PAGODA SYSTEMS

ULApag

CENG 491

INITIAL DESIGN REPORT

Ercan Üret --- 1298045
Çağatay Turkay --- 1298355
Selçuk Tunç --- 1298348
Sinan Mutlu --- 1298025

TABLE OF CONTENTS

1. INTRODUCTION 4
1.1 PURPOSE OF THE DOCUMENT 4
1.2 SCOPE OF THE DOCUMENT 4
2. SYSTEM OVERVIEW 4
3. SYSTEM ARCHITECTURE 6
3.1 DATA STRUCTURES & TECHNICAL DETAILS OF THE EDITOR 6
3.1.1 Object Class 7
3.1.1.1 Library Object 8
3.1.1.2 Scene Object 9
3.1.2 Library Class 11
3.1.2.1 Shared Library 12
3.1.2.2 Current Library 13
3.1.3 Sound Manager Class 14
3.1.4 Text Manager Class 15
3.1.5 Scene Class 15
3.1.6 Frame Class 16
3.1.6.1 Normal Frame 17
3.1.6.2 Key Frames 18
3.1.6.3 Step Frames 19
3.1.7 Object Base Class 19
3.1.8 Timeline Class 20
3.1.9 Event Handler 24
3.1.10 Math Library Class 25
3.1.11 File Operations 26
3.1.12 Window Management, Rendering & User Input 28
3.1.13 Overall Class Relation Diagrams 29
3.2 Run Time Environment 31
3.2.1 FrameBuffer Class 32
3.3 FILE FORMATS 38
3.3.1 .obj File 38

3.3.2 .objh File 39
3.3.3 .ulp File 39
3.3.4 .pgd File 40
4. USER INTERFACE DESIGNS 41
5. CONCLUSION 42

1. INTRODUCTION

1.1 PURPOSE OF THE DOCUMENT

This document is the initial design report of the ULApag. The purpose of this document is to initiate the design specifications of the project and establish a basis for the detailed design processes.
We have built the design of the project roughly; meaning that not every single detail of the design is reported here and there may be some additions and modifications later.
We have decided most of the data structures that will be used in project. In this report we will present our project's technical details with some diagrams to reveal our system architecture.

1.2 SCOPE OF THE DOCUMENT

This document specifies the design issues related to design tool and run time environment. The data passed between the modules and external storages are discussed. Also we have discussed all properties of the editor structures. This document will provide the all system components and the relations among them.

2. SYSTEM OVERVIEW

Our main aim was to design an education tool based on animations and visualizations. With this project, user will be able to create 3D animations in order to help their customers in assembling any mechanical component.
System has been designed with two main parts; design tool and run time environment. Our customers will be able to create 3D animations and their customers will use run rime environment to see the animations that are prepared with the design tool.

One of our targets is to design an education tool that will be easy to use for both the developers and their customers. User will be faced with a partitioned scene like in most 3D modeling tools. These partitions will be different camera views for the user. This editor will basically provide a library and a timeline structure to the user.

There will be two internal parts of the library; current library and shared library. Current library will hold information about objects created on the current scene. There are two ways to add an object to current library, user will either import from a 3ds file or add an object from the shared library. Shared library will include very frequently used objects that have been already imported.

The animation creation process is the most critical part of our program. The animation is all about timing and the best way control time in an editor is using a timeline. Timeline consists of little rectangles each of which representing a frame in the current animation. By making suitable changes in shape location and rotation of objects in consecutive frames, the user will create animations. The structure of the program will be detailed according to these major parts in the following sections.

3. SYSTEM ARCHITECTURE

As our system is structured in two main parts, we will cover them in separate subheadings. In Figure-D1 a Level-0 DFD for our whole project can be seen. We now move on to elaborate our data structures one by one.

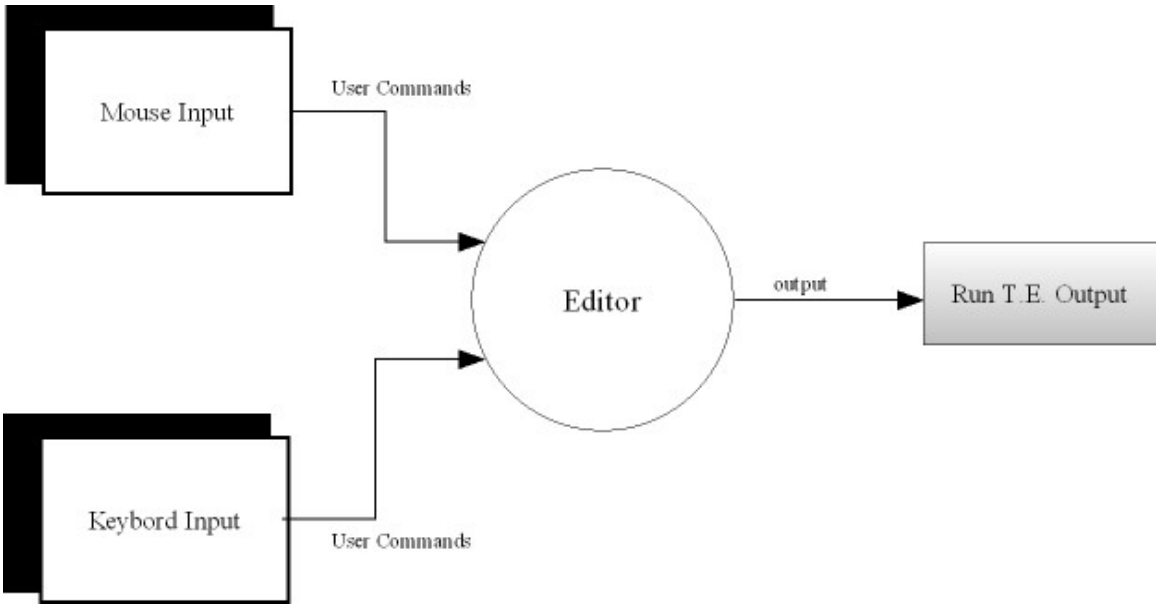


Figure-D1 Level-0 DFD

3.1 DATA STRUCTURES & TECHNICAL DETAILS OF THE EDITOR

In this section the data structures involved in our program are discussed. Each program component is taken into consideration as a separate part, a class. We will include our diagrams directly into this section, instead of having them in a separate chapter. Before beginning the details of the classes, we will give our two general DFDs. and in Figure-D2 we have a more detailed DFD of our editor. At the end of this section, before the details of the classes, we have general class diagrams for explaining the general relations and callbacks between classes. These diagrams are included in chapter 3.1.13.

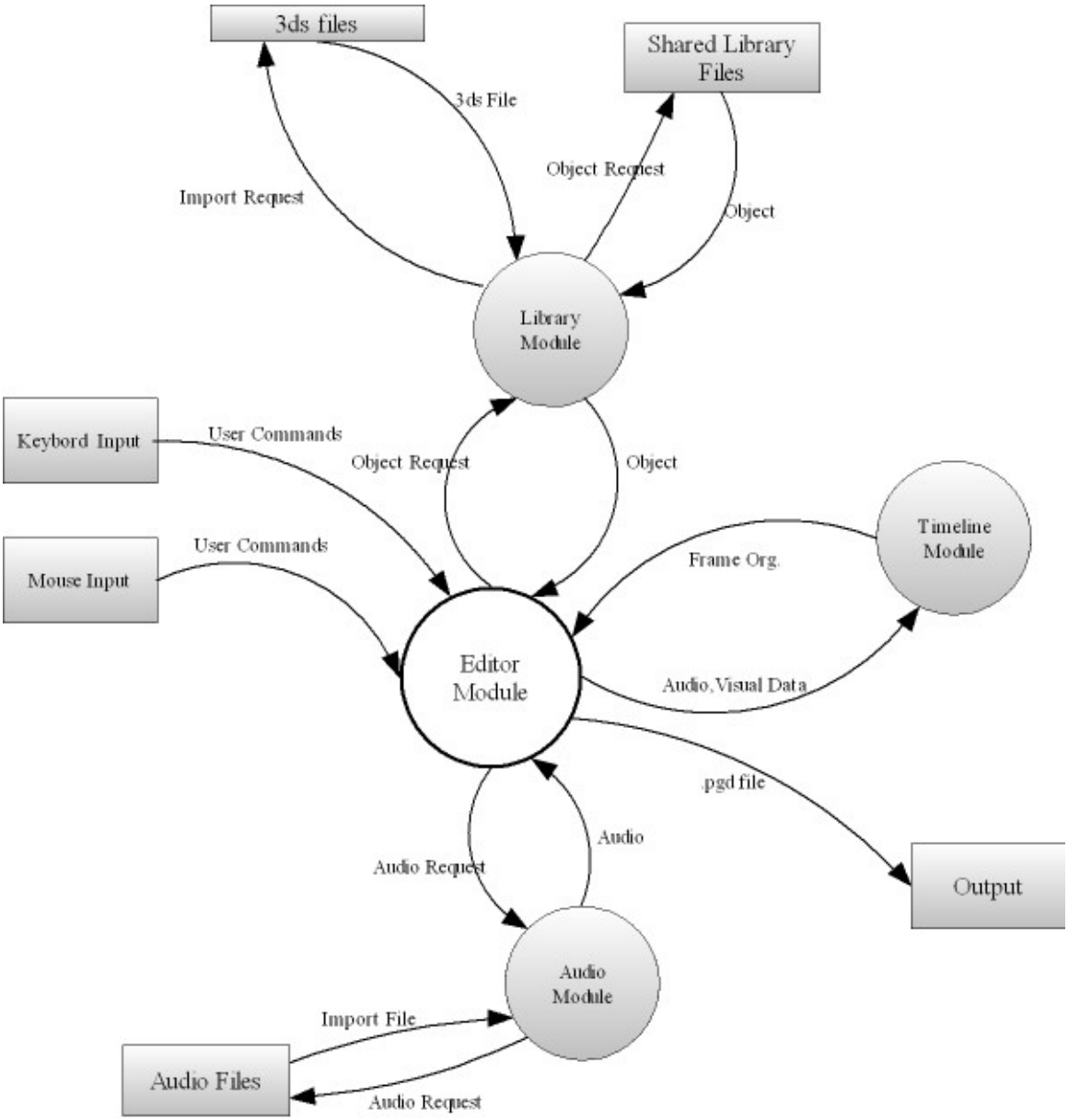


Figure-D2 LEVEL – 1 DFD for the Editor

3.1.1 Object Class

This class is the class which holds information about the objects which are imported from 3d Studio Max files. It will hold all the necessary information to draw an object onto the current scene. This information will be the object's coordinates (vertex, texture), normal vectors, faces and materials. This class will have a draw method which draws the object into the related sub-window.

Also this class will include a name or id which will be used to identify the object. This class will have two inherited classes called *libraryObject* and *sceneObject*. A class diagram explaining the class structures and inheritance relations are included in figure-c1.

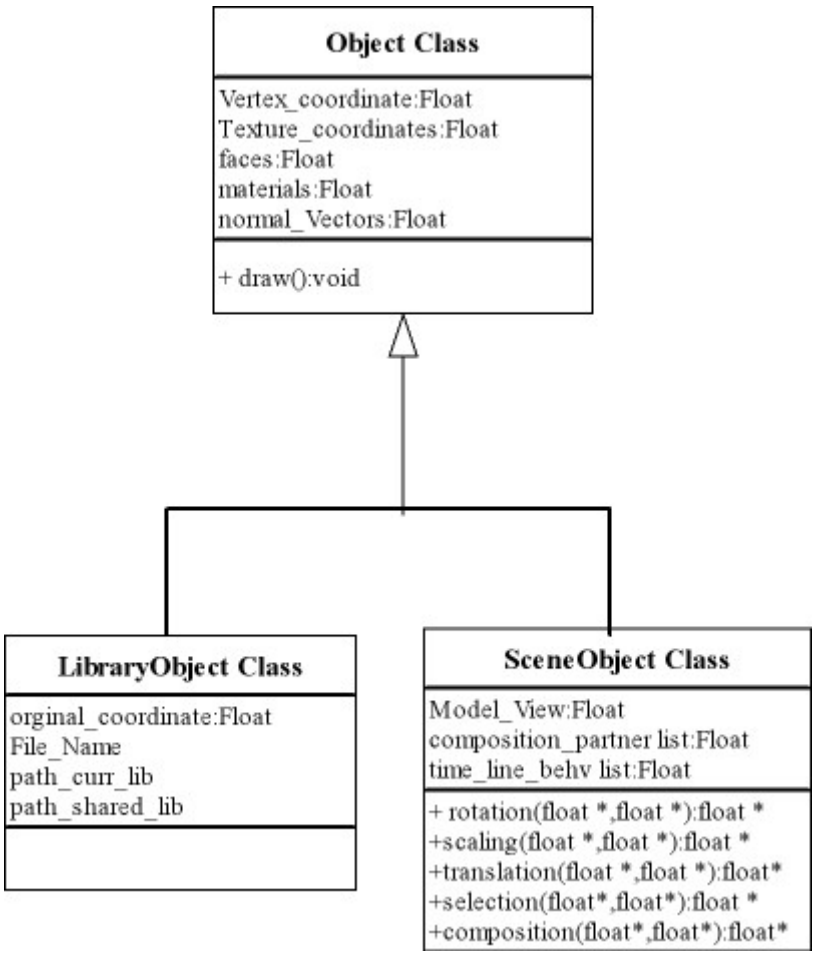


Figure-c1 The Object Class and Inheritance Relations

3.1.1.1 Library Object

This inherited class will hold information about objects which are in the library. The objects in the library will only hold the original coordinates of the imported object which is probably drawn at the origin (people tend to draw objects at the origin, in 3D modeling tools). The library objects will be a database for our program and when dragged to the screen instances of it will be created on the screen. In addition to the fields of its super-class *object*; this

class holds the physical location and filename of the object; this associated file will be our own representation of the object. This file format will be .obj. (Details of an .obj file will be covered in the "File Formats" section.) . We chose .obj because it is a very popular and easy to use file format. In addition, many of the 3D drawing tools support saving in .obj format (Unfortunately, 3ds Max and Maya don't support .obj format). Also, this class will include the path of the object in the library, one path address for its position in the current library part and one other path address for its position in the shared library part. Our library will be constructed in a tree structure and this class will hold where in the tree the object takes place. When the tree leaf in which a particular object lies is opened, all the information which is kept in the file named "filename" will be loaded into the memory. We could have loaded all the files which are lying in the shared library at the first execution of the program but with thousands of objects in the shared library, this could take up so much memory space and also the first execution of the program could take very long. When the user wants to add an object to the current scene, the object's data will be copied into the newly created instance of *sceneObject*. We need the data of the library object included in its variables, because we need to display the object in a mini-display at the top of the library. When a specific object is selected in the library, a small drawing of it will be drawn in the mini-display. That is why the super-class *object* has a member method draw() which will be inherited by both of its sub-classes.

3.1.1.2 Scene Object

A scene object is an object which lies on the current scene. It is an instance of a particular object which is included in the current library.
These scene objects will have their own modelView matrices. And methods associated with them which apply all kinds of modifications like rotation, scaling, translation, selection and composition. Whenever a modification request arrives at the current object, the modelView matrix of the current

object (or objects, in case of a composition) is updated. By having a separate modelView matrix for each object, we will be holding all kinds of transformation matrices in each object's data; this gives us the information of how much an object has been transformed between two consecutive key frames. Using this data we will apply tweens and transformations of objects. Each object has a list of other objects which are in composition with itself. We call this list "composition partners list". Whenever a new object is added to a composition, it is added to all the individual composition partners' lists in all of the objects in the current composition. Whenever a modification is applied to an object which is currently a member of a composition; this modification is applied to all the other objects which are included in the current object's partners list. If the current object is not a member of any composition, then the composition partners list will be empty and the current modification is solely applied on the current object. The member method draw() of its super-class object; will draw the *sceneObject*s to the main sub windows.
The scene objects will contain a list which is representing their behavior in the timeline. For each frame, the object is present in, a modelView matrix representing the orientation of the object will be added to this list. When the object is being drawn with the draw() method, the object will know on which frame it is being drawn and apply the appropriate modelView matrix. This method will give us the optimality to have only one instance of an object. We will be able to modify the object, apply tweens and draw the object easier. The physical instance of the object will only exist in the Object Base class and for all the other classes we will keep pointers to these objects. For example, to render the object between frames one and fifty, we will use the values of the timeline array of the object from index one to fifty. As we have only one instance of the object for the whole timeline, any change on the objects' attributes will be applied directly to all of the object's instances on the timeline.

3.1.2 Library Class

Our library class will be responsible for a number of tasks. These include; importing a 3ds file, managing the shared library records on the disk, managing all the objects added to the scene and to the library. We will have two different parts in our library: the shared library and the current library. Their only difference will be the sources they get data from so we will not have two different classes for these parts. We will use .NET's tree component to organize the structure of these libraries. And shared library and current library will be two distinct folders in this tree. In the library class, we will have two *object* lists; one for shared library objects and one for the objects of the current library. Both of these lists contain pointers to *library object*s. A DFD can be seen in Figure-D3 explaining the data flow in these library components. We also added a use – case diagram explaining the usage of the library components. This figure can be seen at the end of the details of this class (Figure – u1). We have added a class diagram for the Library Class (Figure-c2).

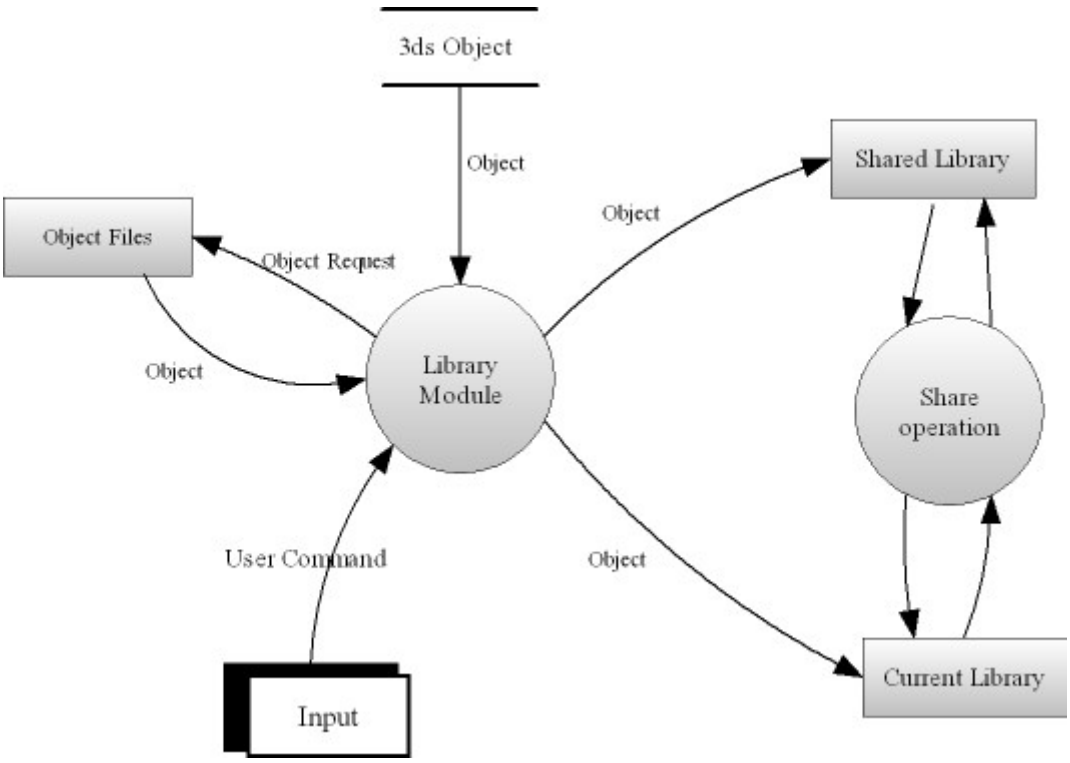


Figure-D3 Level-2 DFD for the Library Module

3.1.2.1 Shared Library

The shared library list will be constructed when the program is executed. We will have a header file which will hold information about the files which have already been added to the shared library. This header file will be explained in the file formats section. The data of all the objects which are included on the shared library will not be loaded at the first execution, but instead only the header file will be loaded to construct the tree view into the library panel. As we will have a small display in our library panel, we will need to load the data of the objects in the library, our preferred method for handling this difficulty is already explained in the library object part, so we are not covering it again here. The main function of shared library is being a cumulative resource for the current file. The user will add an object from the shared library to the current library and then use it in the current scene, so will have methods to copy the object pointers from the shared library list to the current library list. This method will be called via two different callbacks; first, by dragging from the shared folder to the current folder and secondly by dragging from the shared library to the current library, when this action is done, the newly added object is automatically added to the current library.

When the user wants to add an object to the shared library, user will carry a selected object from the current library to the shared library. At this moment the header file will be updated to reflect the changes by the addition of a new object. Also a new obj file is written to the disk which holds the data of the new object. So when the user closes the program and opens a new project after a while, this object will reside in the shared library.

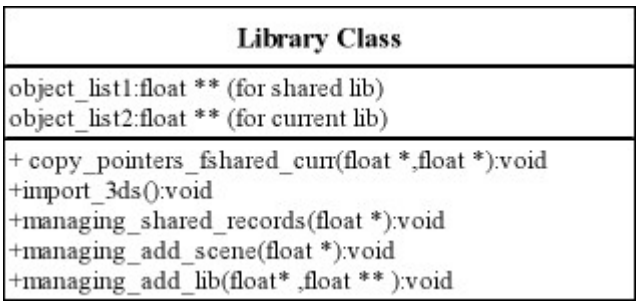


Figure-c2 Library class

3.1.2.2 Current Library

The current library holds the list of objects which are currently drawn on the screen and which are newly imported into the library but not used in the animation already.

One other feature of the current library will be importing from 3ds files to our internal format. At first the 3ds files will not be converted directly to obj files, (we will use obj files to store our objects on the disk) but first reside in the memory in our *library object* objects. Then whenever the user wants to add the object to the shared library, these *library object*s are written to the disk immediately. All the above file operations will be done by the *file operator* class which will be discussed later.

When the user wants to create a new instance of an object he/she will simply drag the file from the library menu into the scene. The newly created object will be added to the current frame's scene and a reference link will be added to the current library's data. The current library will hold information about the instances of a specific object. This will also be displayed in a tree fashion. When you click on an object you will get the list of the instances of this object which are displayed on the scene. So one way to reach and select the objects on the scene will be selecting these references in the library. The user will be able to assign names to each of these objects. By default an instance of the current object will be given a name "the objects name + its order" i.e. if you want to add a "box" object to the current scene and you already have three boxes in your scene, this new box will be given the name "box04". To

accomplish these functionalities we have to keep lists for each objects' instances, whenever a new object is created on the scene, a *scene object* pointer is added to the object's list which resides in the library class. The library class will not have further capabilities than this organizing and importing feature.

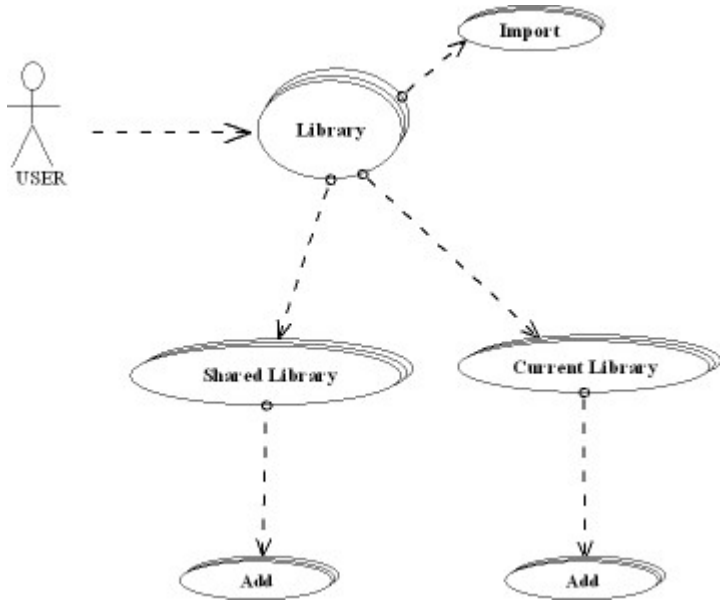


Figure- u1 Use case for the Library Components

3.1.3 Sound Manager Class

This class will mainly use the methods of the fmod library. Fmod library gives us all the necessary tools to load a sound file, adjust its volume and play the sound. This class will only hold the path of the selected sound file and load the file to memory when desired. Also it will hold the data to play it whether in a loop or for a single time. It will be able to call the necessary functions from the fmod library. A key frame will have a sound object associated with it if the user wants to add sound to a key frame. The library gives us the ability to play multiple sounds simultaneously, so that we will use this feature to have some background music and load different speeches on it. The path of the sound file and properties of it will be input by the user from the user interface. A class

diagram showing the properties of the sound manager class can be seen in Figure – c3.

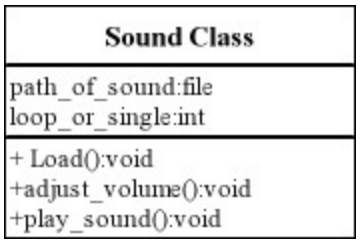


Figure-c3 Sound Class

3.1.4 Text Manager Class

Each key frame can have a text object associated with it. This text object will hold complementary text for the current animation. In this class we will hold all the necessary information about the text's font, size, color and the text itself. The specifications and the text itself will be obtained from a text panel in the user interface. A class diagram showing the properties of the sound manager class can be seen in Figure – c4.

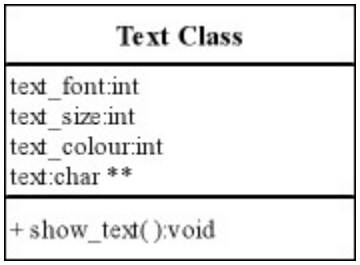


Figure-c4 Text Class

3.1.5 Scene Class

Each frame will hold a scene on it. When we are talking about displaying a frame, we are talking about drawing all the objects specified by the current scene. A scene object will hold the object list, camera list and light list. The object list will contain pointers to objects in Object Base class. The camera

and light modifications will be handled by Open Scene Graph's functions. The scene class will also have a draw() method, which will render its data to the screen, when called, this method will make no further modifications on the object and will directly render them with respect to its lists. This draw method will pass from which frame it is called so whilst rendering the object, the object will know which modelView matrix to apply. A class diagram showing the properties of the sound manager class can be seen in Figure – c5.

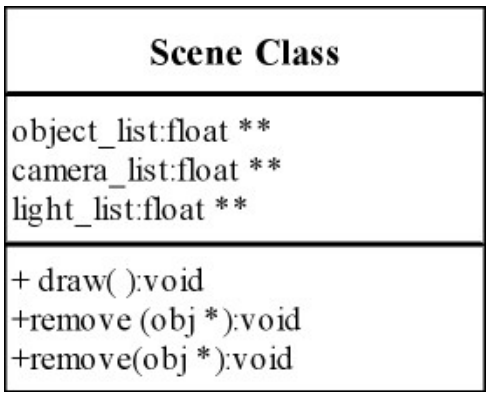


Figure-c5

3.1.6 Frame Class

This class will be holding the data of a unique frame in the timeline. The animation will be the concurrent displaying of frames with the specified frames per second. As we have covered in our previous reports, we will have different frame types, namely; normal frame, key frame and step frame. We will cover these frame types one by one. These will be implemented in an inheritance relation. All of these frame types are sub-classes of the frame class; moreover step frame will be the subclass of key frame.

The common attribute of a frame will be its scene data. All of these types of frames will and should include data about its scene information. As stated above the *scene* class will hold data about all the objects, lights and cameras. Another attribute which is common to all frames will be the update facility. When specific modifications are input by the user, i.e. translation, scaling etc, the selected objects' properties will be updated according to the update parameters which will be passed via callbacks from the user interface. As each

object will have its own methods to deal with modifications, this modifications will be passed to the *scene* class, and from the object list in the *scene* class, to individual *scene objects*. A class diagram explaining the class structures and inheritance relations are included in figure-c6.

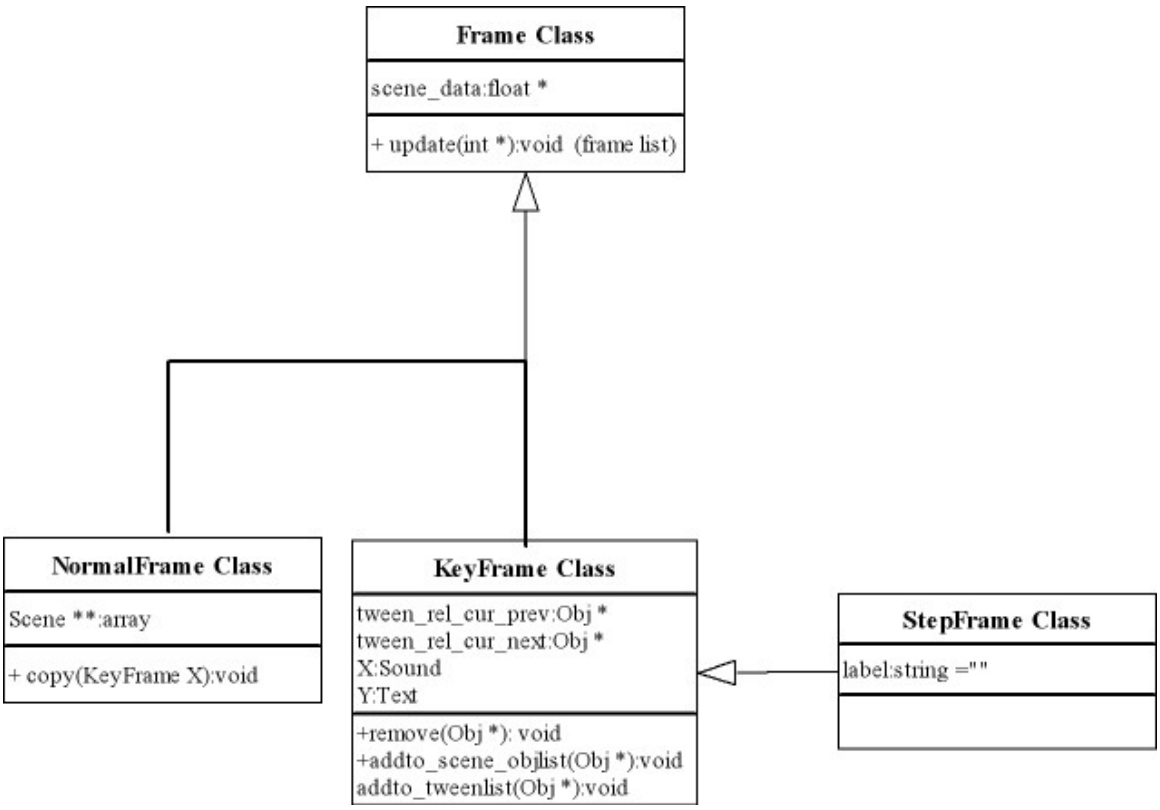


Figure-c6 Frame Class and Inheritance Relations

3.1.6.1 Normal Frame

We were first thinking to have no separate classes for normal frames, because normal frames will carry no specific information. The data of normal frames are fully dependent on the data of the key frames it is between. So at first we planned to calculate their data from its surrounding key frames while displaying the normal frames, but afterwards we thought that this would cease the overall performance of the animation so we decided to add frame objects

for each normal frame. A normal frame will not have any additional attributes. It will contain a scene class which is associated with the current frame and when its draw method is called the current frame is rendered to the screen. The user will not be able to make any changes in normal frames. The modifications on key frames will directly affect *normal frame* attributes via tween applications. The details of how tween operations will be accomplished will be covered later in the report.

3.1.6.2 Key Frames

Key frames are frames which will be the key points in our animation implementations. All the preferred modifications will be done in key frames, in contrast to normal frames, key frames will be modifiable. The user will be able to do all the modifications on key frames. These modifications include, as stated before, selection, deletion, composition and transformations.
When the program first starts, the timeline will include a single key frame. This beginning frame will be at time position zero in the timeline. It will not include any objects, cameras, lights or tweens. Whenever the user adds an object or from the other components, the lists will be populated. With only a single frame existent, the user will not be able to create any tweens, because tweens are defined between two key frames. When the user wants to add a new key frame, let's say, at position five, the three frames between the first frame and the fifth frame will be created as normal frames and the content of the first key frame is copied into all these normal frames and the newly added key frame at the fifth position. As for now, there is no tween defined, all of the frames contain the same scene information.
Let's go into detail of how tweening will be handled. In a *key frame* object we will have two separate lists of which objects are active in a tween operation. One of these lists will be for the tween relations defined between the current and the previous key frames and the other list will be for the current and the next key frames. When the user wants associate a tween with an object (it

means that the current key frame is not at the end of the timeline, it means there is a number of key frames further in the timeline); first the object's pointer is added to the current key frame's tween-with-next-frame list, and then to the tween-with-previous-frame list of the next key frame in the timeline. Whenever an object is removed from the current frame, it should be removed from all the tween lists of the current, previous and next key frames. The user will be prompted with a warning massage when an object involved in a tween operation is tried to be deleted, the user will be asked to remove the tween on that object manually, then delete or these operations will be done automatically. These lists will be modified and managed by the *timeline* class, which has access to all the frames on the current timeline. The pointers in *object class* which are discussed in *object class* part will be used to achieve this pointer management in tweening. The details of tweening will be therefore covered in the *timeline* class.

3.1.6.3 Step Frames

The step frames will be frames related to the output of the program, which will be run by the RTE. Steps are turning points in our last assembly animation and when the user wants to advance in the result animation in RTE, the user will jump from one step frame to the other. Each step frame is also a key frame, so that step frame will be a sub-class of key frame. Instead of having a separate class we could have added only a simple list pointing to step frames but we think that step frames can have some additional attributes which we cannot foresee from now.

3.1.7 Object Base Class

The object Base class will be a main class for managing all the objects which are being rendered to the scene. The object Base will contain the physical objects, not pointers to them. As this class will have access to all the objects,

this class will provide all the control over the objects. These controls will be done by having a selected objects list and with respect to the inputs from the user, the selected objects will be affected from the modifications. Whenever a new object is added to any of the frames of the current timeline, a new object instance is created in this class's object list. As each object has its own methods to handle modifications, the Object Base is responsible for choosing the true object to be affected by any of the modifications. A class diagram showing the properties of the sound manager class can be seen in Figure – c7.

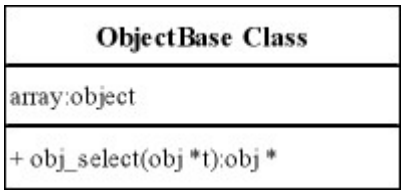


Figure-c7 Object Base Class

3.1.8 Timeline Class

The timeline class will hold the frame list as its data field. As this class will have a general access to all the frames in the object, it will be responsible from all the modifications and animation creation sequences. A class diagram showing the properties of the sound manager class can be seen in Figure – c8.

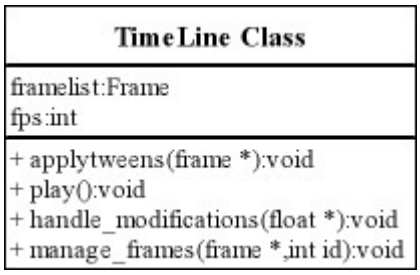


Figure-c8 Timeline Class

The *timeline* class will hold frames which are each having a *scene* object in them. And as stated above the scene class will have pointers to the objects

which are existent in the current frame's scene. This data hierarchy means that the *timeline* class will have access to all the objects drawn on any frame of the timeline. So that, we will send any modification request to individual objects within this class. Moreover and more importantly we will deal with tweens, which will be our main tool for creating animations, will be achieved within this context.

When a call from the event handler comes, the timeline passes this to the object or objects and the modification is achieved.

The most important functionality of this class will be achieving tween operations. To handle tween operations, we will need the properties of the object involved in the tween operation. What we need to define the behavior of an object in a specific object is its modelView matrix. To accomplish tweens we need to define the behavior of an object between two key frames. So, we first get the object's matrix from two consecutive key frames. After that we have to find how much difference has occurred by comparing these two matrices. As all the modifications will be cumulated in the object's matrix, the difference between these two matrices gives us the parameters of the change in the object's attributes. After we have the difference matrix in our hand, we have to apply this modification matrix step by step to each normal frame which lies between these two consecutive key frames. To illustrate, let's say that in a key frame our box object is at position (0, 0) and at time eleven, there is another key frame and our box is at position (10, 0). Also assume that this box is involved in a tween. Between these two key frames we have ten normal frames. In each of these normal frames, the box must make the 1/10 of the total translation, if it makes this translation in one frame, it will appear as moving and will reach the destination point of (10, 0). To calculate these modifications, we first subtract the matrix of box at frame one from the matrix of frame eleven. This gives us the total modification matrix, by dividing this matrix to the number of normal frames between these two key frames; we have a new modification matrix. This matrix; in our case, will be a matrix to translate the object one unit length in the positive x direction. After this

matrix is applied for ten times consecutively, the box will be at the final destination. The same sequence will hold for the rotation and scaling tweens. When the user modifies an object, the timeline class will check whether this object has been involved in any tween and if it does, the above procedure is applied. To apply this procedure, the object's matrix is taken from the nest key frame and the current matrix is taken from the current object. After that the tween calculation method takes these matrices and finds a new matrix to achieve this tween. Then this tween matrix is applied to all of the normal frames in-between the two key frames. This process is applied whenever an object involved in a tween is modified or a new object is made a part of a tween. When an object which was a part of a tween is deleted, the normal frames between the current key frame and the previous key frame are affected. The object's matrix should be updated and must be equal to its matrix defined in the previous key frame. Also when a normal frame is inserted in-between two key frames, the tween matrix should be changed and applied again because instead of "modification / # of normal frames", we will now have "modification / # of normal frames + 1". To reflect this slight but crucial change, we have to apply this new matrix to all of the normal frames. The same procedure also applies for deletion of objects. There will be some more special cases while dealing with tween operations, therefore we will very careful while implementing this part.

One important duty of this class will be managing the timeline. Possible user requests can be adding/removing a key frame, adding /removing normal frames, setting up a fps value and quick view method, to see the current project while it is being constructed. Whenever a change is made to the timeline, the object's attributes have to be modified to reflect the new changes as described above.

The quick view method will play the animation sequence, which has been built so far, in our main sub-window. In this method, the timeline frames' *scene* classes will be rendered to the scene consecutively. This rendering will be in a simpler way than what the RTE will display. This quick view will be done with

objects not having any textures or materials enabled. This can surely be done in wireframe mode.

We have to cover how this class handles frame management. When a new key frame is added, the time between these two key frames will be filled with normal frames. All these normal frames and even the new key frame itself are filled with the data of the previous key frame. Adding a new key frame doesn't only affect the timeline but also the objects. Remember from the *object class* definition that every object include its own timeline behavior matrices as its field. We have to add the new key frame and normal frame values and expand the *object class*'s timeline list. This is done here for each object existent in the current key frame.

A Level-2 DFD can be seen in Figure-D4. This DFD is a supplementary visual for what has been covered in this section; it explains how the data flows in the timeline component.

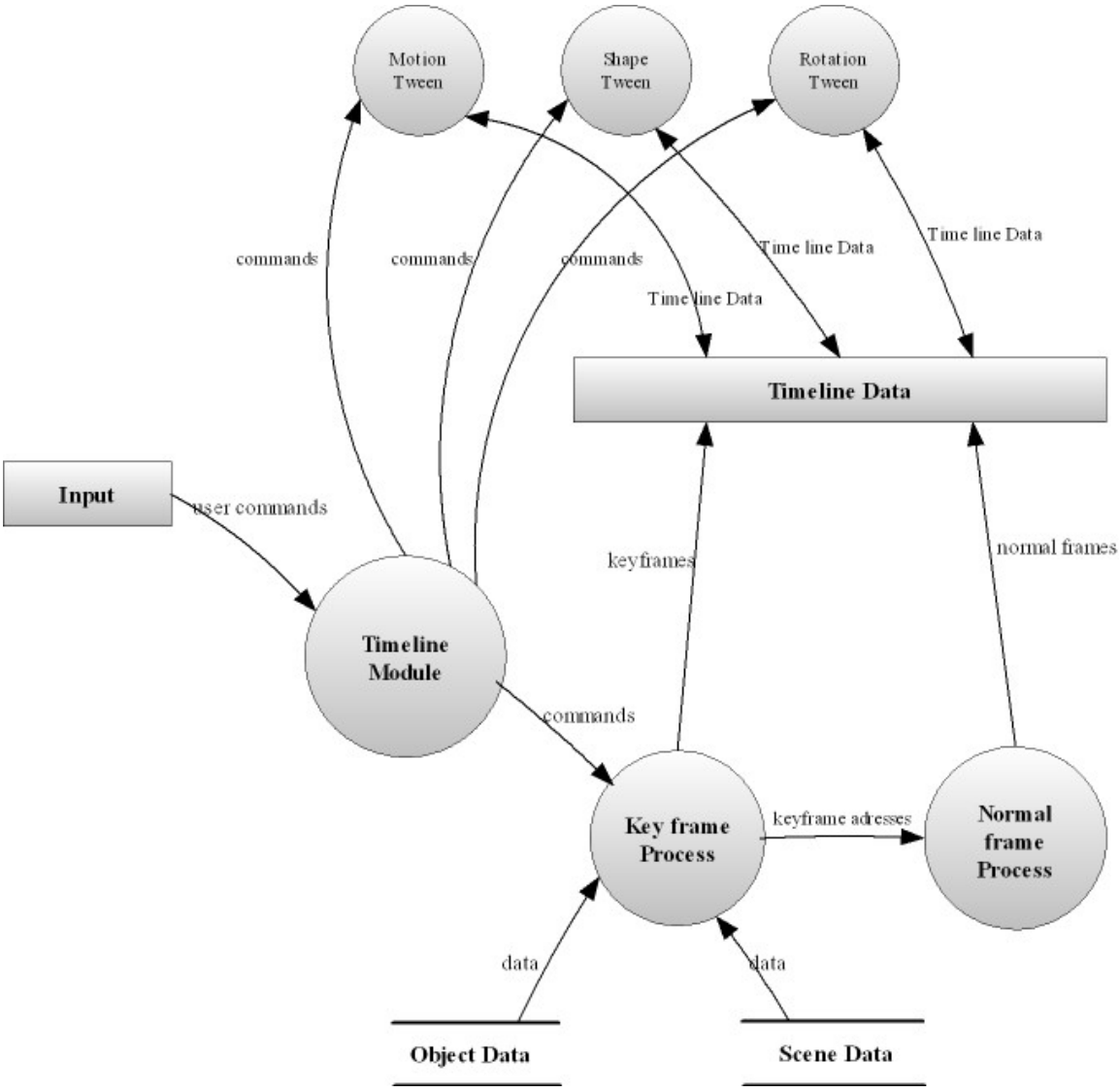


Figure-D4 Level-2 DFD for the Timeline

3.1.9 Event Handler

This class will be responsible for handling all the callback coming from the user interface and user input peripherals. We will have a large number of user inputs coming from the user interface and each of these operations will be handled by different classes and the system will give different responses. In order to handle these callbacks in an organized manner, we will have a main event handler class. This class will have a unique function for directing the

appropriate callback to the correct module. For example, when the mouse button is pressed in order to select an object, the mouse pressed event will be passed to the event handler class and this class will call the Object Base's selection function and the object in consideration will be selected. This class will assign different ids for each callback coming from the user and with respect to this id choose the correct function. We have a DFD to visualize the behavior of this class, which can be seen in Figure - D5. As this class will not act as a standard class but as a collection of handler functions, we have not added this as a class diagram.

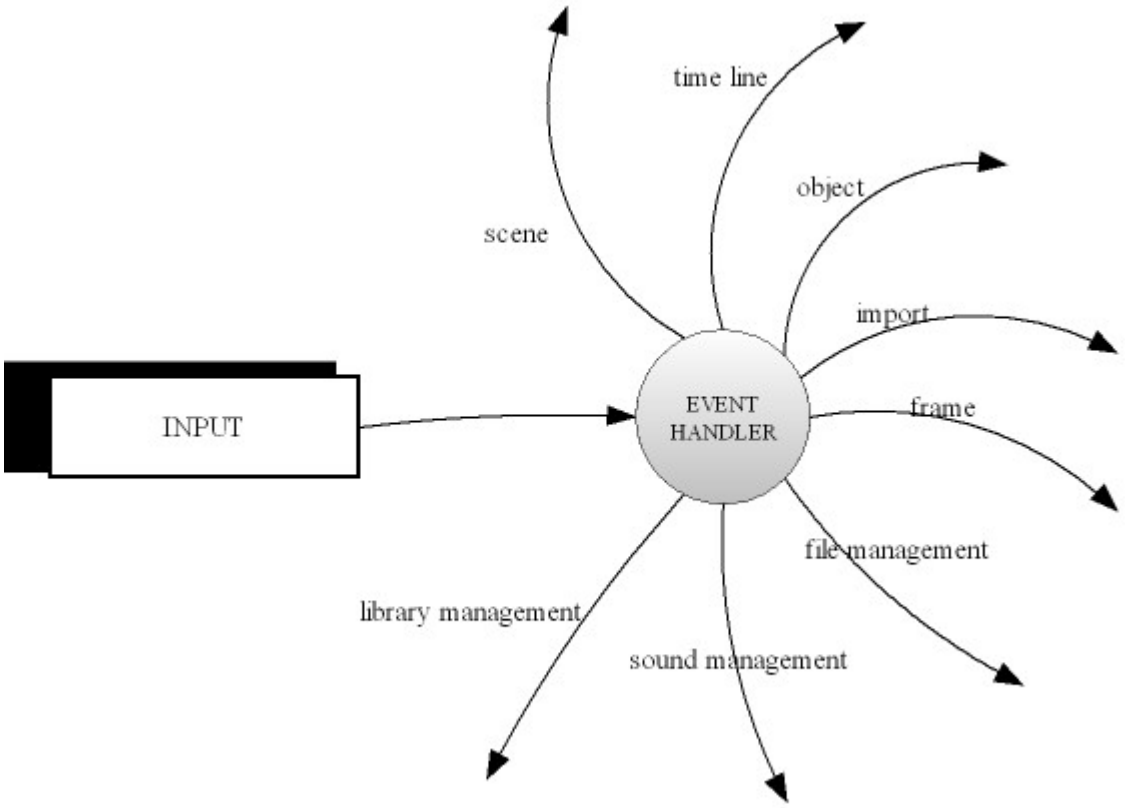


Figure-D5 Level 2 DFD for the Usage of the Event Handler

3.1.10 Math Library Class

This class will contain all the necessary mathematical operations in our project. Matrix multiplications, matrix addition, vector operations, vector –

matrix operations, constructing essential modification matrices, other matrix operations and all types of mathematical operators which are not defined in the standard math library will be implemented here; this class will be available to all the classes which are doing some sort of mathematical operations.
This class will be implemented as static and as many of the classes will need to use this class, we will not need to create lots of instances of this class. A class diagram showing the properties of the sound manager class can be seen in Figure – c9.

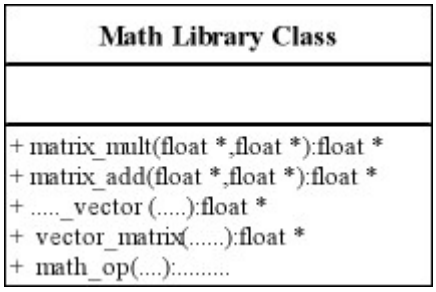


Figure-c9 Math Library Class

3.1.11 File Operations

We will have a number of file operations in our project, these will include:

- Reading a 3ds file for importing purposes
- Writing obj files for storing objects stored in the library
- Writing a header file to organize the files included in the shared library
- Reading obj files to load object data
- Saving the current project into the disk
- Writing the output file which will be played by the RTE

We are now giving details of some of these operations.
For reading a 3ds file, we will use 3ds Max's SDK. There are lots of importers written to fulfill our needs but we will write an importer for our own needs but get help from these ready-made importers.

An obj file is very easy to read and write, so this file format will be very easy for us to handle, brief information on obj files are given in the file part.
While saving the current project on the disk, we will have to save each part separately. For the content of the current library we will write separate obj files for each library object. To hold the scene and object data, we will write the contents of the Object Base's objects' data and the positions of the camera and lights for each frame in the timeline. All the data we need to present the current animation project will be saved in the file. A class diagram showing the properties of the sound manager class can be seen in Figure – c10.

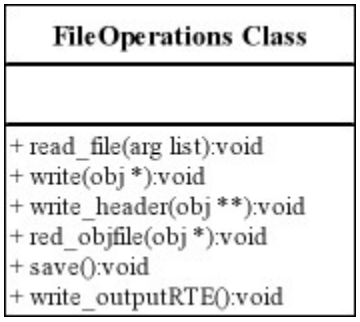


Figure-c10 File Operations Class

For the output of the editor which will be played by our RTE, we will have a detailed file which looks similar to the saved editor file. As the RTE will be a limited version of the editor, the data it will require is more or less the same as the editor.
This class will call the necessary file operation when requested from the user. A use-case diagram illustrates the available operations related to this class (Figure – u2).

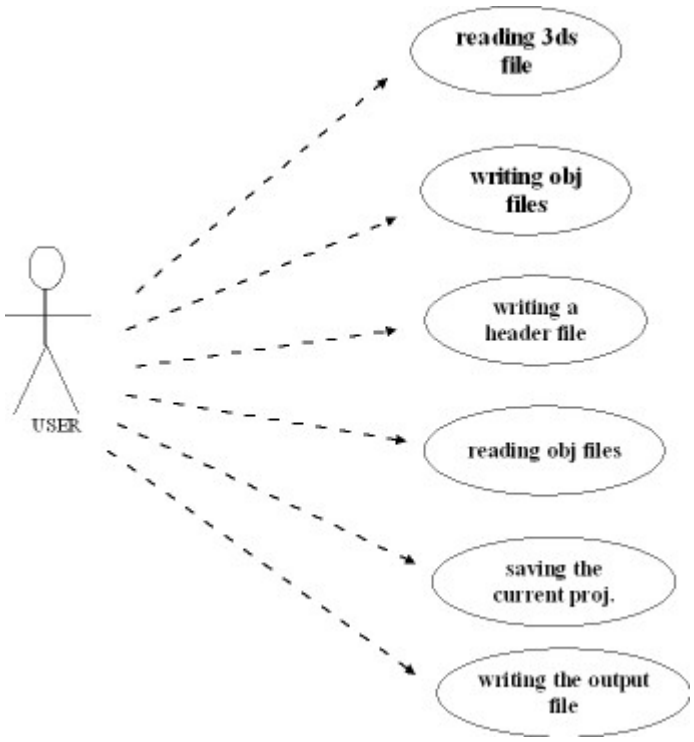


Figure- u2 A use-case diagram for File Operations

3.1.12 Window Management, Rendering & User Input

The window management will be done via .NET's MFC components. We will use MFC to create a main window and sub-windows in it. Also we will use MFC components to create our user interface; therefore we will have pre-made components to handle user input on those fields, i.e. buttons, checkboxes, etc. For rendering we will depend on our Open Scene Graph Engine. It will handle all the light, camera, material and texturing business. We will have different cameras associated with each of the sub-windows. Moreover, we will use osg to get user input from the rendering screen. For example, to detect mouse motion or mouse clicks we will use osg's functions. After getting the required parameters from these functions, we will call the event handler class to handle this user input.

3.1.13 Overall Class Relation Diagrams

In this section we have a general class diagram, covering all the classes covered in the above sections. The first diagram (Figure-c11) covers the relation between the classes object, library, object base and file operations.

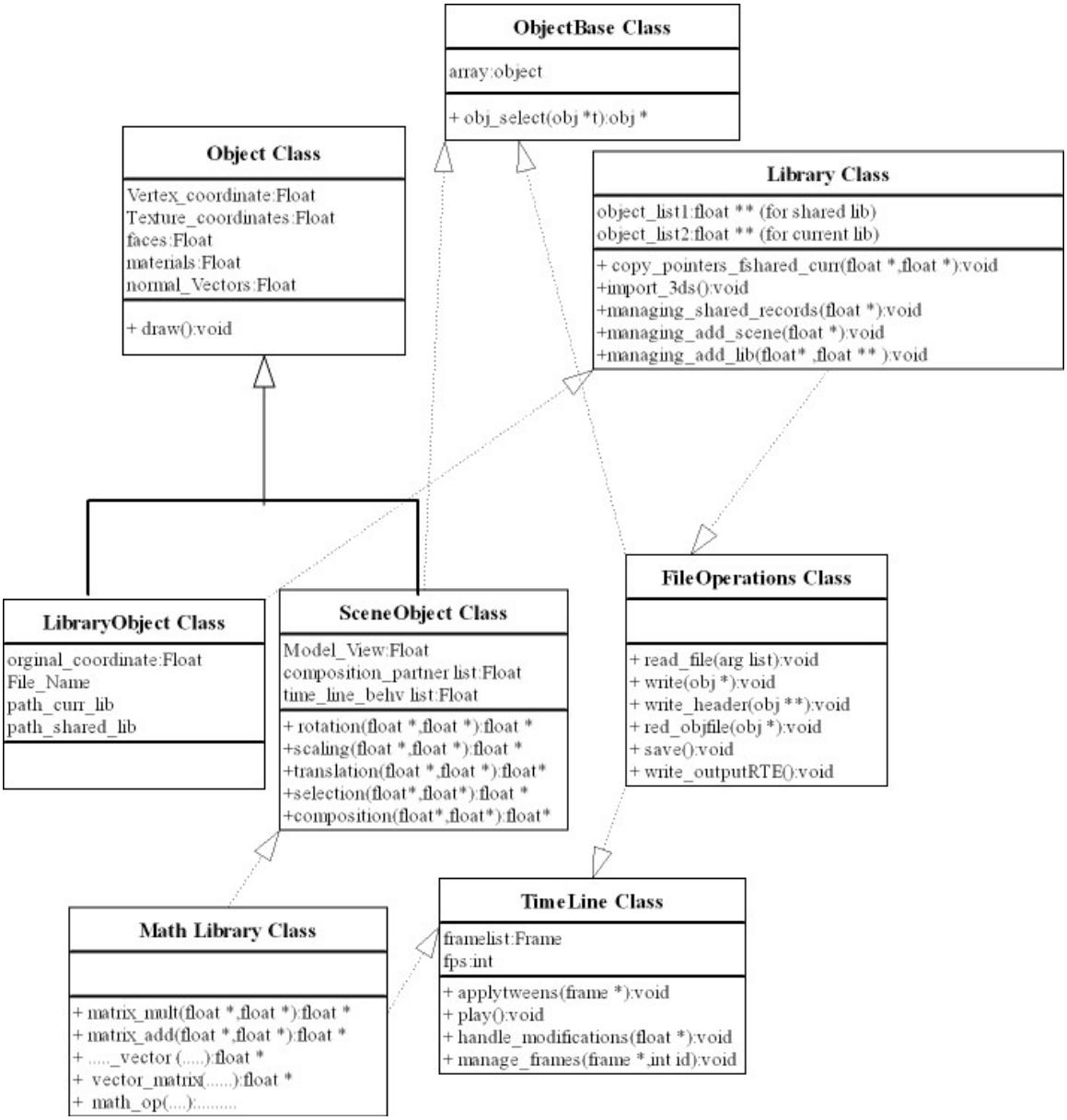


Figure-c11

The second class diagram covers the relations between frame, timeline and scene classes (Figure-c12).

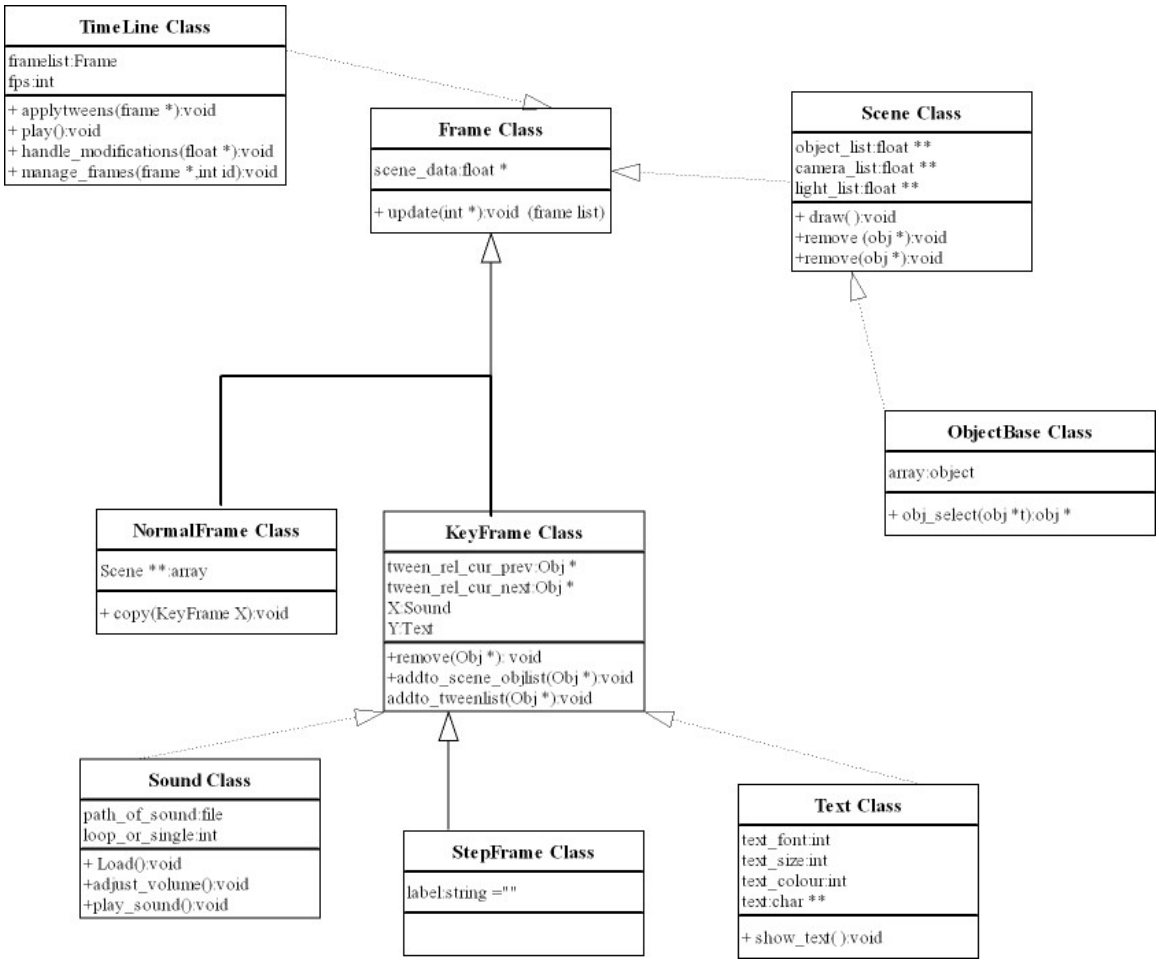


Figure-c12

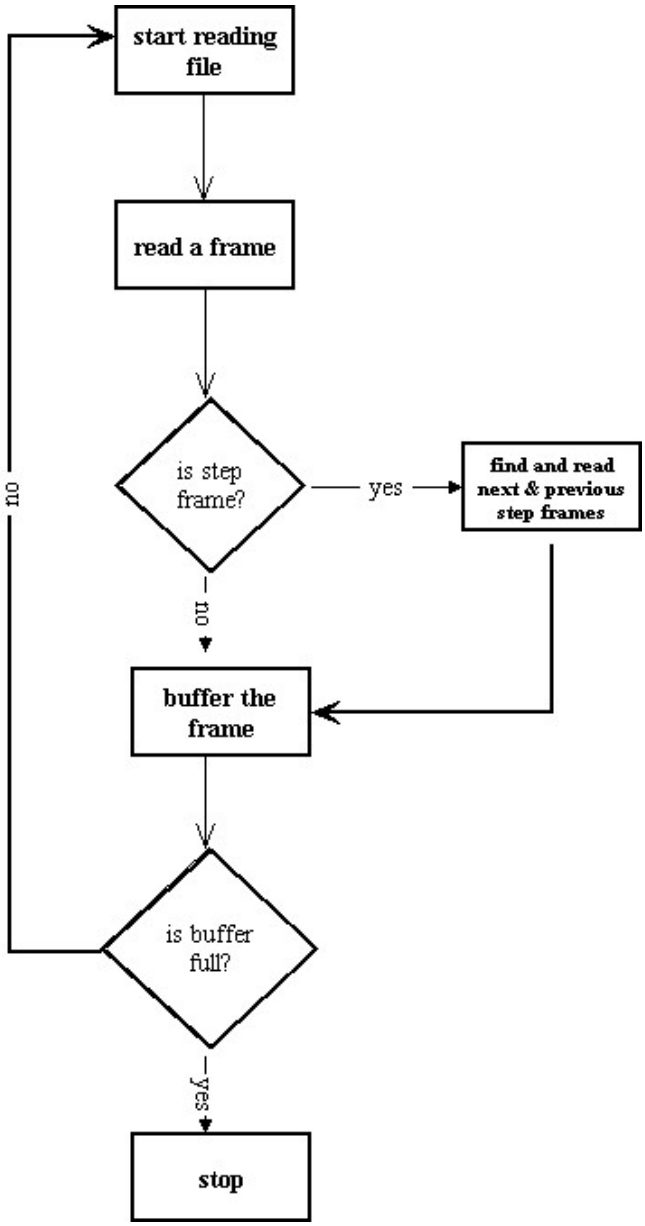
3.2 Run Time Environment

Run time environment application is a complete product of design tool which is our animation editor. Since run time environment is complete product can be also seen as an output of animation editor with all its executable files, (probable dynamically linked libraries) and animation file which will be designed in .pgd file format that is described in this document in detail.
Due to its being just one product or output, run time environment has the capability of running just one specified animation file so that it has no option to run a different animation file. Behind this application, all run time environment subfolders, apart from the file names, all executable files (and probable dynamically linked libraries) will be same.
It is crucial to note that in our program design tool (or animation editor) is our main program that's why it has all capability and functionality that run time environment has. Design tool can run the animation which is just created, can stop the motion, travel in the current frame, view the objects and scene from different aspects, can zoom in and zoom out. Considering all these functionality, our run time environment is very similar to a part of design tool. Our run time environment is using many classes and functions of design tool. Main difference between the run time environment and design tool is run time environment is currently reading the frames from the specified file for animation, but design tool with the help of timeline class is calculating the normal frames and runs the animation which means design tool is reading frames from the memory. Due to this slight difference, run time environment has an additional class to the classes described above which is called FrameBuffer Class.
Run time environment is currently designed to use Sound Manager, Text Manager, Scene, EventHandler, Frame, ObjectBase and FrameBuffer classes.

3.2.1 FrameBuffer Class

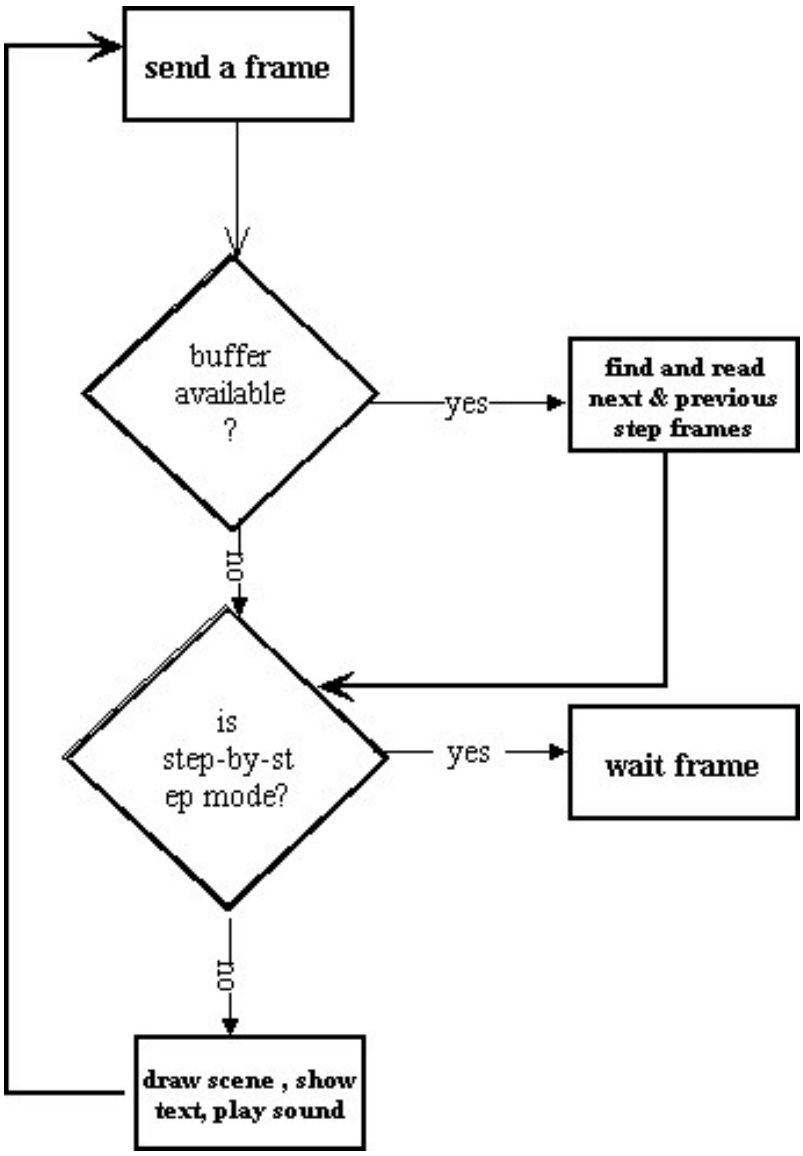
FrameBuffer class has a buffer mission in adjusting file read speed and frame display rate. Class has a frame-list constructed of *Frame* class which is the actual buffer, a waiting frame indicator to quickly select which frame will be next in a condition of just after transition from step-by-step mode to assembly animation mode, an *ObjectBase* class to keep all objects physically, next and previous step frame indicators as *Frame* class, and file reading methods.
File reading starts with *FrameBuffer* class. With the start of reading a file, firstly file heading attributes such as buffer size (number of frames to be kept in frame-list), frame per second information (number of frames in a second is going to be displayed and probable, but not certain yet, properties like regular frame displaying or not) are read by *FrameBuffer* class.
After reading heading attributes *BufferFrame* gets objects specified in the file to the *Objectbase* class and keep them physically in the class.
After all, buffering frame operations start, *BufferFrame* class reads the frames from file and fills the frame-list. While reading the frames, methods of the class check each frame if it is a step frame or not. And if it is a step frame methods automatically find and set the previous and next step frames of current frame. This is because, in a condition of user's selection of previous or next chapter to play, buffer automatically draws the scene of frame to the screen and shows a waiting icon on mouse arrow while *BufferFrame* class empties existing frame-list and fills it up again reading from the next frame of specified step frame and when buffer is full, *BufferFrame* begins its routine reading and sending frames for rendering coherently again.
Reading frames to buffer and sending them for display algorithms stated below. These are separate operations that are handles coherently.

Buffer Read Algorithm



Buffer starts reading a file, after reading a frame checks if frame is step frame or not. If it is step frame, methods find and read next and previous step frames of current step frame in order to smooth transition to next chapter. This operation continues until buffer is full.

Buffer Sending a Frame Algorithm



Buffer sends frames continuously and checks at each step if buffer is available for reading new frames if available starts the buffer read algorithm. And at each step mode of run time environment, if it is in step-by-step mode buffer sending goes into a waiting mode if it is not object drawing algorithm continues.

It is explicitly stated before that run time environment has two run time options namely; step-by-step and assembly animate. In the assembly animate mode user just watches the animation so *FrameBuffer* class and frame drawing methods work coherently but when user selects step-by-step mode, animation holds the currently displaying frame and reach the scene camera position information. After getting this information, all .NET interface functionalities and mouse functionalities such as zoom in, zoom out, left, right, up, down, backward, forward movements are a matter of camera position. Apart from camera, all current objects in the scene are going to be rendered as same.
A use case below shows these activities (Figure-u3).

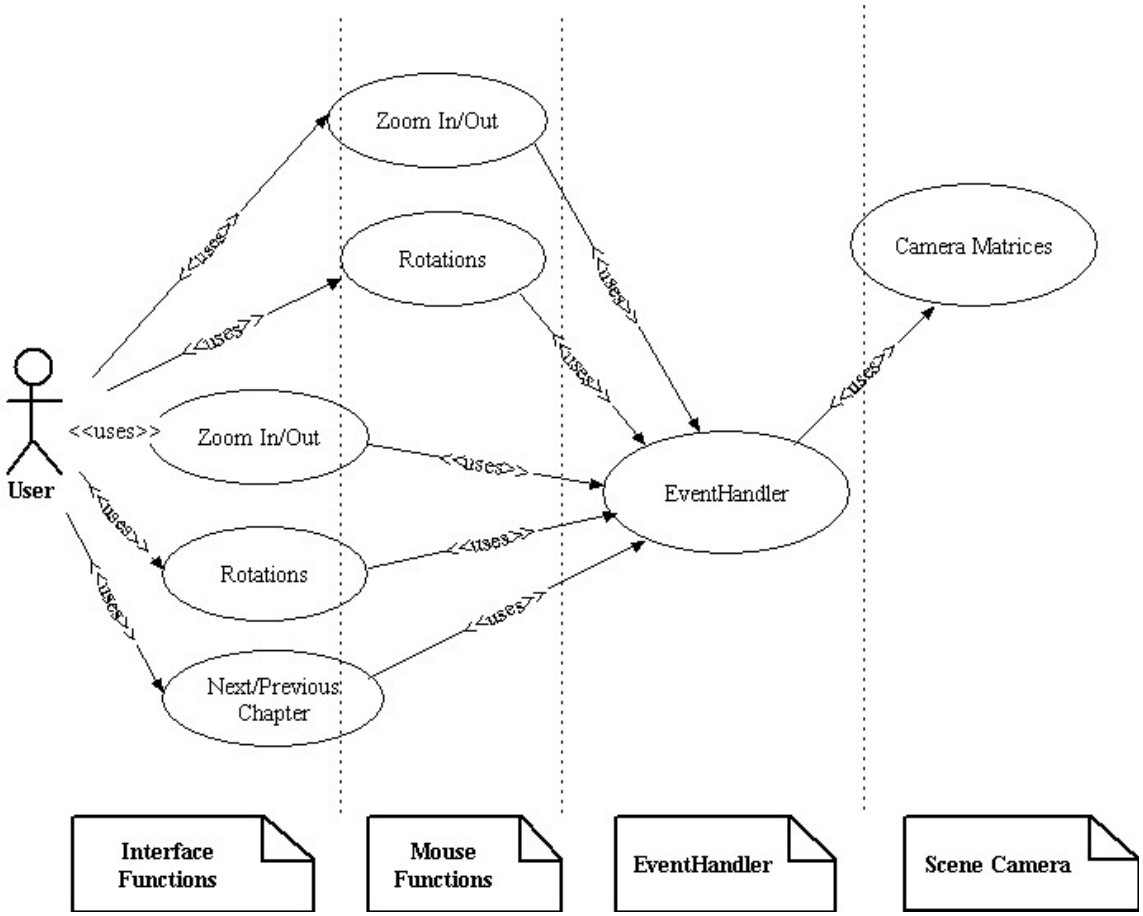


Figure-u3

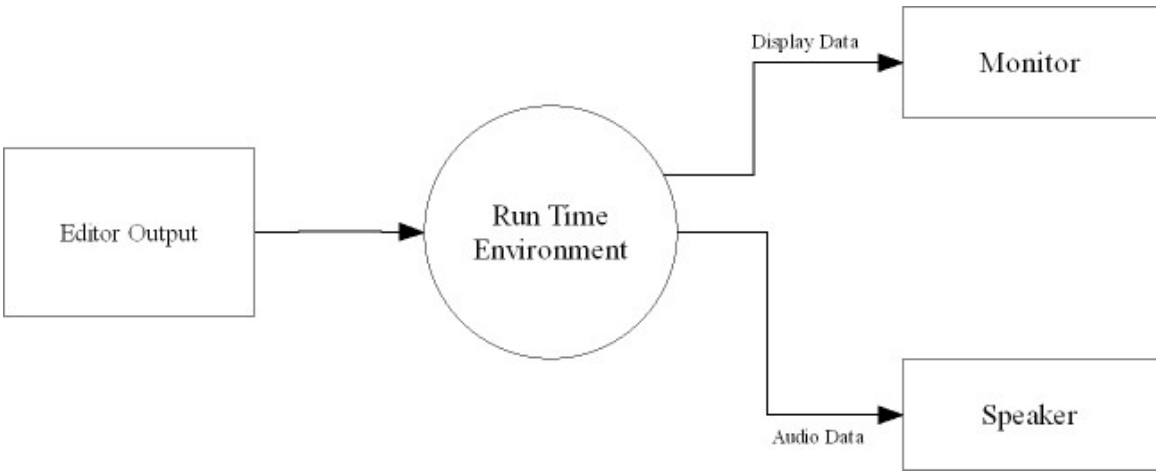
In the use case user manages the functionalities of interface and mouse this information is sent to *EventHandler* class and *EventHandler* class apply this to the camera matrices.

Camera is going to be used to travel inside the scene according to user inputs.

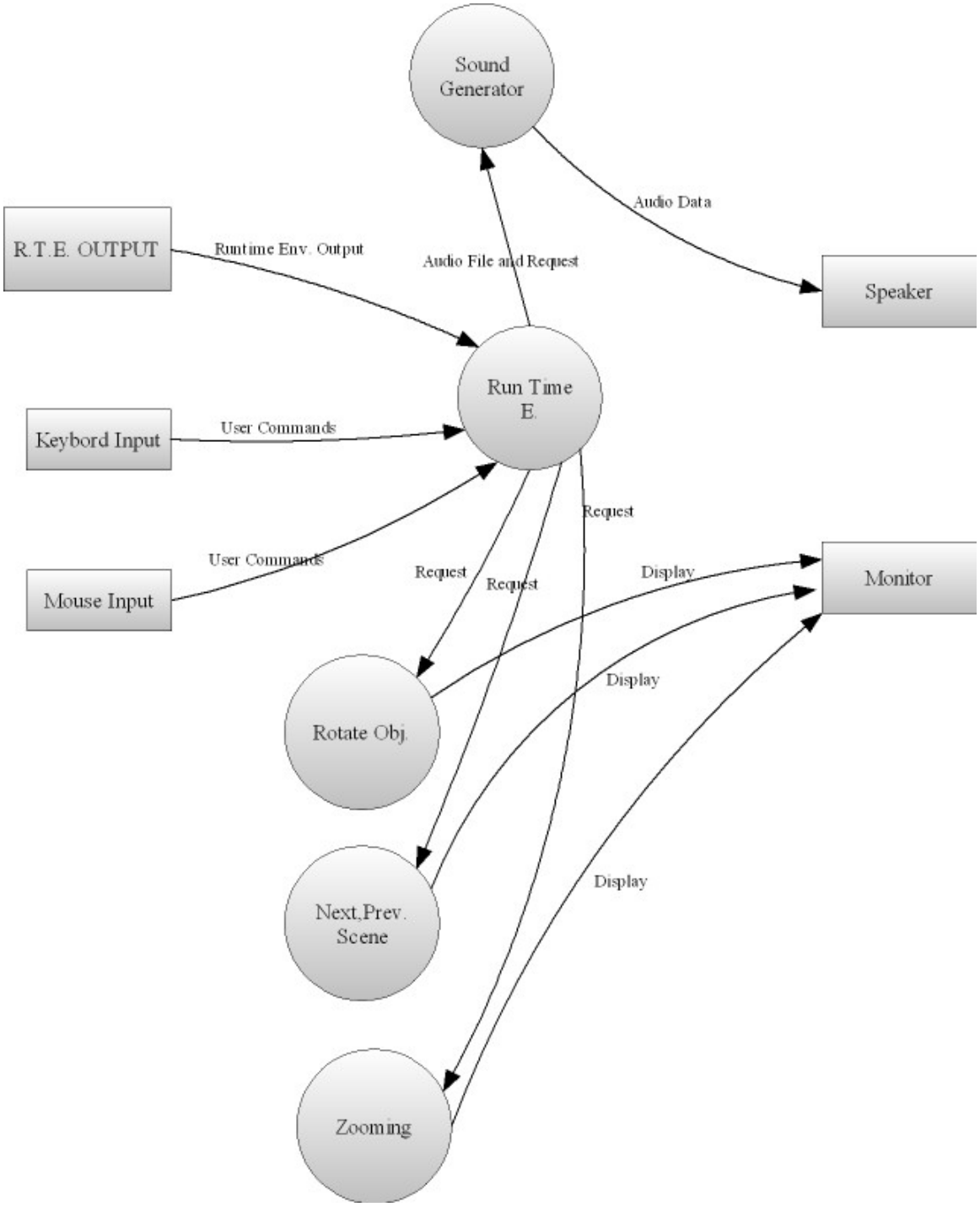
Camera matrix is designed as an osg::Matrixd matrix and defined as the product of two osg::Matrixd cameraRotation and osg::Matrixd cameraTranslation matrices. Rotations will be applied to cameraRotation matrix, translations will be applied to cameraTranslation and the product of two (camera = cameraRotation * cameraTranslation) constructs the camera matrix. With the help of camera matrix, camera movements are accrual by Open Scene Graph Viewer class setViewBymatrix() method. After user selecting the animation assembly mode firstly the waiting frame comes to the scene and *BufferFrame* routine goes on.

Finally data flow diagrams are stated below.

DFD Level 1



DFD Level 2


Sound Generator
Audio Data
R.T.E. OUTPUT
Runtime Env. Output
Audio File and Request
Speaker
Run Time E.
Keybord Input
User Commands
Request
User Commands
Request
Request
Display
Monitor
Mouse Input
Rotate Obj.
Display
Next,Prev. Scene
Display
Zooming

3.3 FILE FORMATS

We will have a number of file formats used in our project. We will cover them one by one:

3.3.1 .obj File

This file format is a very common and popular file format used very widely in 3D applications. We will use this format to keep our objects in our library. Here is a brief overview of how this file will look:

Vertex data

- geometric vertices (v)
- texture vertices (vt)
- vertex normals (vn)
- parameter space vertices (vp)

Elements

- point (p)
- line (l)
- face (f)
- surface (surf)

Grouping

- group name (g)
- object name (o)

Display/render attributes

- material name (usemtl)
- material library (mtllib)

Here is a sample representation of a simple cube in .obj format:

```
v 0.000000 2.000000 2.000000
v 0.000000 0.000000 2.000000
v 2.000000 0.000000 2.000000
v 2.000000 2.000000 2.000000
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 0.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
f 1 2 3 4
f 8 7 6 5
f 4 3 7 8
f 5 1 4 8
f 5 6 2 1
f 2 6 7 3
```

3.3.2 .objh File

This file will be positioned in the same directory of our shared library. It is used to manage and organize the obj files. It will hold the number of objects in the shared library and the names and paths of the objects in the shared library.

3.3.3 .ulp File

This file is our editor's save file. The data it will contain is as follows:

- The list of objects in the current library (the path and names of .obj files in the current library)
- The list of all the objects in the *Object Base* class with all the data of each object (coordinates, attributes and timeline details)
- The list of frames in the timeline (with their individual scene data and object lists and also camera and light positions)

3.3.4 .pgd File

This file will be used by our RTE and be the product of our program. In terms of content it will more or less be the same as our .ulp file.

- The list of all the objects in the *Object Base* class with all the data of each object (coordinates, attributes and timeline details)
- The list of frames in the timeline (with their individual scene data and object lists and also camera and light positions. Also data of the key frames which are holding sound and text attributes)
- Additional attributes for the RTE

4. USER INTERFACE DESIGNS

We have made rough sketches of our user interface. We have done these sketches in a drawing tool called SmartDraw. Here is a sample user interface for our editor (Figure-gui1).

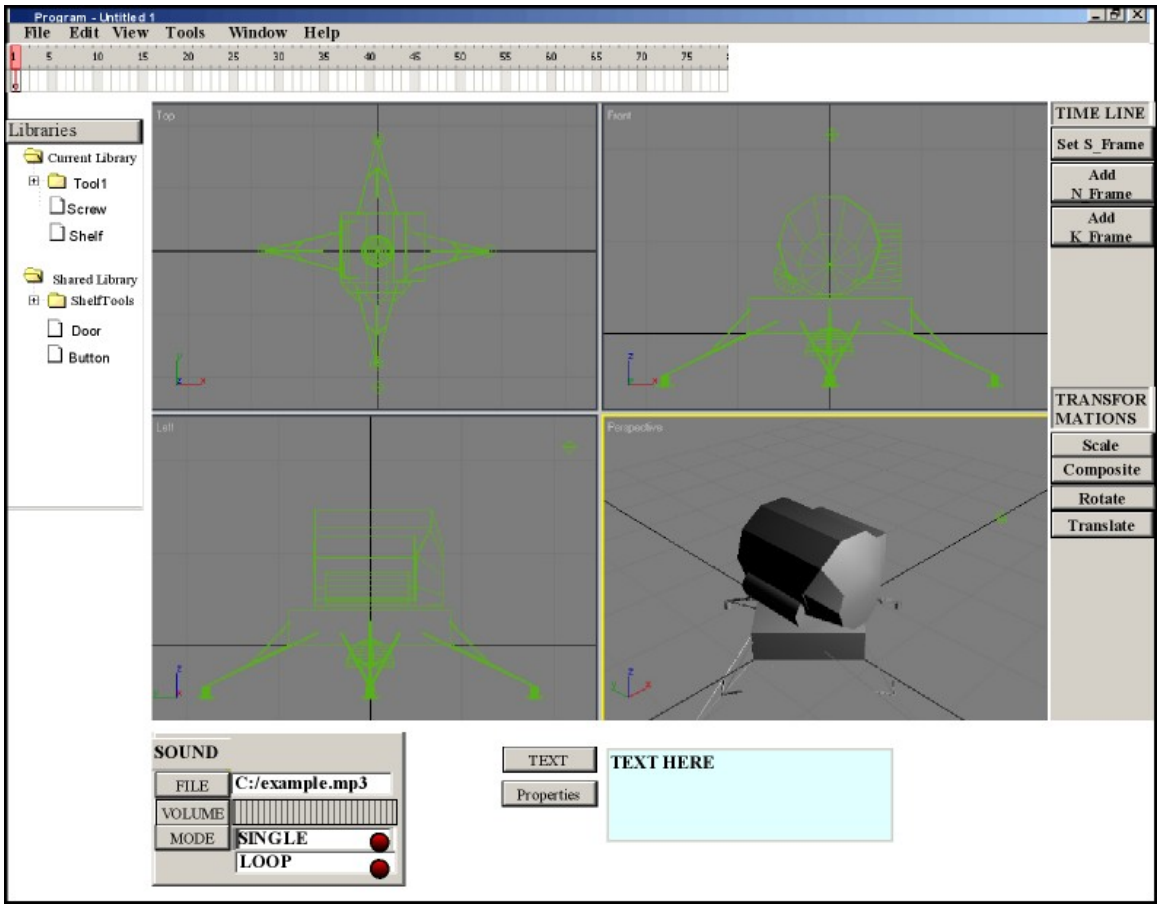


Figure-gui1

And here is a sample screenshot from our run-time-environment (Figure-gui2).

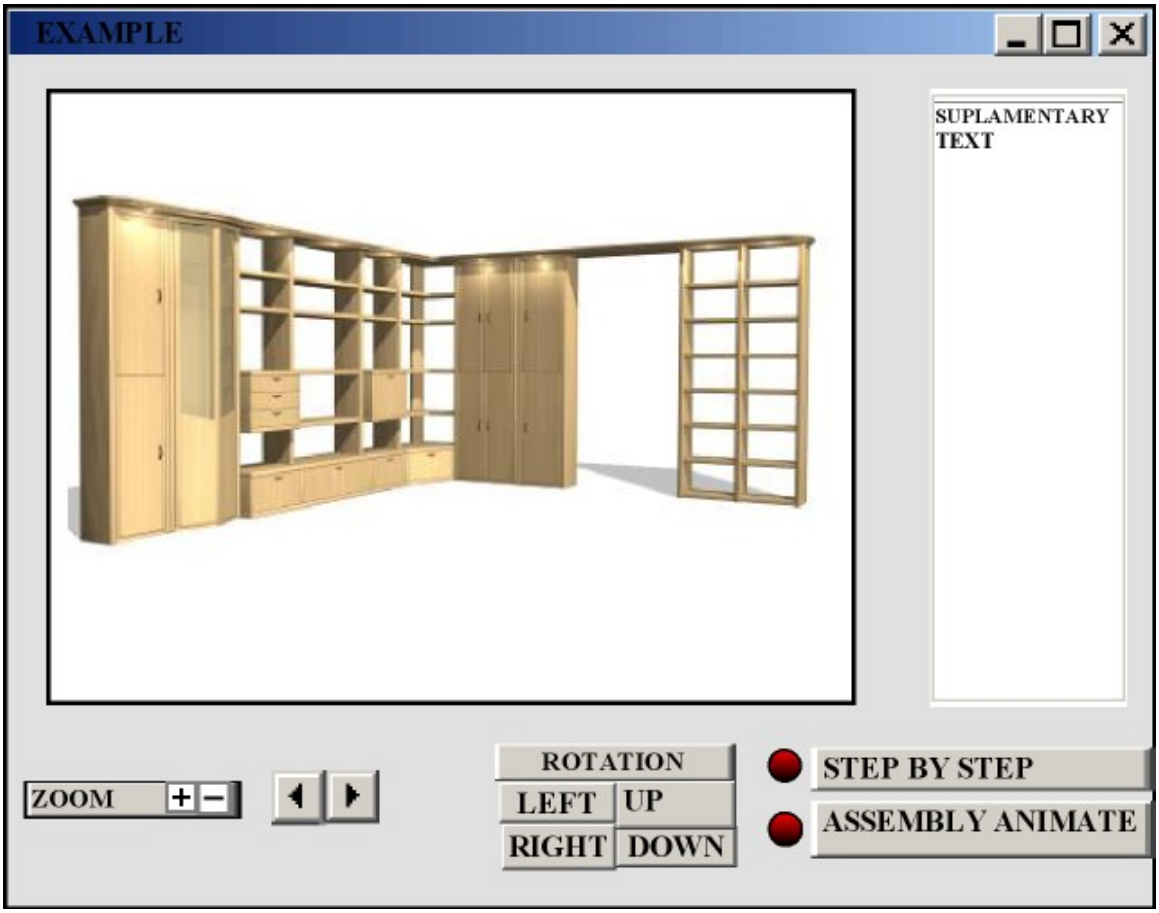


Figure-gui2

5. CONCLUSION

This report gives detailed design specifications of our project. We have tried to cover all the topics but there is a number of missing parts and gaps in our design. We haven't been able to concentrate on these gaps, but we are of their existence. We will try to cover these details better in the final design. This report is with its content; will be a great source for us while writing our final design reports and while implementing the project in the second term.