the Game

- User first selects the game which he/she will play by browsing from the existing ones.
- After the selection the game starts
- Player controls and directs character in the game in order to get to the end which is mentioned in the mission part. Except the character, there will be no controllable object in the game. However, many of the objects will have their own artificial intelligence.
- When the player starts the game and does a mistake by not obeying one of the defined rules, he/she gets an error message defined by the teacher.
- After the error message is appeared, the player returns to the position in which he/she was just before the error is occurred.
- When the player gets to the destination, the game finishes.
- After the learner gets to the end position, a ".log" file will be created.
    - This file will include the mistake and how many times that mistake is done. The mistake definition includes not obeying the defined rule.

2.3 Watching the Animation

The basics and properties in the animation part are:

- The watcher first selects the animation that he/she will watch by browsing from the existing ones.
- During the animation, questions prepared by the teacher will appear as a popup to measure.
- The watcher will answer the question and after that he answered the question he/she will continue to the animation.

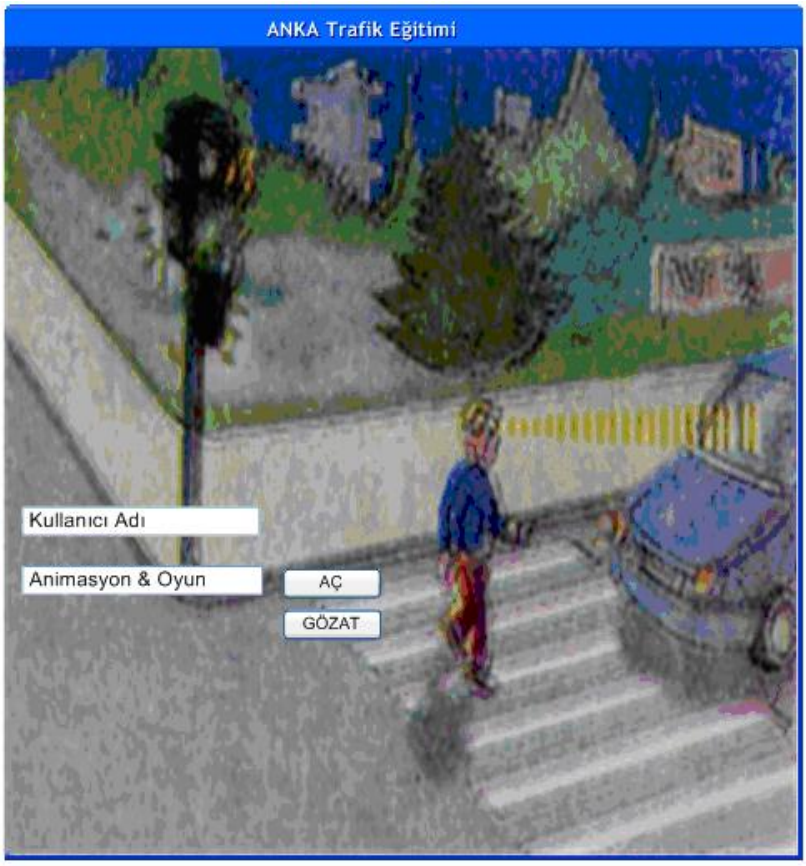
- After the learner watched all of the animation and answered all of the questions defined the animation finishes.
- Then, the tool will form a file which contains the feedback about the pop-up questions' answers. This feedback will include the answers that were given to the questions by the learner.

3 USER INTERFACE DESIGN

There is going to be two executables. One of them gets user information, takes user request, then starts the viewer for an animation or a game. Other serves an editor for user to make a game or an animation. The following subsections explain the interface designs for these two executables. The figures we illustrated there are not the exact forms in final product. They just generalize the user-computer interface.

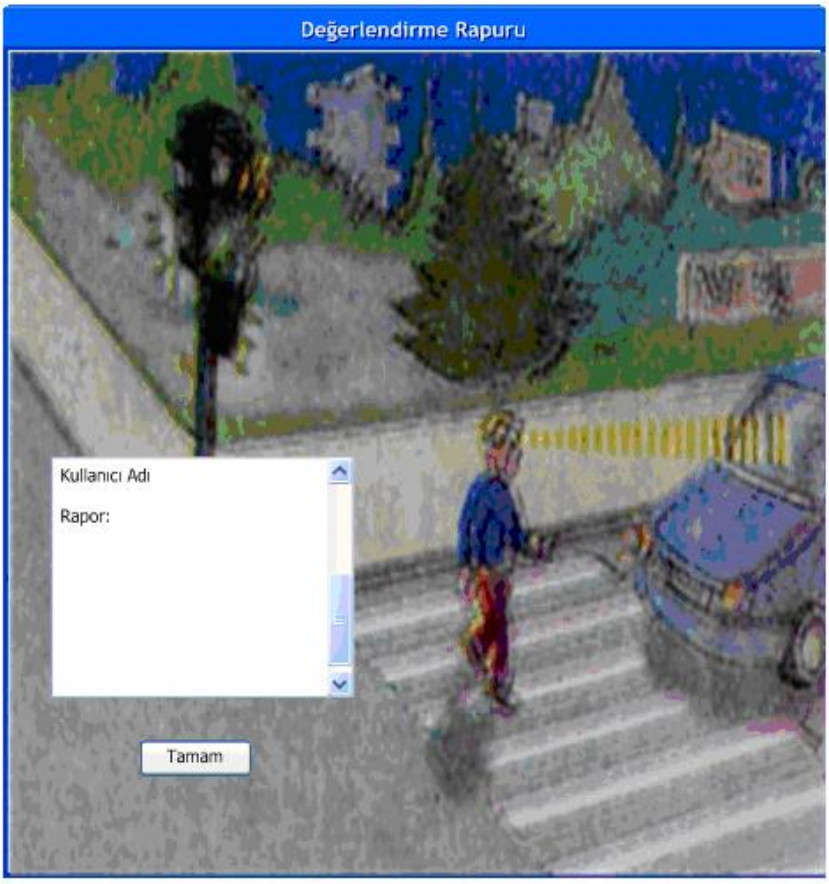
3.1 Player Main Menu Design

The below figure illustrates the login form and starts a game or an animation. Some user information, student id or name, will be taken. This information will be used in messages given to the student for encouraging. All animation and game files are kept in a directory, accessed from there. They are accessed by this menu.



Main Menu for Viewer

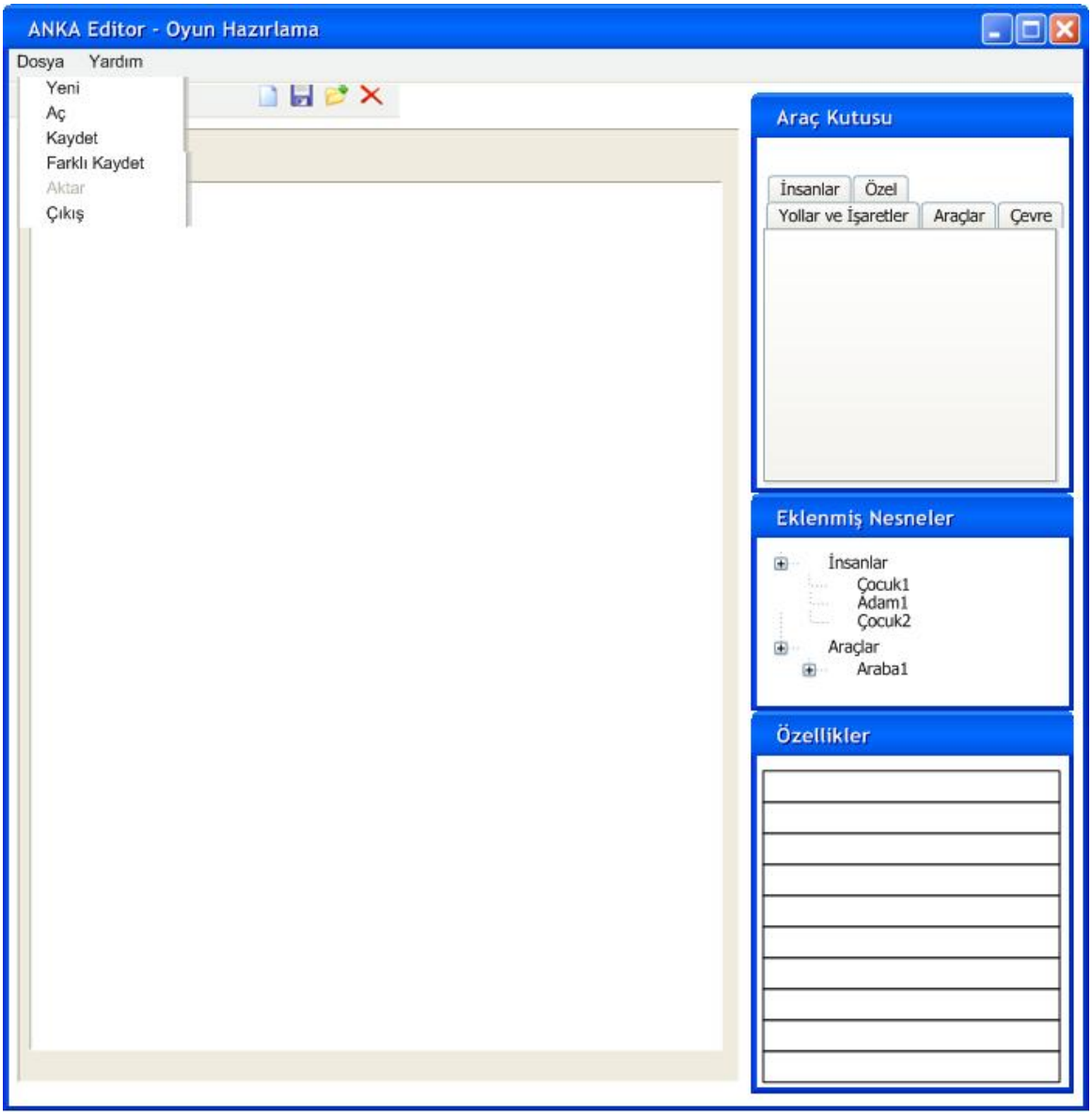
As we stated in our requirement analysis report when the animation ends, an information screen will appear which will tell the user about the point user has taken from questions. The below figure illustrates this screen. Within the evaluation report grade, user predefined comment, the mistakes can be shown.



Report Window

3.2 Editor Design

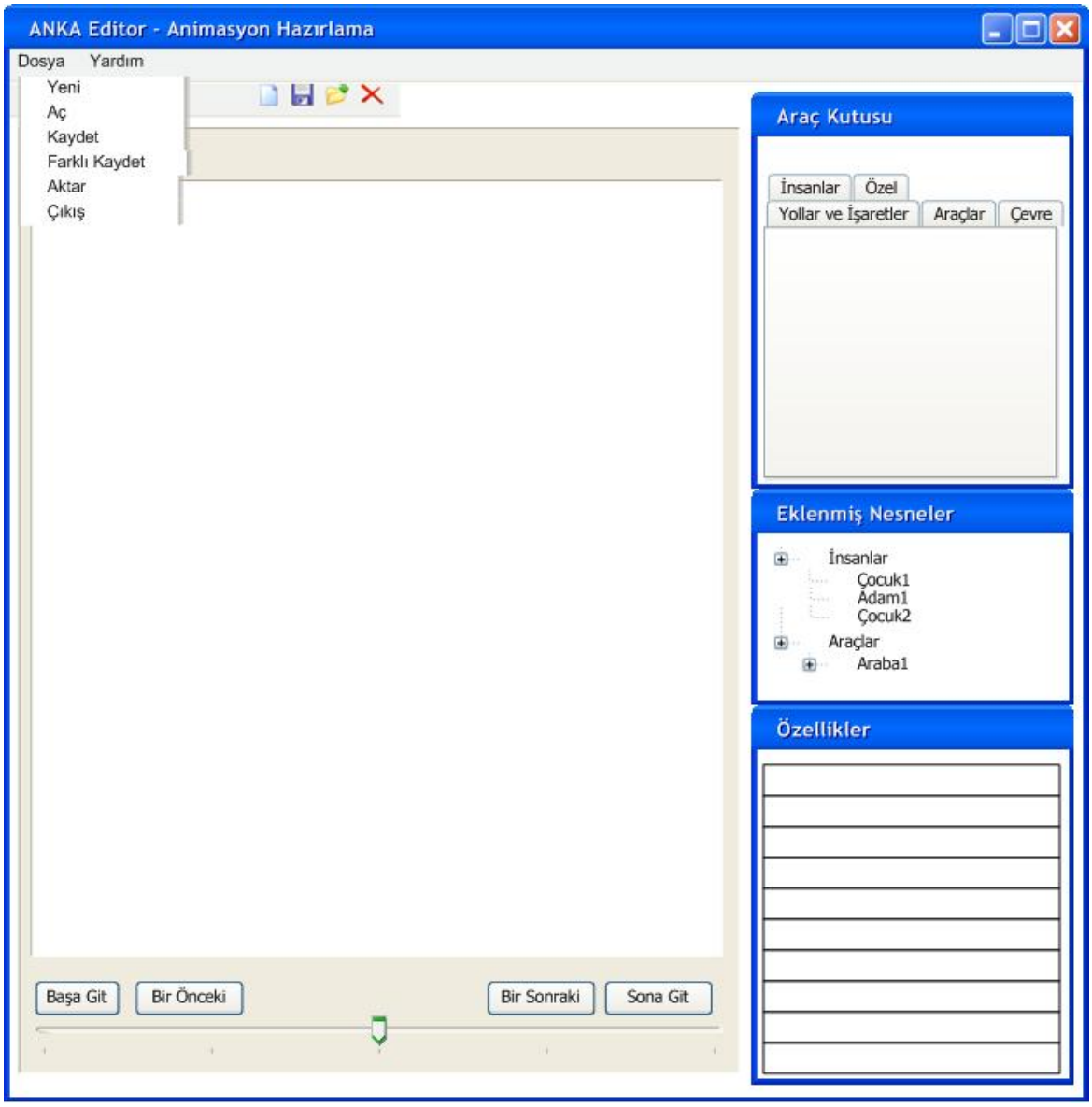
The below figures illustrate the general structure of our editor, and the main difference of two process modes (edit animation and edit game processes).



Game Editor

Instructor can make an animation or a game within a single editor. However, there should be a mechanism to know the current application on editor (i.e. whether the user is making animation or making game). We should know this because we give more rights to the user of the editor on making animation but limited rights on making a game. For example, with game editor user can assign a main character and give him/her start and end position (where he starts his action and where he finalizes it); with animation editor user just sets the actions of characters in the interval between start and end position; however with game we take this information from player.

Settings on such actions are done with properties subpanel, so it should be different for an animation and a game; and we should know the active process to make it different.



Animation Editor

There would be a ruler to cross the frames on 3D view subpart of the editor. This would not include rendering. It just changes the frames. So it uses only graphics and AI to show the objects in correct places. The ruler is going to be set only for the animation

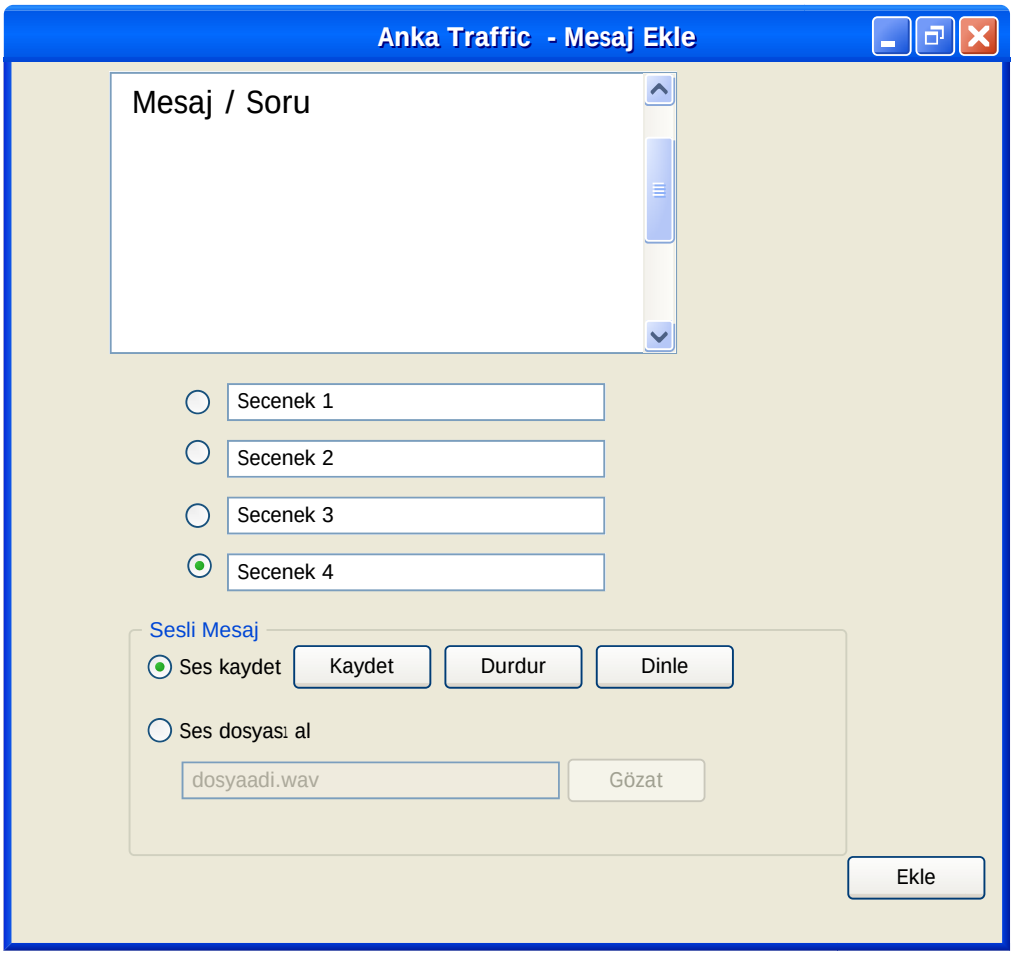
because the next frame is going to be determined by the player at the time of game play.

During the usage of editor, the user can add objects to the 3D view part by changing the camera view using the controls with the keyboard. These controls are:

- *CTRL + Arrow Keys* : to move the camera's look at position
- *Arrow Keys* : to change the camera view without changing the place, only by changing the angle

3.2.1 Adding a User Display Message

After selecting the frame, user can add display messages for students to help, to give a warning or to ask a question. When user wants to add a message following window opens:



To add a sound message user can either can record a .wav file by the check box 'Ses kaydet' or import from a .wav file by the check box 'Ses dosyasi al' by browsing the file.

Adding a sound is optional. However, all messages must have texts. These texts can be either questions or warnings. When the user wants to add answer options to the questions he/she can use the text boxes with radio buttons. He/She should choose the right answer by selecting the radio button next to the answer.

3.2.2 'Yeni' tool-command

If user has done some animation; decided to stop working on it and wanted to prepare game, he changes the animation and game selection on editor. We attached this selection from toolbar on 'yeni' tool-command. Clicking on 'yeni' causes a pop-up menu which includes 2 options: animation or game. Any other selection causes the editor to be reset and the 'mode' variable to change. Before the reset, user is asked for savings, and then the initial configuration for game is set. The same editor exists for animation and game. The difference is the scene on its sub-panels. For example, on a change from game to animation 3D view subpart is cleared.

3.2.3 'Aktar' tool-command

This tool command is active on animation mode. It is used for converting animations to '.avi' format.

3.2.4 'Aç' tool-command

Lists the saved animation and game files. User makes an animation or game selection. It works like 'Yeni' tool-command. But this time the editor is set with respect to the selected animation or game instead of the editor's initial configurations.

3.2.5 'Araç Kutusu' Sub-Window

For each 3D model we designed, there is a thumbnail image on the panel. These images are grouped in tabs. They are grouped into human, special, vehicles, environment and roads and signs objects. The objects added are seen from 3D view part of editor.

3.2.6 'Eklenmiş Nesneler' Sub-Window

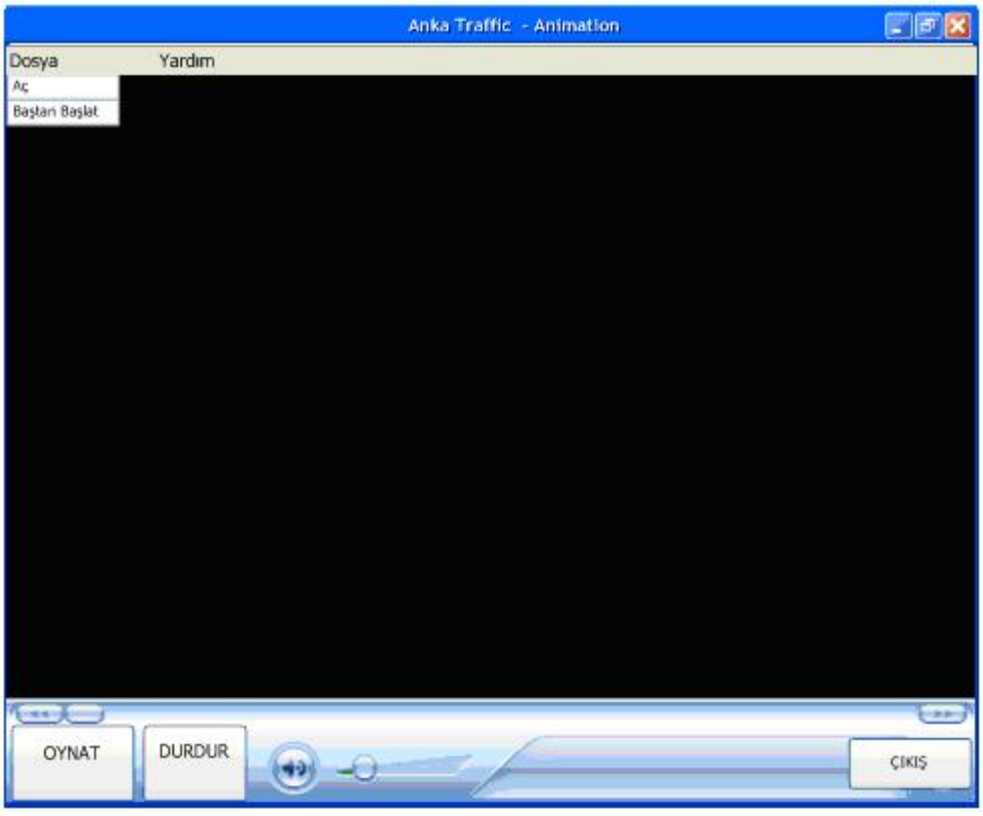
This sub-window shows the objects that are added before. Using this sub-window and clicking the previously added objects, user can select the object and view the properties in the "Özellikler" Sub-Window.

3.2.7 'Özellikler' Sub-Window

This sub-window shows the properties of the selected object. Thus, the user can modify the object easily.

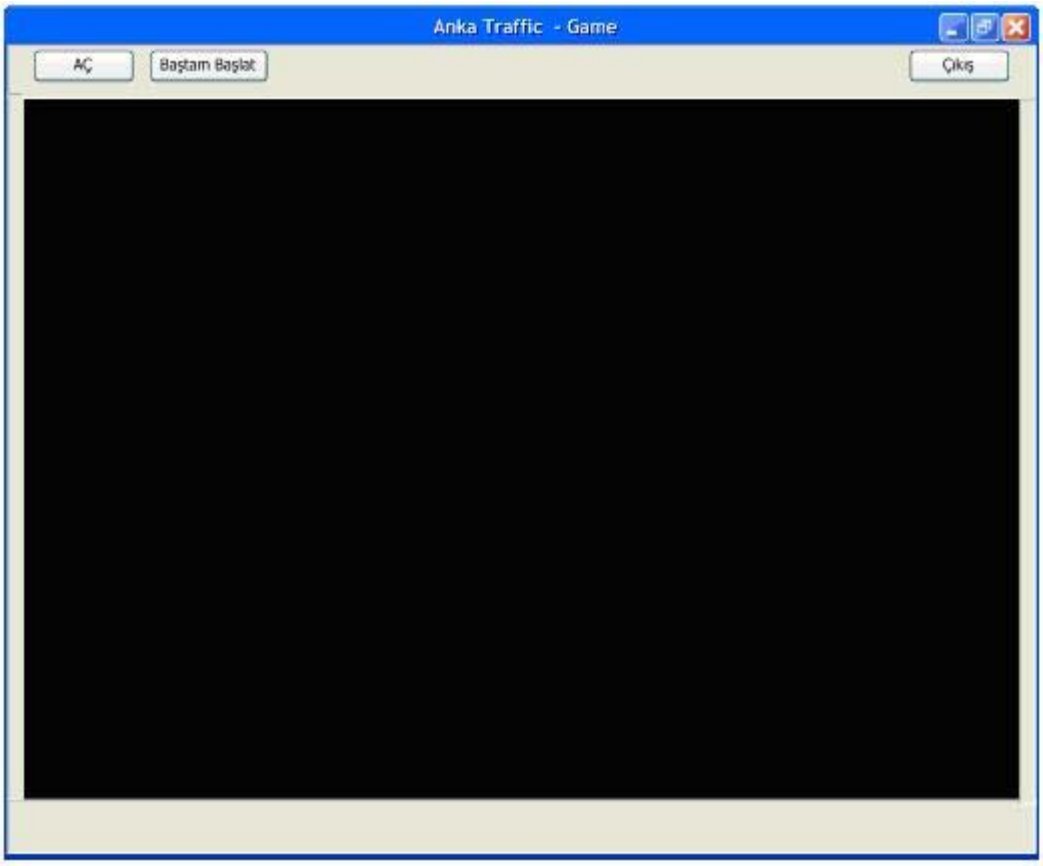
3.3 Viewer Design

The below figure illustrates viewing an animation option of our tool. The user first selects the animation from available ones by clicking on the "Aç" tool command. After the selection the animation is ready to run by clicking on the "Oynat" button. At any time during the animation the user can stop the animation by using the timeline bar. There is a "Çıkış" animation option for exiting.



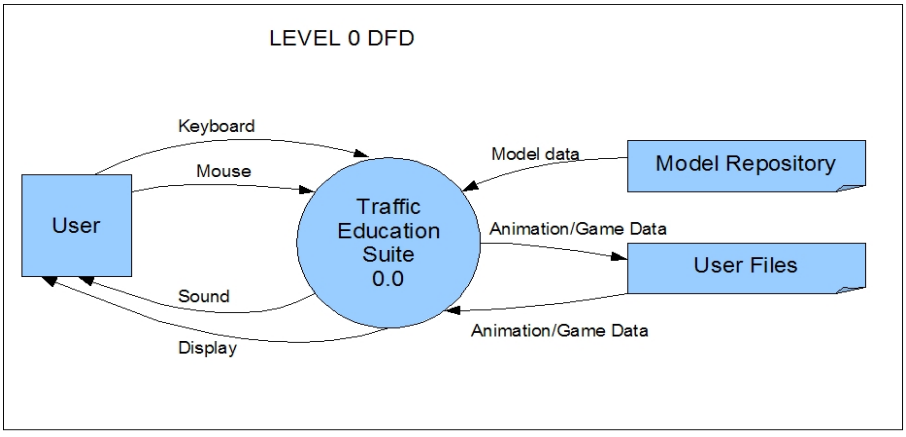
Animation Viewer

The below figure illustrates playing the game option of our tool. The user first selects the game from available ones by clicking on the "Aç" tool command. After the selection, the game is starts to run. There is a "Çıkış" animation option for exiting.



Game Viewer

4 SYSTEM OVERVIEW

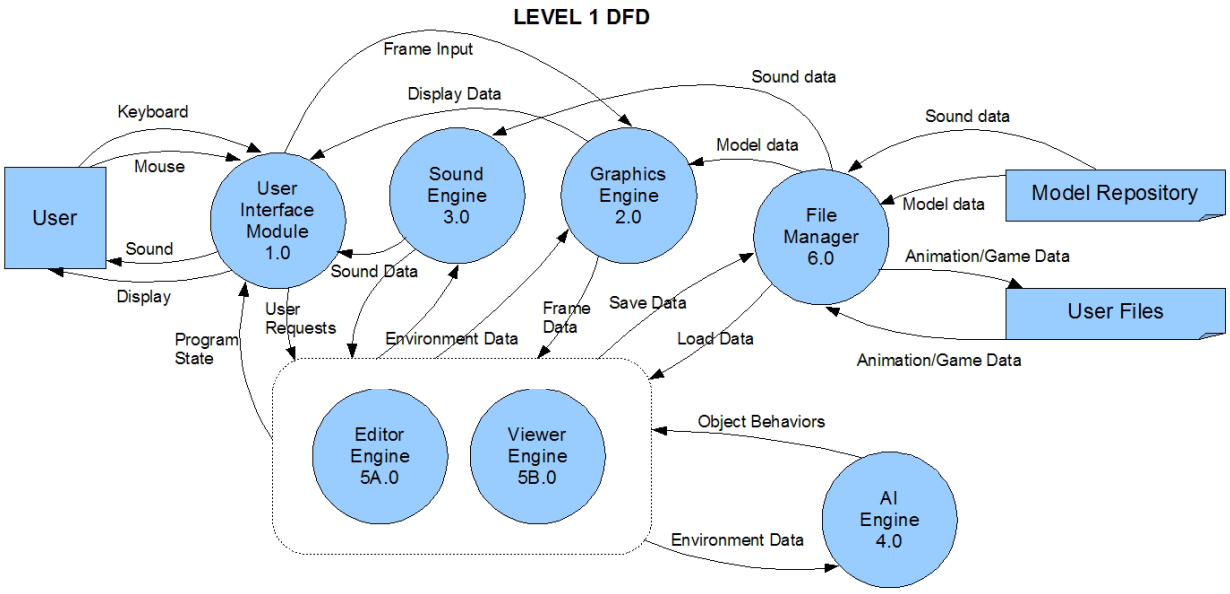


Our application is composed of two parts: Editor and Viewer. The difference between these parts is one of them is used for creating animations and games; the other is used for viewing these animations and games. Beside this difference, they can be thought as almost same in the means of program's internal processes.

As we see in the above graph, our only external entity is the user of the program. The user sends its requests by mouse and keyboard. And he/she gets the output from speakers and monitor of its computer. The details of the user interactions are described in user interface section of this document.

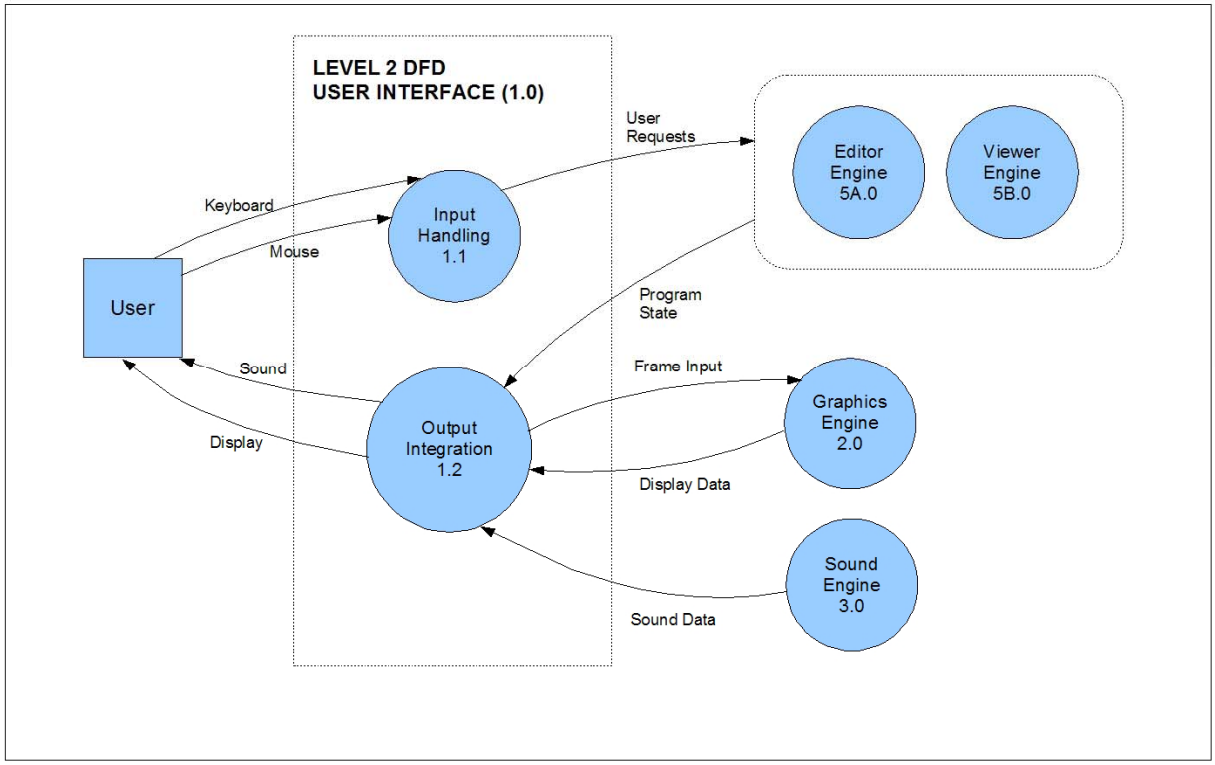
Also we have a data repository to hold 3D models and sounds (environmental sounds) that will be used in the game. The user files are simply the data that user saved for later use (to edit in the editor again, or to view in the viewer). It is just the working folder of the user and it can be anywhere in the computer or on a removable media. All file formats are described in file formats section of this document.

The internal logical architecture of the program can be represented as below:



4.1 User Interface (1.0)

We are using CEGUI (Crazy Eddie's Graphical User Interface) manager for user interface. Actually we planned to use windows API, but later we see that CEGUI is very suitable for using with our graphics engine (OGRE3D). In CEGUI manager, the overlay of the interface is designed before coding the program, then this overlay is imported in the initialization of the program.



4.1.1 Input Handling (1.1)

Input handling is done by callback functions of user interface elements. These callback functions are subscribed during initialization of the user interface. The mouse and keyboard events will be produced by frame listener binded with OGRE window and these events will be consumed by Editor/Viewer Engine.

4.1.2 Output Integration (1.2)

Output integration is simple. The windows of user interface will be showed according to program state. And one of these windows will be binded to Graphics Engine (OGRE3D). The integration of sound with display will be automatically done in hardware level (play functions of sound engine, OpenAL, gives sound directly to user).

4.2 Graphics Engine (2.0)

As we stated before, we will be using Ogre3D graphics renderer as our graphics engine. The ogre engine is used by creating an application object of Ogre class and calling its go function. Then rendering process will continue until we close it. So we will initialize our program (load necessary data) before calling 'go' function. Rendering can be stopped in the frame callback functions of Ogre frame. These callback functions are called after every frame is rendered and before every frame is rendered.

Actually our user interface will be an Ogre frame. Our 3D view will be a child frame of the user interface frame. When 3D view frame interacts with mouse, its frame listener will be activated and response the action.

4.3 Sound Engine (3.0)

We will be using OpenAL as our sound engine. OpenAL is a powerful tool for using sound in applications (like games), but we are not making a game and do not need the complex features of the engine. We need the engine only in two places.

One of our usages is giving environment sounds to the user so that a child playing a game will be able to hear the cars around. This feature will be done simply by calling play functions of OpenAL. The car and other environmental sounds will be hold in model repository.

The second is the message facility of our editors. The teacher will be able to add sound clips to screen messages. This feature will be maintained by calling record functions of OpenAL. The sound data will be saved into the animation/game file.

We are not interested in other features of OpenAL (like 3d sounds, complex sound effects, etc). We simulate the 3d sound effect by calculating the distance between the cars and user, and adjusting volume with respect to distance.

OpenAL uses OpenGL like engine. After initialization in the beginning of the program, it starts playing when we call 'play' and stops when we call 'stop' function. So we do not need anything to integrate it with Ogre, but calling the functions in idle times.

4.4 AI Engine (4.0)

We will use OpenSteer library for artificial intelligence purposes. OpenSteer is a specialized library for determining steering behaviours of automatic objects. We will use it for determining car behaviours and pedestrian behaviours (other than the main character in games).

Initially we will define a terrain map (terrainMap class) in the engine. This terrain map determines where the objects can go where they can not go. Actually this map can be thought as a virtual model of our map data. Later we will add the road data to this object by the Route class of the engine. This route data determines the normal path of the objects (the roads of our animation/game map). Our cars and humans will be inherited from SimpleVehicle class of OpenSteer and will be binded to the terrain map. Later in game or animation, at each idle time interval, we will predict the next behaviours of all objects by calling this terrain map.

In game editor, we will not use artificial intelligence. In animation editor and animation viewer, we will use it for calculating future frames (since we hold the behaviours only, we need to calculate every time we watch the animation. In games, the random movements of pedestrians and cars in traffic are calculated by the engine.

4.5 Editor Engine (5A.0)

Editor Engine is the main part of our program when the program is started in editor mode. All of the calculations that are done is in this part and later passed to the other engines.

Editor engine is not defined as a class or function. It is the combination of all other function calls before rendering, and during rendering. So we can say that it is our entire program other than external libraries (sound engine, graphic engine, AI engine).

4.5.1 Program Core (5A.1)

It is the part of the program that holds all data of the program, like current animation or game data, time information, program states. When the program starts, core is initialized. When new data exists, the core must be updated. It is the combination of global variables and functions that manipulates this global data.

User interface learns what to show by interacting with this part (program state). User interface callbacks (user requests) changes the state by triggering update functions.

When user wants to save or load data, File Manager interacts with this part to get or set the data.

4.5.2 Data Calculations (5A.2)

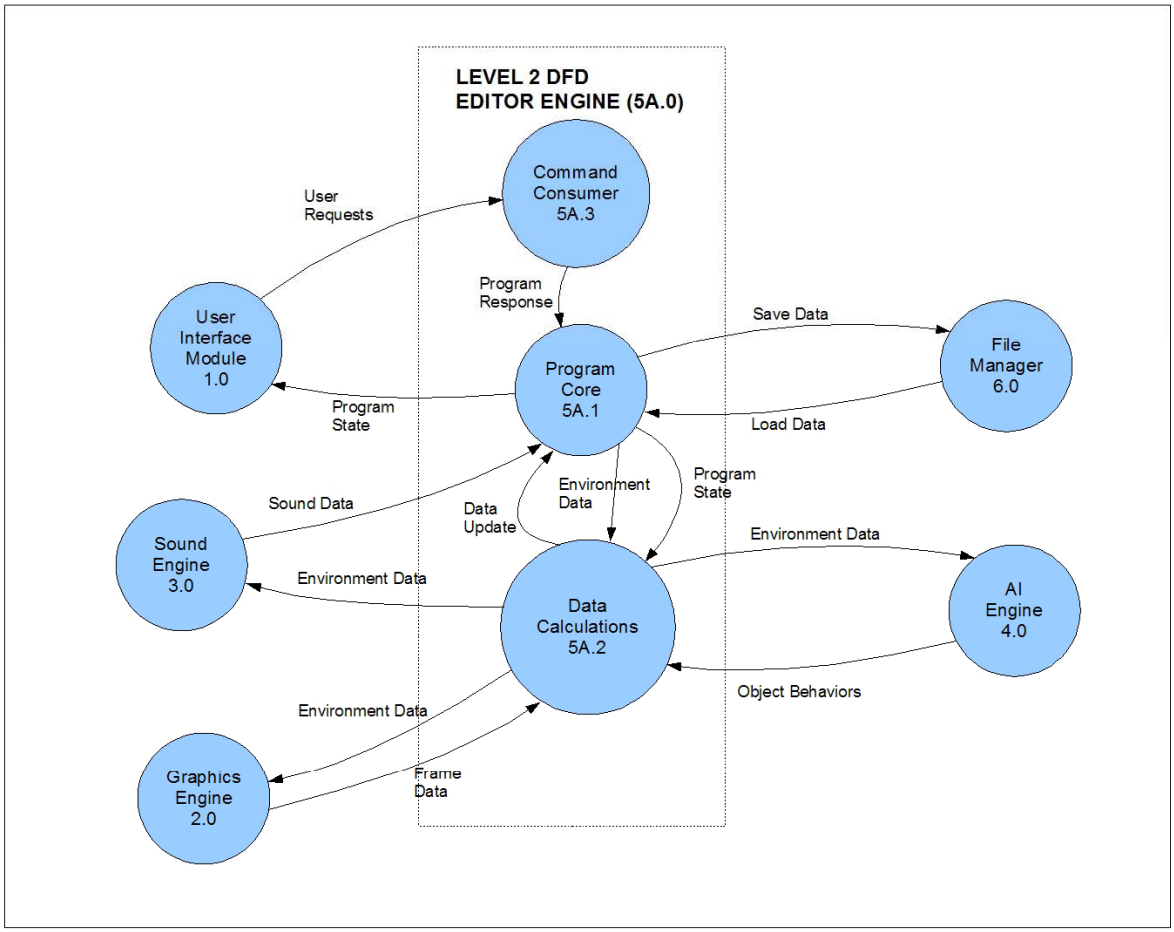
At every step of our program, like user actions and frame rendering, we need to update positions of the objects and the data related with animations or games. This part of the program calculates the necessary changes, updates core data, and triggers the necessary functions from other engines.

After every frame, the program data (program state and environment data) is taken from program core. Then, all object data is passed to AI engine and future movements (object behaviours) are predicted for objects. Later, if necessary, program core is updated (data update). Finally, the changes in graphic and or sound engine is calculated and passed to the sound engine and graphic engine.

This part contains all of the functions, examinations that are done at every idle time interval (in frame callbacks of graphic engine).

4.5.3 Command Consumer (5A.3)

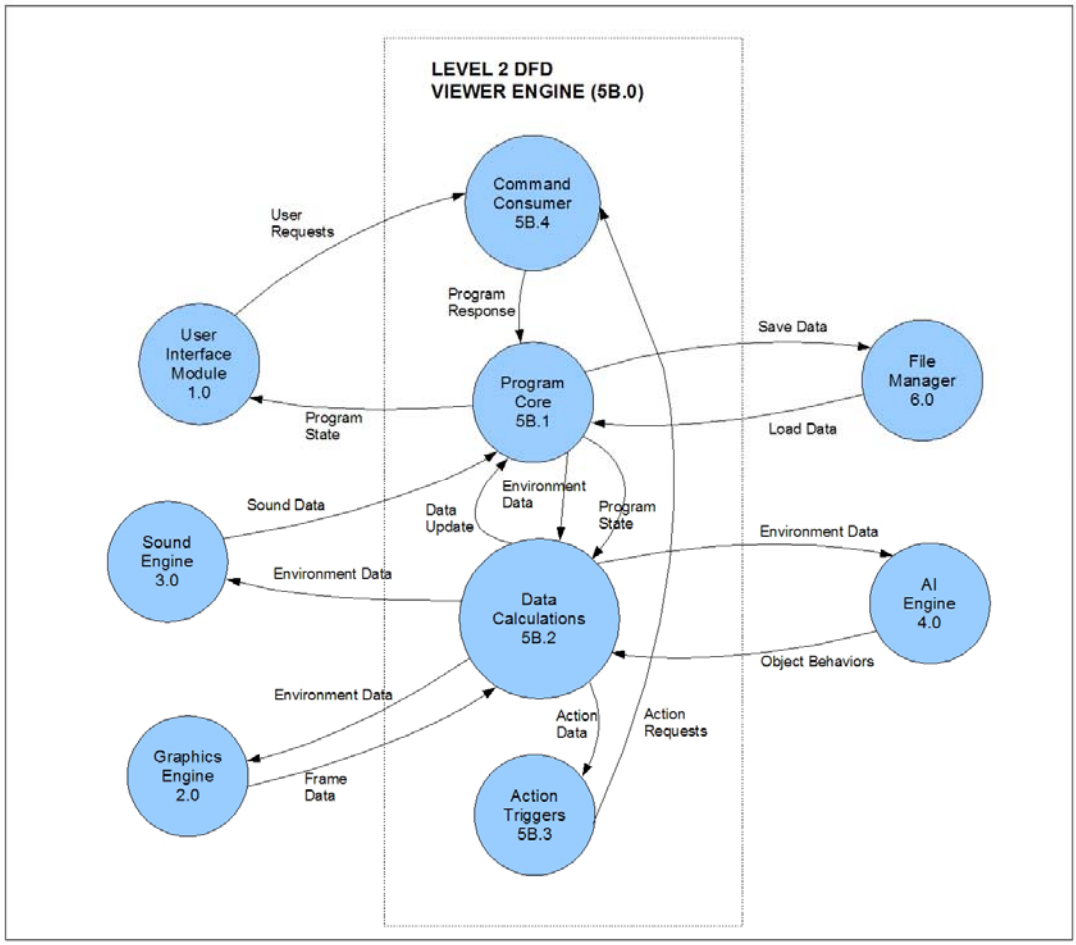
The user commands that are caught by user interface are processed by this part. It will run as an independent thread so that for complex actions, our program doesn't freeze. It will process the commands one by one in the order that commands created. The necessary changes will be passed to the program core.



4.6 Viewer Engine (5B.0)

This part is almost same with editor engine. The only difference is action triggers (5B.3) module.

In games, the user (child) will wander in the virtual map (our 3d world). The main character of the game (the object the user controls), may encounter some special conditions (like violating a traffic rule or stepping on a user defined area, which is defined by teacher by adding blue area to the map). Also in both games and animations message screens will appear and wait for user to interact with these screens. In both of these two conditions, the program behaviour must be changed to reflect the effects of these conditions. Action triggers deals with these situations.



4.6.1 Action Triggers (5B.3)

In viewer engine's data calculations, different from editor engine, all above conditions must be checked by calculating the main character's position (in games) and time (in animations). If any found, this is passed to action triggers (action data). Then, the necessary commands are produced to be consumed by command consumer (5B.4).

4.7 File Manager (6.0)

File manager is the class that contains our file related mechanisms. It will be implemented as a namespace in our application. The file formats are not described in this part.

4.7.1 Save (6.1)

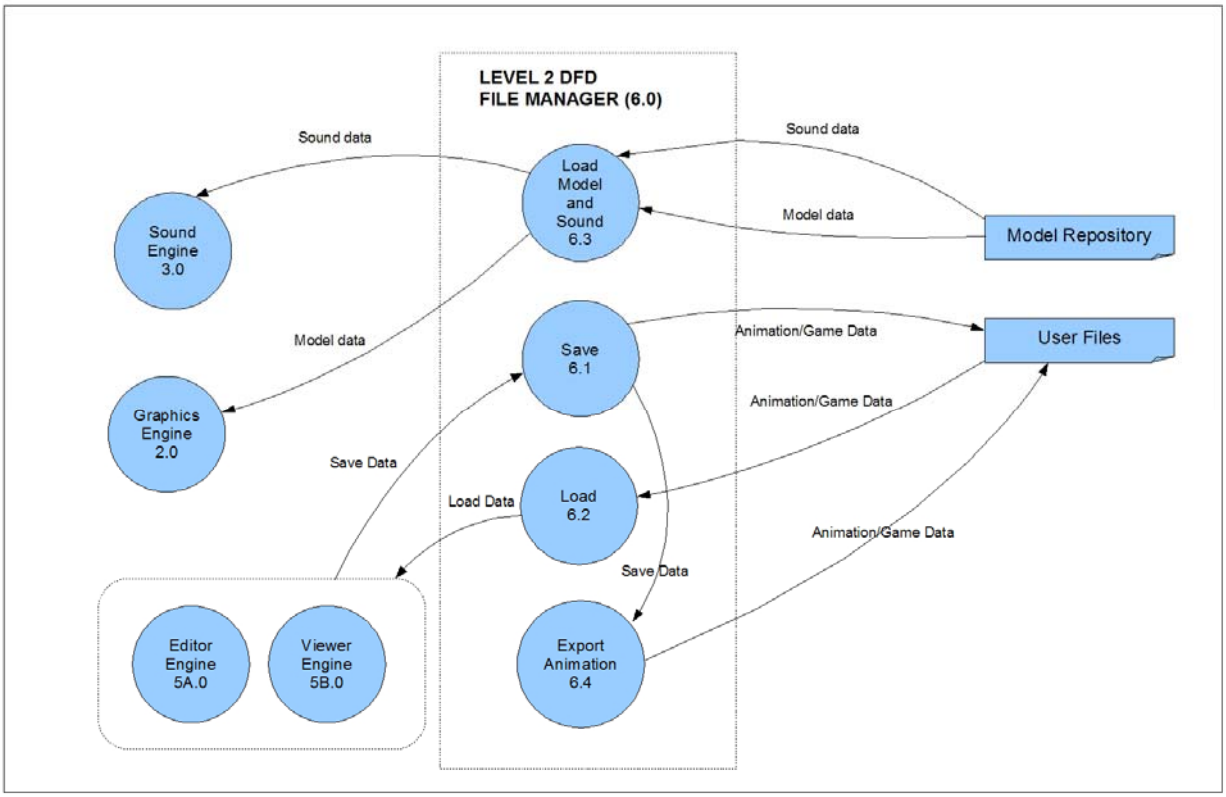
When the user wants to save a game or animation, it is maintained by this module. Also when a game or animation ends (in viewer mode) the log of the game or the animation is saved to the default folder by this module.

4.7.2 Load (6.2)

When the user wants to load a game or animation, it is maintained by this module.

4.7.3 Load Model and Sound (6.3)

This function loads a model or a sound from model repository. This function is needed by graphic engine and sound engine.



4.7.4 Export Animation (6.4)

In animation editor, user is able to export animation. In this function we use the audio and video libraries of 'ffmpeg' multimedia system. The libraries 'libavcodec' and 'libavformat' libraries of 'ffmpeg' provides necessary codecs and tools to combine screenshots taken from graphics and then combine it with the sound recorded by sound engine.

5 DATA DESIGN

5.1 Classes

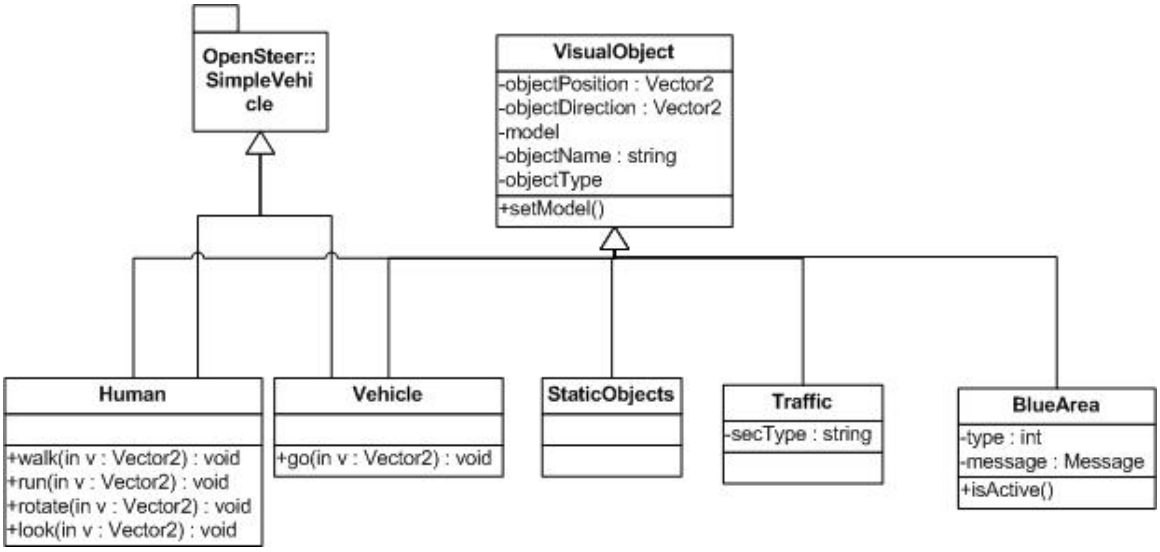
In this part of the document, the data structure we are going to use in our project is explained. Beside the classes we will use from libraries we have designed many classes. All of these classes have 'set' and 'get' functions for every attributes which are not told here. Other properties of these classes are explained below.

All other relationships between classes are displayed in a diagram which is at *Appendix A.*

5.1.1 VisualObject Class

This class is mainly the base class for our objects. All other objects except "Road" object is inherited from this class.

It stores the main properties of an object like the position and direction of our object in our 3d world. Also, model filename is stored here. 'objectName' is unique for every object. The instances of human, vehicle, staticObjects, Traffic, or BlueArea classes are distinguished by 'objectType'.



5.1.2 Human Class

This class uses multiple inheritance and is inherited from our VisualObject class and OpenSteer's SimpleVehicle class. SimpleVehicle class is used for Artificial Intelligence and guessing the next positions of the objects, thus we had to inherit our class from that. Moreover, this class has some extra methods to control our object.

5.1.3 Vehicle Class

This class also uses multiple inheritance like Human class. It is inherited in the same way like Human class. In this class we have an extra method to drive our vehicle to the position we give.

5.1.4 StaticObjects Class

This class is inherited from VisualObject class and defines the static objects in our program. These objects can be listed as: buildings, trees, etc. We do not need any special method for this class because they will be positioned and will not move until users decide to edit the object.

5.1.5 Traffic Class

This class is also inherited from VisualObject class. We designed such a class in addition to designing StaticObjects class since some nonmovable objects needs some special method to apply a traffic rule. So, addition to the properties of the VisualObject class, it also stores the information of which traffic object it belongs to. This information is stored in 'secType'. We differ traffic objects like roads, sideways, crossways, traffic lights, traffic signs etc. from other objects and from each other with respect to the 'secType'. So we apply different rules to different objects. Rules are handled with AI.

5.1.6 BlueArea Class

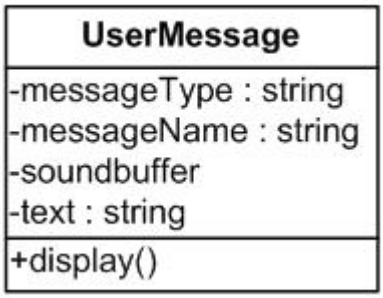
Objects of this class are used for bounding to some places in map. This bounding place is set in base class VisualObject and taken in 'ObjectPosition' variable. This bounded area also bounds to a display message which is defined by the educator. Objects of this class are not visible to students neither in games nor animations but visible to the educators using the editor. In viewer mode if user places it's hero into a blue area, then the corresponding message would be displayed. This message could be a question to the student, a warning, a help statement or any other comment given from the teacher.

To determine the next behaviour of the program we take a 'type variable' of the BlueArea object which can be either 'start', 'end', 'message', 'question'. Since for example if the bounded message is a question, the program will wait in a predetermined time to take the answer, or if blue area corresponds to 'end', then game will end.

'isActive( )' checks whether it is the time for the BlueArea object to display the message or not.

5.1.7 UserMessage Class

This class is the one which will be used to display messages to the user. A display message can also be associated to a sound message. These messages can be warnings, helps or questions.



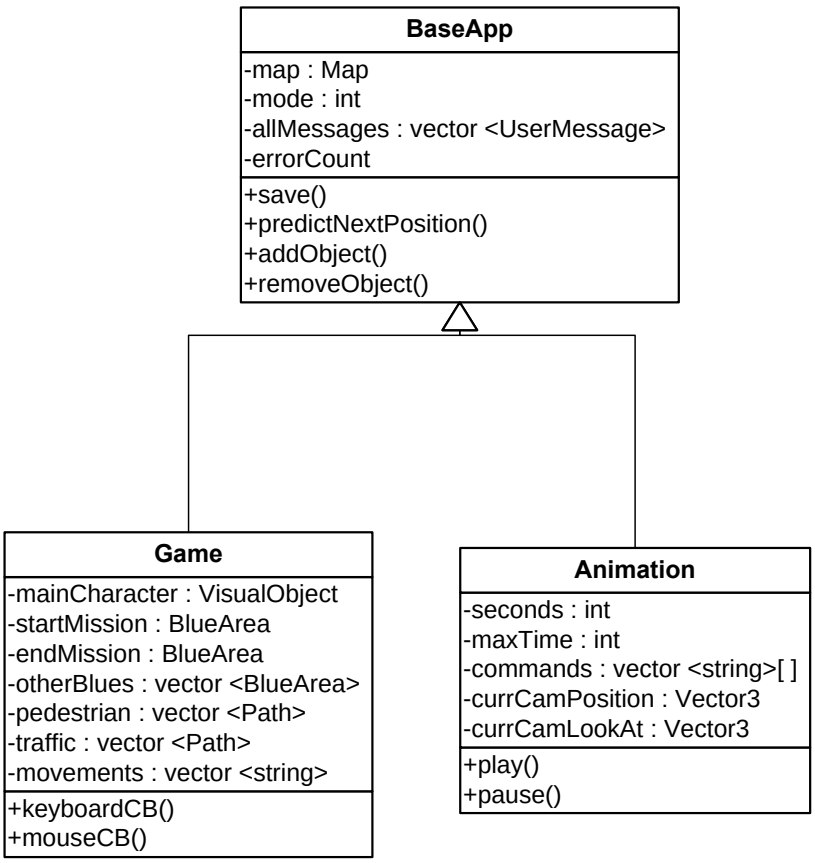
'messageType' defines the type of the message which can be either a "question" or "simple". "simple" type messages are warnings and helps.

'soundbuffer' is the buffer which will be used by openAL for storing the sound either recorded by the educator or imported from a '.wav' file. Its initial value is 'null'. Thus there cannot be any sound messages without texts. Importing from a .wav file is done by browsing in the file system and loading this file in to soundbuffer using the loadSound() method of file manager class.

'display( )' method is the method used for displaying the messages.

5.1.8 BaseApp Class

This class is the base class for our 'Animation' and 'Game' classes. It cannot be created without using these two classes. It stores the common properties of the other two classes like 'map' and 'messages'. 'mode' stores the value that tells whether the objects are in "viewer" or "editor" mode.



It has a 'save ( )' method for the user to save the animation or game he/she created. This method calls FileManager's related method depending on the object to be saved.

'errorCount' is stored to give a report at the end of an animation or game about the progress of the student.

'predictNextPosition( )' is the method that uses AI with OpenSteer functions. It gives us the next positions of 'Human' and 'Vehicle' objects.

'addObject( )' and 'removeObject( )' methods are defined for user to add or remove objects in editor mode.

5.1.9 Game Class

This class is inherited from the 'BaseApp' class. It defines the main structure of our game in our program.

'mainCharacter' is the object which will be controlled by the student during the game.

'camMode' is the mode of the camera view. There are three modes for the camera which can be listed as: first person view, third person view and a view far from the player.

'startMission' and 'endMission' define the start and end of the game. Game starts with the player positioned at 'startMission' BlueArea. When the player reaches the 'endMission' BlueArea the game ends. 'otherBlues' vector defines the other BlueArea objects attached to messages that player can face during the game.

'pedestrian' and 'traffic' vectors control the density and the flow of vehicles and people other than the player. They are controlled by AI on a given path that we used different vectors for them.

'keyboardCB' and 'mouseCB' methods control the movement of 'mainCharacter' according to the input by the player. They only work in viewer mode of the game object.

'movements' vector composed of strings and used to detect the consecutive movements. Control of these movements can be explained as:

In order to detect correctness of the action of the hero we must detect whether he/she obeys the defined rule or not. We have developed the following detection way:

- We use a list structure ('movements') in order to keep track of the last consecutive movements of the hero.

- When we need to evaluate the correction of the movement, we will examine the list that we have defined and update at each movement and compare with the correct action order defined related to the current rule.

For example, we will solve the problem of movements in a fixed area by this approach which can be detecting the order of looking left and right.

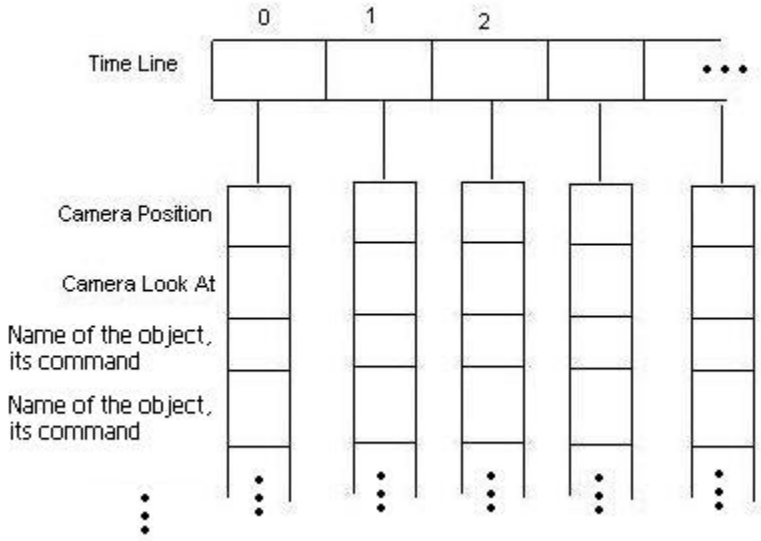
5.1.10 Animation Class

This class is also inherited from BaseApp class just like Game class. It defines the main structure of the animations in our application.

In addition to the attributes of BaseApp class there is 'seconds' attribute for us to control the timeline in our application.
'maxTime' is the maximum duration of the animation given by the user at the beginning of the creation of the animation.

'commands' is an array of string vectors and will store the movements of the objects in our program. The structure of this array is:

- Length of the array is defined according to the 'maxTime' given by the user
- Each vector defines the frame at that time of the animation
- Each element is a vector of strings
- At the beginning of each vector, first two strings are 'camera position' and 'camera look at' at that time
- Other strings will be composed of the name of the object, name of the movement method of that object and parameters of that method in order, ending with a delimiter.

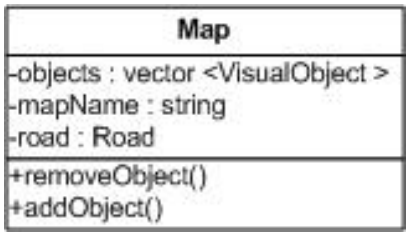


'commands' vector

'currCamPosition' and 'currCamLookAt' are Vector3 objects defining the camera view.

'play( )' and 'pause( )' methods will be used by the student during the viewer mode of the animation.

5.1.11 Map Class



This class stores the objects in our application. Both visual objects and roads are stored in a map object. It is used by BaseApp class.

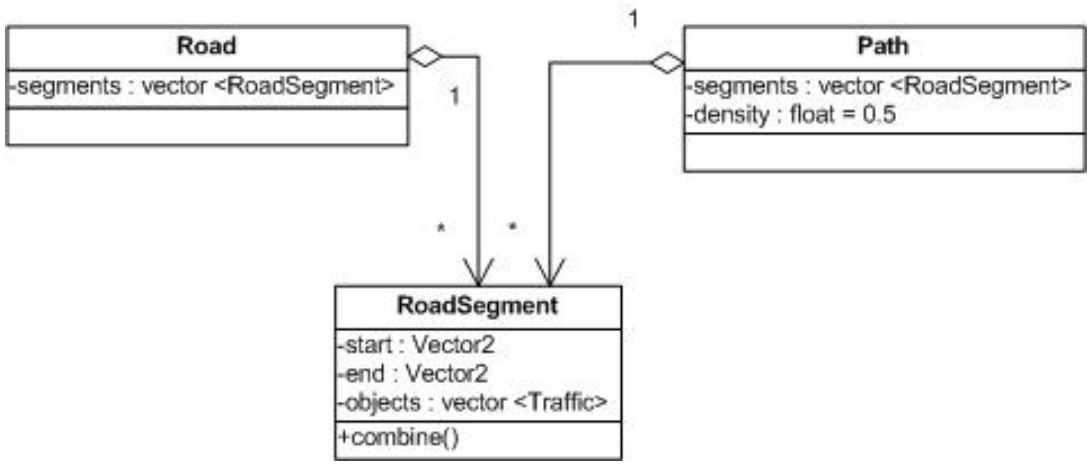
'objects' is the vector that stores the objects put by the user in editor mode.

'road' is the roads defined by the user in editor mode.

'mapName' is a unique name defined for the map.

5.1.12 Road Class

This class just stores the segments of RoadSegment objects in a vector. It stores all of the roads defined by the user in one object to be used by a map.



5.1.13 RoadSegment Class

This class is the main element of a Road object which is a segment composed of straight roads defined by the user. 'start' and 'end' values are stored with a Vector2 object, thus any road added to the map can be queried and combined with the 'combine( )' method.

5.1.14 Path Class

This class is specially designed for the AI to control the movements of objects in a given path. Just like the 'Road' class this class also uses RoadSegment objects to store the path. Moreover, there is a different attribute called 'density' to store the density of vehicles or pedestrians on the given path. This value is initialized to 0.5 and cannot be less than 0 or bigger than 1.

5.1.15 FileManager Class

This class is consisted of just methods. No instance of this class can be created. All methods are static.

'saveAnimation( )' and 'saveGame( )' methods are used for saving the animation or the game made by the educator in the editor. They are stored as '.anim' and '.game' files.

'loadAnimation( )' and 'loadGame( )' methods are used by both the editor and the viewer mode. When they are opened in editor mode, user can make changes on them and make a different animation or a game. On the other hand, in viewer mode, user can watch the animation or play the game. In viewer mode these methods are called by the button callback functions in the main menu or animation and game windows.

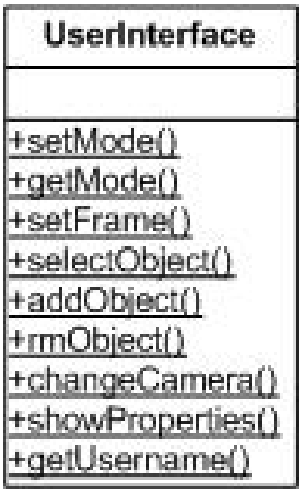
'exportAnimation( )' method exports the given animation to '.avi' format using OGRE functions and FFMPEG. Thus, the animation can be viewed without the need of our program in any video viewer.

'saveLog( )' method is used for saving the user's errors during the animation or game. These errors are saved to 'username.log' file in the directory of the animation or game.

'loadModel( )' method is used for getting the model from the model repository.

'loadSound()' method is used to load a given '.wav' file to the soundbuffer for usage.

5.1.16 UserInterface Class

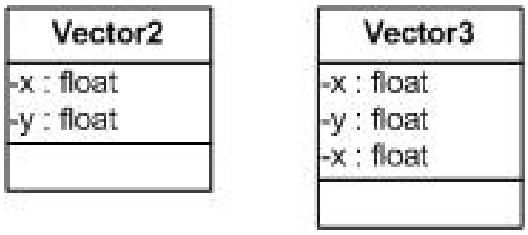


This class contains the main functions that GUI will use. No instance of that class can be created. All the methods of this class are static.

'setMode( )' and 'getMode( )' methods get the mode of the application and display the next window according to this mode. These modes are 'animation' and 'game' mode.

5.1.17 Vector2 Class

This class consists of two floats which correspond to x and z values in the 3D space of OGRE.



5.1.18 Vector3 Class

This class consists of three floats which correspond to x, y and z values in the 3D space of OGRE.

5.2 File Formats

5.2.1 Model Files

We will draw our visual object models in 3D Studio Max. Then these models will be exported to Ogre file conventions. Ogre has a tool for that purpose (3DS Max to Ogre exporter).

Ogre recognizes mesh data in ".mesh" files, skeletal animations associated with that mesh in ".skeleton" files, and material properties of that mesh in ".material" files. Exporter will create each file for our models.

Also, in editor mode (both animation and game), we will need a thumbnail of that model in our object toolbar (so that user can see what will be added to the environment). These thumbnails will be 100 * 100 pixel long ".jpg" files.

There must be a text file (with extension ".desc") for each model. This file will consist of one line only (the type of the object).

There must be another text file (with extension ".txt") for each model. This file will contain the text which will be showed in the toolbar (in editor mode) when user moves the mouse over them.

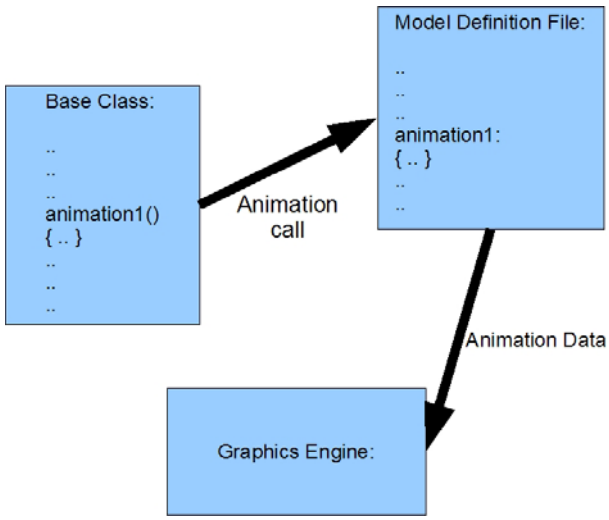
All models will have a unique name, and all these files will be named with that name (modelname.jpg, modelname.mesh, modelname.skeleton, modelname.material, modelname.desc, modelname.txt). Also all of these files will be put in a tar file with extension ".model" (modelname.model).

And all of models are put in a folder named as "models". In later post-release upgrades of the program, newer models also should be copied into this folder.

The relation between model files and VisualObject classes

As we stated above, we will read visual data of visual objects. In both editor mode and player mode, all these objects will be read from files and will be hold as objects of base classes. Distinction of these base classes is done according to animations and logical requirements of objects. These base classes are Human class, Vehicle class, StaticObjects, Traffic, BlueArea classes. Detailed explanation has been done about the classes in the previous section called Classes.

We can think these base classes as a virtual interface for arranging and animating these objects. All of visual objects will derive from one of these objects. Therefore, all models we will read from our model repository, must implement our animation functions. (Design of our models will be explained in another chapter.) Later in our application, we will trigger all these animations, by calling the member methods of these base classes. Here is a demonstration of this process:



Model files will only contain visual animations. Other calculations (such as transformations due to animation) will be in base class methods.

Human Class:

Human class is intended for any walking object on the map. Both children and adults are objects of this class. The detailed information about this class is given classes

section of this document. The models designed for objects of this class must implement these animations in 3D Studio Max:

- walk
- run
- rotate
- look (looking in a direction without moving)

Vehicle Base Class:

Vehicle class is intended for any vehicle that moves on the road. Buses, cars, children with bicycles are objects of this class. The detailed information about this class is given classes section of this document. The models designed for objects of this class must implement these animations in 3D Studio Max:

- go

5.2.2 Animation Files

Animations are hold in a ".tar" file called "AnimationName.anim". These tar file includes other files for the data of animation. These files are given below.

"AnimationName.txt": This file is a text file that contains the length of the animation in seconds in the first line.

"AnimationName.timeline": This file is a text file. It contains all animation details. The lines starting with "#N:" indicates that the lines below that line are the commands that must be executed in the Nth second, until the line starting with "#N+1:". Each line, after that line, is a command string. The structure of command strings are described in Animation class.

"AnimationName.map": This is the file for the map that animation is built on. The structure of this file is given below.

In addition to these files, there is a folder with name "Messages". In this folder, there are two files for each message in the animation.

"MessageName.txt": It is the text of the message with name messageName.

"MessageName.wav": If this file does not exist then it means that there is no sound recorded with that message. If exists, it is the sound clip associated with that message.

5.2.3 Game Files

Games are hold in a ".tar" file called "GameName.game". These ".tar" file includes other files for the data of game. These files are given below.

"GameName.txt": This is a text file. The first line of this text file includes the name of the object, which is set to be the main character in the game. The second line of this folder is the name of the BlueArea object in which the main character of the game starts playing. The third line of this folder is the name of the BlueArea object which the game ends when main character steps on.

"GameName.map": Exactly same as animation case.

"Messages" folder: Exactly same as animation case.

"GameName.blue": This is a text file. Each line describes a blue object. In each line there are numbers and string which describes the object in this order (ObjectName, ObjectPosition, type, Name of the bounded message object)

"GameName.pedestrian": This is a text file. Each line describes a path. First number is the float value which describes the density of pedestrian traffic. The numbers coming after are the starting and ending vectors of road segments. There can be multiple segments for a path.

"GameName.traffic": This is a text file. Each line describes a path. First number is the float value which describes the density of car traffic. The numbers coming after are

the starting and ending vectors of road segments. There can be multiple segments for a path.

5.2.4 Map Files

Map files are text files. In the first part of these files, there are the descriptions of the objects (visual objects with type human, vehicle, and static). Each line indicates only one object. The description of the objects includes these values in a plain string format: (name of the object, type of object, name of the model, object position, object direction).

After objects are done, there will be an indicator "#", which means that road data will come after. The road data is same as pedestrian and traffic files described in previous section.

After road data is done, there will be another "#", which means traffic objects descriptions will come after. These types of objects are described similarly with human or vehicle objects: (name of the object, type of object, secondary type, name of the model, object position, object direction).

5.2.5 Log Files

Log files are the files created after an animation or game is viewed in viewer mode. These are recorded in the same folder with the animation or game. They are simple text files, includes the number or traffic errors, or wrong answers to questions in game or animations. The name of the files are made with the username that user has entered at the beginning of the program. 'username.log'

5.2.6 Storing Files in ".tar" Format

As we know, there is no standard tool for compressing/extracting tar files in windows. We will use the libraries of 7zip program for all archiving purposes. Although it supports several formats, we will be using it for just tar/untar files.

Additional information about 7zip can be found in: http://www.7-zip.org/

6 SEQUENCE DIAGRAMS

6.1 Sequences Related to User Interface

Below are definitions of some typical types we used in our sequence diagrams.

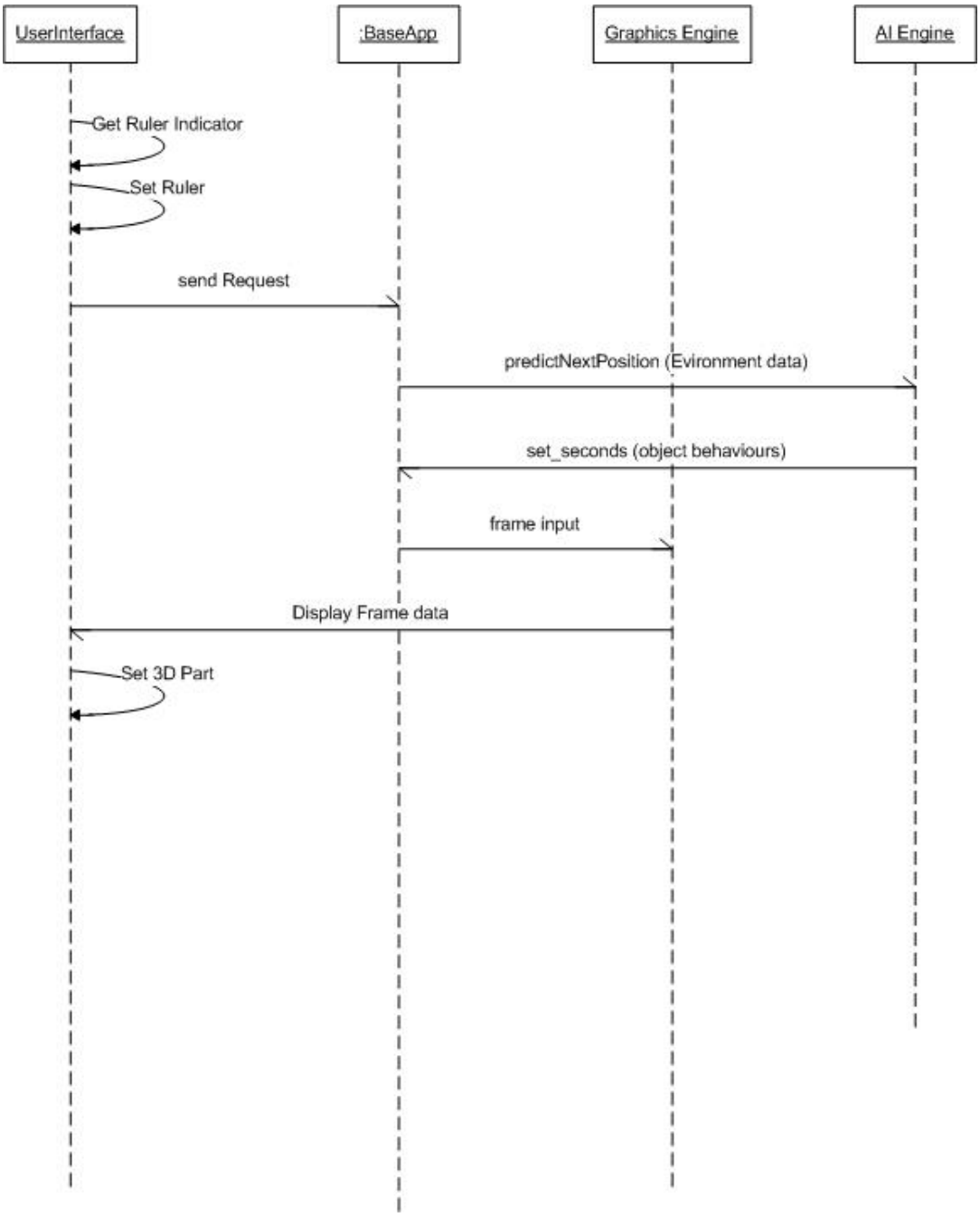
User Interface: It is just user-computer interface described at GUI design part.

A/G File **:** An animation or game file in our own file format.

BaseApp **:** An animation or game object described at class diagrams.

6.1.1 Ruler

As described at GUI design part there is going to be a ruler to cross the frames on 3D view subpart of the editor. When GUI detects any attempt to change the ruler the new position of the ruler is set on the screen and corresponding frame is loaded to 3D part of the editor by using graphic engine and AI (AI decides the correct positions of objects).

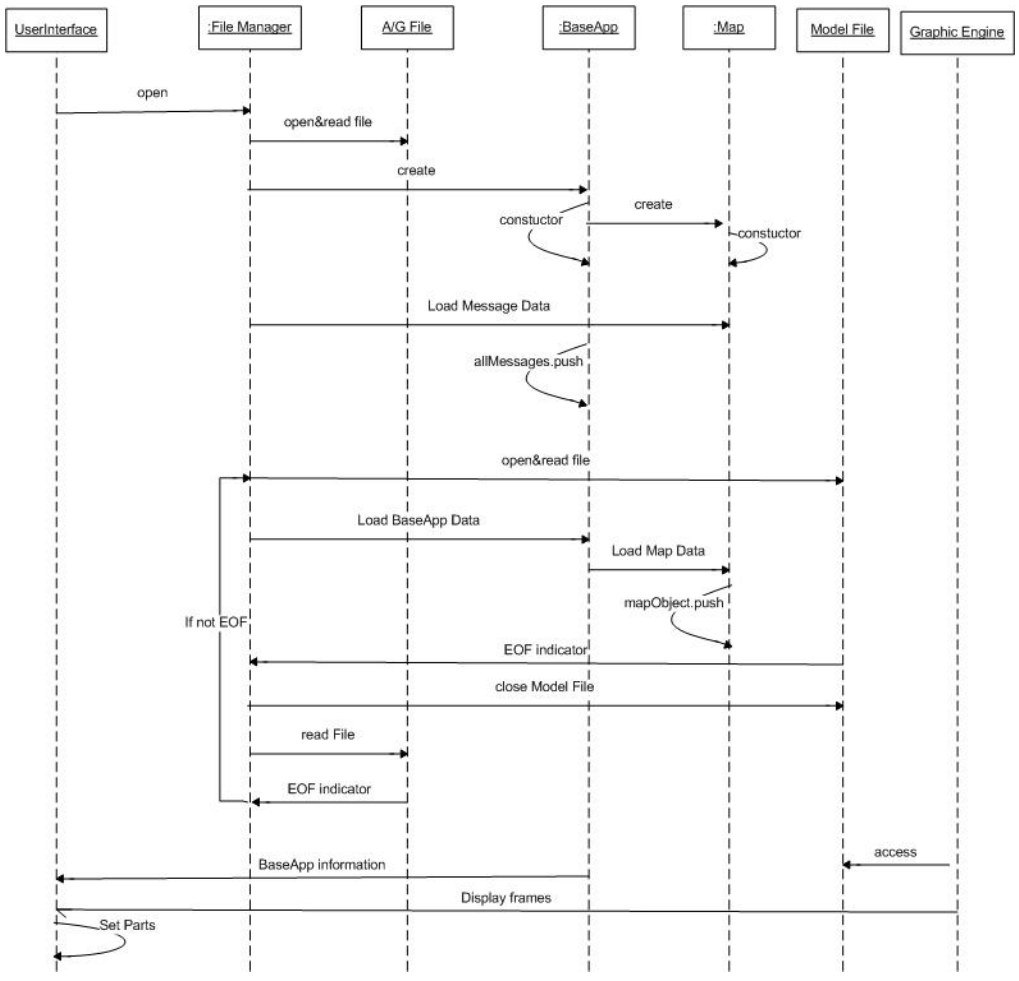


Ruler Sequence Diagram

6.1.2 Load Animation / Game

When GUI detects a user press 'Aç' tool-command as shown on the figure at GUI-Editor Design part , it directs the request to File Manager class. File Manager opens and reads the desired A/G file. Then a BaseApp object is created. The order we load

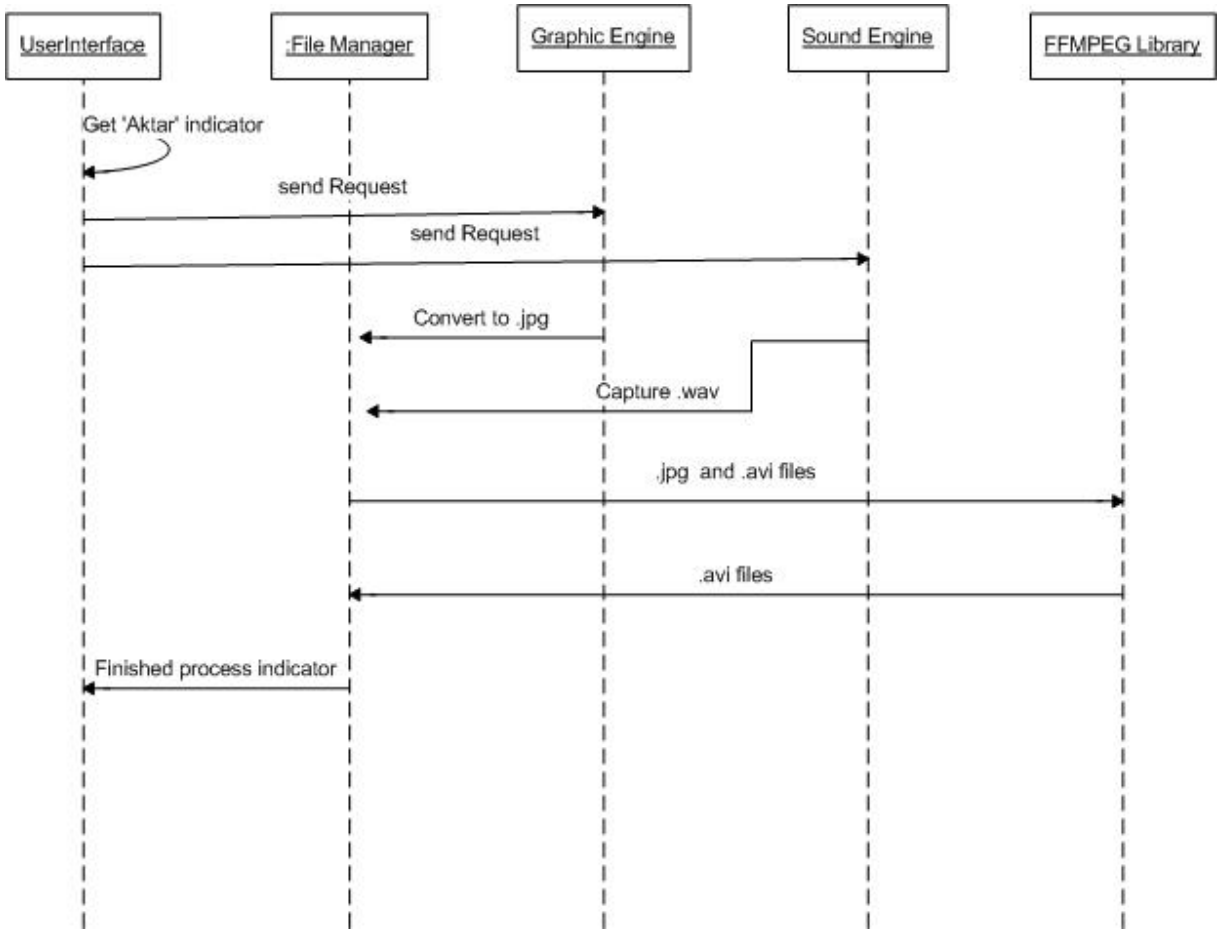
into our A/G file will effect the sequence on the below diagram. Since it is just about the coding, later on we may change the order. For this time it is assumed that information of attached messages in A/G file is kept before the whole objects in animation or game. So in the below diagram first all messages information are loaded to the animation or game object , then information of whole objects are read and loaded to it. Then graphics engine uses our BaseApp object and displays the frames on 3D part of the editor. Also with respect to the properties of BaseApp, the other subparts of the editor are set. For example, with respect to the mode (loaded animation or loaded game), the editor is shaped in a distinct way.



Load Animation/Game Sequence Diagram

6.1.3 Convert Animation to .avi format

When GUI detects a user press 'Aktar' tool-command as shown on the figure at GUI-Editor Design part, it directs the request to graphic and sound engine. Graphic engine converts the animation to ".jpeg" files and sound engine captures the sound. Then ".jpeg" files and ".wav" file are sent to ffmpeg library by the file manager class and ".avi" format is created with ffmpeg. And a feedback is sent to GUI.

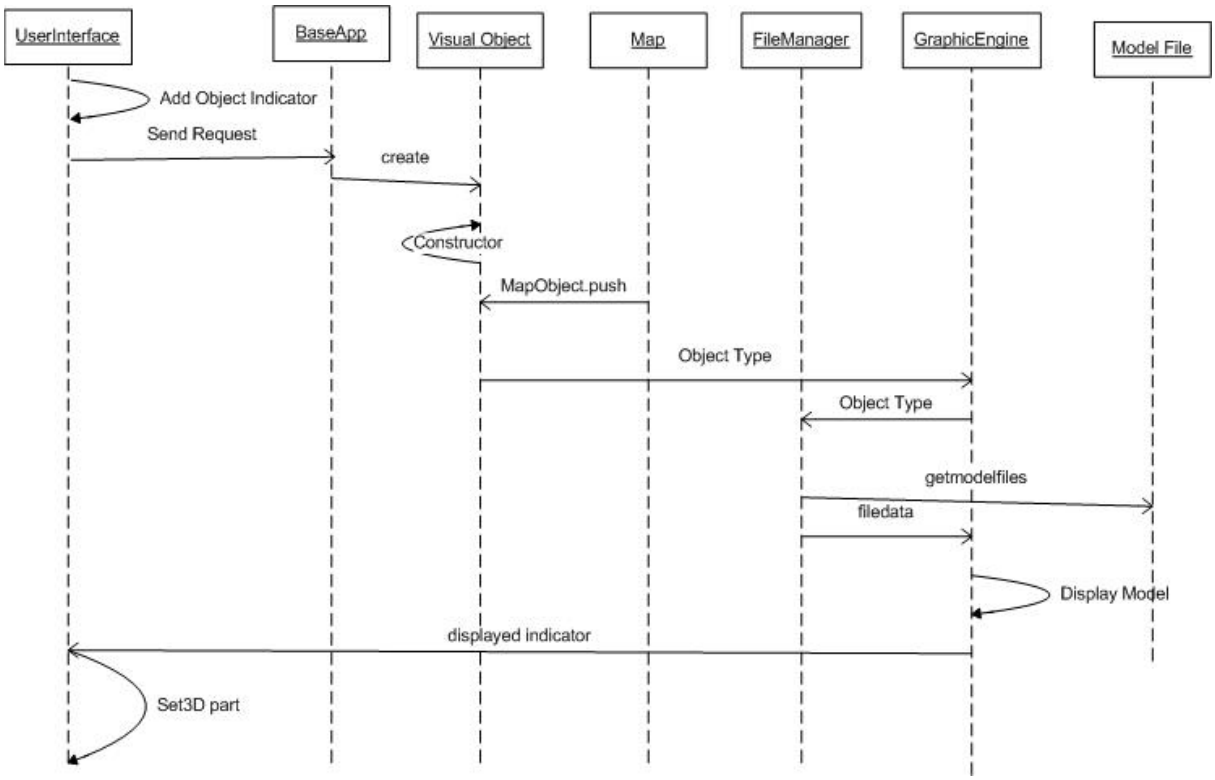


Animation Export to '.avi' Sequence Diagram

6.1.4 Add Object

When GUI detects a user pressing on 'Araç Kutusu' as shown on the figure at GUI-Editor Design part , it directs the request to BaseApp object. Corresponding object is created and added to the objects of Map. And through the File Manager, this

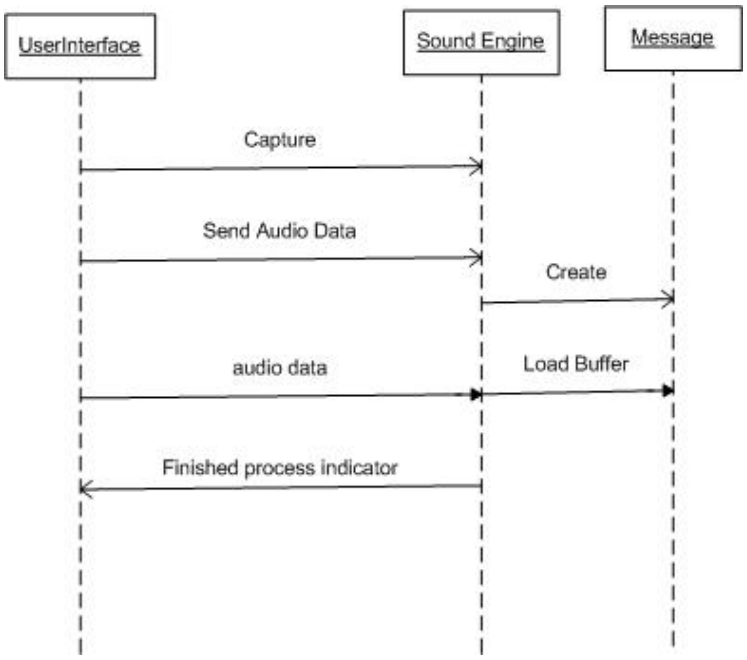
object's model files data are sent to graphics engine. Then graphics engine adds the object to the current display.



Object Addition Sequence Diagram

6.1.5 Capture

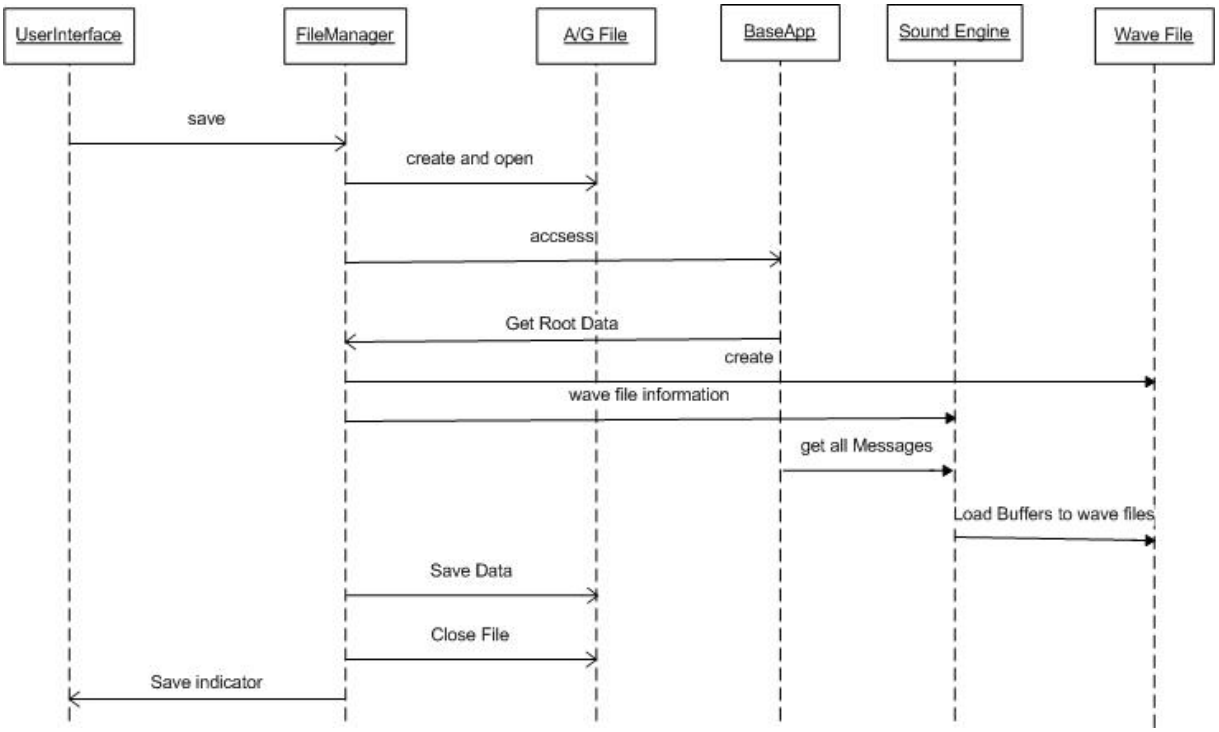
When user wants to capture sound for messages by microphone, sound engine creates a message object. Then, it loads the buffer of this message object with the incoming audio data.



Sound Capture Sequence Diagram

6.1.6 Save Animation or Game File

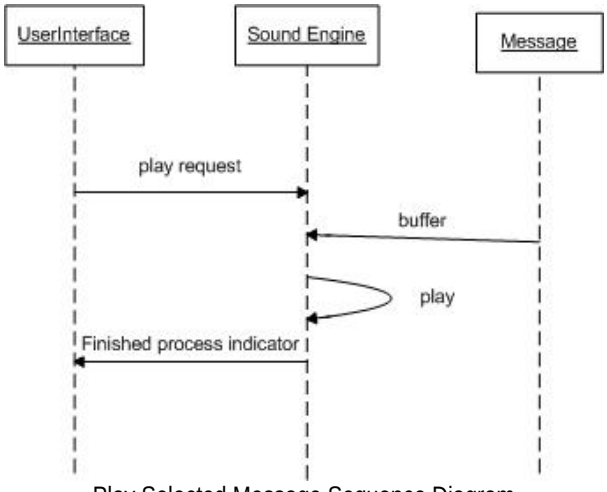
When any save request is detected an A /G file is created as a respond by the File Manager. File Manager accesses the BaseApp class and gets the data to be saved, writes them into the A/G file. All the buffers (in message objects are also saved into .wav files) at the end closes the file and finalize the process.



Save Animation/Game File Sequence Diagram

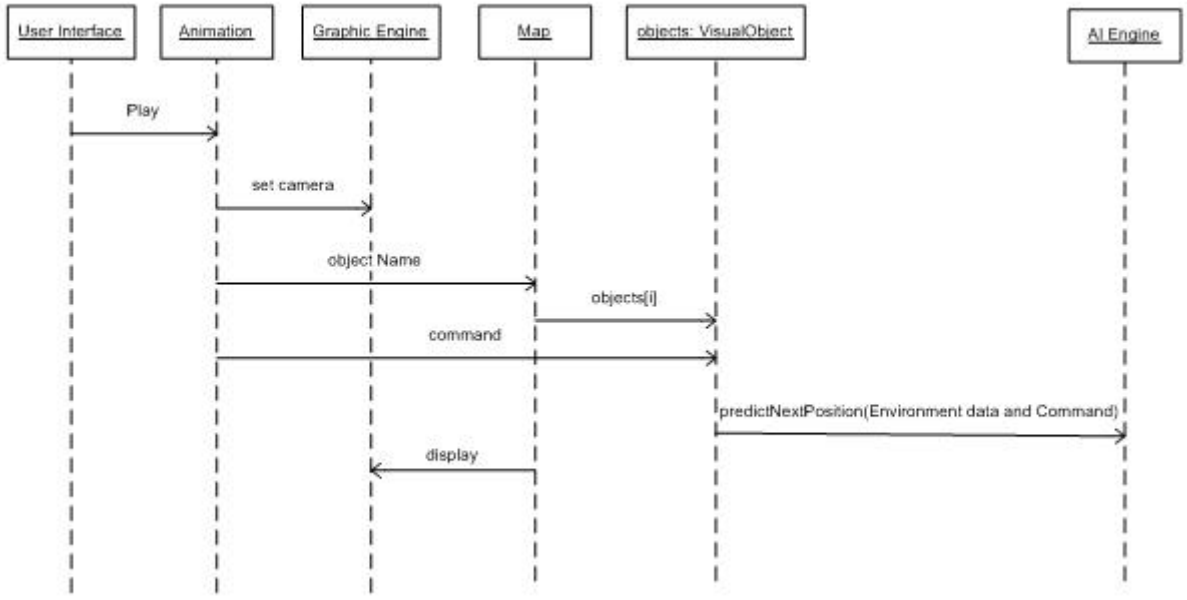
6.1.7 Play Selected Message

Sound Engine plays the wave that was previously loaded into memory.



Play Selected Message Sequence Diagram

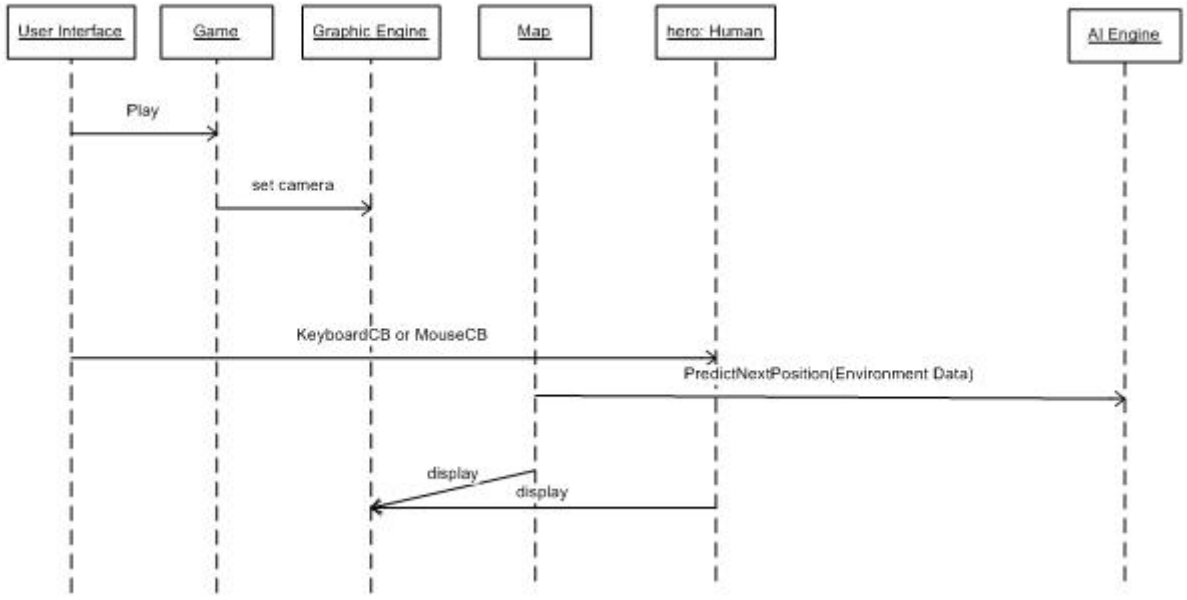
6.1.8 Displaying Animations



Animation Display Sequence Diagram

This sequence diagram summarizes processing only one entry of commands vector in animation class. For all entries in commands, the above procedure would be followed up to the end. As we abstracted the commands vector, it handles consecutive camera positions and objects behaviours like run, walk, turn, stop etc. Once the camera is set by 'Graphic Engine', then the controlled objects are replaced according to the given command by the AI Engine.

6.1.9 Playing Games



Game Play Sequence Diagram

In playing game, we just keep track of the movements of our hero with respect to the keyboard or mouse inputs. However, the other objects have to obey the predefined AI's.

7 SOUND EFFECTS DESIGN

In editor part of our program, we will have only decorative sound effects. These sound effects will be played when user clicks on a button or after an action takes place. Apart from these sound effects, in editor part user will be able to record sound and play this sound before attaching to a message.

In viewer part (both animations and games), we have environmental sounds for realistic ambiance. These sounds will be similar to sounds that we hear everyday in the streets. We will have several different sound effects recorded for this purpose and they will be played randomly in our program. Since it will be playing all of the time, we will start playing those sounds, from start of the game or animation, until play or pause is pressed in animation or the end of the game.

In games, we need children to be aware of their surrounding, so we will play sounds for cars, with respect to their distance to the student. As the distance becomes less, the volume of a car will be larger. Also, we will adjust the frequencies of the sounds with respect to the speed of the car (higher speed higher frequency). We expect the student to listen to the surrounding to detect the incoming cars. Also there will be walking/running sound effects for the movements of player.

In animation, most of the sound effects will be same as the games. The difference will be because of the fact that there will be no player, so there will be no walking/running sound effects. There will be just the sounds of cars with respect to camera position.

Sounds used for effects will be in such a format that it will be a short constant sound which can easily be placed in a loop as long as the animation or game continues. Sounds attached to objects will start playing when the object starts moving and will stop when the object stops.

An important note: In both games and animations, when a screen message appears, we will pause all of the sounds and play the recorded sound of the screen message, if it exists.

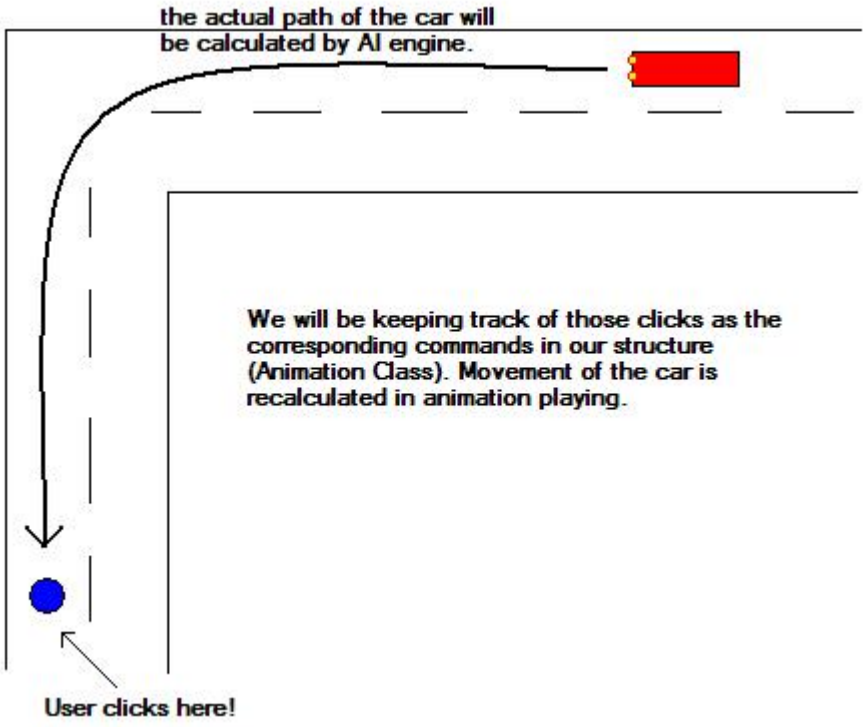
8 ARTIFICIAL INTELLIGENCE DESIGN

In our project we will be using AI in two occasions.

One of them is to determine automated intelligent movements (random movements) of the vehicles and pedestrians. This occurs only in games since in only in games we have random traffic (for both pedestrians and vehicles). It includes: the setting of speed of the car according to other cars on the road, waiting for other cars in road junctions, waiting in the traffic lights, obeying the traffic rules, etc. This occasion of AI will not be used in animations since we don't need cars and pedestrians to obey rules

in animations, because the instructor may want to demonstrate what is wrong to students.

The other occasion is to determine and normalize the standard movements of vehicles and pedestrians. This occurs in both games and animations. It includes: keeping the cars on the roads, smoothing turnings, calculating the paths, detecting the activation of blue areas, etc.



In animations, so instructor will only click on the screen to move cars and pedestrians, to a specific location. He will just choose the ending point of the movement for the chosen object. Then, the path will be calculated and the steering behaviours of vehicles will be determined by AI engine. The graphics engine will use this calculated path afterwards. We will be keeping just tracks of these paths and recreate them while playing animations or editing the animations.

9 ADDITIONAL CONVENTIONS

9.1 Virtual 3D World

Our virtual environment is a 3D world where each element of the map is associated with a two dimensional position vector (since any map element always stands on the floor, i.e. doesn't fly). The floor of our world lies in the positive x-z plane (so all coordinates are positive).

In our world, unit coordinates are assumed to be equal to meters. So 2 meter long car will appear 2 units long (in horizontal coordinates) in our world. Also a 10 meter high building will appear as 10 units in y coordinate.

The floor of our virtual world is flat. So we don't need any mechanism for describing the natural elevation (such as mountains). So we will need only models for our visual objects (humans, vehicles, buildings, etc) while rendering our virtual world.

Our virtual world is finite. So while creating the world, we must give boundary parameters for x and z coordinates. Therefore, the size of the world can be different (which can be set by user while creating the world). Height of our world is fixed to 20 units in y direction (actually it is not important for our program; it just must be higher than the longest building model). So, we can draw our model in a rectangular prism.
Our virtual world is described as entities of Map class (the definition of this class is given in another section). The objects of this class will include data about size of the world and the visual objects placed in that world.

Our animations and games occur in virtual world. So any animation or game is associated with a Map object.

9.2 Camera Modes

We will use different camera views for different parts of our program.

9.2.1 Editor

For the editor part, user will be able to view the world that he is designing with a perspective view. He will be able to zoom in/out and change the camera angle to view the desired place in the map.

9.2.2 Viewer

9.2.2.1 Game

In games, user will have only one view to see the environment. This view will be from the eye of the user. We will use perspective view with an angle of 60 degrees both vertically and horizontally. We provided these angles to have a more realistic look as it is important for us to make user feel in a sensation of reality.

9.2.2.2 Animation

In animation part of our program, field of view will be bound to objects defined by the educator in the editor part. These views will be defined during the creation of animations as follows:

- The educator will choose an object and attach the camera to it. First object added to the animation will be our default camera view.
- After attaching the camera, user will choose the field of view of the camera, which can be either:
    - From front
    - From back
    - From right
    - From left

User will be able to change the camera view for every frame. For example, after selecting a front view for an object a few frames before, he can change that view to a back view a few frames later. These camera views will also be stored in the animation structure.

9.3 Frame Rate Issue in Games and Animations

All applications which include animations have a problem with setting the rates of the animations. When a scene is showed on the screen, it takes long time if there are a lot of objects in the scene; it takes less when there are a few objects in the scene. So it may end with animations and object movements asynchronous in the game, which may be unpredictable and fatal for our application.

We overcome this issue by calculating the next positions of our objects by the help of our AI engine. This is done as follows:

Our AI engine runs as an independent thread in our program. This thread will recalculate the positions of objects at fixed time intervals (let's say 30 times in a second). Frame rendering will be done according to these calculations. Our rendering engine will not keep track of frames per second. It will repeatedly render the objects as much as it can. Since our objects will do fixed (30 times) number of unit movements (it can be calculated as speed of the objects in a second over 30), our frame rate doesn't affect the speeds of the objects. So the transformation values of objects per frame are not fixed within the game.

This method will ensure that our objects will move at a determined speed. But, it may cause frame skipping if the computer's graphics card is not very strong. But we ensure that there will be no fast or slow movement of objects because of computer power. We gave an example below.

The speed of the object is 30 units per second (therefore 1 unit per calculation) and the frame rate is 30 frames per second: then the speed per frame is 1 unit at average. The frame rate is 60 frames per second: the speed per frame is 0.5 unit (which means the object will not move in one of two consecutive frames).

We may adjust the number of calculations to have smoother movement for our objects or to have more performance.

9.4 Traffic Density Issues

We mentioned about traffic density in several sections of this document. We keep the density as a floating point value between 0 and 1 (actually it cannot be equal to 0 since it means there is no traffic and we don't need that path). 1 means a traffic density of maximum, which is equal to 20 objects per minute. The values between 0 and 1 will be interpolated according to maximum value. This value is not a constant value, but an average value. Because in games, cars and pedestrians move randomly, they wait in traffic lights, decrease speed not to crush other cars and pedestrians. This changes the distribution of the traffic at specific location, but at average it should be equal to density value.

9.5 Implementation of Modules

Our application will be composed of two executables: Editor and Viewer. Since these two executables will share most of the modules, our modules will be implemented as dynamic linked libraries (DLL). It also gives us the chance to upgrade our application easily by recompiling DLLs after the release. Our modules are just the modules in our DFD diagrams.

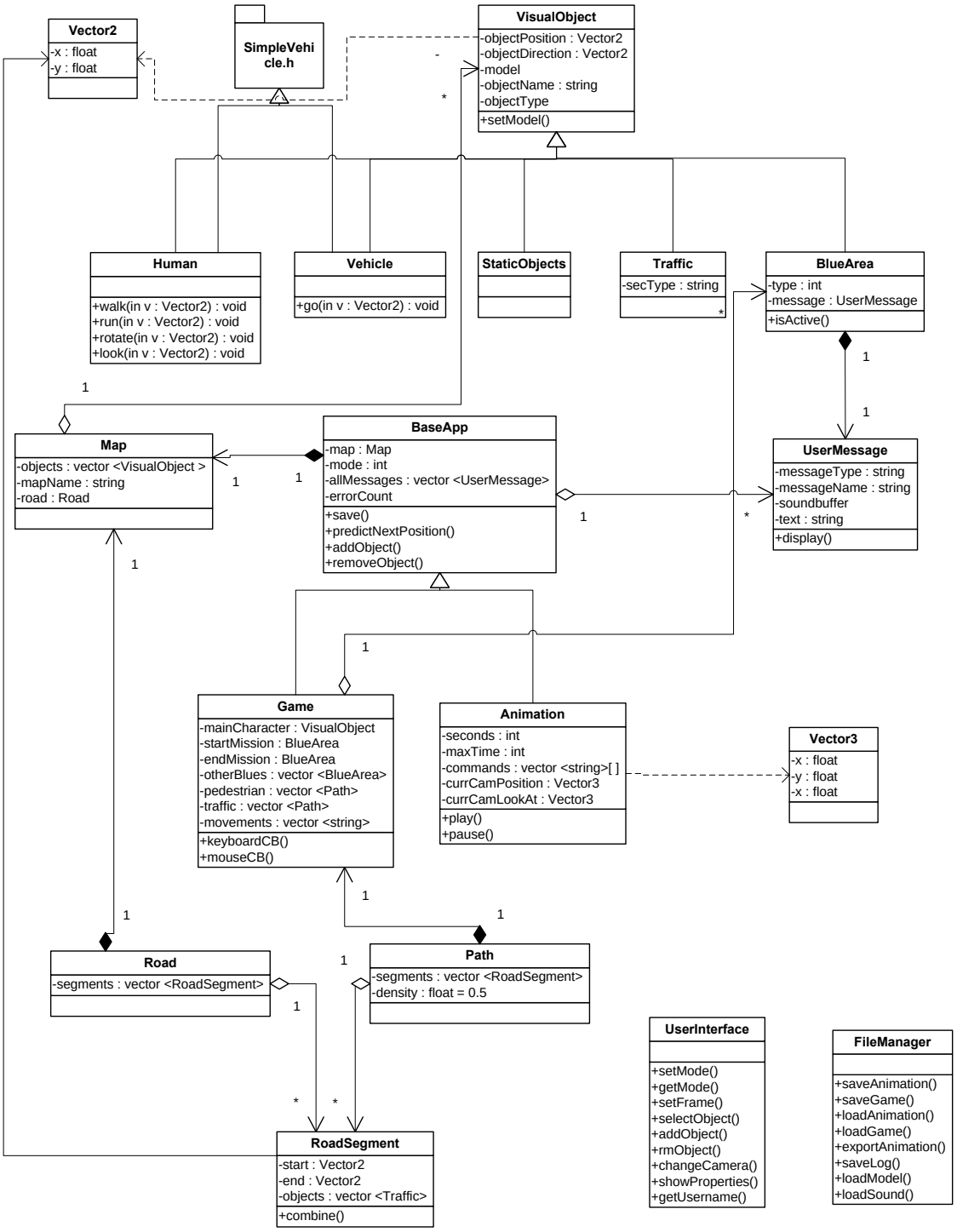
10 CONCLUSION

This document includes the design details of our software tool. During the preparation of the document we examined all of the external tools and libraries which will be used in the development steps of our tool. We have determined the following design details:

- Class structure and classes
- Data structure
- File structure
- The tool and libraries that we will use
- The relation and connections between the external tools, libraries and our main program
- Flow of data which is illustrated by using DFD diagrams
- Our conventions to the application

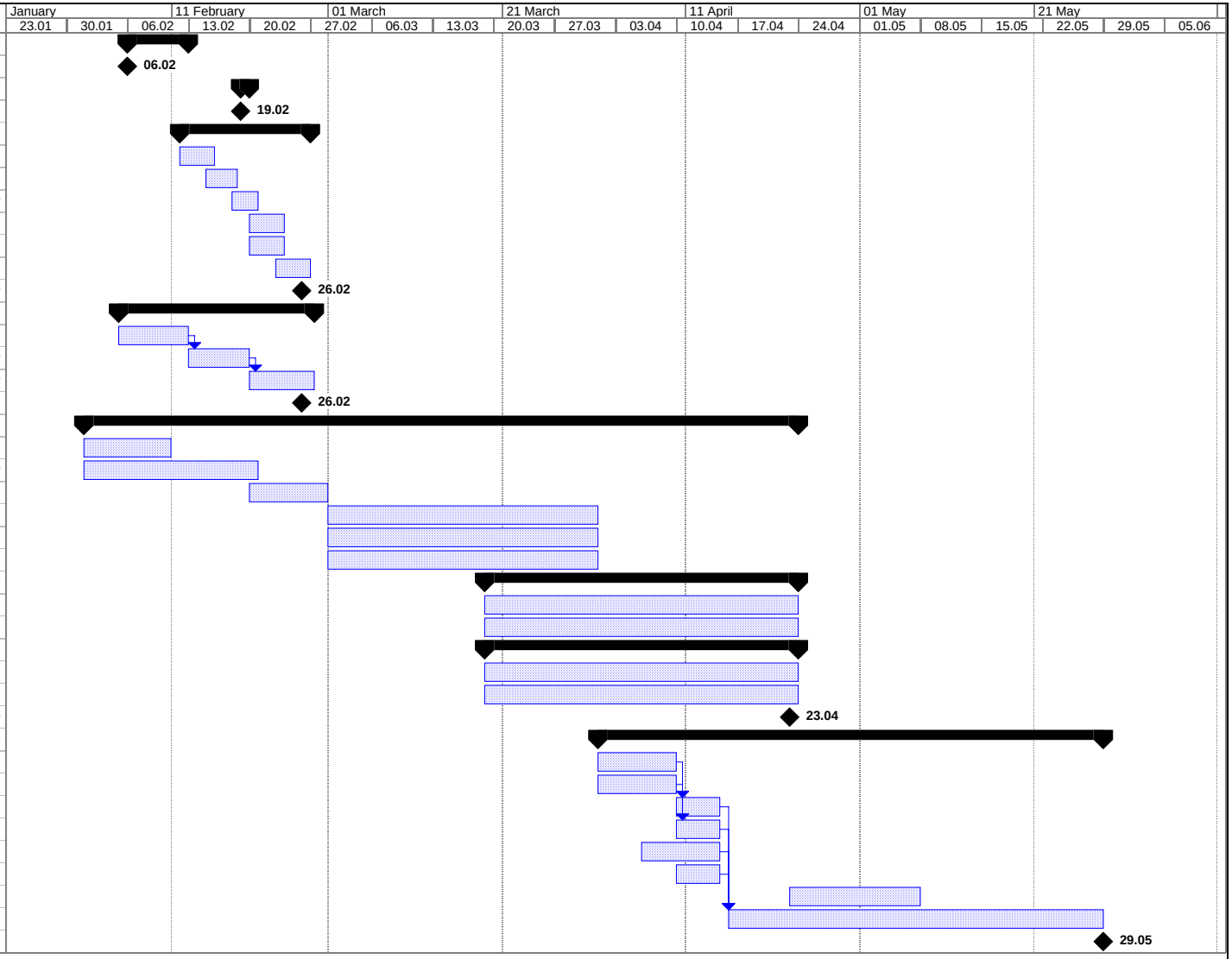
11 APPENDIX

11.1 Appendix A – Class Hierarchy and Relations



11.2 Appendix B – Gantt Chart

ID	Task Name	Start	Finish
1	**Living Schedule**	**Mon 06.02.06**	**Sun 12.02.06**
2	Living Schedule is ready	Mon 06.02.06	Sun 12.02.06
3	**Design Review**	**Sun 19.02.06**	**Sun 19.02.06**
4	Design review is made	Sun 19.02.06	Sun 19.02.06
5	**Configuration Management Plan**	**Sun 12.02.06**	**Mon 27.02.06**
6	Learning CM Details	Sun 12.02.06	Wed 15.02.06
7	Defining Org. CM Framework	Wed 15.02.06	Sat 18.02.06
8	Defining CM Process	Sat 18.02.06	Mon 20.02.06
9	Set up CVS	Mon 20.02.06	Thu 23.02.06
10	Applying Determined Steps	Mon 20.02.06	Thu 23.02.06
11	CM Plan Documentation	Thu 23.02.06	Sun 26.02.06
12	CM Plan Report	Sun 26.02.06	Mon 27.02.06
13	**Designing Web Page**	**Sun 05.02.06**	**Mon 27.02.06**
14	Creating a storyboard	Sun 05.02.06	Sun 12.02.06
15	Planning navigational tools	Sun 12.02.06	Mon 20.02.06
16	Create web page	Mon 20.02.06	Mon 27.02.06
17	*Web page is working*	Sun 26.02.06	Sun 26.02.06
18	**Implementation**	**Wed 01.02.06**	**Sun 23.04.06**
19	**User Interface (CEGUI)**	Wed 01.02.06	Fri 10.02.06
20	**3D Model Drawing**	Wed 01.02.06	Mon 20.02.06
21	File Manager Module	Mon 20.02.06	Tue 28.02.06
22	Integration of OpenAL	Wed 01.03.06	Fri 31.03.06
23	Integration of OGRE	Wed 01.03.06	Fri 31.03.06
24	Integration of OpenSteer	Wed 01.03.06	Fri 31.03.06
25	**Editor Module**	**Sun 19.03.06**	**Sun 23.04.06**
26	Animation Editor	Sun 19.03.06	Sun 23.04.06
27	Game Editor	Sun 19.03.06	Sun 23.04.06
28	**Viewer Module**	**Sun 19.03.06**	**Sun 23.04.06**
29	Game viewer	Sun 19.03.06	Sun 23.04.06
30	Animation Viewer	Sun 19.03.06	Sun 23.04.06
31	*First Release*	Sun 23.04.06	Mon 24.04.06
32	**Test Specifications**	**Sat 01.04.06**	**Sun 28.05.06**
33	Defining testing strategy	Sat 01.04.06	Sun 09.04.06
34	Defining test record keeping strategy	Sat 01.04.06	Sun 09.04.06
35	Defining testing tools	Sun 09.04.06	Sat 15.04.06
36	Defining types of tests	Sun 09.04.06	Sat 15.04.06
37	Defining how to debug	Thu 06.04.06	Sat 15.04.06
38	Defining test schedule	Mon 10.04.06	Fri 14.04.06
39	Test specification plan documentation	Sun 23.04.06	Sun 07.05.06
40	Debugging	Sun 16.04.06	Sun 28.05.06
41	*Final Release and Demo*	Mon 29.05.06	Wed 31.05.06



Project: implSchedule
Date: Sun 08.01.06

Task | Progress | Summary | External Tasks | Deadline
Split | Milestone | Project Summary | External Milestone