DETAILED DESIGN REPORT

by

MIR

N. Özlem ÖZCAN - 1250596
Hamza KAYA - 1250448
Mustafa Öztorun - 1250638
Ozan Şimşek - 1250760

INTRODUCTION

PURPOSE OF DOCUMENT

This document is the detailed design report of the 3D action/adventure game project of MIR. It is written according to results of the steps that carried out by MIR during the first semester of senior design project course of computer engineering in Middle East Technical University. Purpose of this document is to define the design specifications and system solutions for the game design.

SCOPE OF DOCUMENT

The overall architecture of the project is defined in this report. This report includes the detailed description and specification of the modules and the classes which build the game, major constraints of the project, graphical user interfaces, use cases of the player. It also covers the hierarchy between the classes, internal data structures, file formats. The diagrams structuring the game are revised and detailed. In addition, the document provides the customer views and needs. The information gained from customers is well studied and lead MIR to design a game described in the next section. Moreover, a schedule and coding standards of the game is provided at the end of the report.

PROJECT DESCRIPTION

Computer games industry is one of the leading areas in computer technology. Computer game industry takes the advantages of all improvements in main areas of computer science and technology such as computer graphics, artificial intelligence, algorithms, and technological improvements in hardware of computers. Therefore, game programming is a uniquely challenging area of software development that requires a combination of several different disciplines. Concepts like storyline, graphics, sound, controls, and play modes should be put together in consideration of any game design. On the other hand, in user side playing a game is not obligatory, it is for just fun. Since, concepts like game scenario, minimum requirements to be able to play the game, ease of control in game, attractive scenes, good sounds should also be well designed for success of a computer game.

Under the light of above information, MIR aims to develop a computer game named ‘1956’. The game is played in a simulation of 3D environment of Metu campus. As being an adventure type game, our hero will walk around, fight with enemies who have artificial intelligences, explore the environment and use weapons and some tools in the game. The game consists of 7 levels that includes some missions to be completed and puzzles to be solved. Main characteristics of computer games are implemented in the game such as saving and loading game facilities, changeable game options of graphics, sounds and controllers. Game scripting is supported to enable the player change the some parts of the game and ease the implementation of game by members of MIR.

MAJOR CONSTRAINTS

TIME CONSTRAINTS

There are so many complications in a complete computer game. We have limited time to complete the project. We have seven months to finish all implementation, documentation, testing, bug fixings and enhancements. Furthermore as senior computer engineering students we have courses except the project. We have so many interesting ideas about our game but we may not implement all of them due to time constraints. So we should manage the time carefully and organize our works.

PERFORMANCE CONSTRAINTS

We are trying to develop a computer game, not such a program that people have to use. They will take and play our game only for fun. Therefore our game can be played on a moderate PC easily. What we mean by a moderate PC is at least:

- P III 1000 MHz Processor

- 64MB GeForce-4 VGA
- 256MB Memory
- 16bit Sound Card
- Enough free disk space

Hardware resources of the players' machines should be used carefully. The game should not have bugs, flickering screens, lower frame rate etc.

DESIGN CONSTRAINTS

C++ will be used as programming language by letting us to use object oriented concepts. Microsoft's Visual C++ is used as coding environment. Open GL is the main graphics library. For graphical design part we are using Photoshop, 3Dmax and Corel. ODE library will be responsible for the physics calculations. Python is the scripting language that will aid us throughout the game. We said that Boost.Python library will be used for the interfacing of C++ and Python in the initial design report. However, results of the difficulties we faced in while trying to use that library, especially in linking parts, we decided to use another commonly used library, swig. Swig (Simplified wrapper and interface generator) is a open source tool that will ease the interfacing of c++ and python codes of the project. Microsoft Visio is the tool that will be used for technical drawings. CVS will be used for version controlling.

GAME DESCRIPTION

IN GAME VIEW

Game contains parts inside and outside of the buildings. While hero is inside the buildings player sees the world in first-person mode. This mode is mainly used for exploration and puzzle solving purposes. Hero isn't visible in this view mode. While hero is outside, third-person mode is used to show the hero and environment to the player. Camera is placed on the upper-right side of the hero. Camera moves relative to the hero to obtain a static view. Therefore, player always sees the hero with the same angle.

GAME FILES

The game has kinds of files to keep the whole related data on the disk permanently. Some of these files are read from disk when game is started and some is read when required during the game. All data is kept as texts in the files. There are level files, puzzle files, configuration files, script files, 3D model files, files of textures in the game. A save file is written to disk when a player saves the game. Detailed explanation of files and their formats can be found in following parts of the report.

MAP

The game map has a grid structure. Grid is consists of equivalent cells, in other words every cell has same dimensions. This design will ease texturing of the map part by part and routing of the moving objects. For outdoor environment, there will be only one map and each level will take parts of it accordingly. Each indoor environment will have its own local coordinates and local map, but the structure will be same.

The player can access his/her location information on the map by clicking the signboards which are distributed over the game world.

LOAD LEVEL

Game consists of seven different levels. Each of them has its own text files. The level files contain all information about that level. Each level file has two main parts. First part contains the information of static objects such as corresponding part of the map, coordinates of submap. The second part contains all the dynamic information, starting point of the hero, information of

informative animations being played during game, information of animals, vehicles, NHH[1], creatures, potions, money and weapons etc. This distinction is essential for ease of save and load operations. When a new game is started, content of these files is read and all internal data structures initialized accordingly.

SAVE/LOAD GAME

When a saving game operation occurs, a save file is constructed. All global variables and all information of all dynamic objects, our hero and puzzles are written in this file. Each object is responsible for saving its state. For loading a previously saved file, all globals are set again from the file and all information of static objects is read from the corresponding level file and the other dynamic information is read from related savefile. Each object is responsible for getting or setting its state during saving and loading.

PHYSICS

Open Dynamics Engine (ODE) library is responsible for all physics calculations in the game. Most of the objects in the game have a physical body. The collision detection operations between these objects are handled by ODE. In each time interval the collision detection system is called in the callback function and each contact points are calculated. By using these contact point we can specify the objects' position and the surface normal. A joint type which is a contact joint is formed for each point of collision. These joints are put into a joint group to speed up the calculation. But never forget that if the number of point increases the calculation time increases. A step of simulation is done and all contact joints are removed from the system.

AI

In action games AI is mainly used for enemies, partners, support characters etc… In our game we use AI to give specific acts to enemy creatures and other non-hero humans (animals also have a little AI). They can follow a still pattern or act totally randomly or a mixture of them. To give a roaming ability to an object, its velocity or position must be altered based on the position of other objects. The roaming movement of the object can also be influenced by random or predetermined pattern. To implement behavioral AI there is a need to establish a set of behaviors. It can be done by establishing a ranking system for each type of behavior present in the game and then applying it to each object. For example, for each different type of enemies we will assign different percentages to the different behaviors, thereby giving them each different personalities. An aggressive creature might have the following behavioral breakdown: chase 50% of the time, evade 10% of the time, walk in a pattern 30% of the times, and walk randomly 10% of the times. On the other hand, a more passive creature might act like this: chase 10% of the time, evade 50% of the time, walk in a pattern 20% of the times, and walk randomly 20% of the times.

The next movement of a creature (decisions made during the game) is also determined using AI. A cost is assigned to each available movements of a creature and one of the most beneficial movements is chosen among these movements. Because of the geography of the environment and the random distribution of the static objects, we need to implement a path finding method for all moving objects (hero, creatures, animals, non-hero humans etc…) in game. To achieve these goals we need to implement efficient search algorithms, since our search space is rather large. For decision making purposes we need to implement most of the uninformed search methods like depth-first search, breath-first search, iterative deepening search as well as informed search methods like greedy best-first search and iterative deepening A* search. Also shortest path algorithms are used for path finding problems in game.

SCRIPTING

There is a scripting system in our game. Although a game can be built without a scripting system, there are many advantages of using it. First of all, after implementing core part of the code in C++, we will develop some parts of the game like AI, with less effort in coding. It is also useful on some file operations such as loading or saving game. For instance, its pickle module supports serialization that can be used to save or load complex data structures. It provides flexibility and moddability. Also it provides an easy way of level creation and editing. All level information is held in files; hence this data can be changed with scripts that are handled by an interpreter without recompiling code. An interpreter may cause to run slower but it let us to code faster. Because of similar reasons we decided to embed a scripting system in the game. Instead of developing a

[1] Nonhero Human

language, an existing language is used. This will save time and we probably have a more powerful one. Python is used in the game as scripting language. It and its interpreter are open source. Therefore, if there is a need we can customize it according to our needs. Coding is very easy in python, only indentation is required to separate statement groups. Memory management is done by it and it has a powerful set of string operations. Large documentation and support can be obtained from the internet. In addition there are lots of open sources libraries for python over the net.

C++ is a static language which is compiled on the other hand python is a dynamic language which is interpreted. They must be combined in a way. To achieve this, an open source tool, SWIG is used. It acts as an interface between them. It allows user to expose C++ classes and functions to python. It does this operation by using C++ compiler. As a result of exposing a dynamic library file and a python file is obtained. Then exposed classes and functions can be reached by using python. Moreover, by including required header files, we will make python calls directly from C++ code. An example call is made below:

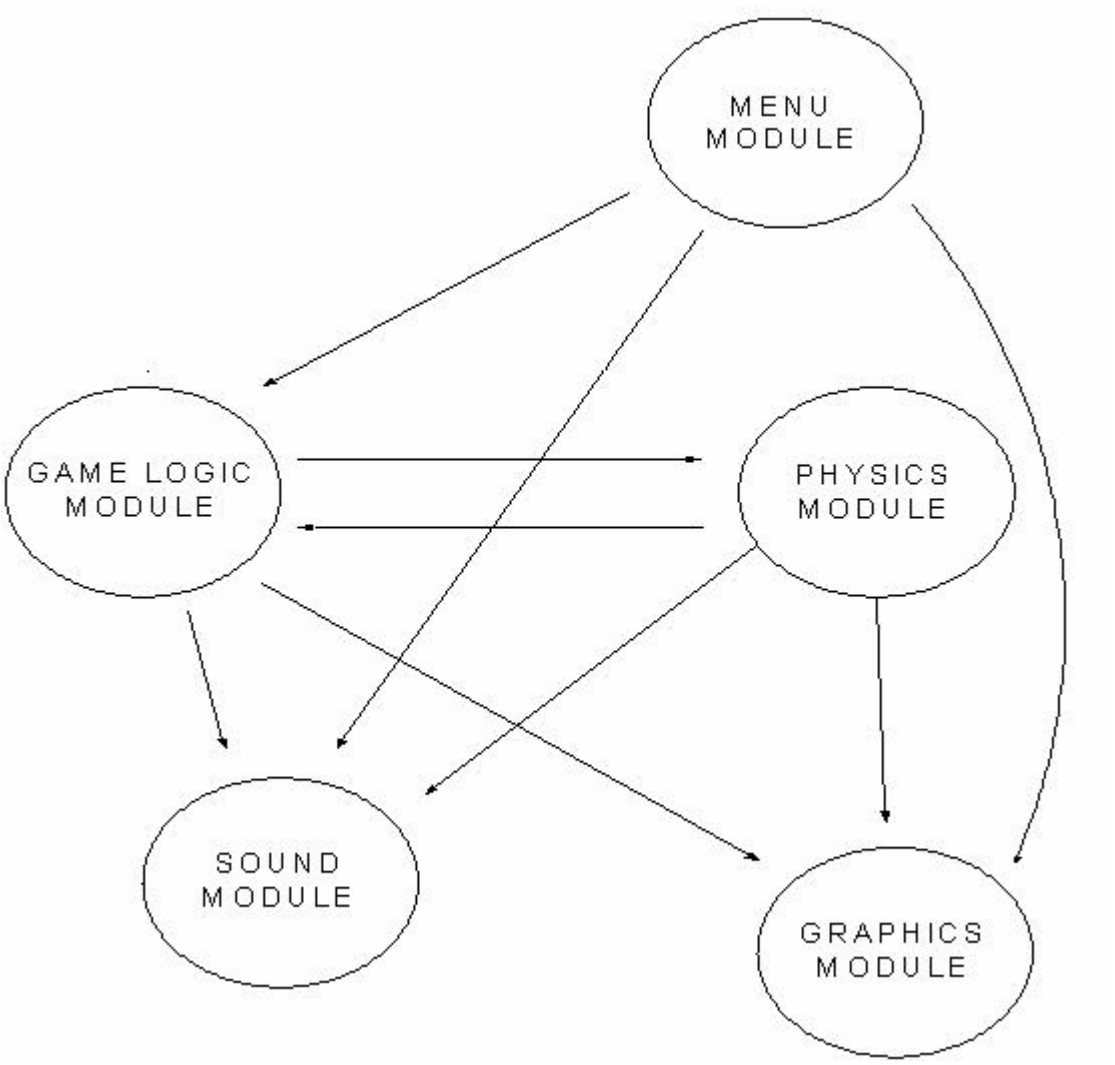
```
// Other header files
#include <python/python.h>

CallerFunction ()
{
   Py_Initialize();
         // Python calls will made here.
   Py_Finalize();
}
```

CUSTOMER INTERVIEW

According to interviews we made with some computer game players (customers) there are many reasons for a game to reach success. Here some common ideas: first of all a game can be played on a moderate pc, it should not require higher hardware requirements. Control of game should be easy and controllers can be customized. It should have a simple menu and walking through the windows of menu should be easy. The player should be free in game environment. S/he should not be forced to follow a predetermined path. Game story should be easy to follow and levels should be related to each other. Music is important in the game. Same background music should not play continuously. It should be melodious with game environment. Camera views are one of the most important parts of the game. There should be some options for cameras. During game some events should occur not related to player, it should have a level of AI. The game should not have similar features with most of the games in the market. There should be informative videos or animations but should not be boring, long conservations. Loading time should be short. It should not be too easy or difficult to finish the game, a balance should be provided. Objects in the game environment should be attractive and exotic. They do not have to be realistic all the time. Saving the game should be easy and there should be no need to save the game frequently.

SYSTEM MODULES



MENU MODULE

The menu module is responsible for the menu operations from the opening of the executable file till the starting of the game and the pause menu in addition. The possible inputs for this module are mouse clicks for buttons and text entries for the name of the hero.

GRAPHICS MODULE

Graphics module handles all of the rendering operations during the game. According to the feedback taken form the game logic module and physics module, this module renders the next frame for each iteration of the game loop.

SOUND MODULE

Sound module is responsible for playing the sound effects and game sound of the game. This module runs parallel with other modules, in other words this module is active during the whole game. Game sound information is gathered from the level data and the actions in the game. On the other hand sound effects are played according to the player inputs, warnings and errors occurred in game.

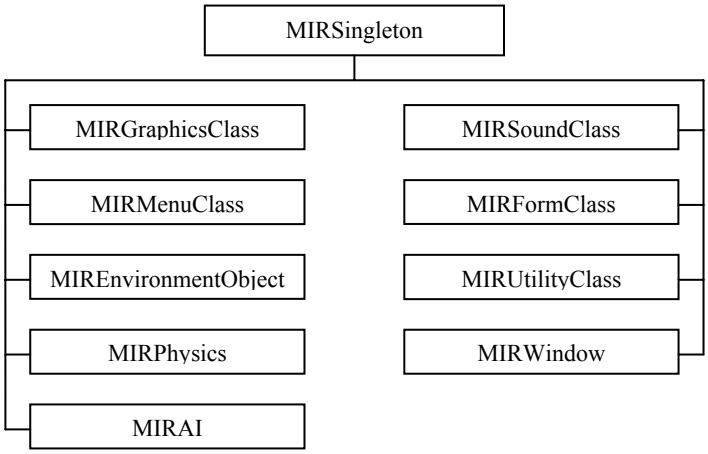
GAME LOGIC MODULE

Game logic module is the main module that deals with the decisions made in game. It's strongly related with AI. The behaviors of the creatures and other autonomous objects are determined by this module. Decisions are made according to the player inputs and the state of other objects.

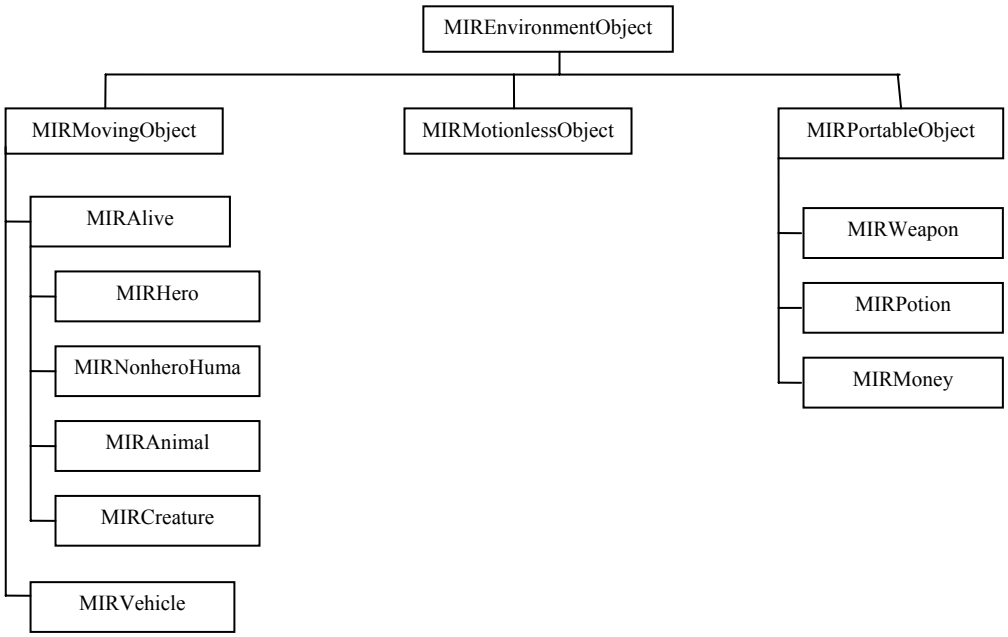
PHYSICS MODULE

This module handles the interactions of the physical objects. Using the feedback of the input handling module, collisions occurred in game environment are detected by this module and the state (position, velocity etc…) of the participating objects are updated accordingly. The updated state is passed to the graphics module for final rendering operation.

CLASS HIERARCHY



There exists a base class MIRSingleton. All other classes are built on this class. This tree is expanded and explained in detail in following part of this section.



MIREnvironmentObject

- m_iaPosition
- draw ()
- getState ()
- setState ()

MIREnvironmentObject class is the base class for all objects that are in the environment where the player controls the hero of the game. It has an array named m_aPosition which holds the rolative position coordinates of the object in the environment and a function named draw that draws the object in game window. getState and setState functions are essential when saving and loading game. getState returns the momentary state information of owner object. setState set the state of the object when loading is being done.

MIRMotionlessObject

- m_ iBodyId

MIRMotionlessObject class is the main class that will handle the static objects in the game like buildings, trees, statues etc.

MIRMovingObject

- m_iDirection
- m_iVelocity
- m_iType
- m_iTrackId
- playTrack( )

MIRMovingObject class is the base class for all objects that can move or be moved in the game environment. Every object of this class has a velocity and a direction.There is also a unique type for each of them. m_iTrackId is the unique identifier of the track which is played by playTrack function of this class.

MIRAlive

- m_iHealth

MIRAlive class is the class for all objects that are living in the game. m_iHealth is a bounded integer that holds the value of health of the objects.

MIRHero

- m_iWeaponInHand
- m_iMagicPower
- m_iStamina
- m_iIntelligence
- m_iSkill
- m_iSpell
- walk( )
- talk( )
- hit( )
- shot( )
- take( )

MIRHero class is for the hero which is controlled by the player. m_iWeaponInHand holds the unique number of the weapon that hero has. Other variables are bounded integers that determines the characteristics and features of the hero. Hero can act according to member functions of this class.

MIRNonheroHuman

MIRNonheroHuman class is the main class for all characters who are living in the game.

MIRAnimal

MIRAnimal class is the main class for all animals that are living in the game.

MIRCreature

- m_iWeaponInHand
- m_iStamina

MIRCreature class is for all creatures which are not human or animal in the game environment. m_iWeaponInHand holds the unique number of the weapon that creature has. m_iStamina is a bounded integer that holds the value of stamina of the creature.

MIRVehicle

MIRVehicle class is for all kinds of vehicles such as bus, car, minibus, truck, plane which are moving in the game environment.

MIRPortableObjects

- m_ iBodyId

MIRPortableObject class is the class for all objects that does not move or be moved by hero in the game environment. It only has an int named m_ iBodyId which is unique for each object.

MIRWeapon

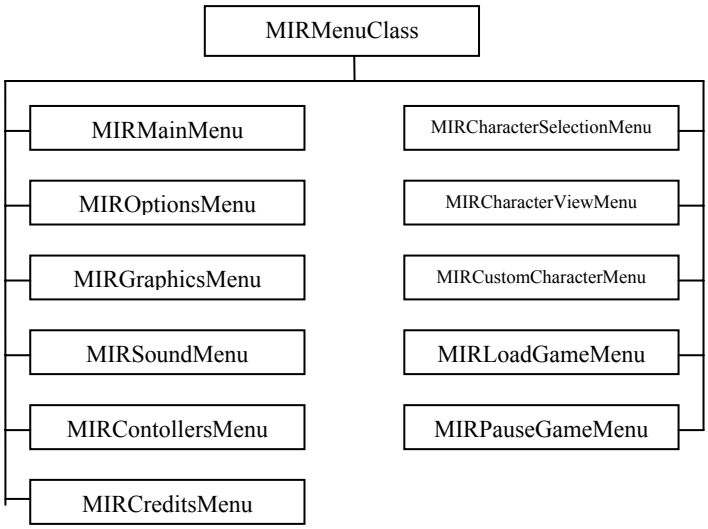
MIRWeapon class is the class for all weapons that are used by hero in the game environment.

MIRPotion

MIRPotion class is the class for all potions that are used in the game environment.

MIRMoney

MIRMoney class is the class for money that are used by hero in the game environment.



MIRMenuClass

- draw ()
- parseInput ()

This class is the base class of all other menu classes. It has a draw method which is responsible for rendering the menu to screen. parseInput method directs the input to the related object in related menu class.

MIRMainMenu

- m_mirButtonforOptions
- m_mirButtonforNewGame
- m_mirButtonforLoadGame
- m_mirButtonforExit

All members listed above are of type MIRButton. They are the actual buttons on Main Menu screen. Click on a button leads the player to the screen of the related menu class. Click on m_mirButtonforExit closes the game.

MIROptionsMenu

- m_mirButtonforGraphics
- m_mirButtonforSound
- m_mirButtonforControllers
- m_mirButtonforBack

All members listed above are of type MIRButton. Different from the buttons in MIRMainMenu class, there is m_mirButtonforBack click on which leads the player to the previous menu screen.

MIRGraphicsMenu

- m_mirScrollBar
- m_mirButtonforBack

m_mirScrollBar is an instance of MIRScrollBar. It enables the player specify the screen resolution for the game. m_mirButtonforBack leads the player to the previous menu screen.

MIRSoundMenu

- m_mirScrollBarforGameSound
- m_mirScrollBarforEffectSound
- m_mirButtonforBack

With these scroll bars, the player can change the sound volume for both game and effects. m_mirButtonforBack leads the player to the previous menu screen.

MIRControllersMenu

Controllers of the game can be customized with MIRControllersMenu class.

MIRCreditsMenu

Credits menu gives a brief information about us.

MIRCharacterSelectionMenu

- m_mirButtonforCharacter
- m_mirButtonforCustomize
- m_mirButtonforBack

This menu is a demonstration menu for character selection. m_mirButtonforCharacter, m_mirButtonforCustomize and m_mirButtonforBack are instances of MIRButton. Onclick method of m_mirButtonforCharacter button leads the player to MIRCharacterViewMenu screen. There is not a single m_mirButtonforCharacter button. The number of this kind of buttons is the same as the number of built in characters of the game. Onclick method of m_mirButtonforCustomize button leads the player to MIRCustomCharacterMenu screen. m_mirButtonforBack leads the player to the previous menu screen.

MIRCharacterViewMenu

- m_mirTextBox
- m_mirInfoBox
- m_mirButtonforPlay
- m_mirButtonforBack

m_mirTextBox is an instance of MIRTextBox. The player enters the name of his/her hero in this text box. m_mirInfoBox is an instance of MIRInfoBox. The information of the character is shown with this infobox. m_mirButtonforPlay and m_mirButtonforBack are instances of MIRButton. Onclick method of m_mirButtonforPlay button directs the player to watch the intro of the game. m_mirButtonforBack leads the player to the previous menu screen.

MIRCustomCharacterMenu

- m_mirInfoBox
- m_mirCheckBox
- m_mirNumericUpDown
- m_mirTextBox
- m_mirButtonforPlay
- m_mirButtonforBack

m_mirInfoBox is an instance of MIRInfoBox. There are actually more than one MIRInfoBox instance in MIRCustoCharacterMenu. The number of MIRInfoBox instances is the same as the number of built in characters of the game. This representation is for simplicity. Information about the character is displayed in m_mirInfoBox. If the player wants to custom a character, s/he uses the m_mirCheckBox to select that character. m_mirNumericUpDown is an instance of MIRNumericUpDown and enables the player to change the properties of the selected character. The player enters the name of his/her hero in the text box. m_mirButtonforPlay and m_mirButtonforBack are instances of MIRButton. Onclick method of m_mirButtonforPlay button directs the player to watch the intro of the game. m_mirButtonforBack leads the player to the previous menu screen.

MIRLoadGameMenu

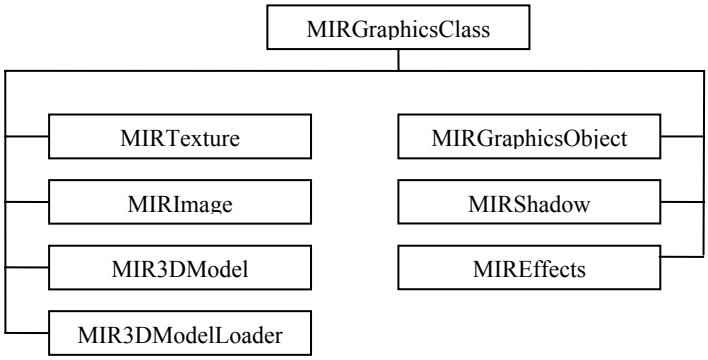
- m_mirTextBox
- m_mirButtonforPlay
- m_mirButtonforBack

In the m_mirTextBox, the player specifies the name of his or her character. With this name, the related save file is found and loaded when the player clicks on the m_mirButtonforPlay. m_mirButtonforBack leads the player to the previous menu screen.

MIRPauseGameMenu

- m_mirButtonforExit2Windows
- m_mirButtonforExit2MainMenu
- m_mirButtonforResumeGame
- m_mirScrollBar

m_mirButtonforExit2Windows, m_mirButtonforExit2MainMenu and m_mirButtonforResumeGame are instances of MIRButton. Onclick event of m_mirButtonforExit2Windows enables the player to exit game totally and return to Windows OS. All windows of the game are closed. When the payer clicks on the m_mirButtonforExit2MainMenu, only the active game is closed. Also the player can continue his or her game by clicking on m_mirButtonforResumeGame button. Besides these exit options, the player can specify the sound volume with the m_mirScrollBar.



MIRGraphicsClass

This class is the base class of all other graphics related classes. It has no special methods and member variables.

MIRImage

- m_iImageWidth
- m_iImageHeight
- m_puiImageData

This class gives an interface for loading images that will be used as textures. An instance of this class will parse the given image file and load its data into the m_puiImageData. Image dimensions will also be stored in this class.

MIRTexture

- m_pmirImage
- m_uiTextureId
- bind ()

MIRTexture is the main class that will be used for textures. It has two main variables. m_pmirImage is a pointer of type MIRImage. It points to the image that will be used for texturing. m_uiTextureId is the texture id that will be used to identify this texture – its name. bind () is the main function that creates the texture.

MIRGraphicsObjects

- drawBox ()
- drawSphere ()
- drawTriangle ()
- drawCylinder ()
- drawCappedCylinder ()
- drawLine ()

This class supplies a list of functions that are used to draw common complex objects like sphere, cylinder etc… It functions as a utility library for rendering operations.

MIREffects

This class gives an interface for special effects (lighting, glowing, drawing auras etc…) that will be used in game.

MIR3DModelLoader

- importModel ()

This class handles the entire 3D model loading operations. importModel function will handle all of the details related to the 3D model file parsing.

MIR3DModel

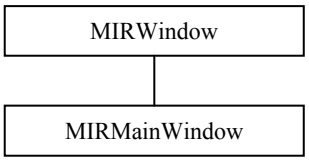
- m_pmir3DModelLoader
- loadModel ()
- renderModel ()

This is the model class used in game. m_pmir3DModelLoader is a pointer to MIR3DModelLoader class. It's used to load the model to memory in loadModel method. renderModel method is responsible for drawing the model.

MIRPhysics

- m_WorldId
- m_SpaceId
- createWorld ()
- createSpace ()
- detectCollisions ()

This is the main interface that handles the physics of the game. Functions of ODE (Open Dynamics Engine) library will be used to implement this class. m_WorldId and m_SpaceId are the world and space ids respectively. createWorld and createSpace functions are responsible for creating the space and world where the game will take place. detectCollisions is the most crucial function of this class. It's the main function that handles the collisions that occur in the game loop.



MIRWindow

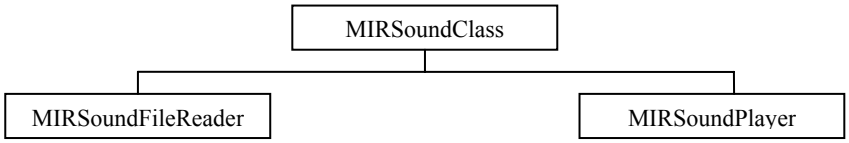
- m_hRc
- m_hDc
- m_hWnd
- m_hInstance
- m_pmirSubWindow
- createGLWindow ()
- initGL ()
- handleInput ()
- drawingLoop ()

This class is responsible for creating a window and a rendering context using MFC. m_hRc, m_hDc, m_hWnd and m_hInstance are the handles of rendering context, private GDI device context, window and instance of the application respectively. createGLWindow creates a window and sets all of the initializations required to draw a GL scene. initGL initializes the GL environment. handleInput is the main callback that is used to pass the keyboard and mouse inputs to the correct module in the game. Finally this class supplies the drawingLoop method. This method is the main drawing loop of this window.

MIRMainWindow

- m_pmirMessageBox
- m_pmirPauseMenuWindow
- m_pmirPanel

This class is the child of MIRWindow. An instance of this class will be used to create the main window of the game. m_pmirMessageBox is used for displaying custom message boxes. m_pmirPauseMenuWindow is the child of main window. This sub window will be used to display pause menu. m_pmirPanel will point to the game panel instance.



MIRSoundClass

- m_amirSoundPlayer
- loadSounds()
- playSound()

This class keeps all ready-to-play sound list. loadSounds() will fill in the m_amirSoundPlayer array with sounds. playSound() function plays the correctsound in m_amirSoundPlayer list.

MIRSoundPlayer

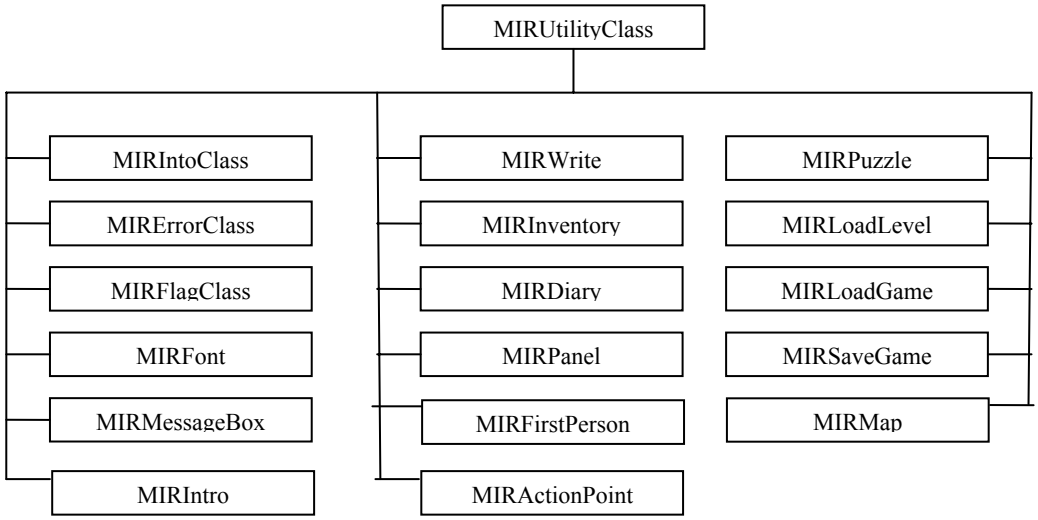
- m_mirSoundFileReader
- initSoundBuffer()

The sound file names to be played throughout the game are used by m_mirSoundFileReader after initSoundBuffer() creates sound buffer.

MIRSoundFileReader

- m_iNumberOfBytes
- loadSoundFileToBuffer()

MIRSoundFileReader is responsible from reading sound files. loadSoundFileToBuffer reads m_iNumberOfBytes . But if m_iNumberOfBytes is 0(zero) then it reads the whole file.



MIRUtilityClass

This class is the base class of all utility classes. It has no special methods and member variables.

MIRIntro

- m_iIntroId
- playIntro();

There will be a brief introduction before the game and before each level. MIRIntro is the class that will be responsible from displaying the correct intro show in the correct place. Looking at the m_iIntroId value, playIntro() will distinguish the correct show.

MIRDiary

- m_iLevel
- m_aszDiaryContent
- displayDiary()

At each level the player will have several missions. He will have a diary to help him/her with clues about the missions. According to m_iLevel part of m_aszDiaryContent will be displayed.

MIRPanel

While playing the game, there will be several items on the screen each having a particular functionality. MIRPanel is the class that will be responsible from functionality of these items.

MIRActionPoint

- m_pScriptFile
- m_vPosition
- m_iLevel

There will be several action points throughout the game. MIRActioPoint is the interface class for these action points. All action points will inherit this class. m_iLevel and m_vPosition will be used to define whether to run the script file pointed by m_pScriptFile.

MIRPuzzle

- m_bState
- m_iLevel
- m_vPosition
- drawPuzzle()
- m_pScriptFile

There will be several puzzles throughout the game each having peculiar properties. MIRPuzzle is the interface class for these puzzles. All puzzles will inherit this class. m_iLevel and m_vPosition will be used by drawPuzzle() function to display the correct puzzle in the correct level at the correct place. m_bState will decide whether the puzzle is solved or not. m_pScriptFile will hold the scripts required for the game.

MIRLoadLevel

- m_aszLevelNames
- m_iLevelId
- loadLevel()

All levels of the game will be kept in different files. MIRLoadLevel class will use m_saLevelNames list and loadLevel() function to use this file name list to load it true file.

MIRFirstPerson

- m_pmirMap
- static object arrays

MIRFirstPerson handles the data required for the game when game is in 1st person mode.

MIRSaveGame

- saveGame()

Throughout the game there will be specific positions that will be used to save the game, these will be the times that MIRSaveGame class will take action. saveGame() will ask the player to enter a file name and the game will be saved.

MIRMap

- m_iC1x, m_iC1y, … m_iC4x, m_iC4y
- m_iTC1x, m_iTC1y, … m_iTC4x, m_iTC4y
- m_pFilePointer

MIRMap handles the grid data.

MIRIntro

- m_ppFiles
- m_pSoundFile

MIRIntro has two members. m_ppFiles is the array of image file pointers that will be displayed during the intro. m_pSoundFile will keep the information about the background music.

MIRLoadGame

- m_saLoadedFileNames
- m_iFileId
- loadGame()

The player will be able to load a previously saved game as well as starting a new game. MIRLoadGame will achieve this by choosing file name from m_saLoadaedFileNames according to m_iFileId, and calling loadGame() function.

MIRMessageBox

- m_szMessage
- displayMessage()

Any message will be displayed by MIRMessageBox class. displayMessage() will display m_szMessage.

MIRErrorClass

Any error throughout the game will be handled by MIRErrorClass. It is going to have special methods for error types.

MIRFlagClass

Those flags that are going to be used in the game will be encapsulated by MIRFlagClass.

MIRFont

- m_iSize
- m_sColor

MIRDrawFont class will define the fonts used in the game

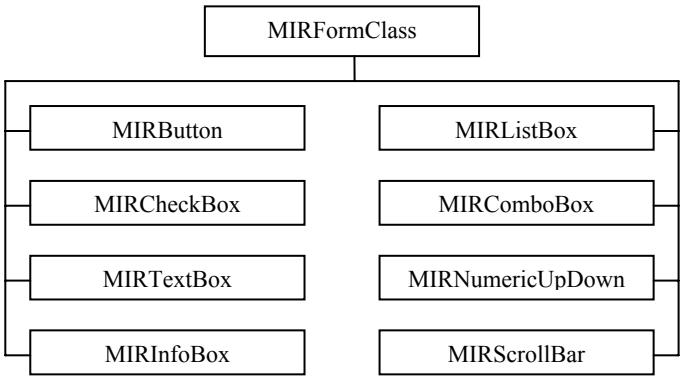
MIRWrite

Writing texts to screen is handled by MIRWrite class

MIRInventory

- m_mirWeapon
- m_mirMoney
- m_mirPotion
- displayInventory()

At any instance in the game, the player will be able to check the inventory and use them. MIRInventory class will keep information about each inventory and will display them to the player when s/he wants.



MIRFormClass

- draw()
- onClick()

This is the base class of all form elements. It contains two methods. draw method draws the form element to screen. onClick method and the following form classes have primitive definitions used in every form based implementation. Therefore only the names of classes are listed. MIRInfoBox needs explanation and it is given below.

MIRButton
MIRCheckBox
MIRTextBox
MIRListBox
MIRComboBox
MIRScrollBar
MIRNumericUpDown

MIRInfoBox

This class implements the place for demonstrating both image and text.

OVERALL WINDOW STRUCTURE

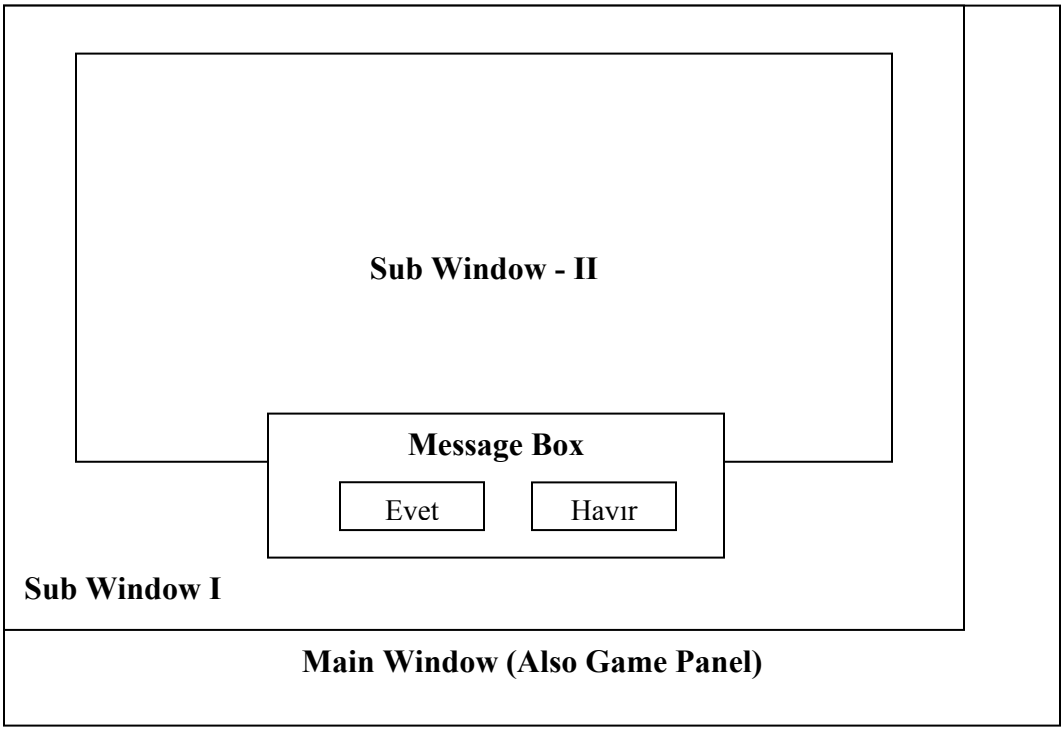


Figure: Window Layers in Game

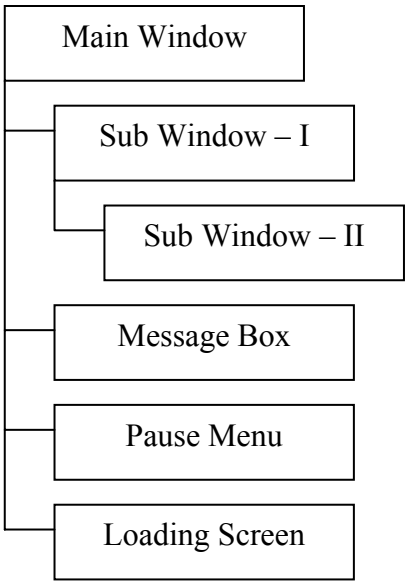


Figure: Window Hierarchy

There will be many window-related operations during the game. Main operations are creating a sub window, rendering some scene to the screen and displaying messages on screen (i.e. message boxes). To achieve these tasks different types of windows are needed. These types are explained below in more details. Windows are created using windows API calls.

MAIN WINDOW

This window is the parent of all other windows. This window is simply a global instance of MIRWindow class. It should be created just after the game started, and will survive until the end of the game.

Each window (i.e. instance of MIRWindow) in the game is capable of creating only one sub-window; however the main window will be capable of creating two sub-windows, and also displaying custom message boxes using MIRMessageBox interface.

Game will be mainly divided into two parts. First one is the menu part. During this part all of the menu drawings will be displayed directly on the main window. No sub windows will exists. However, message boxes may be displayed during this phase. When a message box is needed the m_pmirMessageBox member variable of the main window is used to display the message.

There are three entrance points for second part. These points are the load game screen, custom character creation screen and character view screen. From these screens player will be able to enter the second phase by clicking on the 'Start Game' button. Before moving to the second phase some initializations should be done. During the initializations all interactions with the user should be halted. To achieve this, main window's m_pmirSubWindow variable is used. All of the inputs are directed to this sub-window. So while loading the game a "Loading" screen will be displayed on top of the main window. This sub-window will be destroyed after the initialization completes.

The game panel will be displayed in main window during the game loop. So during the game loop, main window will mainly have a 2D view.

SUB WINDOW I

This window is the child of the main window. After the "Loading" screen destroyed, this window is created instead. This window is the main window that all of the 3D renderings will take place. "Sub Window – I" and game panel is visible during the whole game loop.

SUB WINDOW II

This window is the child of the "Sub Window – I". This window isn't static, and created on demand. Game facilities like the diary of the hero will be displayed in this window. Each time the player clicks on the diary icon this window is created. This window will not release the focus until the player closes the diary (i.e. clicks on the buttons available on this window.)

PAUSE MENU WINDOW

This window is the second child of the main window. When the player presses the escape key during the game, m_pmirPauseMenuWindow variable of main window will be initialized and displayed. This window will also not release the focus until the player closes this window. When this window is active, the whole game state will be paused (graphics, physics, AI…). This will be achieved by pausing the game loop until this window releases the focus.

OVERALL GAME STATES

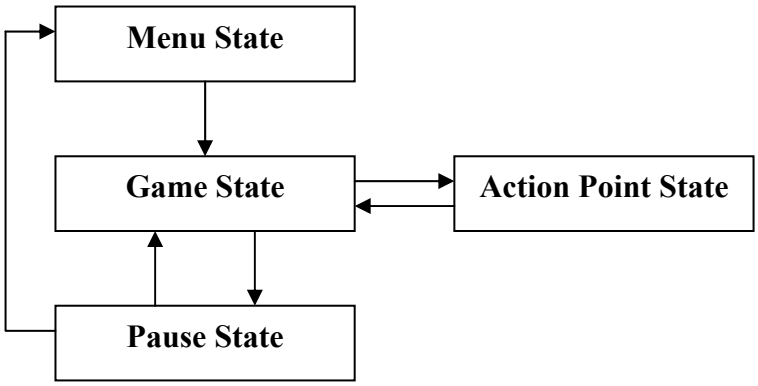


Figure: Game State transitions

In order to handle the transitions between different parts of the game we define four different game states. These states are discussed in more details in the following paragraphs. Only current state of the game will be kept. And this state will be changed only at transition points [i.e. when player clicks on the 'Start Game' button than the state will change from MENU to GAME.]. Game state will define the modules that are needed and handle them. For example, if the state is MENU than AI will not be active, however Sound module will have to be active.

MENU STATE

This state is the initial state of the game. When game starts, a menu is displayed which will give the player the opportunity to change settings of the game, start a new game or load a previously started game.

When the game started the current state will set to this state until player enters the game. Player may enter the game in three different ways: from "Load Game" menu, from "Custom Character" menu and from the "Character View" menu. There is no other way – unless player exits the game – to change the state while MENU state is active.

During this state only a menu will be displayed[2]. And some sound will be played. So we do not need any instances of AI and Physics engine here. Also there will be no sub windows in this state. Only when leaving this state a "Loading" screen will be displayed using a child window of main window. During this state a background music will be played so Sound engine should be alive. Also some sounds will be played when player wastes his/her valuable time on surfing through the menu [i.e. button click sounds].

[2] For the detailed menu structure refer to the Figure MenuTree

Main loop and message handling code for MENU state will be as follows:

```
/*In main windows main loop*/
If Current state is MENU state than
   Play background music
   If  Current menu isn't null than/*Initially this has already set to Main menu*/
      Call current menu's drawing function

/*In main windows message handler*/
If Current state is MENU state than
   If mouse is clicked than
      For all elements of current menu do
         If mouse is on this element than
            /* Callback functions will handle the entire job. For example if the clicked
               element is the 'Back' button than this callback will set the current menu
               pointer to parent of this menu. It will also play a 'Click sound'. Game
               state will change by three of these callback functions.
            */
            Call this elements call back function and return
      /*No elements found! Do nothing.*/
   If key pressed than
      For all registered shortcut keys do
         If pressed key is equal to a shortcut than
            Call shortcut kev's callback function.
```

GAME STATE

This state is the second visited state during the game lifecycle. During this step, player will enjoy the game with his/her hero(ine).

After clicking on the 'Start Game' button, player will change the state to GAME state. Game will leave this state when player opens the pause menu or enters an action point. There is no other way – other than killing the game of course – to leave this state.

We need everything in this state. In other words all of the modules are active during this state. So before entering this phase, all of the stuff must be initialized[3]. In this state during each step of the game loop three main tasks have to be completed. These are parsing inputs, updating scene and drawing scene.

[3] Details of initialization are given in "INITIALIZATION OF THE GAME" part.

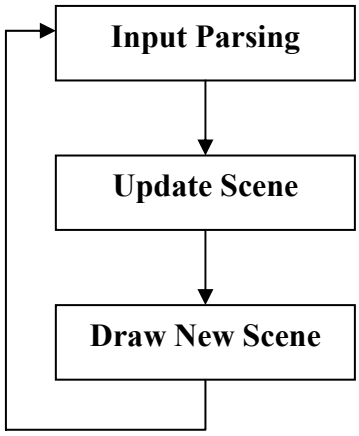


Figure: Steps in GAME state.

Inputs gathered are directed to the window/sub window that has the focus by Windows. Each window has its own input handling unit (i.e. a function registered to handle messages). These units/functions determine the way that the window reacts to the messages. For example a mouse click directed to Sub Window – I will makes the hero move a specified position, but a mouse click directed to Main Window will make the diary visible.

During the GAME state we need more than one window. So sub windows are created to fulfill these needs. At any time in this state we need to render both 2D (game panel) and 3D (game scene) graphics. To achieve this, a child window of main window (Sub Window – I) will be created for 3D graphics. 2D graphics will be directly drawn on main window. Also to display diary we need another 2D rendering area. This area will be created as a child of Sub Window – I (Sub Window – II).

When focus is on Main Window, keyboard presses are checked to see whether they are previously assigned shortcut keys or not. If so these key's callback function is called. Mouse clicks are handled in a similar way. All of the members of game panel are queried to find out which element is clicked. If an element is found its callback function will be called.

When focus is on Sub Window – I, depending on the view of the camera keyboard presses are checked and related callback function is called (i.e. if currently game is in 3rd person mode, pressing 'E' will do nothing, however if game is in 1st person mode pressing 'E' means open the door – if there exists one). Mouse clicks are highly used in 3rd person mode. In 3rd person mode player may click somewhere in the world to make the hero move there. To achieve this some checks have to be done. First of all, no obstacles should be at the clicked position (during the initialization phase, all static objects are kept globally[4]; here we need to check all static objects.). If there is an object, hero will move as near as possible to the destination. Otherwise, the coordinates are again tested. This time we look for something to attack. If a creature is found[5], hero will attack on it. Attacking has two modes depending on the weapon that hero holds. Range attacks require hero to move at least to specific distance. If a creature not found, than we continue searching. There remain three possibilities: NHHs, action points and a safe position to go. When a NHH is found, the action taken isn't different than any other static object if hero doesn't hold a weapon. Otherwise he/she will attack them – possibly result in the death of NHH[6]. Total count of NHHs is constant in a level. When a NHH is killed, simply its position will change, in other words it will appear another place in the world. If an action point is found, hero will move there and the related script file will executed by the engine. In the third case, simply hero moves there.

Before changing the game state from GAME to ACTION POINT we need to save the camera information (position and rotation) and light information (position, attributes…), otherwise changing the state from ACTION POINT to GAME would be problematic.

[4] For more information on the data structures that these objects are kept, refer to the "INITIALIZATION OF THE GAME" part.
[5] This is again achieved by checking all creatures for the specified position. If a creature is found than its id will be returned.
[6] NHHs will also have hit points.

Updating scene is a complex task. It requires the contribution of many modules. First of all according to the inputs gathered, AI will make decisions on how to react these inputs. In the simplest yet widely used case player will point a monster and request the hero to attack this monster. AI has to decide whether this monster should flee or attack to the hero. This is done by executing the monsters decision script. After AI make the decision the world will change. But these changes should be done in a logical way. While fleeing, monster should not fly, nor pass through the walls – if it hasn't such ability. So actions taken should comply with physics rules of our world. This is handled by the physics engine. Physics engine determines the positions of all objects – if they have a physical body – during each iteration.

After scene is updated, we need to display it to the player. This is where graphics engine takes place. All of the models and user interfaces should be re-drawn. The positions of the objects which have a physical body should be gathered from the physics engine. This is done by querying the position information of the object by its body id[7].

[7] Each object is assigned a body id when registered to the physical world.

Main loop and message handling code for GAME state will be follows as:

```
/*In main windows main loop*/
If Current state is GAME state than
   Play background music
   Call game panels display function

/*In main windows message handler*/
If Current state is GAME state than
   If mouse is clicked than
      For all elements of game panel do
         If mouse is on this element than
            Call this elements call back function and return
      /*No elements found! Do nothing.*/
   If mouse is pressed than /*Drag*/
      For all elements of inventory do
         If mouse is on this element than
            Move this item to the mouse position.
   If mouse is released than /*Drop*/
      If currently an item is hold than
         If release position is in Sub Window – I than
            Remove item from inventory
   If key pressed than
      For all registered shortcut keys do
         If pressed key is equal to a shortcut than
            Call shortcut key's callback function.
      /* Here we assure that shortcut keys of main window do not overlap with the
keys of Sub Window – I */
      For all registered shortcut keys of Sub Window – I do
         If pressed key is equal to a shortcut than
            Call shortcut key's callback function.

/* In main loop of Sub Window – I*/
If Current state is GAME state than
   Detect collisions in physical world
   For all static objects do
         Draw object
   For all dynamic objects do
         Get positions from physics engine
         Draw object
   For all NHHs do
         Find next node of the walking path and draw NHHs
   Calculate the position of hero and draw hero
```

```
/*In Sub Window – Is message handler*/
If Current state is GAME state than
   If mouse is clicked than
      For all static objects do
         If this object is at that position than
            Calculate the nearest position where hero can move
            Make hero move there and return
      For all creatures do
         If creature is at that position than
            /*this will determine the behavior of the creature*/
            Run decision scripts and return
      For all NHHs do
         If NHH is at that position than
            If hero is holding a weapon than
               Shoot NHH and return
            Else
               Go to the nearest position where hero can move and return
      For all Action Points do
         If there is an A.P there than
            Save camera & light info
            Change state to ACTION POINT state
            Run related action script and return
      Make hero move this position.
   If key pressed than
      For all registered shortcut keys do
         If pressed key is equal to a shortcut than
            Call shortcut key's callback function.
      /* Here we assure that shortcut keys of main window do not overlap with the
keys of Sub Window – I */
      For all registered shortcut keys of Main Window do
         If pressed key is equal to a shortcut than
            Call shortcut key's callback function.
```

PAUSE STATE

Game will enter this state after player pressed a pre-defined key (Blizzard uses Escape, may be we should do that too!). This state has two exit points: One returns back to the game, and the other returns to the MENU state.

During this state all of the game will be paused. This means that when game is in PAUSE state no requests for collision detection or script execution will made. Other jobs done are the same as MENU state.

ACTION POINT STATE

This state is only reachable from the GAME state. When hero reaches a pre-defined position in the world, the script file related to that position will execute and changes the state of the game from GAME to ACTION POINT. When 'action' is

completed state will changed to GAME automatically. Also player may leave this state by clicking on a button. No other state transitions are possible form this state! This means that player may not pause the game nor s/he can leave the game.

Action points are coordinates at where some sort of 'action's are occurred. These 'action's are entering a building (transform from 3rd person mode to 1st person mode), leaving a building, starting a normal puzzle (required to make progress in a level) and starting a hidden puzzle (gives bonuses).

Leaving a building and entering the same building may seem to be confusing. Since both actions have the same coordinates. But actually they have different coordinates. Our level design requires local coordinates for sub maps (inside the buildings). So an entrance in the outdoor world will have different coordinates than the same entrances indoor coordinates. Entrance actions will simply set the new camera view and load new scene. This is actually the same process as loading a new level. This processes will handled by scripts.

Another type of actions is puzzles. The required action is stored in the related script file. This file is responsible for drawing the puzzle scene – of course by C++ calls and give a solution to the puzzle. For example let's consider a sample puzzle. During a level hero is required to go to the 8th floor of a building that has 20 floors using an elevator. Puzzle information has already been loaded at the start of the level. Puzzle has elements (models to be drawn) and their positions. It also has some 'sensible' points. When clicked on these points something will happen. For our case there are three elements: an up button and a down button and a display (0 is displayed initially). Also there are two 'sensible' points located at the same coordinates of these buttons. When the up button is clicked display will be increment by 5 and when the down button is clicked display will be decremented by 3 – which means that our elevator is broken! At each mouse click script checks whether the display is 8 or not. If so than the puzzle is solved.

Finally there will be hidden puzzles. These have the same logic as the ordinary puzzles but player does not have to solve these puzzles. They are optional.

Main loop and message handling code for ACTION POINT state will be as follows

```
/*In main windows main loop*/
If Current state is ACTION POINT state than
   Play background music
   Call game panels display function

/*In main windows message handler*/
If Current state is ACTION POINT state than
   Ignore all messages

/*In main loop of Sub Window – I*/
If Current state is ACTION POINT state than
   Execute related script file

/*In Sub Window – Is message handler*/
If Current state is ACTION POINT state than
   If action type is puzzle than
         If mouse clicked or key pressed than
            If mouse click is on the 'Leave puzzle' button than
                  Load back the camera & light info
                  Change state to GAME
                  Exit the puzzle
            Execute the action specified in script file
```

OVERALL MESSAGE HANDLING

Each window is responsible for handling their inputs. If they have the focus then all of the messages are directed to them – this is done by Windows.

MAIN WINDOW

If a menu is displayed (current menu is set to some value), then input is directed to this menu. For example after mouse is clicked, all of the members of the menu is checked whether the mouse click is on them or not. If an element is found then its related callback function is called.

During the game main window is only contains the game panel. Game panel handles the inputs like any other menu does.

SUB WINDOW I

This window is where the game takes place. So during the game mostly focus will be on this window. There is a complication here. Some of the inputs directed to this window are actually inputs of the main window. For example F5 is a shortcut for opening the diary. Normally diary is opened by clicking on the diary icon which is displayed on the game panel. So when player presses F5 (focus is on Sub Window – I) s/he actually aims to send this input to the main window (game panel). To overcome this complication message transfer interfaces will be supplied. Main window will already have a part that will handle the F5 input. So Sub Window – I will simply send this input to main window using a function provided (i.e. g_MainWindow.KeyPressed (F5)).

OVERALL USER INTERFACE STRUCTURE

MENU

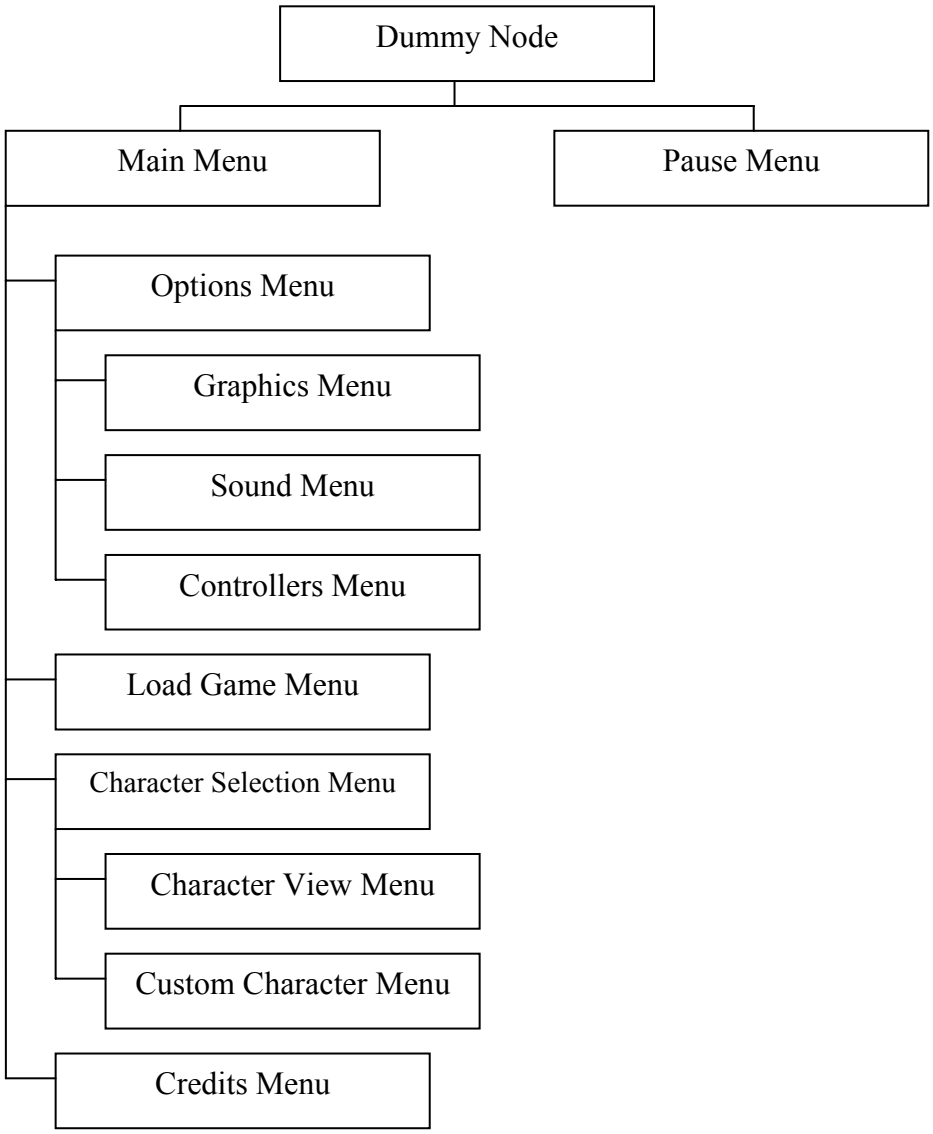


Figure: Menu Tree

In order to handle the menus of the game a menu tree will be created after the creation of the main window. Depending on the state of the game (i.e. currently menu is being displayed, game is started or game is paused.) either the current menu screen is displayed or the game is displayed. Each node of the tree drawn above is of type MIRMenuClass. This structure is global and initially all of the nodes are not initialized. Also a global pointer of type MIRMenuClass is needed to point the currently active menu. After creating the tree – this is done just after the window created – the first node of the menu is initialized. In other words the main menu is initialized and the current menu pointer is set to this node. During each iteration of the drawing loop simply the current menu's drawing functions are called.

When user sends an input (clicks on a button) the related callback function will be called. This function will initialize the related menu tree node and sets the current menu pointer appropriately. During the next iteration of the drawing loop, the new menu will be displayed.

Menu tree is initialized at the beginning of the game however nodes are initialized on demand. Returning to a previously created menu (i.e. returning from load game menu to main menu) will not destroy the current menu!

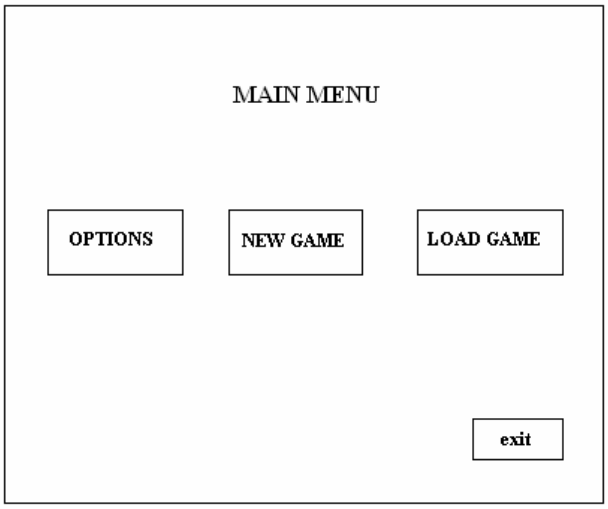
GAME PANEL

When game is in "Game State" there will be a static panel displayed on the main window. This panel will display some useful information for the player like his/her hero's health, mana, inventory etc… Panel will be initialized after exiting the menu state and survive until the end of the game state. Also panel will directly be drawn on main window and will have a 2D view. Some of the elements of the panel will have shortcuts. When player presses on of the shortcut keys the related part will be activated. For example clicking on the diary icon or pressing some pre-defined key will make the "Sub Window – I" create a sub window and show the diary information of the hero on that screen. Mouse clicks are handled like the menus. When a mouse click is captured, all of the elements of the panel are tested and related elements callback function is called.

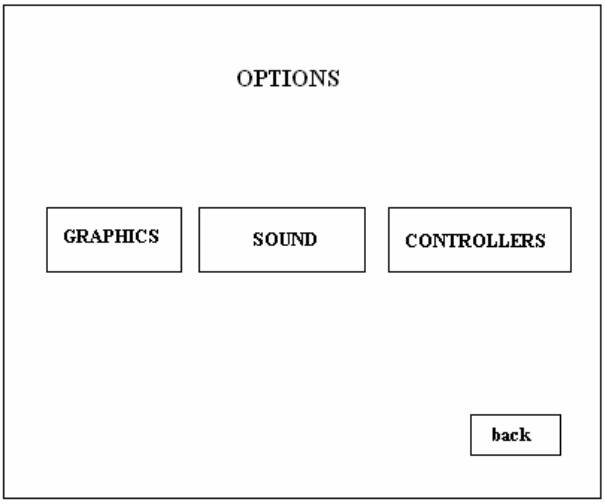
Player will have the opportunity to drag-n-drop an item that is in his/her inventory. Displacement of the objects may be occurred in inventory or between the inventory and "Sub Window – I". In the first case when user drags an item the item will be dropped on the new slot – if the action is valid. In the second case the item will be dropped directly to the 3D environment. If the coordinates that the player drops the item are in the "Sub Window – I" then the item will be removed from the inventory.

GRAPHICAL USER INTERFACES

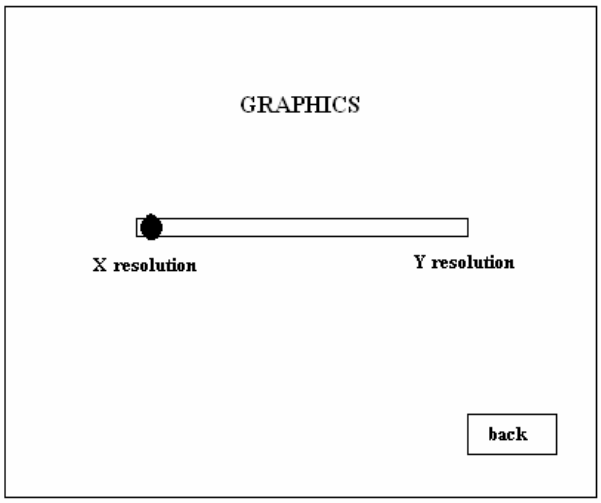
These interfaces are not the final designs. They are prepared in order to generate a general idea of how the game interface will look like.



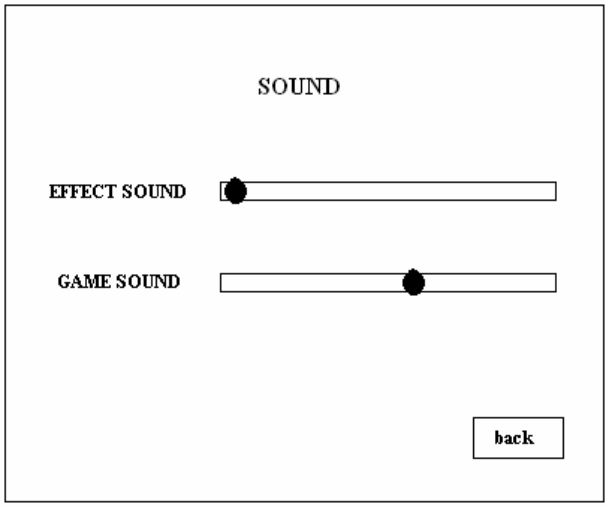
The first interaction with the player is through the Main Menu. If the player wants to change the options of the game, s/he presses the OPTIONS button and the following screen opens.



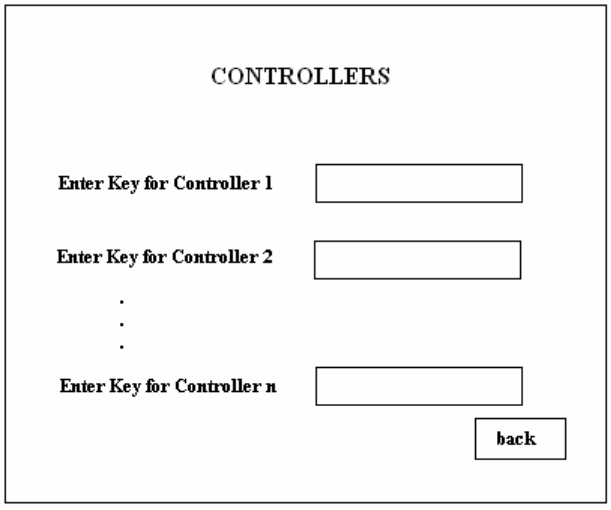
From this menu, the player can change game settings in three ways. The first one is the Graphics menu.



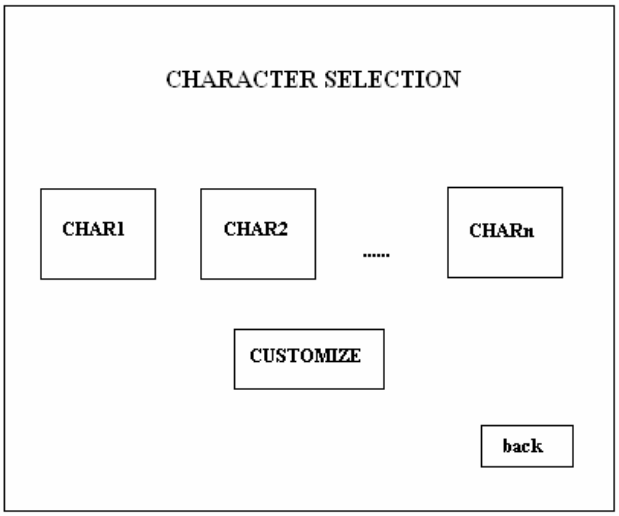
The player has the opportunity to play the game in two different screen resolutions. Besides the second group of settings the player can update is sound.



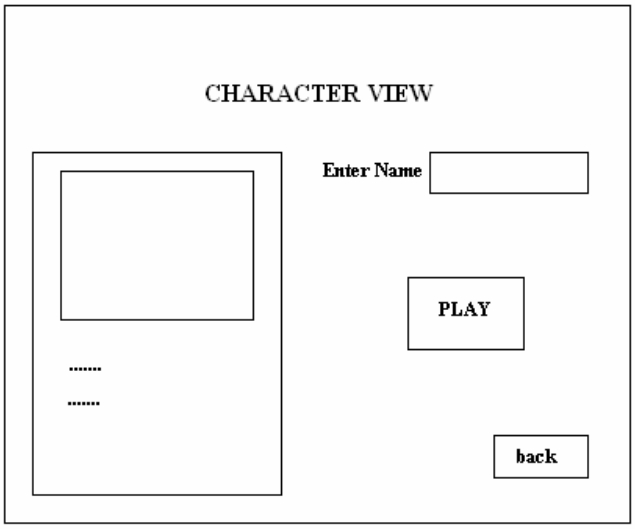
Sound options are divided into two categories. One is the game sound which refers to the soundtrack of the game. The other one is the effect sound which contains warnings, errors, ingame effects etc. The last setting group which can be changed by the player is the game controllers.

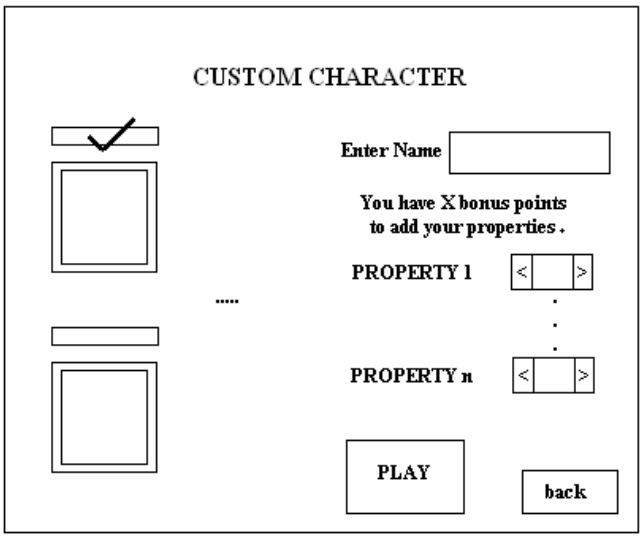


The game has both mouse and keyboard controls. In the Controllers menu, the player can define his or her own control key settings for each controller.

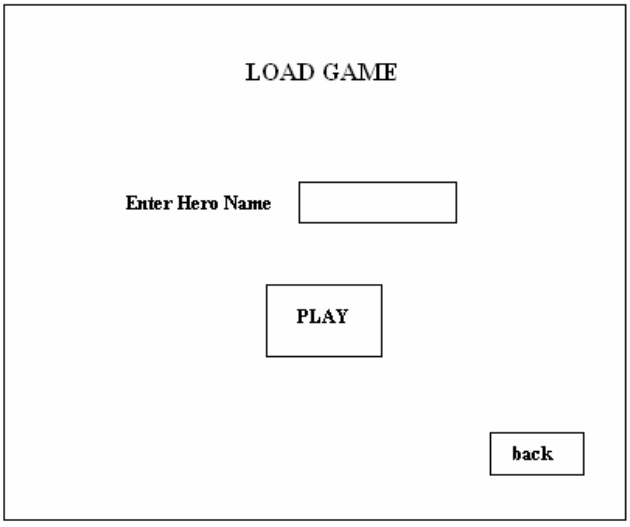


If the player clicks on the NEM GAME button in the Main Menu, the Character Selection menu is opened. From this menu, the player can specify his or her hero from the designed characters or s/he can customize one of the designed characters.





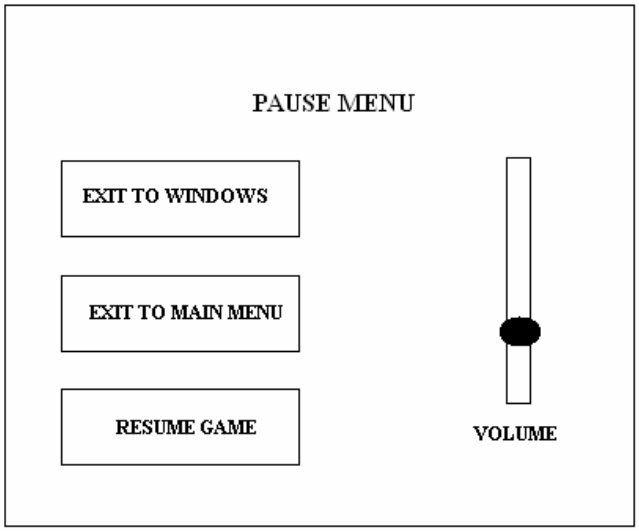
For both of the possible hero selection ways, the player enters a name for his or her hero. Customizing is carried out by changing the reserved properties of a designer character.



If the player clicks on the LOAD GAME button in the Main Menu, the Load Game menu is opened. The player can load a saved game by entering the name of his or her hero.

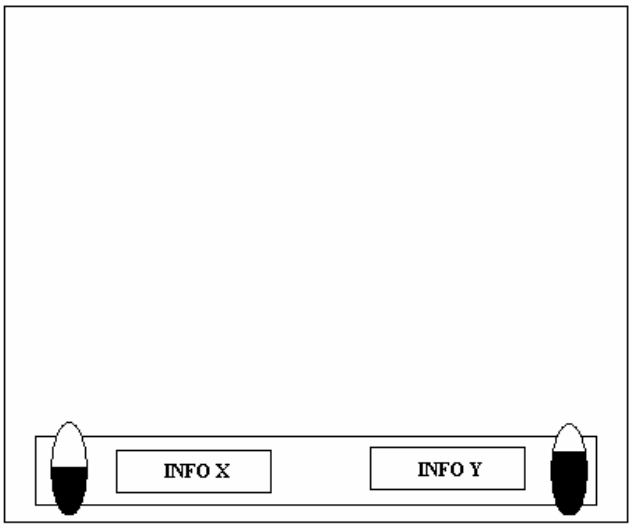
The PLAY buttons in the above screens leads the player into the game.

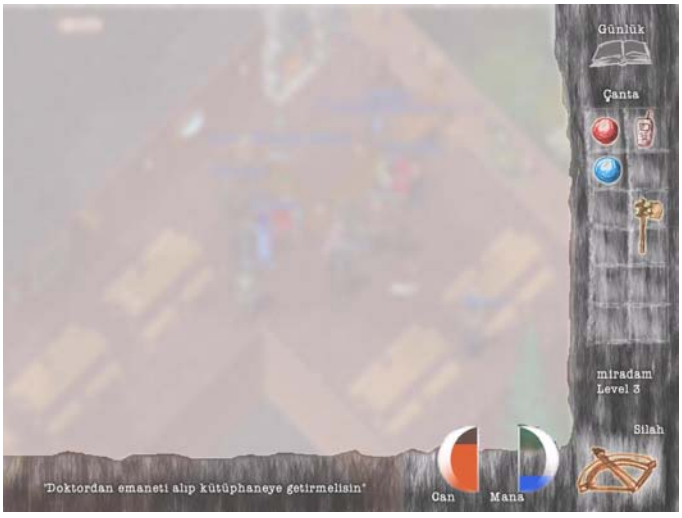
During the game the user has a pause option. If s/he pauses the game, the Pause Menu is opened.



In the Pause Menu, the player has exit options either to Windows or to Main Menu. Or the player can ust change the volume of game sound and resume game.

During the game, there is a static game panel on the screen. The player can see some properties of his or her hero on this panel.





FILE FORMATS

LEVEL FILE

As it is common in almost all games, our game will have different levels. In each level we will have an environment and there will be specific objects, inventories, creatures and puzzles in this environment. It is a problem to load all these information into the game if you don't have a good file format. In order to solve this problem we have developed a file format for levels.

Our level files will keep information about: Name of the level, intro of the level, missions of the level, an object list of the level, information about the non-player characters, sound that is going to be played through the level, textures and their positions in the environment, action-point list. What we mean by action-point is that there will be specific locations in the environment that will trigger scripts to run or that will show the player a puzzle. Let us go over these items.

We have already declared that our game will be a mixture of first-person and third-person. So we are going to differentiate these modes and the objects in these modes by tags. Those objects, sound, puzzles, textures that will be shown in third person mode will be enclosed by <thirdperson></thirdperson> tags and those objects, sound, textures we are going to see in first-person modes will be enclosed by <firstperson></firstperson> tags. So general structure of the level file will be:

```
<levelname>MM'de dehset</levelname>
<intro>...</intro>
<missionlist>...</missionlist>
<diary>...</diary>
....
<thirdperson>
      …….
</thirdperson>
<firstperson>
      ……..
</firstperson>
      .
      .
<firstperson>
      ……..
</firstperson>
```

Before each level we are going to show the player an introduction slide show that will give brief information about the level. This is going to be an image slide show. So we are going to keep the image filenames, the order of these files and sound that is going to be played during the intro in our level files. And also each level will have a list of missions that will be displayed at panel. We should also keep this information in our level file.

```
<intro>
      <image>img1.jpg</image>
      <image>img2.jpg</image>
      …
      <sound>introsound</sound>
</intro>
<missionlist>
      <mission>Kutuphaneye git</mission>
      <mission>Kitabı bul ve getir </mission>
       …
</missionlist>
```

We provide a diary to the player to get clues about the game. The content of the diary extends according to the global level progress variable. The corresponding diary texts are given in level files.

```
<diary1>bidibidi</diary1>
<diary2>bidibidi</diary2>
<diary3>bidibidi</diary3>
```

The light information is kept with four components: position, diffuse, ambient and specular. This information is kept in level file in the following structure.

```
<light>
    <posX>10</posX>
    <posY>55</posY>
    <posZ>60</posZ>
    <diffuse>
        <1>1.0</1>
        <2>1.0</2>
        <3>1.0</3>
        <4>1.0</4>
    </diffuse>
    <ambient>
        <1>0.0</1>
        <2>0.0</2>
        <3>0.0</3>
        <4>1.0</4>
    </ambient>
    <specular>
        <1>1.0</1>
        <2>1.0</2>
        <3>1.0</3>
        <4>1.0</4>
    </specular>
</light>
```

There are going to be many textures in the environment. The information about the textures are kept in the level files.

```
<textures>
    <grass>grass.bmp</grass>
    <concrete>concrete.bmp</concrete>
    ….
</textures>
```

We divide the environment into grids in order to make efficient texture mapping. Each grid has four coordinates in game world, four coordinates for texture file and a texture name referencing a texture between the above tags.

```
<map>
    <grid1>
        <g1X>24</g1X><g1Y>24</g1Y>
        <g2X>25</g2X><g2Y>24</g2Y>
        <g3X>25</g3X><g3Y>25</g3Y>
        <g4X>24</g4X><g4Y>25</g4Y>
        <t1X>24</t1X><t1Y>24</t1Y>
        <t2X>25</t2X><t2Y>24</t2Y>
        <t3X>25</t3X><t3Y>25</t3Y>
        <t4X>24</t4X><t4Y>25</t4Y>
        <texture>grass</texture>
    </grid1>
    ….
```

```
    <gridn>
    ….
    </gridn>
</map>
```

There are moving objects in the game environment. They will walk or in general move in specific paths. These paths are set of continuous grids.

```
<paths>
    <road1>
        grid61
        grid62
        grig63
        grid64
    </road1>
    ….
</paths>
```

We are going to have sound played throughout the game. There is continuous background music. But background song is not constant, may shuffle in a group of songs or change on purpose at specific action points. Besides the background music, there will be button click sound, animation sounds of hero and other moving objects. All these sound information will be kept in the level files for future use.

```
<soundlist>
    <introsound>intro.wav</introsound>
    <footstep>fstep.wav</footstep>
    <cat>miyav.wav</cat>
    <car>car.wav</car>
    <carbreak>carbreak.wav</carbreak>
    <scream>scream.wav</scream>
    <clapping>clapping.wav</clapping>
    <roar>roar.wav</roar>
    ….
    <gender1>sound11.wav</gender1>
    <gender1>sound12.wav</gender1>
    ….
    <gendern>soundn1.wav</gendern>
    <gendern>soundn2.wav</gendern>
    ….
</soundlist>
```

We are going to load 3d models in our game so the information about these models should be kept in the level files. We keep an object list in the level file that will keep a list of all static, moving objects and creatures and all the objects in the environment will instantiate these object list elements.

```
<objectlist>
    <statics>
        <tree>tree.3ds</tree>
        <building>building.3ds</building>
        ….
    </statics>
    <inventoryobjects>
        <money>money.3ds</money>
        <pistol>pistol.3ds</pistol>
        ….
    </inventoryobjects>
```

```
    <moving>
        <femalestudent>female.3ds</femalestudent>
        ….
    </moving>
    <creatures>
        <weakcreature>weak.3ds</weakcreature>
        ….
    </creatures>
</objectlist>
```

For those static objects we keep the instance name, location, rotation and translation of the instances in the level file.

```
<staticobjects>
    <object>
        <class>tree</class>
        <posX>213</posX>
        <posY>100</posY>
        <posZ>11</posZ>
        <rotationangleX>30</rotationangleX>
        <rotationangleY>30</rotationangleY>
        <rotationangleZ>30</rotationangleZ>
        <translationX>3</translationX>
        <translationY>3</translationY>
        <translationZ>3</translationZ>
    </object>
    ….
</staticobjects>
```

For those moving objects and creatures other than position information we will keep velocity, sound and path information. All moving objects have one path and walk ar move on this path. For every animation of the object, a sound can be assigned. For creatures and NHH, additionally an AI script is assigned to describe the creature's manner.

```
<movingobjects>
    <object>
        <class>car</class>
        <posX>22</posX>
        <posY>30</posY>
        <posZ>2</posZ>
        <velocity>20</velocity>
        <path>road2</path>
        <animation1>car</animation1>
        <animation2>carbreak</animation2>
    </object>
    <object>
        <class>femalestudent</class>
        <posX>22</posX>
        <posY>30</posY>
        <posZ>2</posZ>
        <velocity>6</velocity>
        <path>road1</path>
        <manner>femalestudent.py</manner>
        <animation1>scream</animation1>
        <animation2>clapping</animation2>
    </object>
    ….
</movingobjects>
```

```
<creatureinstances>
    <object>
        <class>weakcreature</class>
        <posX>22</posX>
        <posY>30</posY>
        <posZ>2</posZ>
        <velocity>7</velocity>
        <path>road1</path>
        <manner>weakcreature.py</manner>
        <animation1>roar</animation1>
    </object>
    ….
</creatureinstances>
```

The reason for keeping the static and moving objects in different tag headers is because of the fact that we are going to keep their information in different data structures. This is done to ease the search of the mouse click.

And finally the most important part, action points. In order to keep the consistency of the game we must keep information about the actions, which is when to show a puzzle or to be more specific when to open a surprise door. The main point to determine the action will be the position. For example when the hero comes in front of a wall and click on a specific brick a script will run and a secret door will be opened. This is a pass to first person mode and a script file has to be run to achieve the pass. A similar case happens for the puzzles. Therefore we need to keep action points and action point information is kept as follows.

```
<actionpoint>
    <actionscript>opendoor.py</actionscript>
    <level>2</level>
    <posX>231</posX>
    <posY>213</posY>
    <posZ>12</posZ>
</actionpoint>
<actionpoint>
    <actionscript>puzzle1.txt</actionscript>
    <level>1</level>
    <posX>231</posX>
    <posY>213</posY>
    <posZ>12</posZ>
</actionpoint>
```

Some of the puzzles will have dependent puzzles. We chose to keep this dependency by grouping puzzles in levels. So a puzzle can not be solved before all puzzles having lower levels are solved. But in puzzles in same level have no precedence upon each other. To give an example if there are to puzzles one is for finding a key and another is opening the door, before finding the key, we are not going to show “open door” puzzle to the hero although the hero is at the “open door” action point.

The level files are read when new game is being created or when the player chooses to load a previously saved game. Therefore the information in level files is divided into two groups. First group consists of information about missions, diary, textures, map, paths, sound, lights, objectlist, static objects and action points. These will be encapsulated with special characters and read both in new game and load game. The rest of the information explained above is considered as dynamic information about the level. The idea may become more clear with an example. During the game, the hero kills creatures and the number of creatures decreases. If the player saves and exits the game, s/he expects not to meet the dead creatures when the saved game is loaded.

PUZZLE FILE

In each level the hero will come up with several puzzles. All these puzzles are kept in different files. What we keep in puzzle files are: the objects that are going to be drawn on the screen, and a script file for the solution of the puzzle.

We keep the objects that are going to be displayed to the player as the puzzle. Like in level file format, we will keep an object class at the top and instantiate them.

```
<objectlist>
    <button>button.3ds</button>
    …..
</objectlist>
<puzzleobjects>
    <object>
    <class>button</class>
    <posX>12</posX>
    <posY>132</posY>
    <posZ>2</posZ>
    <rotationangleX>30</rotationangleX>
    <rotationangleY>30</rotationangleY>
    <rotationangleZ>30</rotationangleZ>
    <translationX>3</translationX>
    <translationY>3</translationY>
    <translationZ>3</translationZ>
    </object>
    ….
</puzzleobjects>
```

And finally we must have information about the solution procedure of the puzzle. In order to support the user change the solution of a puzzle, we keep a script file to define the solution. So every puzzle has it's own script file defining the solution procedure.

```
<solution>puzzle1.py</solution>
```

SAVE FILE

Throughout the game, in each level there will be some specific points that we will allow the user to save the game. In such cases we must keep all information about the game state. This includes the global variables of the game. Specifically level variable, level progress variable. In addition to these, the states of all moving objects, creatures and the MIRHero instance. Lastly the state of action points, like a puzzle is solved or not, are written in a save file.

Whenever the player saves the game we must keep all information about the hero. When the player loads the game, according to these information we fill in the hero class.

```
<hero>
    <name>mir</name>
    <posX>21</posX>
    <posY>45</posY>
    <posZ>3</posZ>
    <weaponinhand>pistol</weaponinhand>
    <health>50</health>
    <magicpower>32</magicpower>
    <stamina>38</stamina>
    <money>3000</money>
    <inventoryobject>telephone</inventoryobject>
```

```
    <inventoryobject>shield</inventoryobject>
    <inventoryobject>spellA</inventoryobject>
    <inventoryobject>money</inventoryobject>
    ….
</hero>
```

The states of moving objects and creatures are read from the linked lists of related arrays and saved in a structure same as the level file structure.

```
<movingobjects>
    <object>
        <class>car</class>
        <posX>24</posX>
        <posY>32</posY>
        <posZ>4</posZ>
        <velocity>20</velocity>
        <path>road2</path>
        <animation1>car</animation1>
        <animation2>carbreak</animation2>
    </object>
    <object>
        <class>femalestudent</class>
        <posX>24</posX>
        <posY>32</posY>
        <posZ>4</posZ>
        <velocity>6</velocity>
        <path>road1</path>
        <manner>femalestudent.py</manner>
        <animation1>scream</animation1>
        <animation2>clapping</animation2>
    </object>
    ….
</movingobjects>
<creatureinstances>
    <object>
        <class>weakcreature</class>
        <posX>24</posX>
        <posY>32</posY>
        <posZ>4</posZ>
        <velocity>7</velocity>
        <path>road1</path>
        <manner>weakcreature.py</manner>
        <animation1>roar</animation1>
    </object>
    ….
</creatureinstances>
```

The information about which puzzles are solved and which missions are completed must be written in a save file, so that while loading the saved game we don't show same puzzles to the player. We save these information by keeping the index of the action point array where the solved variable of the MIRActionPoint instance is 1.

```
<completepuzzles>
    <index>5</index>
    ....
</completepuzzles>
```

CONFIGURATION FILE

In our menu options we allow the player to play with the graphics settings, sound settings and keyboard short-cuts. Rather than doing this setting operation every time the player starts the game, we keep these information in a configuration file. At the beginning of the game the options are loaded according to this configuration file.

```
<sound>-2000</sound>
<graphics>
        <width>1024</width>
        <height>768</height>
 </graphics>
 <shortcuts>
        <forward>up-arrow</forward>
        <backward>down-arrow</backward>
        <right>right-arrow</right>
        <left>left-arrow</left>
         ….
 </shortcuts>
```

When the player changes the options either before starting the game or by pausing, the changes will be transmitted to the configuration file.

STARTING OF THE GAME

When the player starts the executable of the game, first of all a MIRWindow instance with the default graphics options is created[8]. Global state is set to "menu", a predefined integer value[9]. The blank menu tree is created. The menu tree is a tree with MIRMenuClass pointers in its nodes. There is a reserved node for every menu instance that can be created. In blank state, all pointers are NULL and the global current menu pointer is also set to NULL. Two instances of MIRFont, for active and inactive states of the menu fonts, are added to font array. Every MIRFont instance has a unique id so no search method is needed to find the correct instance. Font on a menu button is activated when the mouse is over that button. After the font instances, an instance of MIRMainMenu is created. The first child of menu tree and the current menu pointer is set to point to the main menu instance. So the content of the first window is ready. Two instances of MIRSoundClass, one for background music and one for button-click sound, are created with the default sound options.

drawingLoop() method of MIRWindow instance is called. As the global state is "menu", draw() method of the menu instance referenced by the current menu pointer is called in drawingLoop(). When the player clicks on a button related with a new menu, an instance of that menu type is created and referenced by a child node of the current menu node in the menu tree. The current menu pointer is set to point to the newly created menu instance. So when drawingLoop() method of MIRWindow instance is called, the draw() method of the newly created menu instance is going to be processed.

If the player makes changes in sound, graphics or control options, these changes are directly saved in configuration file. Sound options are passed to sound instances. The situation is not so easy with graphics options. Game window has to be destroyed and recreated with the new graphics settings in configuration file.

INITIALIZATION OF THE GAME

There are two types of initialization processes in our game. First one happens when the player starts the game from new game button, the second happens when s/he starts the game from load game button in main menu. Both types are explained in detail in this part of our report.

---

[8] Details of window creation can be found in "OVERALL WINDOW STRUCTURE" part.
[9] Details of game state information can be found in "OVERALL GAME STATE" part.

NEW GAME

The player selects a character from character selection menu and gives a name to his/her hero in character view menu or custom character menu. 3ds model file of that character is imported. The data is stored in new MIRHero instance. Initial position of the hero is known for all levels. Sound information of hero is also known. These are gained from the game code.

An instance of MIRPhysics is created. Global level variable is set to 0. All the contents needed to initialize the game environment are loaded from the level file. The right level file is selected according to the global level variable. A global progress variable is created with initial value 1. This global progress variable keeps track of the hero's progress during the level. This variable is updated according to puzzles. While the level file is being parsed, corresponding global variables and MIRFirstPerson instances are created.

<levelname></levelname>
The descriptive name of the level is read and saved in global level name string.

<intro></intro>
An instance of MIRIntro is created. The file pointer array of MIRIntro instance is created and elements of this array are assigned to the files listed in the level file. The sound file pointer of the MIRIntro instance is set to point to the sound file listed in the level file.

<missionlist></missionlist>
A string array is created to hold the mission texts. These missions are printed to screen according to the level progress variable.

<diary></diary>
A string array is created to hold the mission texts. Diary information is displayed according to the level progress variable. At a level progress value of 3, the contents of first three indexes of this array can be reached.

<textures></textures>
All the textures used in the game, except the textures of 3d models as they come with models, are read between these tags. An array of structure including a file pointer and a string is created and pointers are assigned to texture files listed in level file and the name of the name of the tag is saved in string part.

<map></map>
A matrix of MIRMap instances is created according to the size information in the level file. Grid information is parsed and saved in this matrix.

<paths></paths>
For each road, an array of MIRMap pointers is created and the pointers are assigned to the map matrix elements constructing the road.

<light></light>
An array of MIRLight object pointers is created. Each time a light tag sequence is met, an instance of MIRLight is created and the next element of the light array is assigned to this instance. The position, diffuse, ambient and specular of the light are saved in MIRLight instance.

<soundlist></soundlist>
An array of structure including a file pointer and a string is created and pointers are assigned to the files listed in level file and the name of the name of the tag is saved in string part.

<objectlist></objectlist>
3d models used in the game are read between these tags.

<statics></statics>
An array of structure including a MIRStaticObject pointer and a pointer to linked-list of structure including position, rotation angles and translation information is created. For every object read between these tags, an instance of MIRStaticObject is

created and filled with information of the 3d model. After that, MIRStaticObject pointer in the next element of the array is assigned to this instance.

<inventoryobjects></inventoryobjects>
An array of structure including a MIRPortableObject pointer and a pointer to linked-list of positions is created is created. While reading between inventoryobject tags, MIRPortableObject instances are created at random positions. Of course positions are checked for validity before rendering. This makes levels somehow dynamic.

<moving></moving>
An array of structure including a MIRMovingObject pointer and a pointer to linked-list of structure including position, velocity, path, animation sounds and a file pointer is created. For every object read between these tags, an instance of MIRMovingObject is created and filled with information of the 3d model. After that, MIRMovingObject pointer in the next element of the array is assigned to this instance.

<creatures></creatures>
An array of structure including a MIRMovingObject pointer and a pointer to linked-list of structure including position, velocity, path, animation sounds and a file pointer is created. For every object read between these tags, an instance of MIRMovingObject is created and filled with information of the 3d model. After that, MIRMovingObject pointer in the next element of the array is assigned to this instance.

This may seem confusing with the above array, but it improves efficiency in searches. If an animal or NHH is killed, the instance isn't destroyed but the position and path information is changed. As our hero is expected to fight with the creatures, number of creatures can decrease. And during these fights, hit search favors the distinct arrays.

<staticobjects></staticobjects>
For every object, the related element of static objects array is found according to the class information. After that, a new node is added to the linked list. The position, rotation angles and translation information is saved in this node.

<movingobjects></movingobjects>
For every object, the related element of moving objects array is found according to the class information. After that, a new node is added to the linked list. The position, velocity, path and animation sounds information is saved in this node. If the object is NHH there will be a manner file assigned. The pointer of this script file is saved in this node, too. In other moving objects, this pointer is set to NULL.

<creatureistances></creatureinstances>
For every object, the related element of moving objects array is found according to the class information. After that, a new node is added to the linked list. The position, velocity, path and animation sounds information is saved in this node. There is also a manner file assigned to the creature. The pointer of this script file is saved in this node, too.

<actionpoint></actionpoint >
An array of pointers to MIRActionPoint instances and an array of pointers to MIRPuzzle are created. Each time an actionpoint tag is met, according to the file extension between actionscript tags an instance of MIRActionPoint or MIRPuzzle is created and filled with the information between tags. If the action point refers to a puzzle, the solved variable of MIRPuzzle instance is set to 0. After that, the pointer in the next element of related array is assigned to this instance.

Hero and other object instances are registered to MIRPhysics. The global state variable is set to "game". drawingLoop() method of MIRWindow shows the intro because the global level progress variable is 0. At the end of the intro, the global level progress variable is set to 1. Therefore in the next drawingLoop() of MIRWindow, game environment is drawn. A subwindow is created10.

[10] Details of subwindow creation can be found in "OVERALL WINDOW STRUCTURE" part.

LOAD GAME

When the player wants to continue from a previously saved game, a sequence similar to above one is triggered. So only the points that differ are mentioned here.

When the player enters a name to load game, the corresponding save file is started to be parsed. The level and level progress variables are read and assigned to global variables. Then corresponding level file is started to be parsed. Static part of the level file is read11. The level file is closed and parsing of saved file continues.

<hero></hero>
Information of hero is passed to the newly created MIRHero instance.

<movingobjects></movingobjects>
For every object, the related element of moving objects array is found according to the class information. After that, a new node is added to the linked list. The position, velocity, path and animation sounds information is saved in this node. If the object is NHH there will be a manner file assigned. The pointer of this script file is saved in this node, too. In other moving objects, this pointer is set to NULL.

<creatureistances></creatureinstances>
For every object, the related element of moving objects array is found according to the class information. After that, a new node is added to the linked list. The position, velocity, path and animation sounds information is saved in this node. There is also a manner file assigned to the creature. The pointer of this script file is saved in this node, too.

<completepuzzles></completepuzzles>
In the action point array, the pointed elements in the indexes between these tags have their solved variable set to 1.

PROJECT MODELS

USE CASES

Configure Graphics Card Options

In order to change the graphics card settings the player should enter the “Options” menu by either using mouse or keyboard. From this menu, player will use “Graphics” option to customize these settings. In this new screen player will be able to change the resolution and the color depth of the game. By clicking the “OK” button, player will save the settings. The player will use “BACK” button to turn back to main menu.

Configure Sound Card Options

In order to change the sound card settings the player should enter the “Options” menu by either using mouse or keyboard. From this menu, player will use “Sound” option to customize these settings. In this new screen player will be able to change the volume of the game music. Player can save these settings by clicking “OK” whereas “BACK” will end up with main menu.

Configure Game Controller Options

In order to change the controller settings the player should enter the “Options” menu by either using mouse or keyboard. From this menu, player will use “Customize Controls” option to customize these settings. In this new screen player will see a set of actions and shortcut keys that are assigned to these actions. To change a shortcut key, player have to first click on the

[11] Details of static part of level file can be found in “LEVEL FILE” part.

action and than press the key (or mouse button) that will replace the old one. Also player can reset the shortcuts to their default values by clicking the "Defaults" button. The role of "BACK" is not different than those above.

Load a Game

Player should choose "Load" from "Main Menu" to load a previously saved game. In load game screen the names of all saved games will be displayed. Moving the mouse over a saved game will display the last frame of the saved game. The player has to double-click on the file to load it. Clicking "Play" will start the game while "BACK" will make the player turn back to main menu.

Save a Game

After each puzzle the player will be asked whether he/she wants to save the game or not. After choosing "Yes" the player will be asked to provide a file name. If the player select "No" he/she will continue with the next level. During the game player will not be able to save the game in any other way.

Move Character

Game will have two different views. First one will be a third-person view. During this mode player will move his/her hero by using mouse. Clicking an available target area will make the hero move there. The other view will be first-person. In this view player will use the keyboard to route and mouse to look. Player can make the hero run by pressing a previously specified shortcut key.

Solve Puzzles

In every level player may face a set of puzzles. When the player comes up with a puzzle, a new window will display the puzzle to be solved. Mouse will be used for clicking buttons, making logical decisions, selecting special objects, activating some mechanical structures; on the other hand keyboard will be used only for entering text into appropriate places.

Take & Drop Objects

In the third-person view, player should click on the item in order to make the player walk over the item and put it into his/her inventory. In the first-person view, player has to walk over the item to get it. For both view modes player can drop an item by drag & drop technique. There will be a special key reserved and an icon on the game panel for displaying inventory list. The player will be able to check his/her inventory any time during the game.

Interact with 'others' (monsters, friendly creatures …etc)

In the third-person view, player will click on the 'others' in order to attack/talk… The hero will have either a spell or a weapon with him/her. When the hero has the weapon in his/her hand the player will click on objects to shot them. Similarly, to use the spell on an object, clicking it will be sufficient. Also the enemies attack the hero and the hero dies if his or her health point decreases to zero. Player should go besides the friendly creatures to make them talk.

Read Diary

Player will have a diary containing all the quests and major events. To reach this diary player will press a predetermined key. The player may use the icons to turn the diary pages. Diary window will be reached by an icon on the panel as well.

Improve Skills

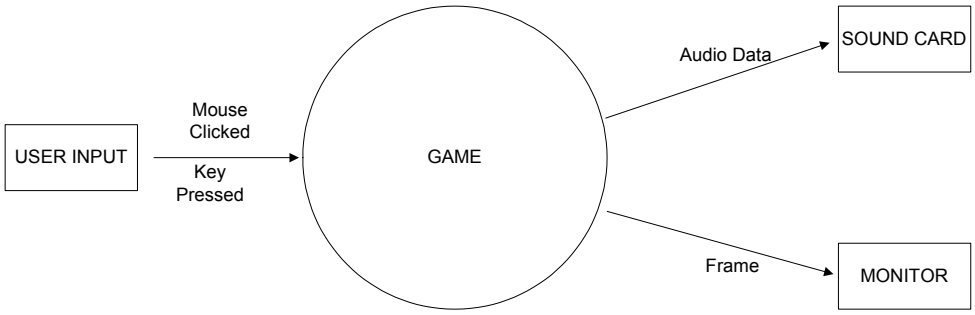
Player will be given a number of skill points at specific game states which will be displayed to the player on the game panel throughout the game. Player can distribute these points among the skills that are displayed on the screen by clicking on the icon on the sides of skills. Pressing "+" will increase the skill whereas "-" will decrease it.

Changing Spells

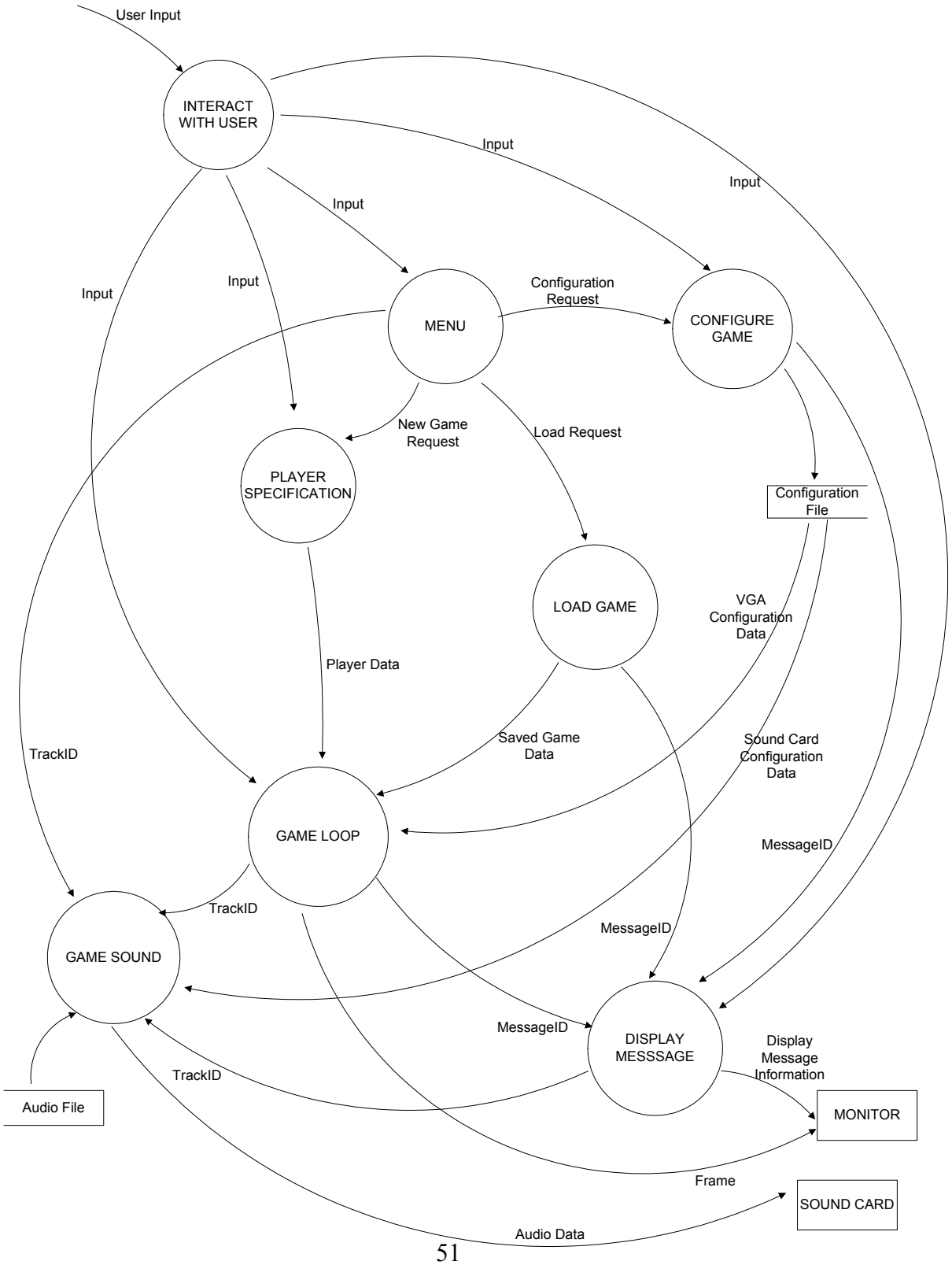
Throughout the game the player will be able to learn new spells which will be hidden at specific positions. He/she will use the game panel to change the spell.

FUNCTIONAL MODEL and INFORMATION FLOW

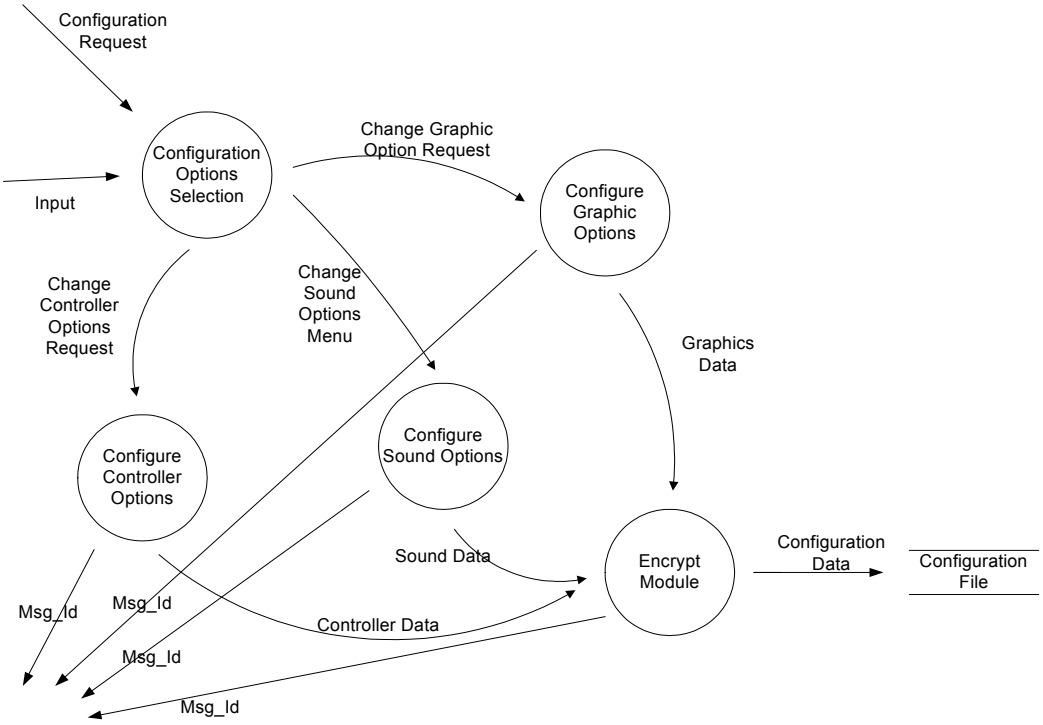
DFD LEVEL 0



DFD LEVEL 1

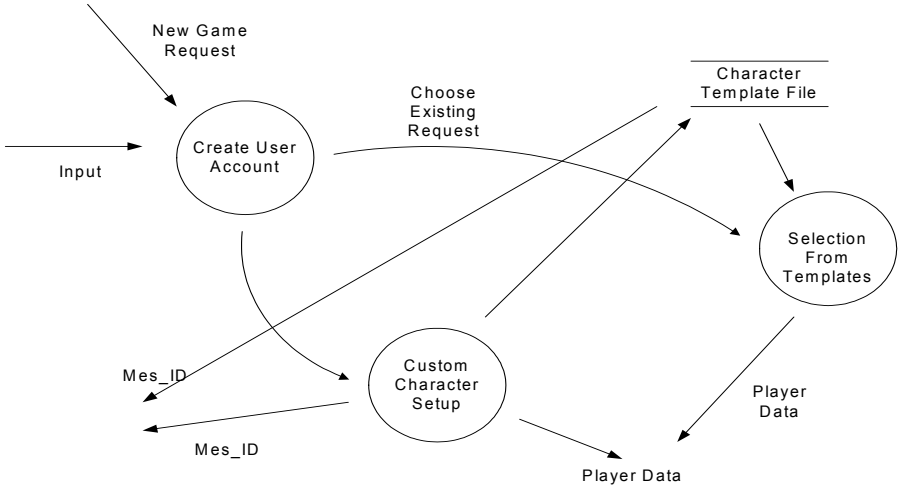

User Input
INTERACT WITH USER
Input
Input
Input
Input
Input
MENU
Configuration Request
CONFIGURE GAME
New Game Request
Load Request
PLAYER SPECIFICATION
Configuration File
LOAD GAME
VGA Configuration Data
Player Data
TrackID
Saved Game Data
Sound Card Configuration Data
GAME LOOP
MessageID
TrackID
MessageID
GAME SOUND
MessageID
DISPLAY MESSSAGE
Display Message Information
TrackID
Audio File
MONITOR
Frame
SOUND CARD
Audio Data

DFD LEVEL 2 Configuration



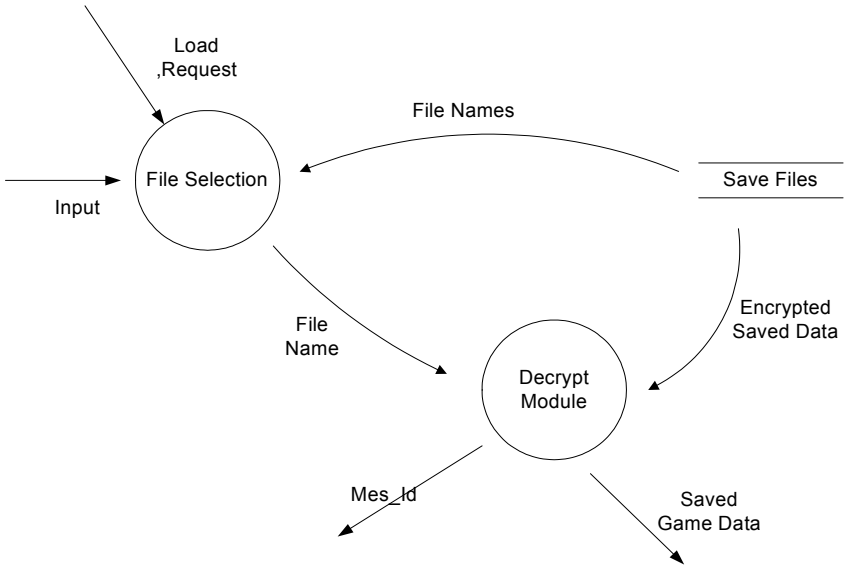
In the main menu when the player presses the configuration button he comes up with a configuration options selection menu. This is the menu which directs the player to different configuration options like sound, graphics and controller settings. Choosing the graphics settings sub-menu makes the user define graphics options like resolution whereas clicking on the sound button is all about the sound options like the volume on-of. On the other hand controller options menu helps the player customize keyboard settings. These three options are all evaluated by the encrypt module and a configuration file is created. Any kind of error throughout these processes is sent to display message module.

DFD LEVEL 2 Player Specification



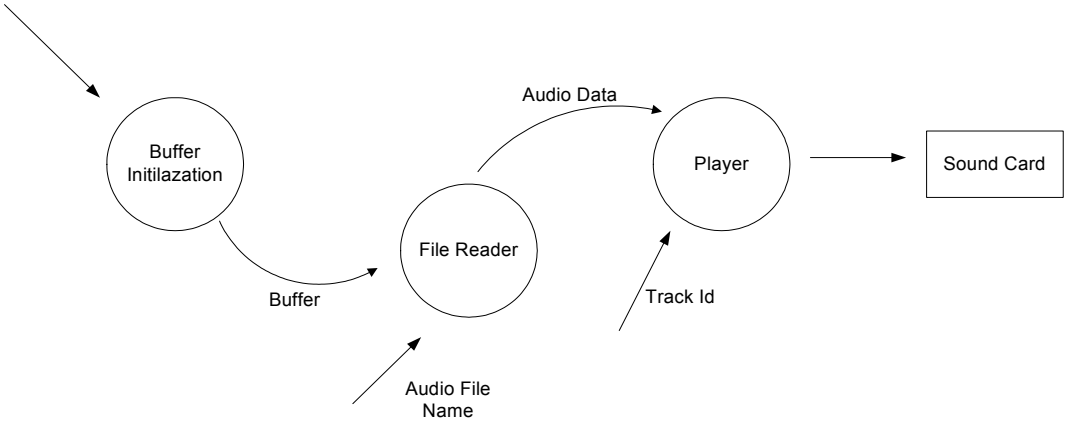
In case of choosing the new game option, the player will face a window asking him/her to specify his/her player. At that moment the player will have two options: either s/he will choose a character from the character template or will s/he customize his/her own character. I the player chooses the latter, the character s/he customized will be saved to the character template. In either case the player will end up with play the game option.

DFD LEVEL 2 Game Load



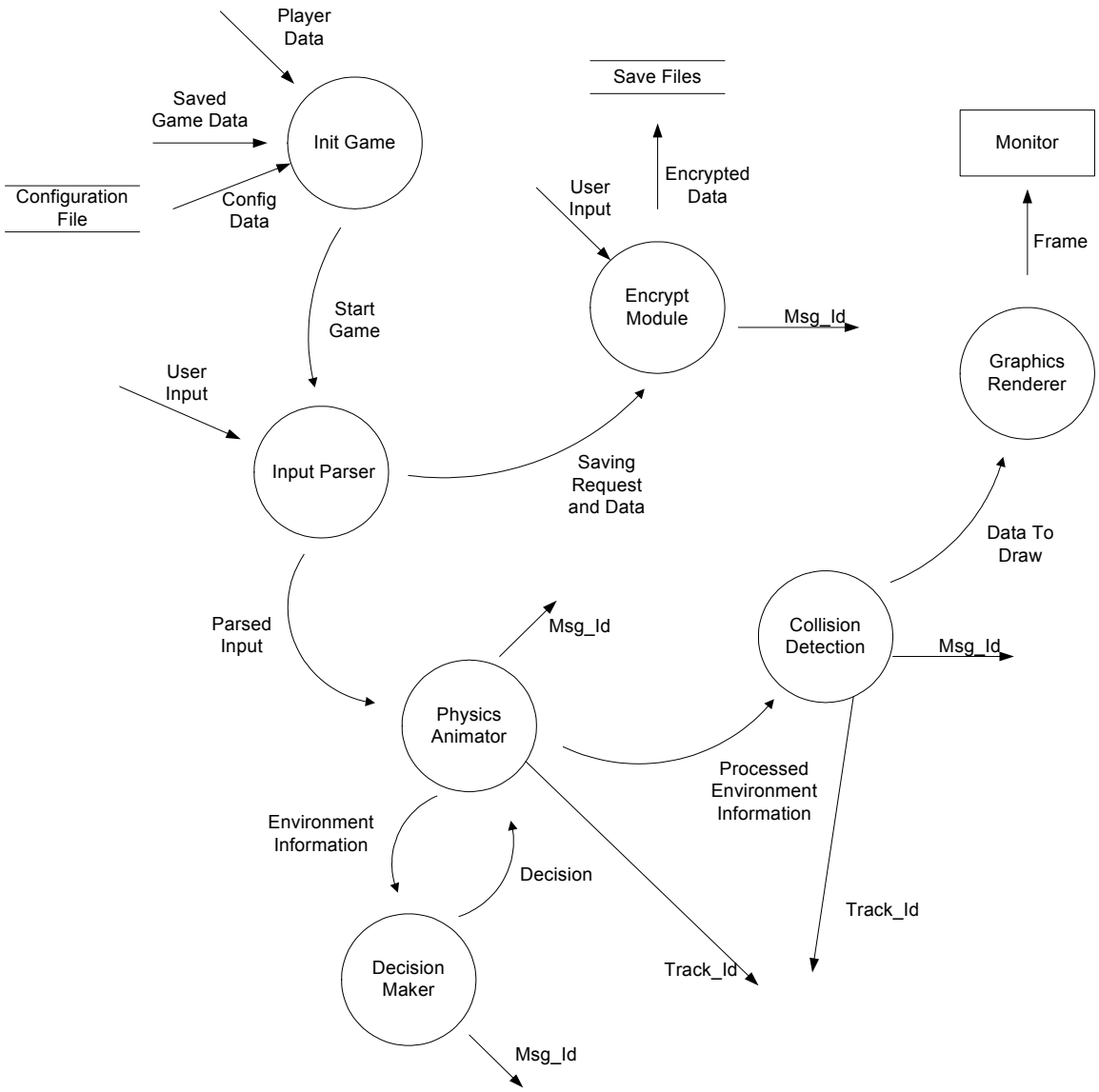
Load game is another option that main menu should handle before starting the game. This selection directs the player to an archive of previously saved files. By clicking on the name of the file the player will be able to decrypt the saved file and continue his/her game.

DFD LEVEL 2 Sound



Sound is one of the inevitable parts of a game. The sound processing starts with initialization of the sound buffer. After the initialization the sound files are read from the file system. And finally all ready-to-play sound data is controlled in player module. Player module plays the sound.

DFD LEVEL 2 Game Loop



Game loop can be thought as the brain of the game. It starts with a “play the game” click and end with a great pleasure of the player. The initialization of the game requires player information and configuration settings. After loading the game the player uses keyboard and mouse to give commands to his/her player. These commands are parsed by the input parser to give a meaning to each command. After the parsing operation the parsed input is passed to the physics animator. The role of the animator is to determine possible effects of the command and decision maker tells the animator what to do next. Collision detection checks whether there will be any collision in such an action and passes the final drawing information to the renderer. Renderer just draws the scene.

CFD LEVEL 1


User Input
INTERACT WITH USER
Input
Input
Input
Input
Input
MENU
Configuration Request
CONFIGURE GAME
New Game Request
Load Request
File is OK
PLAYER SPECIFICATION
Configuration File
Message Flag
LOAD GAME
VGA Configuration Data
Game Paused
Player Data
Game Over
Intro flag
Saved Game Data
Sound Card Configuration Data
TrackID
Sound Flag
Puzzle Flag
Exit Game
GAME LOOP
Message Flag
MessageID
Valid Action
TrackID
MessageID
GAME SOUND
Level OK
Has Message
MessageID
DISPLAY MESSSAGE
Display Message Information
TrackID
Audio File
MONITOR
Data is OK
Frame
SOUND CARD
Audio Data

At every step of the game, the computer will look at the environment state and make a decision. Control flow describes what the computer will do in certain cases. When the player starts the game he/she will go into a flow of menus, necessary for the initialization of the game. Throughout this menu-passing procedure the player will do sequential operations ending up with game loop. Apart from the load menu, in which success of the loading operation is important, there is little control over the menus. But the things change in game loop.

When the player selects “play the game” button, according to whether this is a loaded game or a “new game” the intro is showed to the player, and then the hero starts to play the game. There will be specific locations in the game where the player will have to solve a puzzle which is determined by a puzzle flag. The sounds that will be played in the game may vary according to the location of the player and the environment in which our hero is currently. Sound flag controls which sound to play in which situation. When the player intends to move the hero according to the game mode he/she will click on a point in the screen (in third person) or use the keyboard (in first person). What controls whether to move the hero or not is determined by the valid action flag. The player may want to pause the game and/or exit in the game, pause and exit flags keep this information and make the game loop behave accordingly. Whether the player deserved to finish and jump up to the second level is determined by the level flag.

Message flag interrupts the flow of the program and makes the program show a message to the user when there is an error or a message to the player throughout the game. When the game is over either because the hero is killed or the hero finished all levels, game over flag will be asserted and the program flow will change accordingly.

WORK PACKAGES AND MILESTONES

Milestone 1

4-12/Oct/2004
Project Planning and Product Statement

- Meeting with supervisor
- Identify needs and project constraints
- Write the proposal
- Definition of the product

Milestone 2

13/Oct/2004-8/Nov/2004
Project Requirements

- Inspecting computer game industry and market games
- Requirements gathering
- Inspecting library and tool options
- Defining game context
- Defining system properties
- Defining software specifications
- Defining data, functional, behavioral model
- Writing analysis report
- Completion requirement analysis

Milestone 3

9/Nov/2004-30/Nov/2004
Initial Design

- Reviewing software specifications
- Data design
- Architectural design
- Designing naming convention
- Revising data, behavior and function diagrams
- Writing initial design report
- Completion initial design

Milestone 4
1/Dec/2004-10/Jan/2005
Detailed Design
- Reviewing software specifications
- Reviewing data design
- Reviewing architectural design
- Revising data, behavior and function diagrams
- Writing detailed initial design report
- Completion design

Milestone 5
1/Jan/2004-19/Jan/2005
Prototyping
- Implementing the prototype of the game
- Presenting prototype
- Completion prototype

Milestone 6
1/Feb/2005-02/May/2005
Implementation
- Implementing modules
  - Work package 1 < Implementation of classes related to environment class>
  - Work package 2 < Implementation of classes related to menu class>
  - Work package 3 < Implementation of classes related to sound class>
  - Work package 4 < Implementation of classes related to graphics class>
  - Work package 5 < Implementation of classes related to form class>
  - Work package 6 < Implementation of classes related to physics class>
  - Work package 7 < Implementation of classes related to utility class>
  - Work package 8 < Implementation of classes related to window class>
- Developing scripting mechanism
  - Work package 9 < Implementation of scripts >
- Designing graphical user interfaces
  - Work package 10 <Designing graphical user interfaces >
- Shaping 3D models and textures
  - Work package 11<Designing 3D objects, textures and map>
- Releasing first version

Milestone 7
02/May/2005-31/May/2005

Testing
- Black box and white box testing
- Debugging application
- Preparing final package

Milestone 8
1/Jun/2005

Final Package
- Preparing documentation
  - Work package 12 <Documentation>
- Final release
  - Work package 13 <Preparing final package>
- Presenting the product

CONCLUSION

This report is the detailed final design report of the 3D-adventure game project. The detailed design report is the major milestone in the schedule of the project. It is prepared to establish a connection between the design and implementation of the 3D action/adventure game project of MIR. The modules, classes, hierarchical details, diagrams, data structures, file formats and other design products given in this document are produced in order to guide MIR during the implementation of the game. We have mentioned interface design including graphical user interface. A planned schedule for second semester is also included in the report.

APPENDIX

CODING STANDARDS

HEADER FILES

All of the header files should start with a comment block which gives information about the file. Mainly this block should contain the name of the author of the file, the version of the file, the last modification date of the file and a brief explanation of the file.

Header files should conform to the following template:

```
/*
 * File Name:
 * Author:
 * Version:
 * Last Modified:
 * Description:
 */
#ifndef __CLASSNAME_H
#define __CLASSNAME_H

// All includes goes here.
// All declarations goes here.

#endif // end of __CLASSNAME_H
```

SOURCE CODE FILES

Similar to the header files, all of the source files should have a comment block at the top of the file. It should contain the same information that is required in header files.

Source code files should conform to the following template:

```
/*
 * File Name:
 * Author:
 * Version:
 * Last Modified:
 * Description:
 */

// Source code goes here…
```

GLOBAL FUNCTIONS

Before each global function there should be a comment block. This comment block should contain the information about the function. Also each word of global function names should start with a capital letter. All of the parameters passed to the function should conform to the local variable coding standards. Each code block should be written one tab indented.

Example:

```
/*
 * Function Name: GetSquare
 * Parameters: double dNumber
 * Return Value: double
 * Description: This function computes the square of the given number.
 */
double GetSquare (double dNumber)
{
        return dNumber * dNumber;
}
```

MACROS and CONSTANT VARIABLES

All of the macros and constant variables should be written with upper-case letters. If the macro or constant variable names contain more than one word underscores should be used between words.

Example:

```
#define VERSION 0.001
#define MAX_NODES 1000
const int MAX_DEPTH = 20;
const double RADIUS = 1.17:
```

LOCAL VARIABLES

All local variables should start with a prefix indicating the type of the variable. List of these prefixes is given below. Each word of local variable names should start with capital letters. For commonly used loop variables like i, j, k there is no coding convention.

Int	iVariableName		unsigned int	uiVariableName
Short	sVariableName		unsigned short	usVariableName
Long	lVariableName		unsigned long	ulVariableName
Char	cVariableName		unsigned char	ucVariableName
Float	fVariableName		double	dVariableName
Boolean	bVariableName		null terminated string	szVariableName
Pointer	pVariableName		double pointer	ppVariableName
Array	aVariableName		handle	hVariableName
Word	wVariableName		double word	dwVariableName
enumerated value	eVariableName			

Example:

```
int iDefaultValue = 20;
char aszStr [255] = {'\0'};
double *apdDegrees [100];
HWND hWnd;
int i = 0, j = 5;
```

GLOBAL VARIABLES

Global variables start with the ‘g_’ prefix. Also all of the naming conventions for local variables are also applied to these variables.

Example:

```
bool g_bIsRunning = false;
unsigned char g_aucKeys [255];
```

STATIC VARIABLES

Static variables start with ‘s_’ prefix. All naming conventions of local variables are also applied to these variables.

Example:

```
static float s_fRadius = 1.0f;
static char s_cCapitalA = ‘A’;
```

CLASSES

Class names should start with ‘MIR’ prefix. First letter of each word of the name should be capital. No function definitions should be given in class declarations. All of the member variables should start with the ‘m_’ prefix. All naming conventions of local variables are also applied to these variables. Private and protected member variables should start with an underscore. First word of the methods should start with lower-case letters. All other words should start with upper-case letters. Private and protected methods should also start with an underscore. Instance of a class should start with ‘mir’ prefix. Also all other variable naming conventions should be applied to the created instance.

Example:

```
class MIRFileManager
{
public:
        MIRFileManager ();
        ~MIRFileManager ();
        bool openFile (char * pcFileName);
        bool readFile (void);
        unsigned int m_uiClassType;
private:
        bool _parseFile (void);
        char *_m_pcBuffer;
        MIRFileStream *_m_pmirInputFile;
protected:
        char * _getBuffer ();
};
```

INSTALLATION

We consider to put all the components of the system into a self-extractable file. When it is executed, first it extracts the components. Then, it installs the components automatically.Therefore, our system will have an easy installation.

FILE FORMATS

LEVEL FILE FORMAT

```
<levelname>MM‘de dehset</levelname>
<intro>
      <image>img1.jpg</image>
      <image>img2.jpg</image>
      …
      <sound>introsound</sound>
</intro>
<missionlist>
      <mission>Kutuphaneye git</mission>
      <mission>Kitabı bul ve getir </mission>
       …
</missionlist>
<diary1>bidibidi</diary1>
<diary2>bidibidi</diary2>
<diary3>bidibidi</diary3>
….
<thirdperson>
        <light>
            <posX>10</posX>
            <posY>55</posY>
            <posZ>60</posZ>
            <diffuse>
                <1>1.0</1>
                <2>1.0</2>
                <3>1.0</3>
                <4>1.0</4>
            </diffuse>
            <ambient>
                <1>0.0</1>
                <2>0.0</2>
                <3>0.0</3>
                <4>1.0</4>
            </ambient>
            <specular>
                <1>1.0</1>
                <2>1.0</2>
                <3>1.0</3>
                <4>1.0</4>
            </specular>
        </light>
        <textures>
            <grass>grass.bmp</grass>
            <concrete>concrete.bmp</concrete>
            ….
```

```
</textures>
<map>
    <grid1>
        <g1X>24</g1X><g1Y>24</g1Y>
        <g2X>25</g2X><g2Y>24</g2Y>
        <g3X>25</g3X><g3Y>25</g3Y>
        <g4X>24</g4X><g4Y>25</g4Y>
        <t1X>24</t1X><t1Y>24</t1Y>
        <t2X>25</t2X><t2Y>24</t2Y>
        <t3X>25</t3X><t3Y>25</t3Y>
        <t4X>24</t4X><t4Y>25</t4Y>
        <texture>grass</texture>
    </grid1>
    ….
    <gridn>
    ….
    </gridn>
</map>
<paths>
    <road1>
        grid61
        grid62
        grig63
        grid64
    </road1>
    ….
</paths>
<soundlist>
    <introsound>intro.wav</introsound>
    <footstep>fstep.wav</footstep>
    <cat>miyav.wav</cat>
    <car>car.wav</car>
    <carbreak>carbreak.wav</carbreak>
    <scream>scream.wav</scream>
    <clapping>clapping.wav</clapping>
    <roar>roar.wav</roar>
    ….
    <gender1>sound11.wav</gender1>
    <gender1>sound12.wav</gender1>
    ….
    <gendern>soundn1.wav</gendern>
    <gendern>soundn2.wav</gendern>
    ….
</soundlist>
<objectlist>
    <statics>
        <tree>tree.3ds</tree>
        <building>building.3ds</building>
        ….
    </statics>
    <inventoryobjects>
        <money>money.3ds</money>
        <pistol>pistol.3ds</pistol>
        ….
    </inventoryobjects>
    <moving>
```

```
            <femalestudent>female.3ds</femalestudent>
            ….
        </moving>
        <creatures>
            <weakcreature>weak.3ds</weakcreature>
            ….
        </creatures>
</objectlist>
<staticobjects>
        <object>
            <class>tree</class>
            <posX>213</posX>
            <posY>100</posY>
            <posZ>11</posZ>
            <rotationangleX>30</rotationangleX>
            <rotationangleY>30</rotationangleY>
            <rotationangleZ>30</rotationangleZ>
            <translationX>3</translationX>
            <translationY>3</translationY>
            <translationZ>3</translationZ>
        </object>
        ….
</staticobjects>
<movingobjects>
        <object>
            <class>car</class>
            <posX>22</posX>
            <posY>30</posY>
            <posZ>2</posZ>
            <velocity>20</velocity>
            <path>road2</path>
            <animation1>car</animation1>
            <animation2>carbreak</animation2>
        </object>
        <object>
            <class>femalestudent</class>
            <posX>22</posX>
            <posY>30</posY>
            <posZ>2</posZ>
            <velocity>6</velocity>
            <path>road1</path>
            <manner>femalestudent.py</manner>
            <animation1>scream</animation1>
            <animation2>clapping</animation2>
        </object>
        ….
</movingobjects>
<creatureinstances>
        <object>
            <class>weakcreature</class>
            <posX>22</posX>
            <posY>30</posY>
            <posZ>2</posZ>
            <velocity>7</velocity>
            <path>road1</path>
            <manner>weakcreature.py</manner>
```

```
                <animation1>roar</animation1>
            </object>
            ….
        </creatureinstances>
        <actionpoint>
            <actionscript>opendoor.py</actionscript>
            <level>2</level>
            <posX>231</posX>
            <posY>213</posY>
            <posZ>12</posZ>
        </actionpoint>
        <actionpoint>
            <actionscript>puzzle1.txt</actionscript>
            <level>1</level>
            <posX>231</posX>
            <posY>213</posY>
            <posZ>12</posZ>
        </actionpoint>
</thirdperson>
<firstperson>
        //similar to thirdperson content
</firstperson>
.
//more than one firstperson modes can exist
.
<firstperson>
        //similar to thirdperson content
</firstperson>
```

SAVE FILE FORMAT

```
<level>2</level>
<progress>3</progress>
<hero>
        <name>mir</name>
        <posX>21</posX>
        <posY>45</posY>
        <posZ>3</posZ>
        <weaponinhand>pistol</weaponinhand>
        <health>50</health>
        <magicpower>32</magicpower>
        <stamina>38</stamina>
        <money>3000</money>
        <inventoryobject>telephone</inventoryobject>
        <inventoryobject>shield</inventoryobject>
        <inventoryobject>spellA</inventoryobject>
        <inventoryobject>money</inventoryobject>
        ….
</hero>
<movingobjects>
        <object>
            <class>car</class>
```

```
            <posX>24</posX>
            <posY>32</posY>
            <posZ>4</posZ>
            <velocity>20</velocity>
            <path>road2</path>
            <animation1>car</animation1>
            <animation2>carbreak</animation2>
        </object>
        <object>
            <class>femalestudent</class>
            <posX>24</posX>
            <posY>32</posY>
            <posZ>4</posZ>
            <velocity>6</velocity>
            <path>road1</path>
            <manner>femalestudent.py</manner>
            <animation1>scream</animation1>
            <animation2>clapping</animation2>
        </object>
        ….
</movingobjects>
<creatureinstances>
        <object>
            <class>weakcreature</class>
            <posX>24</posX>
            <posY>32</posY>
            <posZ>4</posZ>
            <velocity>7</velocity>
            <path>road1</path>
            <manner>weakcreature.py</manner>
            <animation1>roar</animation1>
        </object>
        ….
</creatureinstances>
<completepuzzles>
        <index>5</index>
        ....
</completepuzzles>
```

PUZZLE FILE FORMAT

```
<objectlist>
        <button>button.3ds</button>
        …..
</objectlist>
<puzzleobjects>
        <object>
        <class>button</class>
        <posX>12</posX>
        <posY>132</posY>
        <posZ>2</posZ>
        <rotationangleX>30</rotationangleX>
        <rotationangleY>30</rotationangleY>
        <rotationangleZ>30</rotationangleZ>
```

```
        <translationX>3</translationX>
        <translationY>3</translationY>
        <translationZ>3</translationZ>
        </object>
        ….
</puzzleobjects>
<solution>puzzle1.py</solution>
```

CONFIGURATION FILE FORMAT

```
<sound>-2000</sound>
<graphics>
        <width>1024</width>
        <height>768</height>
</graphics>
<shortcuts>
        <forward>up-arrow</forward>
        <backward>down-arrow</backward>
        <right>right-arrow</right>
        <left>left-arrow</left>
        ….
</shortcuts>
```