INITIAL DESIGN REPORT

by

MIR

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

TABLE OF CONTENTS

INTRODUCTION 3
  PURPOSE OF DOCUMENT 3
  SCOPE OF DOCUMENT 3
GAME DESCRIPRION 3
CUSTOMER INTERVIEW 5
MAJOR CONSTRAINTS 5
SYSTEM MODULES 6
CLASS HIERARCHY 8
GRAPHICAL USER INTERFACE 20
PROJECT MODELS 26
  USE CASES 26
  FUNCTIONAL MODEL and INFORMATION FLOW 27
CONCLUSION 34
APPENDIX 35
  CODING STANDARDS 35

INTRODUCTION

PURPOSE OF DOCUMENT

This document is the initial design report of the 3D action/adventure game project of MIR. The purpose of this document is to define the design specifications and system solutions for the game design.

SCOPE OF DOCUMENT

The architecture of the project is defined in this document. This includes the detailed description and specification of the modules and the classes which build the game. This document also covers the hierarchy between the classes. 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.

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, player has two camera modes. In first mode, camera is placed on the upper-right side of the hero. Camera moves relative to the hero to obtain a static view. So player always sees the hero with the same angle. In second mode, camera is placed behind the hero so player has the hero's view. In both modes, hero is fully visible and the size of the hero is approximately one-seventh of the screen height.

GAME FILES

All game files are stored in encrypted and compressed format. Therefore reading a file needs an uncompressing and decryption.

MAP

The game map is a list that contains static objects and the ground. It has a global coordinate system which has a origin at the left bottom corner of the map. There is only one map for whole game and each level takes parts of it according to global coordinates.

Map information is held in a text file and this file contains names of files of all static objects (models) and their locations over the map. 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, puzzle information. 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, nonhero humans, creatures, potions, money and weapons. 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.

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 time, and walk randomly 10% of the time. 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 time, and walk randomly 20% of the time.

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 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, Boost.Python 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.

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.

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:

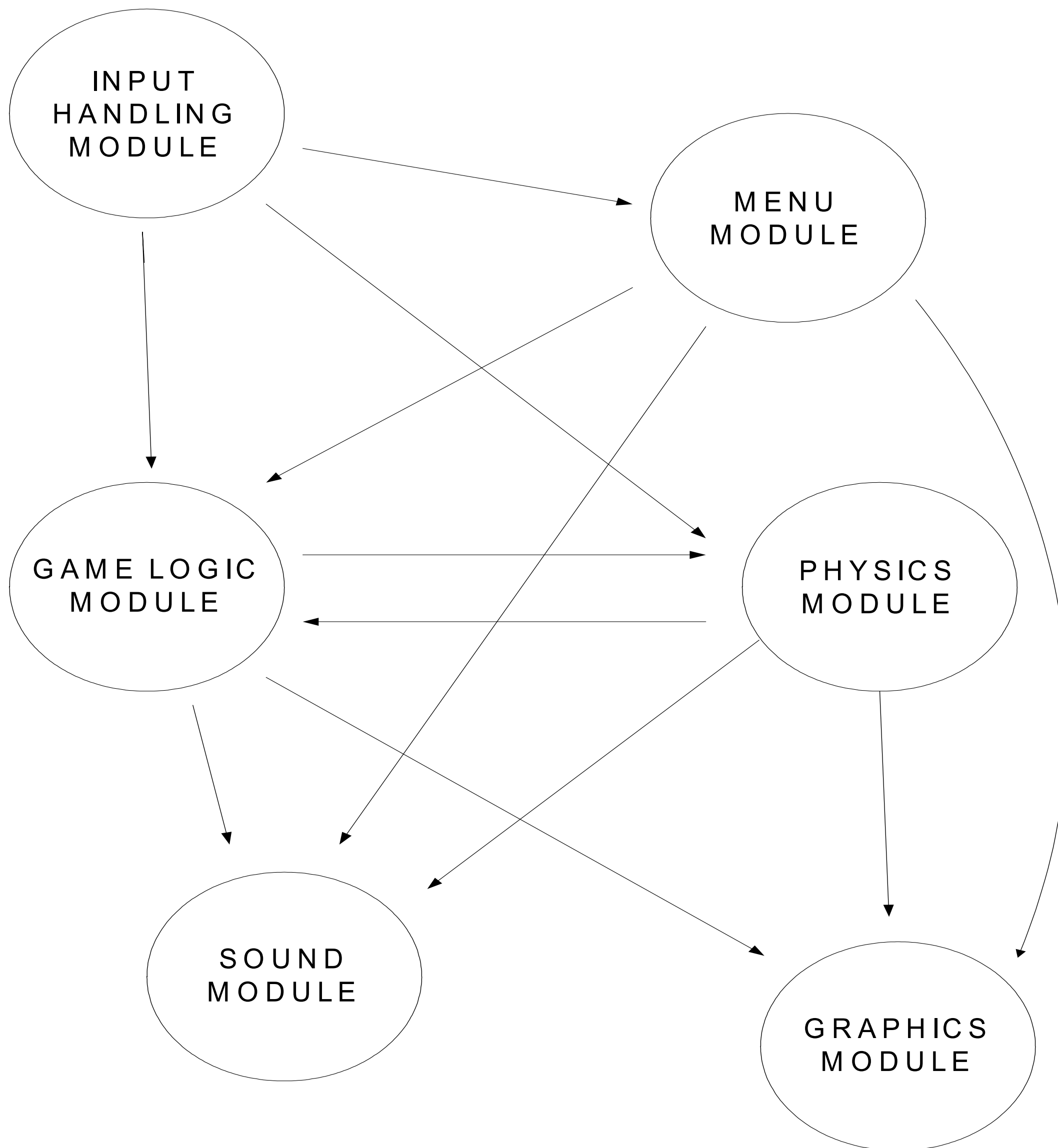
- P III 800 MHz Processor
- 64MB GeForce-4 VGA
- 256MB Memory
- 16bit SoundCard
- Enough free disk space

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

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. Boost.Python library will be used for the integration of C++ and Python. Microsoft Visio is the tool that will be used for technical drawings. CVS will be used for version controlling.

SYSTEM MODULES



INPUT HANDLING MODULE

All player inputs to control the hero are processed and passed to the related module by the input handling module.

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 in 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.

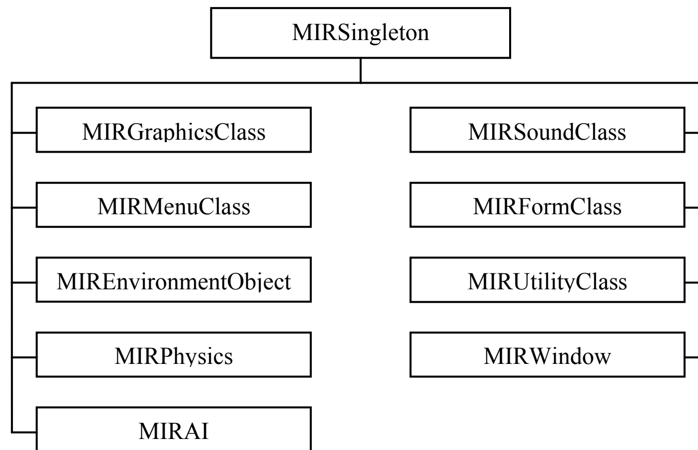
GAME LOGIC MODULE

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.

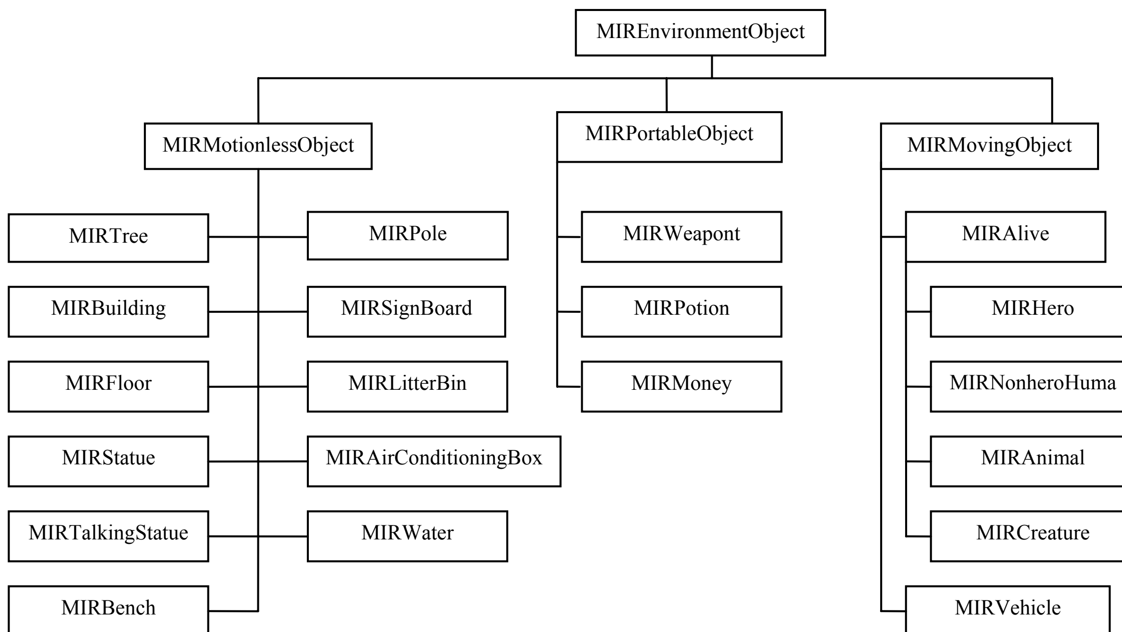
PHYSICS MODULE

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 base class for all objects that do not move in the game environment. It only has an int named m_ iBodyId which is unique for each object.

MIRTree

MIRTree class is for all kinds of tree in the environment.

MIRBuilding

MIRBuilding class is for all types of buildings and alike structures.

MIRFloor

MIRFloor class is for all kinds of trees in the environment.

MIRStatue

MIRStatue class is for all kinds of statues in the environment.

MIRTalkingStatue

- m_iTrackId
- playTrack( )

MIRTalkingStatue class is for statues that have ability to talk in game. m_iTrackId is the unique identifier of the track which is played by playTrack function of this class.

MIRBench

MIRBench class is for all kinds of benches in the environment.

MIRPole

MIRPole class is for all kinds of poles in the environment.

MIRSignBoard

MIRSignBoard class is for all kinds of signes like traffic signs in the environment.

MIRLitterBin

MIRLitterBin class is for all kinds of litterbins in the environment.

MIRAirConditioningBox

MIRAirConditioningBox class is for all kinds of airconditioning boxes around the environment.

MIRWater

MIRWater class is for all kinds of objects that has water in/on it such as river, pool, rill in the environment.

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

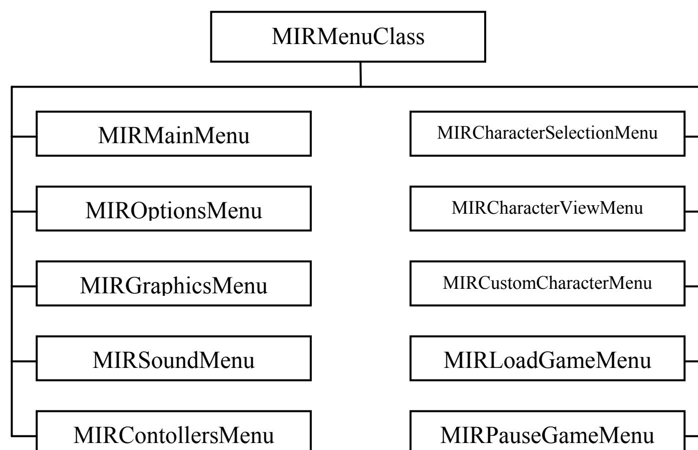
MIRWeapon class is the class for all weapons that are used by hero in the game environment.

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.

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

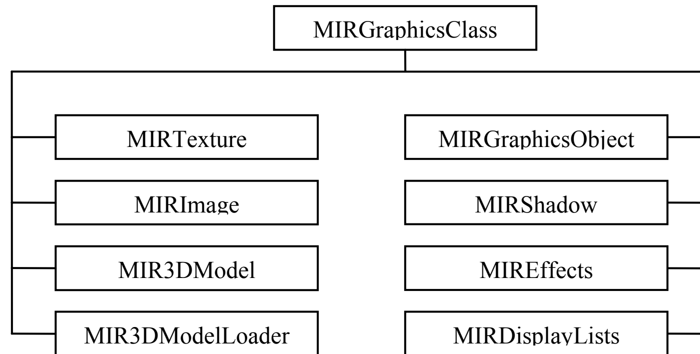
In the m_mirTextBox, the player specifies the name of his or her character. With this name, the related save file is found and loaded when the player clicks on the m_mirButtonforPlay. m_mirButtonforBack leads the player to the previous menu screen.

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.

MIRDisplayLists

This class will be used to handle all the display list operations 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

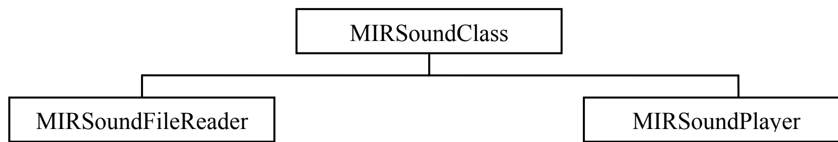
- m_WorldId
- m_SpaceId
- createWorld ()
- createSpace ()
- detectCollisions ()

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
- 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.



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

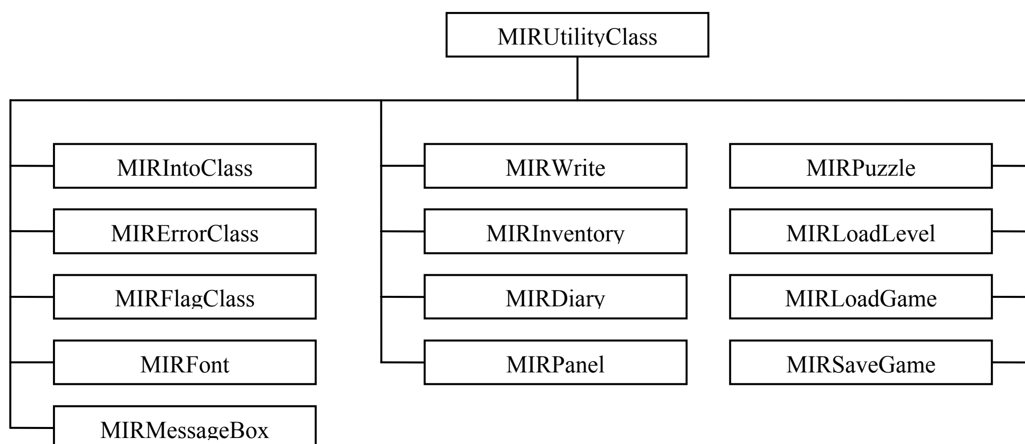
- m_mirSoundFileReader
- initSoundBuffer()

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_aszMissionList
- m_iMissionId
- m_iLevel
- displayDiary()

At each level the player will have several missions. He will be able to check the next mission. MIRDiary will keep the "mission" information for each level. m_iLevel and m_iMissionId together will be used
to choose select the "mission" string from the m_saMissionList and displayDiary() will print the "mission".

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.

MIRPuzzle

- m_bState
- m_iLevel
- m_vPosition
- drawPuzzle()

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.

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.

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.

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

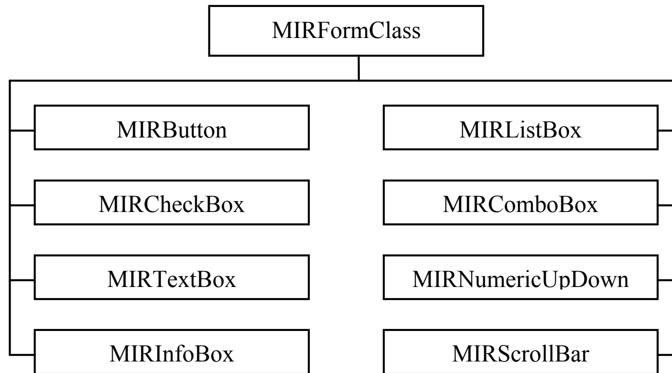
MIRWrite

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 name of classes are listed. MIRInfoBox needs explanation and it is given below.

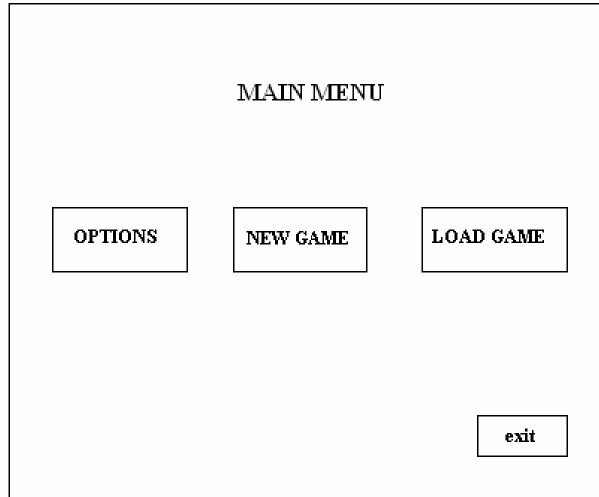
MIRButton
MIRCheckBox
MIRTextBox
MIRListBox
MIRComboBox
MIRScrollBar
MIRNumericUpDown

MIRInfoBox

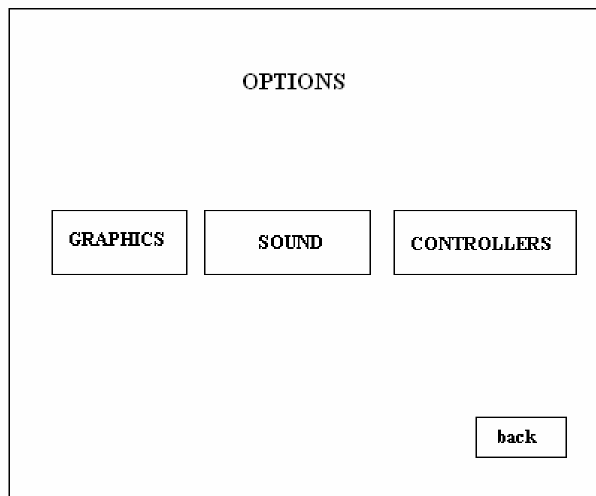
This class implements the place for demonstrating both image and text.

GRAPHICAL USER INTERFACE

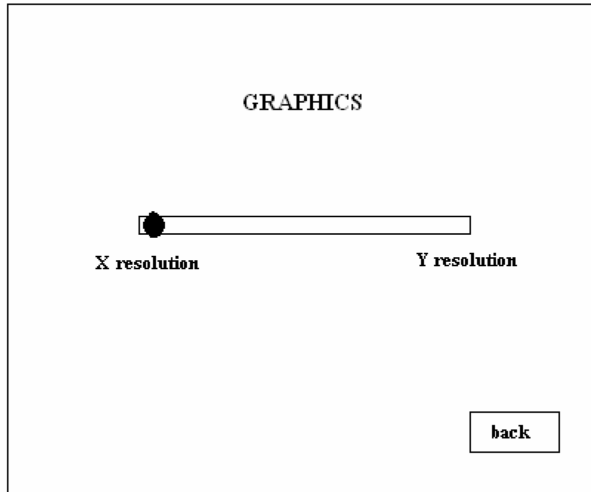
These interfaces are not the final designs. They are prepared in order to generate a general idea of how the game interface will look like.



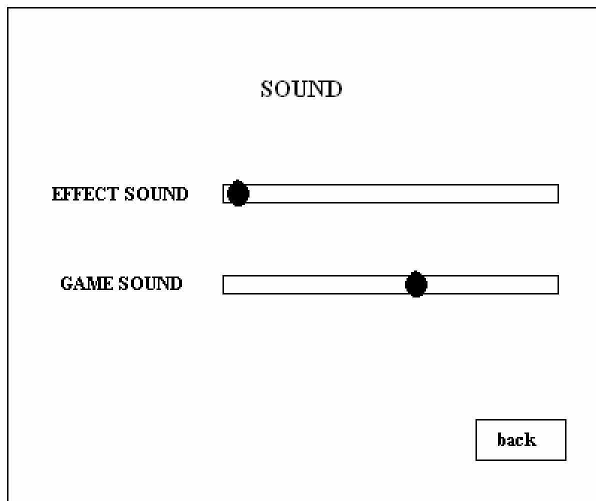
The first interaction with the player is through the Main Menu. If the player wants to change the options of the game, s/he presses the OPTIONS button and the following screen opens.



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.

CONTROLLERS

Enter Key for Controller 1

Enter Key for Controller 2

.
.
.

Enter Key for Controller n

back

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.

CHARACTER SELECTION

CHAR1 CHAR2 ...... CHARn

CUSTOMIZE

back

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.

CHARACTER VIEW

Enter Name

PLAY

....

....

back

CUSTOM CHARACTER

Enter Name

You have X bonus points
to add your properties.

PROPERTY 1 < >

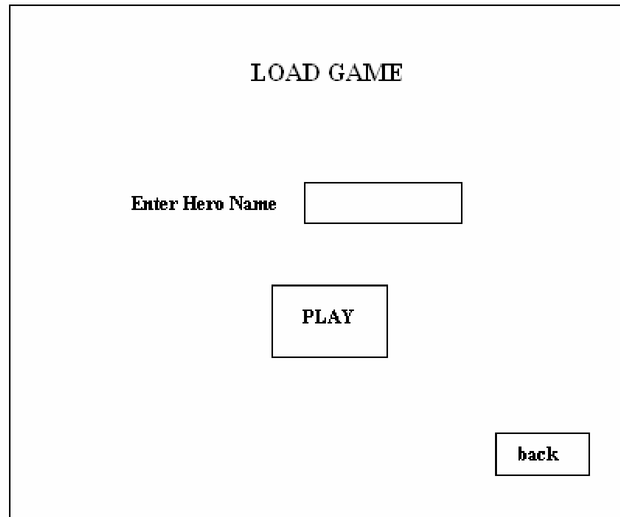
.....

PROPERTY n < >

PLAY

back

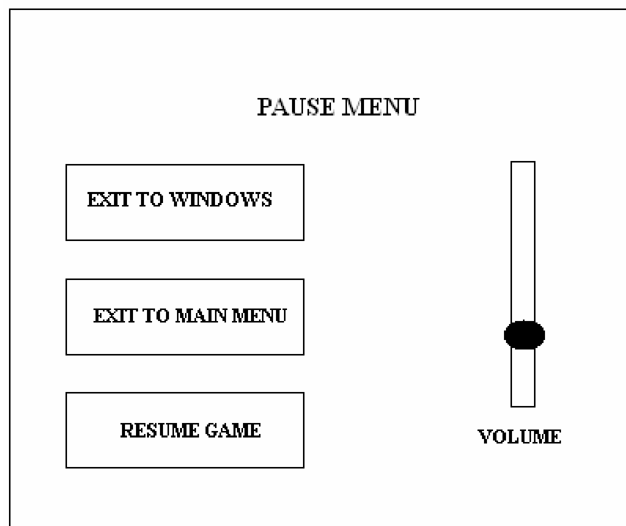
For both of the possible hero selection ways, the player enters a name for his or her hero. Customizing is carried out by changing the reserved properties of a designer character.



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.

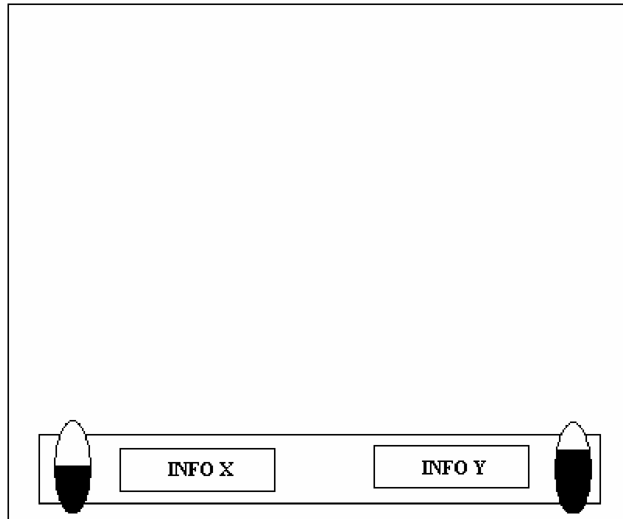
The PLAY buttons in the above screens leads the player into the game.

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.



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 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

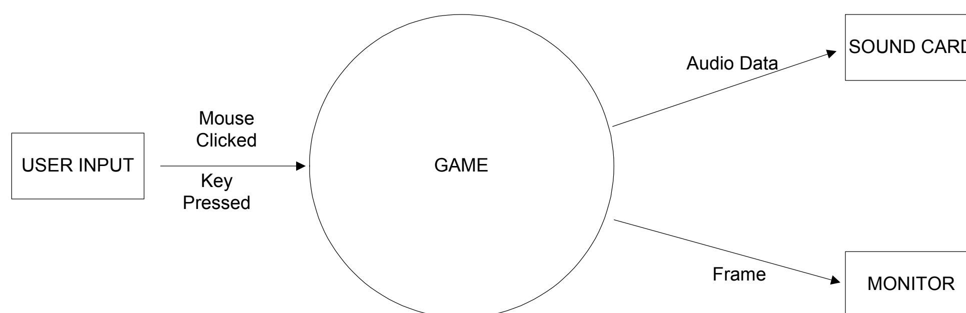
Player will be given a number of skill points at specific game states which will be displayed to the player on the game panel throughout the game. Player can distribute these points among the skills that are displayed on the screen by clicking on the icon on the sides of skills. Pressing "+" will increase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

Changing Spells

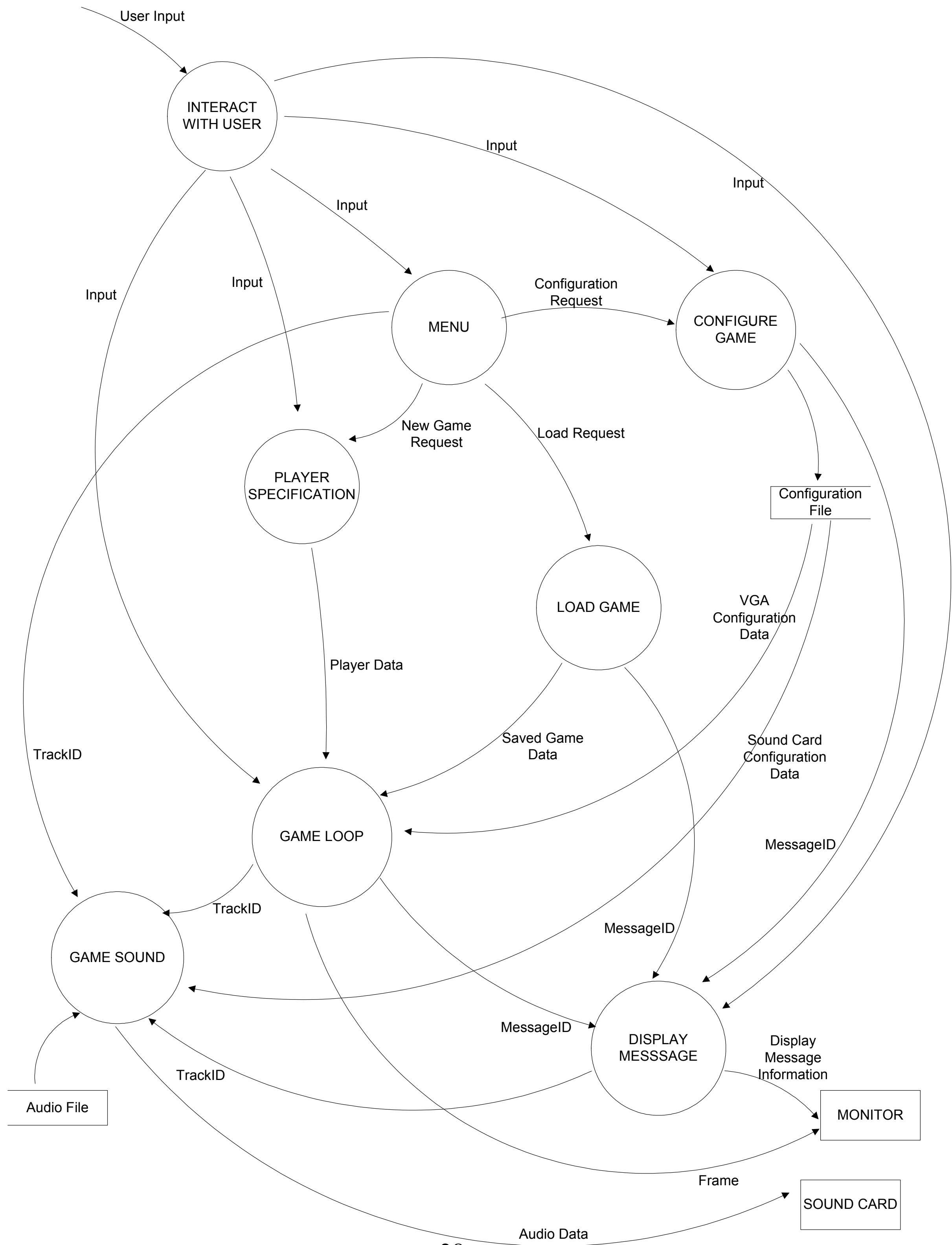
Throughout the game the player will be able to learn new spells which will be hidden at specific positions. He/she will use the game panel to change the spell.

FUNCTIONAL MODEL and INFORMATION FLOW

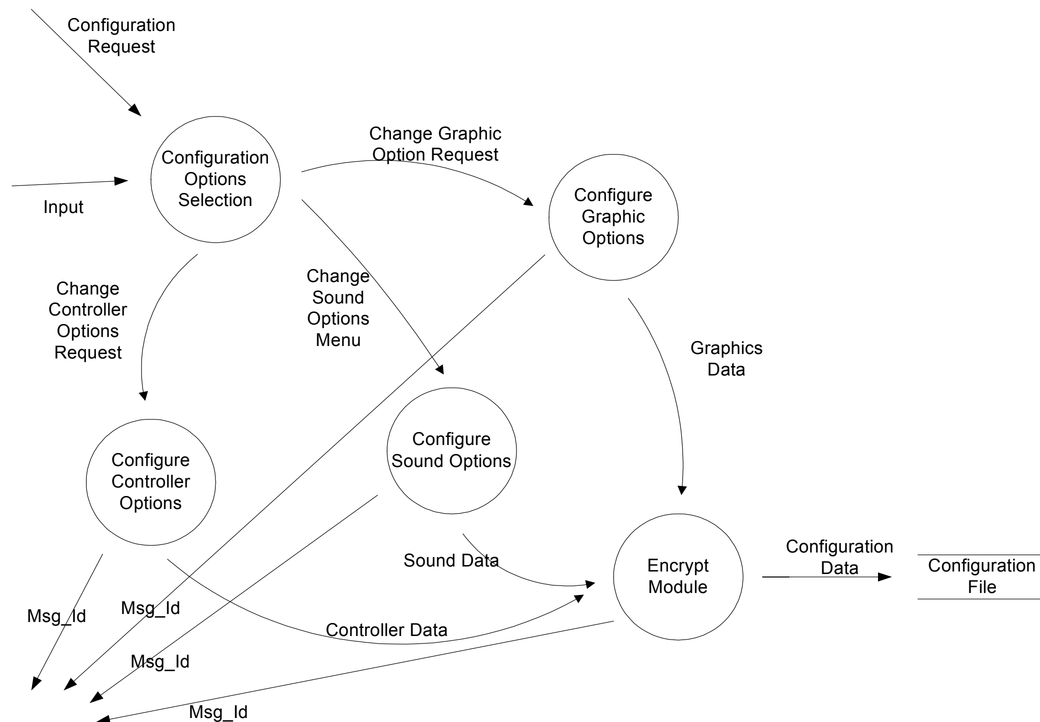
DFD LEVEL 0



DFD LEVEL 1

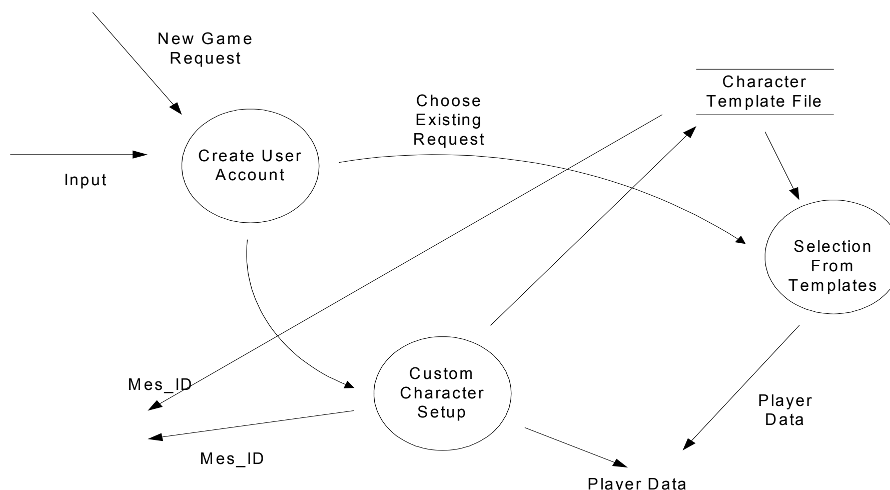


DFD LEVEL 2 Configuration



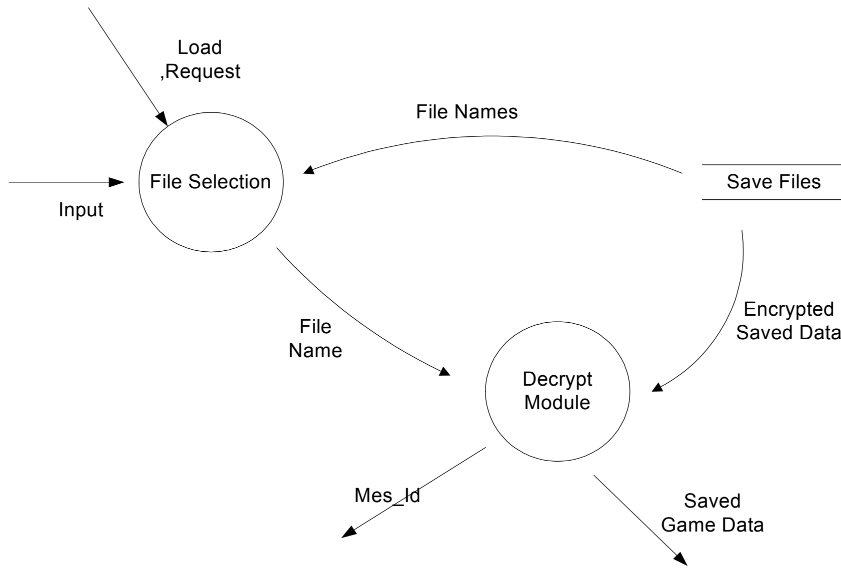
In the main menu when the player presses the configuration button he comes up with a configuration options selection menu. This is the menu which directs the player to different configuration options like sound, graphics and controller settings. Choosing the graphics settings sub-menu makes the user 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



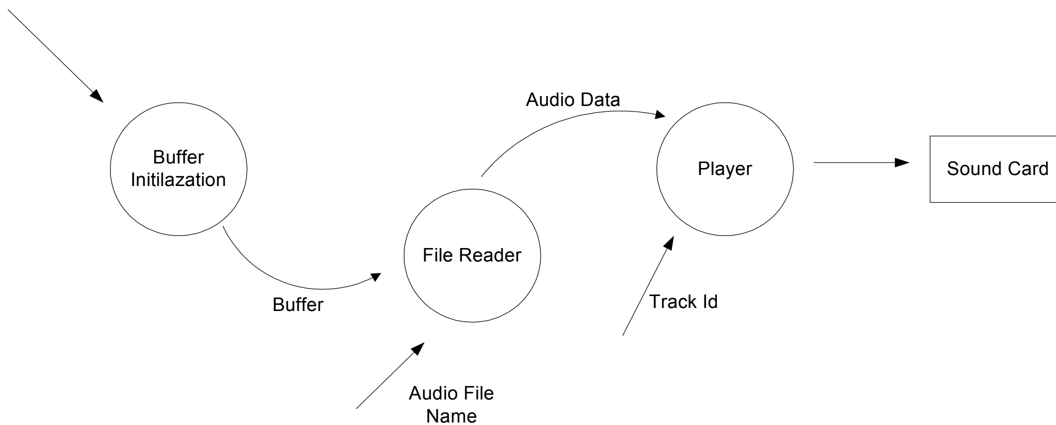
In case of choosing the new game option, the player will face a window asking him/her to specify his/her player. At that moment the player will have two options: either s/he will choose a character from the character template or will s/he customize his/her own character. I the player chooses the latter,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



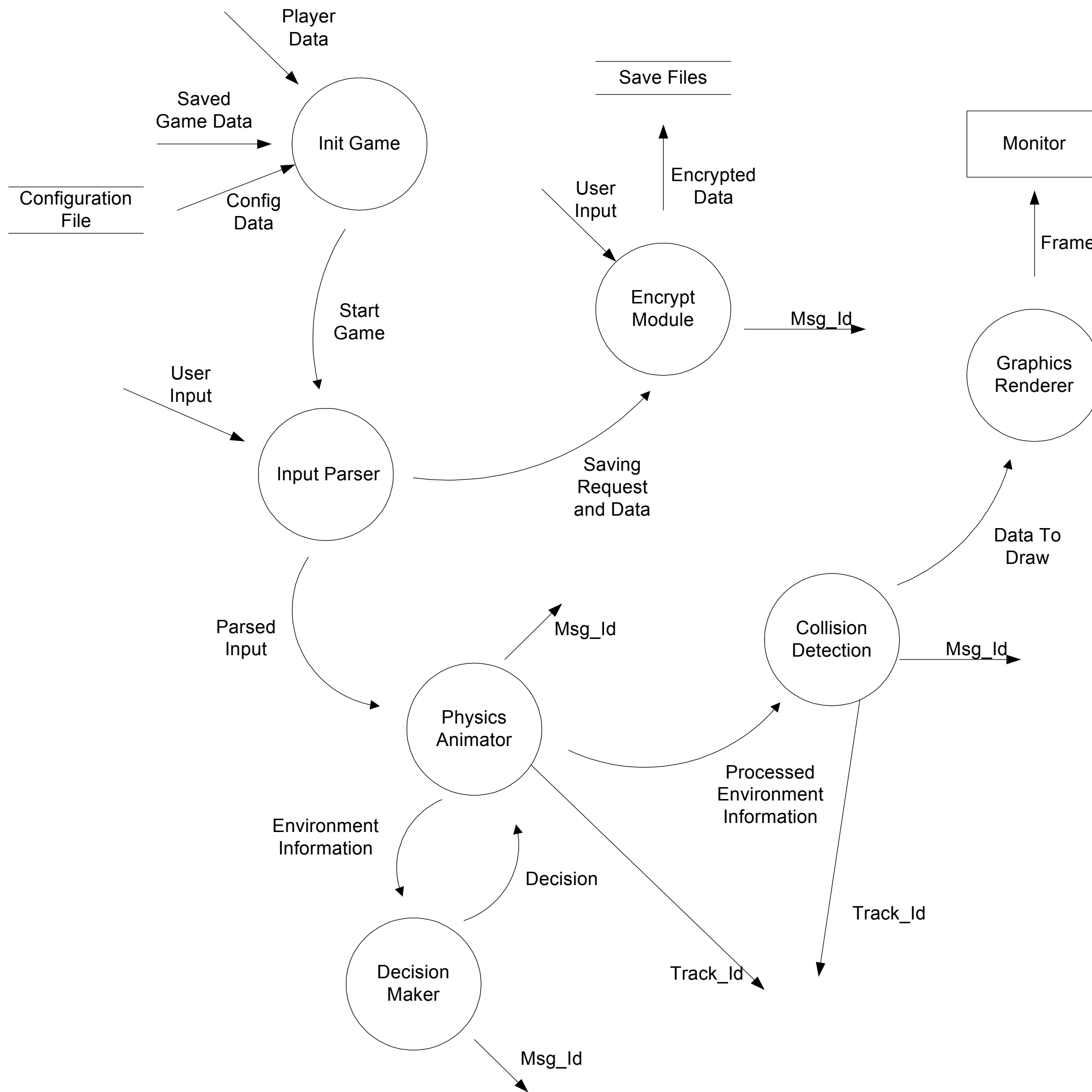
Load game is another option that main menu should handle before starting the game. This selection directs the player to an archive of previously saved files. By clicking on the name of the file the player will be able to decrypt the saved file and continue his/her game.

DFD LEVEL 2 Sound



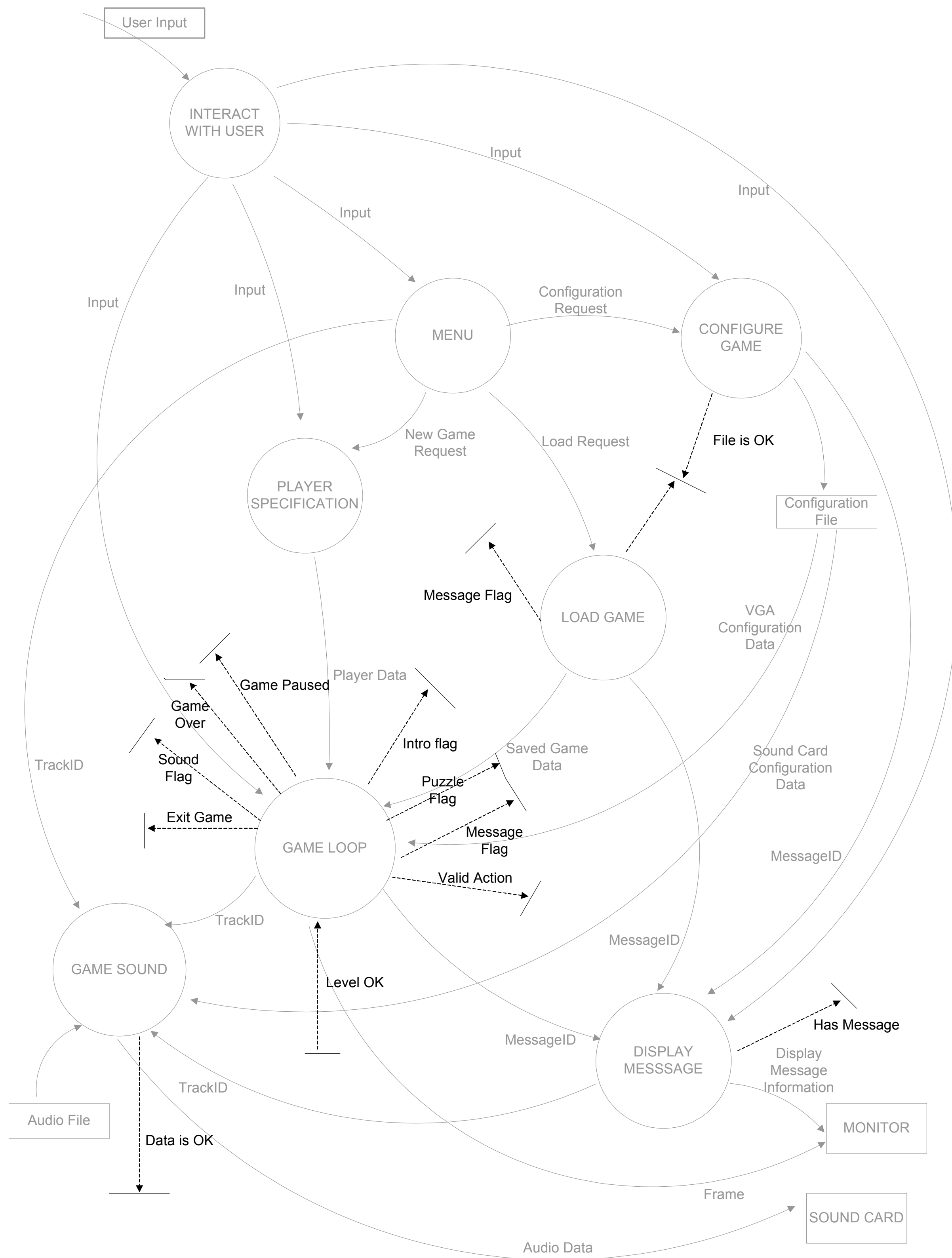
Sound is one of the inevitable parts of a game. The sound processing starts with initialization of the sound buffer. After the initialization the sound files are read from the file system. And finally all ready-to-play sound data is controlled in player module. Player module plays the sound.

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



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.

CONCLUSION

The initial design report is a 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 and other design products given in this document are produced in order to guide MIR during the implementation of the game. The information in this document is to be revised again and the fully detailed descriptions and specifications are going to be given in the detailed design 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 ();
};
```