



**MIDDLE EAST TECHNICAL
UNIVERSITY**



**COMPUTER ENGINEERING
DEPARTMENT**

**FINAL DESIGN
REPORT**

CENG 491



1. INTRODUCTION	4
1.1. PROJECT DESCRIPTION	4
1.2. PROJECT GOALS AND SCOPE	4
1.3. PROJECT FEATURES	4
1.4. DESIGN CONSTRAINTS	5
1.5. PURPOSE OF THE DOCUMENT	5
2. PROJECT REQUIREMENTS	6
2.1. FUNCTIONAL REQUIREMENTS	6
2.1.1. MENU REQUIREMENTS	6
2.1.1.1.General Requirements	6
2.1.1.2.Menu Items Requirements	7
2.1.2. GAME FLOW REQUIREMENTS	8
2.1.2.1.Game Logic Requirements	8
2.1.2.2.Environment	9
2.1.2.3.Player’s Car	9
2.1.2.4.Player – Game Interaction Requirements	9
2.1.3. OPERATIONAL AND STRUCTURAL REQUIREMENTS	10
2.1.3.1.Game Engine	10
2.1.3.2.Graphics	10
2.1.3.3.Sounds	10
2.1.3.2.Artificial Intelligence	11
2.1.3.3.Physics	11
2.1.3.4.Game Data	11
2.1.3.5.Networking	11
2.2. NON-FUNCTIONAL REQUIREMENTS	11
2.2.3. USABILITY AND PLAYABILITY	12
2.2.4. RELIABILITY AND SECURITY	12
2.2.5. PORTABILITY	12
2.3. SOFTWARE REQUIREMENTS	12
2.4. HARDWARE REQUIREMENTS	12
3. GAME SCENARIO	12
3.1. TRAINING	13
3.2. CONNECT AND PLAY WITH OTHER USERS	14
4. INTERFACE DESIGN	15
4.1. MAIN MENU DESIGN	15
4.2. ONLINE GAME MENU DESIGN	16
4.2.1. LOGIN MENU	16
4.2.2. SIGN UP MENU	16
4.2.3. FORGOT PASSWORD MENU	17
4.3. TRAINING GAME MENU DESIGN	17
4.4. OPTIONS MENU DESIGN	18
4.4.1. AUDIO MENU DESIGN	19
4.4.2. CONTROLS MENU DESIGN	19
4.4.3. GRAPHICS MENU DESIGN	20
4.5. CREDITS DESIGN	21
4.6. GAME SCREEN DESIGN	21

4.7. CAR MARKET MENU DESIGN	22
4.8. BUY FUEL MENU DESIGN	23
5. INFRASTRUCTURE	23
5.1. DELTA3D	23
5.2. CAL3D	24
5.3. OPEN DYNAMICS ENGINE	24
5.4. OPENAL	25
5.5. ARTIFICIAL INTELLIGENCE ENGINE	26
5.6. NETWORK	26
5.7. DATABASE	27
6. DETAILED DESIGN	28
6.1. OVERALL GAME STRUCTURE	28
6.2. NETWORK STRUCTURE	32
6.3. PHYSICS ENGINE	36
6.4. PLAYER RELATED STRUCTURE	37
6.5. ENVIRONMENT	38
6.6. CAMERA RELATED STRUCTURE	40
6.7. MENU STRUCTURE	42
6.8. SOUND ENGINE	44
7. ACTIVITY DIAGRAMS	45
8. UML	47
8.1. SEQUENCE DIAGRAM	47
8.2. COLLABORATION DIAGRAM	51
8.3. DATA FLOW DIAGRAM	53
8.3.1. DFD Level 0	53
8.3.2. DFD Level 1	54
8.4. USE CASE DIAGRAM	55
8.5. STATE TRANSITION DIAGRAM	57
9. DATABASE DESIGN	59
10. SYNTAX SPECIFICATION	65
10.1. DATABASE NAMING CONVENTIONS	65
10.2. FILE NAMING CONVENTIONS	66
10.3. CLASSES	66
10.4. METHOD AND FUNCTION DEFINITIONS	66
10.5. GENERAL CODING PRINCIPLES	66
10.6. VARIABLE NAMING CONVENTIONS	66
10.7. COMMENTING	67
11. GANTT CHART	68

1. INTRODUCTION

1.1. PROJECT DESCRIPTION

The main purpose of the project is to develop a massively multiplayer 3D online game which is played via internet through all over the world. The game is formed considering real life conditions so the player can feel the reality and live his own virtual life by gaining money and spending it to necessary things to keep going his life in the game. The title for the game is “The TAXI”. Because initially all users are given a commercial taxi and the game is mainly played with this car.

1.2. PROJECT GOALS AND SCOPE

The goal of this project is to design and implement an interactive, massively multiplayer online 3D game up to a position such that it can be released for commercial purpose. The resulting program will be real-time, realistic with its graphical environment and car physics.

During development of project, we will follow the below methodology.

- ✚ Analyzing the current car racing games
- ✚ Analyzing all the requirements and specifications for the game
- ✚ Design of a game according to the defined criteria
- ✚ Detailed search on the requirements and solutions
- ✚ Implementation and testing of the game
- ✚ Technical support and documentation

1.3. PROJECT FEATURES

Discarding the details, our project will have mainly following features:

- ✚ **3D Computer Graphics:** Since visualization is in vital importance in a 3D game we will make a special effort on graphics design. We will provide advanced graphics by using some high level OpenGL libraries.
- ✚ **Artificial Intelligence:** Our game is not a pure car racing game but also a strategy game, so AI is highly important. We model our artificial intelligence system according to our needs. For example there are artificial customers, traffic and some logical objects. Then we will make a good organization among them.
- ✚ **Updatable 3D Tracks and Environment:** We will provide a plug-in to load the newly designed roads as patches. By this way we can achieve to make user playing the game continuously. Since we will try to do everything dynamic as much as possible it is, we can add new features to our game.
- ✚ **Animation:** We have modelled the scenario according to real life, so for our game to be realistic it will support animation. This can be a person animation, an environment object animation or a car animation which is highly needed to make smooth transitions between positions while the data packages are coming through network.
- ✚ **Game Physics:** Game physics is required to make the effects appear more realistic to the observer, in our case the player. We will model the car physics, collision detection and also the physics of the other objects.

- ✚ **Network Part:** Our game will be played via internet so there will be a server to which all users have to connect to play the game. The server will form the core part of the multiplayer concept. It will have artificial intelligence inside as well as the connection to client part. There will also be a big database which keeps the account information of the registered users and another one holding the online users at that time.

1.4. DESIGN CONSTRAINTS

Software has many constraints as any other real life jobs. Since our project is a time limited project and we are lack of manpower, constraints are more important in our project.

- ✚ **Project Schedule**

The schedule of our project is explained in the Gantt-Chart part. According to this time chart, the project has to be designed and implemented in 8 weeks. Workload must be scattered equivalently because of the demos that we are supposed to make, which are in fact small milestones for us.

- ✚ **Language Constraints**

We will use an object-oriented language to program the game, because the best granularity can be achieved by the help of OO languages. Since the development environment we will use supports C++, we have chosen it.

- ✚ **Data Constraints**

As we process a heavy visualization and store the data for other users we need a huge amount of data storage on users' PC. Server side is not so different from the client side. Since we will hold map info, registered users info, car info, environmental info etc. on server, there will be heavy data traffic between server and clients.

- ✚ **Execution Speed**

To be realistic especially in a massive online game, data transfer should be fast and minimal. Also to obtain a perfect smooth transition between frames we need to supply minimum 24 frames per second refresh rate. Since we need to pass info about the traffic in city to every user, we must optimize the data packages that will be sent. This should be obtained by sending traffic info of only a limited area which is closer to player's car. Near the fact that data transfer should be quick, graphic display must also be optimized. Since we will generate 3D scenes, we must use optimized algorithms so as not to consume CPU very much, which may decrease the performance in advance. While using optimized algorithms in graphics, we should also take care of the trade off between graphics quality and execution speed.

- ✚ **User Interface**

The user interface will be simple and easy to use. Menus should be provided in a logically categorized way. User should be able to play game without need to enter several menus, because many people do not like visiting several menus so as to play a game.

1.5. PURPOSE OF THE DOCUMENT

The goal of this document is to explain the initial design about the game by covering the following parts:

- ✚ Game concept
- ✚ Game scenario
- ✚ Game user interfaces and explanations

- ✚ Game class diagrams
- ✚ Game DFD diagrams (also the other diagrams)
- ✚ Game engines
- ✚ Game network
- ✚ Game database

2. PROJECT REQUIREMENTS

Software requirements is the documentation that completely describes the behavior that is required of the software—before the software is designed, built, and tested. Any coherent and reasonable project must have requirements that define what the project is ultimately supposed to do.

Since we have changed our scenario according to the customer’s request, we need to re-specify the requirements of our project. Despite the fact that our ex-scenario was also related to cars (namely it was a car racing game), there are many important differences which forced us to make a new requirement analysis. So we will state the requirements here again for “The TAXI” project. Since our new scenario is explained in detail explicitly in this report, we will only define the requirements related to the scenario here.

2.1. FUNCTIONAL REQUIREMENTS

Functional requirements define the internal workings of the software: that is, the calculations, technical details, data manipulation and processing and other specific functionality that show how the use cases are to be satisfied.

Functional requirements fall into three groups of types, namely normal requirements, expected requirements and exciting requirements.

(*) Normal Requirements: The objectives and goals that are stated by the instructor and the assistant of the course. These features are expected from a massively multiplayer online game.

(**) Expected Requirements: These requirements are implicit to the product and so fundamental that the customer does not explicitly state them.

(***) Exciting Requirements: These features go beyond the customer’s expectations and prove to be satisfying when present. These features are not guaranteed to be implemented because of the time and manpower constraints.

2.1.1. MENU REQUIREMENTS

These are the requirements related to the menu that is displayed when player enters to the game. This requirement is composed of two subtitles, namely general requirements for the general properties of the menu, and menu items requirements for the functionalities of the menu items.

2.1.1.1. General Requirements

This Main menu is displayed when player enters to the game and when a race is finished and turned back to main menu.

- ✚ (*) This menu must be designed esthetically so as to be attractive for the player. Suitable colors must be selected and elliptic shapes must be used instead of cornered shapes.

- ✚ (*) Player can move on sub-menu titles with keyboard by clicking up-down arrows. And also user can freely move and use mouse for entering sub-menus.
- ✚ (**) When player comes on a sub-menu title, some graphical changes must be done on menu, such as flaming up a new transparent car object or logo on the menu screen. Meanwhile a sound may also be played.
- ✚ (**) A suitable music is played when user stays inside this menu.

2.1.1.2. Menu Items Requirements

✚ Training

- ✓ (*) A new user will need to learn how to play the game. Since there is some specific concepts and roles that are new to people, new players will be able to play some duties one by one.
- ✓ (**) User will be able to select the training that s/he will play.
- ✓ (**) Training menu leads to a training sub-menu in which user selects the training. Trainings will be listed here and their titles are: City Tour, Serving Passenger, Buying Fuel, Maintaining Car, and Visiting Car Market.
- ✓ (**) When a training is selected, game will start in training mode. In this mode player will be informed with a bubble menu about the movements that s/he should do for completing the mission.

✚ Online Game

- ✓ (*) When an internet connection exists, user will be able to connect to the main server, and play online game with people from all over the world.
- ✓ (**) When clicked, a sub menu will appear for carrying out the connection to the server. User will be able to connect to the online game if s/he enters the right username/password combination.
- ✓ (**) If user doesn't have an account, s/he can get a new account by clicking the "Sign Up" button. This button will lead user to a new sub-menu in which s/he will be asked for "Username", "Password", "Password Confirmation", and "E-mail Address". User can form a new account by clicking "OK" button. If an error occurs, user will be informed about the progress.
- ✓ (**) If user has an account but forgot the password, s/he will be able to get his/her password. For this purpose, s/he must click the "Forgot Password" button. And from the coming sub-menu, s/he must click the "Send My Password to My Mail" button. By that way, password of the user will be sent to the e-mail address of the user that s/he had assigned when s/he first sign up.
- ✓ (**) By clicking "Login" button, user will connect to the online game.

✚ Options

- ✓ (*) Users can adjust the general options related to the game.
- ✓ (**) User can change the speed unit that is displayed on the race screen. Choices are km/h and mph from the "Speedometer" section.
- ✓ (**) User can make changes on audio settings. A submenu exists in options menu for audio. In this menu "Menu Music", "Car Radio", and "Sound Effects" sections exist, which are used to increase and decrease the sound of the related kind.
- ✓ (**) Users will be able to visualize and change the game controls (which will only be on keyboard).
- ✓ (**) Users will be able to change the graphics options from the submenu that is opened by clicking the "Graphics" button. This submenu includes "Screen

Resolution”, “World Detail”, “Car Detail”, “Texture Detail” and “Viewing Distance”.

- ✓ (**) Screen resolution will be able to set to either 800x600 or 1024x768 which are suitable for most of the screen cards used widely today.
- ✓ (**) User will be able to adjust world detail, car detail and texture details to high, medium or low. Best quality will be gained at “high” level, but due to the hardware constraints, user may want to set these to medium or low.
- ✓ (**) Viewing distance will be able to be set by the user as “far” and “near”. This option will define how far distance will be rendered. For a more realistic feeling, this must be set to far. But it may be set to near because of hardware constraints of the user.

Credits

- ✓ (**) This menu will list the names of the members of the project team, related information about our company, and names of the individuals and corporations that we have taken help from.
- ✓ (**) This list will slide slowly on the screen while music is played at the background.

Quit

- ✓ (*) User can return to the operating system by clicking “Quit”.

2.1.2. GAME FLOW REQUIREMENTS

These are the requirements that are related to the game play. We examined game flow requirements in four subtitles: Game logic requirements, Environment requirements, Car requirements, and player-game requirements.

2.1.2.1. Game Logic Requirements

-  (*) Since this is a game we are producing, players should not get bored while playing, besides game should make them play more as they play. We will provide this fact by exciting the avidity of human to money and success.
-  (**) Our game is mainly a massively multiplayer online game. So we provide user to play with other players all over the world. User also will be able to play offline game, but this will be limited game just for training of the user.
-  (**) User will earn money by serving customers to destinations given and when selling his/her car, and lose money when buying fuel, paying traffic taxes, maintaining car, buying a new car.
-  (**) When user first starts game, s/he will be a free taxi driver, i.e. s/he will not be member of any cabstand. But as player earns money, s/he will be able to enroll to a cabstand in exchange for a reasonable amount of money. In short player and the owner of the cabstand will obtain a contract. (Owner of a cabstand is the server as default)
-  (***) Player may buy shares of a cabstand, a fuel station, a car maintenance shop. By that way user will be able to earn more money. But there will be trade-offs. Shares of the owner should gain or lose value, thus making player feel more excitement.
-  (***) Player can buy traffic insurance.
-  (***) Crashing will give damage to both cars, responsible player from the crash will pay the money of the counter side, where as his/her car will be damaged and this will make him/her lose customer unless s/he repair his/her car. If the player has traffic insurance, damage will be supplied by the insurance company.

2.1.2.2. Environment

✚ Terrain, Buildings and Objects

- ✓ (*) Game must have a terrain which gives the feeling of realism to player.
- ✓ (**) Terrain will consist of sky, ground, road, buildings, skyscrapers, trees, and some extra objects such as statue, street lamps, dustbin etc.
- ✓ (**) Car is not permitted to get out of the road. This will either be satisfied by putting protective embankment on borders of the road, or by just preventing the car from getting out of the road as if it can not climb over pavements.
- ✓ (**) There will be people on road, some of which will be the customers waiting for a taxi.
- ✓ (***) Some external objects can be drawn such as a passing helicopter, plane, ship etc.

✚ Cars

- ✓ (*) There will be other players' taxis and city traffic on the game environment.
- ✓ (**) Cars can crash. Crashing will give damage to both cars.
- ✓ (**) Wheels of the cars will turn left and right according to the direction given by the player.
- ✓ (***) Smoke can arise from the car in serious crashes.
- ✓ (***) Sparks can rise from car when it crashes with a barrier.

2.1.2.3. Player's Car

✚ It has movement capabilities

- ✓ (*) Go forward, go backward.
- ✓ (*) Turn left, turn right.
- ✓ (*) Stop suddenly by using handbrake.

✚ It has external effects

- ✓ (**) It can sound the horn.
- ✓ (**) When stepped on break, back break lights are turned on.
- ✓ (***) When stepped on break suddenly, car tires will mark the road.

2.1.2.4. Player – Game Interaction Requirements

✚ (*) Player controls the car with the keyboard.

✚ (*) Game uses monitor and speakers to send output to player.

✚ (**) Player can view the game from different outside camera views, namely they are near and far views.

✚ (**) Player can view the game from inside the car, i.e. view the steering wheel, speedometer, etc. as in real car.

✚ (**) Player can see the road map, back of the car from mirror, fuel status, speed, motor spin and gear of the car. All these are located in suitable positions on screen.

✚ (**) Player will be informed about important events such as obligation to pay tax, or need to serve a customer from the cabstand etc. via a message bubble located in a suitable position of the screen.

✚ (**) Player will be able to monitor the money that s/he has.

✚ (**) Player will be able to monitor state of the career of himself/herself (i.e. if player satisfies the customer, his/her career will be successful, and it will become unsuccessful otherwise).

✚ (**) Player will be able to monitor the status of car damage and also the status of the driving license that s/he has. All these four status viewing sections(money, career, car

damage, driving license) will be located altogether at upper right corner of the screen, and user will be able to show/hide this menu with just one click and also with a keyboard shortcut.

- ✚ (***) Player will hear motor sound, changing according to the motor spin.
- ✚ (***) Player will take a shaking view when a crash occurs and camera view is from inside of the car.
- ✚ (***) Player can pause the game by clicking only if s/he is stopped the car near the road, by clicking ESC key. Then ESC menu will appear. Pressing ESC while car is moving is not allowed since this may affect other players. User may leave the game via this menu.
- ✚ (***) Player can adjust the radio settings such as changing channel, decreasing volume etc. from the sub-menu button located at the bottom of the screen.
- ✚ (*) Player can exit from race, exit from game directly from the escape menu of the race.
- ✚ (*) Player can resume the game from the escape menu of the race.
- ✚ (***) Player can see the statistics of his/her car via the submenu that is opened when “My Car” button is clicked which is located at the bottom of the screen.
- ✚ (***) Player can follow status of his/her extra jobs such as the state of the shares of the fuel station, cabstand etc. via the submenu which is opened when “Extra Jobs” button is clicked that is located at the bottom of the screen.

2.1.3. OPERATIONAL AND STRUCTURAL REQUIREMENTS

These requirements form the basics of the game. These objects affect the reality of the game, thus give user enthusiasm to play more and more. We examined these requirements in six subtitles: Game engine, graphics, sound, AI, physics, game data, networking.

2.1.3.1. Game Engine

- ✚ (*) The core functionality typically provided by a game engine includes a rendering engine for 2D or 3D graphics, a physics engine or collision detection, sound, animation, artificial intelligence, networking, and a scene graph.
- ✚ (*) Combines the subcomponents of the game.

2.1.3.2. Graphics

- ✚ (*) Renders the scene in 3d mode.
- ✚ (***) Maps the textures on objects.
- ✚ (***) We need to apply some tricks to improve the performance of the game without disturbing the user. For instance, instead of using many lights, we should use textures that have the light effect on themselves and so on.
- ✚ (***) Draws the objects that are far to camera with low quality than that are near.

2.1.3.3. Sounds

- ✚ (***) Menu clicks; passing from menu to menu actions produces sound.
- ✚ (***) In main menu and escape menu, music is played at the background.
- ✚ (***) Motor sound is played during race, changing according to motor cycle.
- ✚ (***) Collision sound is played on car crashes.
- ✚ (***) Cars can horn; this sound will also be played.
- ✚ (***) Music is played during game by the help of car radio component.

2.1.3.4. Artificial Intelligence

- ✚ (*) There will be city traffic which is controlled by the AI.
- ✚ (**) Server will distribute customers to the free taxi owners and also to the cabstands.
- ✚ (**) Computer will judge the guilty side on crashes, and give punishment accordingly.

2.1.3.5. Physics

- ✚ (*) Car must obey to the real world physics rules.
- ✚ (***) Car should wave by help of shock absorber's movement.

2.1.3.6. Game Data

- ✚ City
 - ✓ (*) Terrain
 - ✓ (*) Objects
 - ✓ (*) Buildings
- ✚ Sounds
 - ✓ (*) Menu clicks
 - ✓ (**) Speeches
 - ✓ (**) Music
 - ✓ (**) Motor/Horn/Collision/Crash
- ✚ Player Info
 - ✓ (*) Account info
 - ✓ (**) Properties of the car, status of money, career, driving license, fuel etc.
 - ✓ (**) Free taxi or member of a cabstand
 - ✓ (***) Extra job info
- ✚ (**) Game settings info. General game settings must be saved, and changes that user has made must be permanent (i.e. not change to default when game is started each time from the OS.)
- ✚ (*) Textures
- ✚ (**) Car features/photos/statistics
- ✚ (*) Car models(3D objects)
- ✚ (*) Images
- ✚ (*) Settings

2.1.3.7. Networking

- ✚ (*) Players will connect to a network game via internet
- ✚ (**) Server will collect the coordinates and statistics of each player, combine data, manipulate collisions, add city traffic into account, then serve the data to every client. This process will be repeated periodically. For increasing performance of network, we need to send as limited information as possible. This may be gained by sending only the coordinates of the cars that are nearer to the player.
- ✚ (**) Server will be able to apply AI to city traffic.

2.2. NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements support functional requirements, by imposing constraints on the design or implementation (such as performance requirements, security, quality standards, or design constraints).

2.2.1. USABILITY AND PLAYABILITY

Since games are for fun, it must be easy to learn and use so as not to be boring for the user. So we thought that menus must be designed as understandable as possible. Also we planned to follow the traditional approaches in many designing menus, because players will compare our game to car racing games that s/he has played earlier. So in design we inspired by some other games on market.

We will add training sections for the game, so as to educate new players. By that way user will not be unfamiliar to game concepts when s/he is playing with his/her account on the internet. In training sections, basic concepts of the game will be introduced.

Near usability, playability is also very essential. User must not get bored when riding car. Game will give the feeling of competition and use the greed of human for money and success. Since people do not get bored when running after the greed, we will benefit from that side of humans. Game will not be too hard or too easy. Namely becoming rich won't be easy and it will require much work. But also user will not get poor easily in game unless s/he doesn't work totally (i.e. computer will give chances to player to serve customers). Control of the car will be efficient and easy as it is in most of the car related games that are on market today.

2.2.2. RELIABILITY AND SECURITY

Game must be debugged very carefully. Product must be checked in many aspects so as not to cause any problem to the end user. Since our game is also an online playable game, security is very important for us. Protocols that we use must be integrated to system very carefully. There shouldn't be any open backdoor left on computers of players during and after plays.

2.2.3. PORTABILITY

Since time is a big concern for us on this project, we will not be able to make a platform independent game. Because this will increase our work from two to three, which will make us fall behind our targets. Instead we will make our game platform dependent. And since Windows has dominance on market, and project group is more familiar with Windows concepts, we will make our game work well with Microsoft Windows.

2.3. SOFTWARE REQUIREMENTS

Windows operating system is needed for running our game. This is essential because OS provides the connection between programs and low level hardware.

2.4. HARDWARE REQUIREMENTS

- + A PC with Pentium 4 class CPU
- + Internet connection via modem or wireless or Ethernet card
- + Sound card

3. GAME SCENARIO

In this part the scenario will be explained in details.

Although this is an online game it has to be installed to all users computer and then played with other users by activating the online account.

When user starts the game first time, he is asked to create an account only by entering a new username, and then the game is loaded for that user. If he has already an account, he

selects his username and the game is configured for that user. Then there available two options to user:

- ✚ Training or,
- ✚ Connect and play with other users

3.1. TRAINING

This part is designed to help user to learn how to play the game by visual and audio interaction with user. When user selects training, he is redirected to the training menu. There are some options which user can choose considering which part he wants to play or learn. For example some possible options are:

- ✚ Explore the city
- ✚ Learn how to get fuel
- ✚ Learn how to sell and buy car
- ✚ Learn how to maintain and customize the car
- ✚ Learn how to a member of a cabstand
- ✚ Learn the tips of making money
- ✚ Learn how to communicate with other online users

If user chooses to explore the city, he just goes around the city and learns the roads, streets and sees where some important buildings are which are highlighted on the map may be on the right of the screen.

Purpose of the second option is to make user to try to get fuel for his car by going to petrol station. There will be visual aids and sound interaction with user in this part and the following parts which give user some commands to do. These are small missions to help user to get the points correctly. When user enters the petrol station, a menu appears on the screen where user can decide the type and amount of the fuel he wants to take. Since different type of cars uses different type of fuels, there appears only the suitable ones for that car. And also the user can see the cost of the choice he has decided.

In the third training part, user is taught to sell a car as well as to buy a car from the autobazaar. User is commanded to run the car to the bazaar and enter the menu. On the menu, there are two options: sell the existing car or buy a new one. When sell option is selected, the value and the details of the user's car is displayed on the screen so the user can decide whether to sell at that cost. The value of each car is determined according some realistic measurements such as the damage it has, the production year, the maintenance rate, the motoring fine and etc. So when user customizes his car or pays the fines, he can sell his car at a higher value. On the other hand, user can also buy a car by just clicking the buy button. At this stage, the properties of the available cars are shown on the screen such as acceleration, top speed, handling or other extra ones we may add, and user can go around them via keyboard or mouse. Also the cash the user has and the cost of the car also displayed so the user can see whether he can afford it.

At maintenance part, user is given the order to drive the car to the maintenance shop or building and enter it. In this menu user is informed about the current parts of his car and also the parts that he can buy for his car, the costs of the parts are displayed on the screen. There is another option in the menu by which user can make his car get rid of from the damage as much as he pays, so he can make his car healthy which is important to attract the customers.

At being a member of a cabstand stage, user learns how to register himself to a cabstand and the benefits of it. While driving the car, user can always see the available cabstands and the short descriptions about them. The description includes some small information such as the place of the cabstand, the popularity, the profit rate, how many members it has as well as the money it requires to be a member of it. We have not decided yet which informations will be displayed during free roam and while at that cabstand. But it is certain that to be a member user has to go to that cabstand and accept the contract. The contract includes some conditions about the membership such as the cost, the leaving compensation and some other restrictions. The benefits of the cabstand will be explained in the following sections.

In training, user also can learn some basics of how to make money which is required for the continuity of the career. If user loses all of his money, he has to reset his career. In this part, user is given some tips to spend less money or gain more. Some examples are “select the shortest path while going somewhere, so the customer may give much money in consideration of the time and you use less fuel”, “obey the traffic rules so no punishment is given” and other else.

User also will be given the details of how to communicate with other users and how to play together or against some other players. All the messaging is done via chat which is provided in the game. User can send message during the game without returning to the main menu but with some restrictions. He can see his messages and send new ones after stopping the car at the side of the road. Since our game is real life adapted, user can also guess what to do, when and where to do.

3.2.CONNECT AND PLAY WITH OTHER USERS

When user wants to play online, he is asked for the username and password to connect to the server. Then he resumes his career. He can see the career status on the screen, the map, the status of the car and some extra things we provide. Playing online is not different from the training part basically. Unlike from the training part, user has to be more careful about his money, because he can end his career. So he has to fulfill even the simple rules to not to lose money. The main goal of the game is make user to learn and apply the real life conditions to this virtual environment and somehow feel the difficulty of the real life. He has to obey the traffic rules, make the customer happy with extra things, pay the taxes regularly, attract the customer by either upgrading his car, his fame or providing different things that other taxis don't have, give less damage to car and etc.

User also has to be a member of a cabstand to make big money, because the cabstands have both fixed and more customers. By selecting and being a member of a cabstand which has high profit rate or not so many members, s/he can double his wealth.

After having enough money some users may come together and establish their own cabstand and try to increase its fame among customers. This provides them to get the all profit and hire whoever they want. The members of the new cabstand have to select a manager. But the cabstand has to be managed very well to compete with the other ones. But due to the time constraints, this part is not guaranteed to be manipulated.

There are also some extra features that we have not decided yet but can be good to increase competition among users. The user can buy share from some buildings such as petrol station, maintenance shop or others. He does not have to buy all of the shares. The more he

pays, the more the shares and the profit he gets from that one. Manager is the one having the highest share on that building. The manager is responsible for the prices, taxes, penalties and etc. If he follows a good strategy on managing, he can double the profit. The more the money, the easier the life.

4. INTERFACE DESIGN

Design of the user interface is in high importance, because interface is the vitrine of the product with respect to the user. A shop with well designed vitrine attracts more customers; same is valid for a software product. Furthermore this is a vital issue in computer games, because players are seeking for better graphics quality.

In this part we will give information about the game menus. Since we give detailed information about menus and their functionalities, we will not go into detail in this part. But just give brief information about the menus and menu transitions.

4.1. MAIN MENU DESIGN



When game starts, main menu screen greets the user. This screen is designed in a way so as to mirror the soul of the game in one sight.

User can start an online game from online game button, a training game by clicking training button. For displaying and adjusting the game settings, options menu button must be clicked. Then options menus will appear. For viewing the development team and info about game preparation, credits menu button must be pressed. For exiting from the game to the operating system, quit button must be pressed.

We thought that full screen changes will not be very cute for starting a game. Because of this reason, we designed most of the submenus to appear not full screen but inside screen. Only options and credits menus are full screen submenus. Submenus related to game play are all small submenus, which give user the feeling of quickness for playing the game. This is essential because many people don't like games because they open late.

4.2. ONLINE GAME MENU DESIGN

When user clicks the online game button first login menu appears.

4.2.1. LOGIN MENU



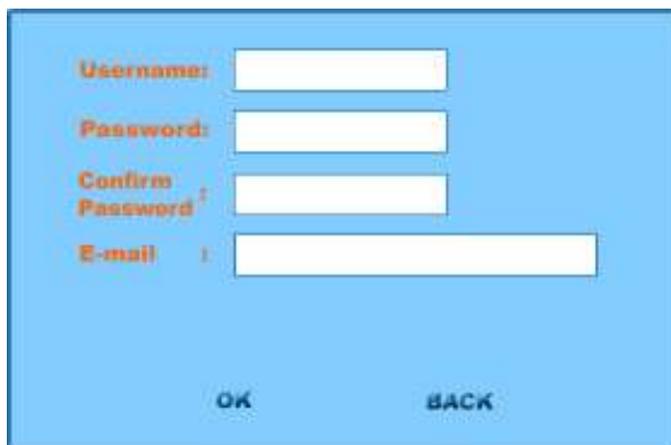
The screenshot shows a login menu with a light blue background. It features two input fields: "Username:" and "Password:". Below the input fields, there are four buttons: "LOGIN", "SIGN UP", "FORGOT PASSWORD", and "BACK".

User can login to the server and thus to the game by typing the username and password to the related fields and then pressing login. If information entered are incorrect, user will be informed about the situation.

If user does not have an account, s/he can get a new account from the menu that appears when sign up button is pressed.

If user has an account but forgot the password, s/he can take his/her password via forgot password menu.

4.2.2. SIGN UP MENU



The screenshot shows a sign up menu with a light blue background. It features four input fields: "Username:", "Password:", "Confirm Password", and "E-mail". Below the input fields, there are two buttons: "OK" and "BACK".

User can obtain a new account by entering a username, a password, then confirm password, and e-mail address. If username that the user wanted to obtain has taken before, user will be informed, and requested to enter a new username. If password confirmation is not

satisfied, user will be requested to re-confirm password. When Ok button is pressed and no error occurred, account will be constituted.

4.2.3. FORGOT PASSWORD MENU



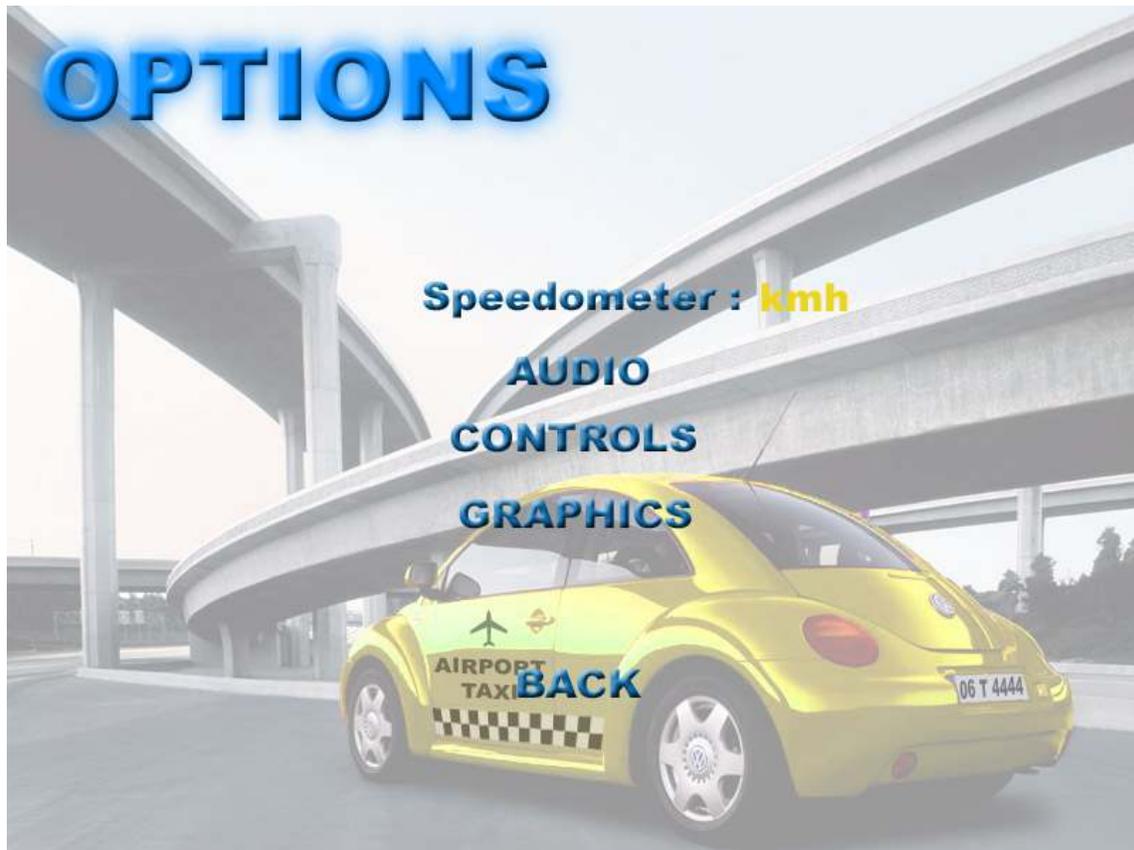
If user has an account but forgot the password, s/he can get the password by clicking send password to e-mail button. When clicked, password of the user is sent to mail address of the user that s/he assigned when taking the account.

4.3. TRAINING GAME MENU DESIGN



User can play different training games. For making a free city tour in order to learn the streets, city tour training is designed. For learning how to serve a passenger, how to buy fuel, how to maintain the car in case of a car crash or when a modification is wanted to be done, how to visit the car market so as to sell/buy car, related buttons will be pressed and game will start immediately.

4.4. OPTIONS MENU DESIGN

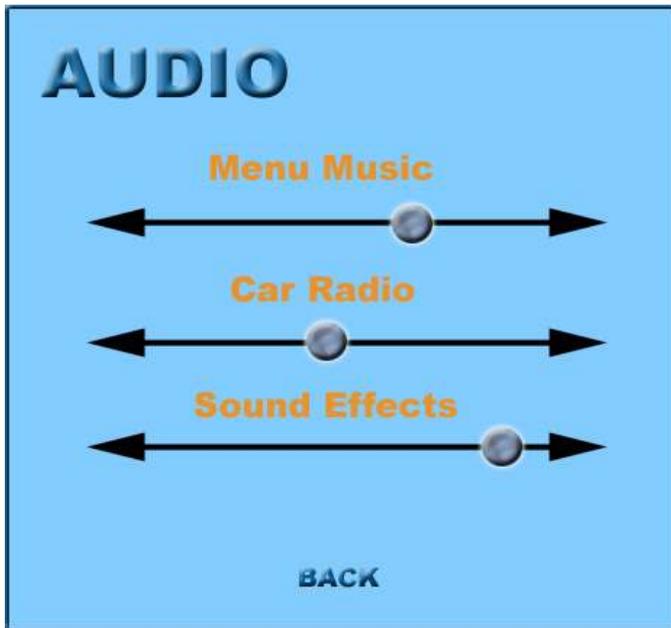


Settings related to game play are adjusted from the submenus of options menu.

Type of the speedometer is may be set to either kmh (kilometer per hour) or mph (mile per hour) from speedometer button, either by clicking mouse button or hitting left/right arrows of the keyboard.

Audio settings, keyboard control settings and graphics settings may be done from the submenu that appears when related button is clicked.

4.4.1. AUDIO MENU DESIGN



Volume of the tune that is played when menus are active can be adjusted from “Menu Music” field. During game, user will be able to listen to music, as it is the case in real life. We named this concept as car radio, and volume of the radio may be adjusted from this menu. Adjusting radio volume is also possible from inside the game. Volume of the sound effects can also be adjusted from that menu.

4.4.2. CONTROLS MENU DESIGN



Our car is controlled by the keyboard. And the controls for the keyboard can be adjusted from “Controls” menu. For adjusting a control, user will first click the control that

s/he wants to change with mouse, then hit the key that s/he wants that action to be set. After setting, when back button is clicked, program will turn to the main options menu by saving last key configurations. Also user can load default key values for this menu, by clicking “Set Default Keys” button.

4.4.3. GRAPHICS MENU DESIGN



User can adjust screen resolution either to 800 x 600 or 1024 x 768, by either clicking with mouse or by hitting left/right arrow. Texture quality may be adjusted to low, medium, or high. World detail and car detail may also be set to low, medium or high. View distance may be set to either near or far.

4.5. CREDITS DESIGN



This screen shows the names of the development team, and the people that we have taken help from.

4.6. GAME SCREEN DESIGN



This is the game screen. Player will be able to view the game from three different camera views: Inside the car, outside near and outside far. View seen above is the outside far view.

Up left corner of the screen shows the values of the fuel, speed, motor spin and gear.

Up right corner of the screen shows the status of the player: money that the player has, career of the player, damage of the car, and the driving license point. Customer satisfaction will affect the career of the player, at the beginning career will be 50 %, and change according to customer satisfaction. Car damage display will start with 0 % at the beginning, denoting no damage. When a crash occurs, car gets damaged and this info is displayed in “Car Damage” section. When user first starts the game with a new account, s/he has 100 points for driving license. As player commits traffic crime, points will be reduced from his/her driving license.

Down right corner of the screen shows the city map. Location of the car and the location that the customer must be served will be indicated on the map.

Down left corner of the screen has the in game menu. Extra jobs button makes it possible for the player to watch the state of his/her extra job. This part of the game is stated in excited requirements. In short this button will give user chance to view the state of the shares of a fuel station, or a cabstand etc. My Car button makes user learn about the statistics of his/her car. Player can change the tune, open-close radio; adjust radio volume by the menu that is opened by clicking Radio button. From the menu that is opened when Menu button is clicked, player may exit the game.

4.7. CAR MARKET MENU DESIGN



In the game, user will have interaction the environment. For example so as to take a customer from the road, player must stop the taxi in a suitable area in front of the player. With the same logic, for entering car market, player must stop the taxi in the indicated area in front of the car market. Thus first user must go to the car market, and then park the car to a suitable place, and then this menu will appear.

By the help of this menu, user can sell his/her car to the server (not to a real person yet, this is an exciting requirement) by clicking “Sell Car” button. When sold, cost of the car will be added to cash of the player.

User can buy a new car, but a user is not allowed to have two cars (this is also an exciting requirement). So, if user has a car first s/he must sell it. When doesn't have a car, user can buy a new car by first selecting the car that s/he wants. If cost of the car is less than the cash of the player, s/he will be able to buy this car. By clicking “Buy Car” button, user will own the selected car. This menu shows the properties of the car; acceleration, top speed, breaking, handling, fuel consumed per km, type of fuel that car uses. Also picture of the car, name and production year of the car is displayed on this menu.

4.8. BUY FUEL MENU DESIGN



The image shows a light blue rectangular menu with a thin black border. At the top, it displays "Liter Cost : 1.3 \$". Below this, the text "Fill" is followed by a white rectangular input field, and then "liters of Fuel". Underneath, it says "Total Cost: \$". At the bottom, there are two buttons: "BUY" on the left and "BACK" on the right, both in blue capital letters.

When user enters to a filling station and park the taxi to the pre-indicated place, this menu will appear. User will enter the amount of fuel that s/he wants to buy, then click “Buy” button.

5. INFRASTRUCTURE

5.1. DELTA3D

Delta3D is a full-function game engine appropriate for a wide variety of modeling & simulation applications (training, education, visualizations, and entertainment).

The Delta3D Engine

Delta3D is an Open Source engine which can be used for games, simulations, or other graphical applications. Its modular design integrates other well-known Open Source projects such as Open Scene Graph, Open Dynamics Engine, Character Animation Library, and OpenAL. Rather than bury the underlying modules, Delta3D just integrates them together in an easy-to-use API -- always allowing access to the important underlying components. This provides a high-level API while still allowing the end user the optional, low-level functionality. Delta3D renders using OpenGL and imports a whole list of diverse file formats (.flt, .3ds, .obj, etc.).

Supported Platforms

Currently, Delta3D is developed and tested on Windows XP using Microsoft Visual Studio .NET (7.1) and Fedora Core 4 using gcc 4.0.0. All the underlying dependencies are cross-platform as well, so just about any platform should be compatible with some modifications to the source (only Win32 and Linux are supported at the moment).

Key Features of the Delta3D Game Engine

- ✚ High level C++ API with Python bindings
- ✚ STAGE: A powerful editor for building 3D worlds and scripting sequences.
- ✚ Supported on Windows XP and Linux.
- ✚ Particle systems
- ✚ 3D audio rendering
- ✚ Supports numerous standard 3D file formats (.3ds, .flt, .osg, .obj, etc.)
- ✚ Game-style client/server networking
- ✚ Extensible terrain architecture
- ✚ Terrain generation directly from source DTED
- ✚ Tool suite: STAGE, Particle System Editor, 3D Model Viewer
- ✚ 3D content exporters for Max 7, Maya 6, and Blender 2.37

5.2. CAL3D

Cal3D is a skeletal based 3D character animation library written in C++ in a way that is both platform-independent and graphics API-independent. Cal3D's essentials can be divided into 2 parts: the C++ library and the exporter. The exporter is what we would use to take our characters (built in a 3D modeling package) and create the Cal3D-format files that the library knows how to load.

The exporters are actually plug-ins for 3D modeling packages. This allows 3D artists to use the modeling tools that they're already comfortable with. The C++ library is what we would actually use in our application, whether it's a game or a VR application. The library provides methods to load our exported files, build characters, run animations, and access the data necessary to render them with 3D graphics.

Cal3D doesn't do graphics, so we are responsible for making a bridge between Cal3D and whatever graphics API we want to use. This includes things like loading textures, handling materials, and rendering the models to the screen. Making a decent character will require the use of a modeling package like 3D Studio MAX. Since we are thinking to use 3D MAX, if we use animation in our game -i.e for pedestrian- we will possibly use this library. Delta 3D has also some animation libraries but Cal3D make animations more effectively.

5.3. OPEN DYNAMICS ENGINE

ODE is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools. So we are going to use this engine for instance our car physics. Basic features of ODE include :

- ✚ World
 - Gravity and time integration
 - Can be on its own thread

- ✚ Space
 - Optimizing collision detection
- ✚ Body
 - Physical object
 - Size, mass and position
- ✚ Geometry
 - Collision detection
 - Rendering

There are 6 types of geometries: Sphere, box, cylinder, ray, triangle mesh, and user defined. For our car, body is a box for example.

5.4. OPENAL

Audio features in games are indispensable. It takes users at the center of it. Therefore the attractiveness of the game depends on the realism of the environment and the flow of action. Sound effects are one of the major features that affect the realism of the atmosphere. Besides, playing music during the game increases the entertaining factor. For this purpose the software must be able to play audio files.

Furthermore, playing videos during the game not only provides realism but also helps attracting the players to the flow of the story. As a design issue, we are going to implement a multimedia module that answers these needs. The module consists of a static class which stores and manages the audio files or video file that is played. We will define methods to play menu music (played only in menus), background music (played during the game-play), and play various sound effects (sounds that are caused by the elements in the game - car, objects and collisions).

For these capabilities we will make use of free libraries such as DevLib and OpenAL to handle these issues easily and efficiently. OpenAL is a cross-platform 3D audio API appropriate for use with gaming applications and many other types of audio applications. The library models a collection of audio sources moving in a 3D space that are heard by a single listener somewhere in that space. The basic OpenAL objects are:

- ✚ a Listener
- ✚ a Source
- ✚ a Buffer

There can be a large number of Buffers, which contain audio data. Each buffer can be attached to one or more Sources, which represent points in 3D space which are emitting audio. There is always one Listener object (per audio context), which represents the position where the sources are heard -- rendering is done from the perspective of the Listener.

On the other DevLib is a relatively new but much more talented library which also provides classes for playing both music files and video files. Basically DevLib is an object-oriented framework written in C++. The main advantages of using DevLib is that it provides user friendly abstraction of heavily used resources such as fonts, images, 3D meshes, files, xml, zip-archives, sounds, videos. DevLib library itself makes use of DevIL, FreeType 2, LUA, ODE, libjpeg, libmpeg2, libpng, TinyXML, unzip, ZLib, SDL, DirectX 9, FMOD, GLEW and STL libraries to fulfill its requirements. One may save/load images, export meshes from LightWave, 3D Studio MAX and Maya, play sound files, show videos, execute LUA scripts and manage consoles which allow updating the value of user-specified keys at run-time. DevLib supports OpenGL and DIRECTX as render system, PNG, JPG, BMP, TGA

file extensions in image management DevLib is fully compatible with Microsoft's Visual C++ 2003.

5.5. ARTIFICIAL INTELLIGENCE ENGINE

As computer games develop, the demand for more sophisticated computer controls increases. At this point AI gains more importance day by day. In an artificial intelligence engine there should be some requirements. An effective artificial intelligence engine should support agents that are:

- + Reactive
- + Context Specific
- + Flexible
- + Realistic
- + Easy to Develop

Reactive agents are methods that compute just one next action in every instant based on the current context. Context specific agents ensure that their actions are consistent with past sensor information and the agent's past actions. Flexible agents have a choice of high level tactics with which to achieve current goals and a choice of lower level behaviors with which to implement current tactics. Realistic agents behave like humans. More specifically, they have the same strengths as human players as well as the same weaknesses. Finally, an artificial intelligence engine can make agent development easier by using a knowledge representation that is easy to program and by reusing knowledge as much as possible. Our game will also provide some AI features. For example in the streets there will be cars and pedestrians that are produced by server. There are some artificial intelligence engines but Delta3D also has a library called dtAI which provides this capability.

5.6. NETWORK

In a head-to-head death match multiplayer game, players form ad-hoc and short-lived sessions. Any client can be selected as a server and the speed is much higher than an online game. There is no persistent state in these games.

However in a MMOG there must be a server other than the clients computers and connection should be persistent. So the infrastructure in server should be handled in a good way.

Since we are going to implement a MMOG, we will consider 3 main things while developing our game in terms of networking such as: Timing, Identification and Determinism.

+ **Timing:**

We need to be able to synchronize time to be able to identify when things happen in the game and when messages arrive. The most complex timing problem is a real-time system, for obvious reasons.

+ **Identification:**

Identification is being able to direct the correct data to the correct area in our game. We do not need to worry about routing with most networking implementations, but we need to come up with a way to uniquely identify each machine and have all other connected machines aware of that identification and "who they're talking to". We also need in-game identification to make sure we're sending info to/from the right objects.

+ **Determinism:**

Determinism is the predictability of our game. We only need to send as much info across the network as we need to ensure matching game states across the board. So game

will only require small amounts of information (ie: the exact choices the user made, because these are the only points that difference can exist in the system).

We are going to implement a multi-threaded server with (possibly) UDP network protocol as our game can be regarded as a real-time system. Delta3D also supports and use some network libraries, but it supports only 256 players concurrently. So we will implement our own network infrastructure using C++ socket programming in order to supply players more than 256. Our server will be the heart of the game. It will keep track of all the objects, players, events and ensure that each client gets the information it needs, as well as processing incoming data from the clients.

5.7. DATABASE CONCEPT

Online games must be refreshed with new content constantly to keep the game alive and exciting for the players. The easiest and most convenient method for this is simply downloading the new content onto the player's computer via the patching process. Since we are planning to develop an online game, it is a must for us to use a database. For database there are some choices for us – i.e. MSSQL, MYSQL, ORACLE and etc...

In our database we will store some data. These data will highly affect our data transfer rate and game performance. Some data will include:

- ✚ User accounts
- ✚ User statistics
- ✚ Game data

For example regarding to user account we can store user's name, password, mail attributes. As to user statistics, we keep users total money, cars features such as model, damage level. Game data include:

- ✚ Terrain features
- ✚ Sound effects
- ✚ Menu options
- ✚ Map, cars, city data

Of course accessing database and retrieving data require cost and decreases the speed. So some information can be manipulated without storing data in the database. For example objects that are not changeable can be stored in a file. Since server capacity is restricted, there will be some restrictions. For example if a user do not participate to game throughout 2 months, his account will be wiped in the database. Once the server is initiated, each user, to join the virtual world, uses a client to connect to the server by notifying the central database regarding the user's login information. If a new user logs in, the central database will register a new account for the user, otherwise, the user will be checked for authorization. The state information of each user should be stored on the server, where the user first created his account.

MMOGs with a client/server architecture have a single huge “game state” which is the game world's data usually held exclusively by the server. This data is extremely important to the game and to access; changes to the data are strictly monitored by the server software.

6. DETAILED DESIGN

6.1. OVERALL GAME STRUCTURE

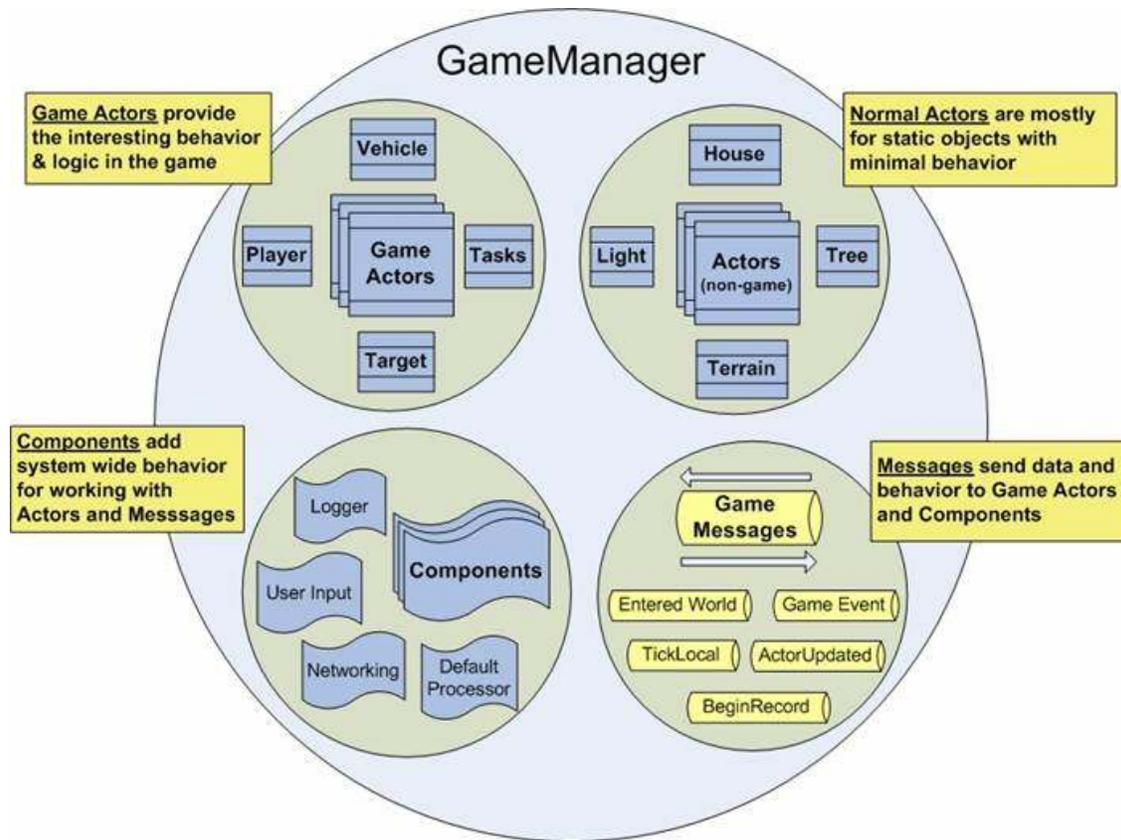


Figure 6.1.1 – The Game Manager

Since we will use Delta3D game engine to develop our game, it is important to introduce how a game is logically implemented by Delta3D. The above figure generally illustrates the architecture of the game engine. Let's explain the details of the figure.

The Game Manager

The first thing to notice is that the Game Manager owns all the Game Actors, normal Actors, Components and Messages. Specifically, it manages the existence and interaction between all the actors and components in your world. It is the heart of your application.

Delta3D gives you almost all the pieces you need including: scene management, physics (not so much), audio abilities (OpenAL), the ability to load objects, environmental effects, lighting, terrain support, cameras and character animation.

Now you have all the pieces you have (except networking which we will handle because our game will be massive!), the biggest problem here is that how we are going to put it all together to make our game running. How do we hold onto the list of all objects in the game? And, how do we manage communication between objects? How does our new car get into the world so the player can pick it up? How do we understand and send messages about a crash? And, how do we inform players about each other? These hows can be longer. And

there is a lot of stuff to worry about to manage our game. Fortunately the Game Manager provides the core architecture to make it all happen.

Actor and Game Actor

An Actor is any game object that has properties. An Actor can truly be anything we can imagine, from flying planes to pulsing lights to containers to trees. Actors are the specialized objects that help define our game.

A Game Actor is an Actor that is designed to work directly with the Game Manager. The difference between a Game Actor and a regular actor or “non-game” actors is that the latter is basically static objects in the game world. We might use it for static meshes like houses, non-moving lights, trees and may be terrain. The Game Actors are what bring life to our game. Game Actors get ticked, can process Game Messages, and can generally interact with the world!

The only way that a Game Actor or actor communicate with Game Manager is Game Actor Proxy. Here, we need to define what a proxy is. An ActorProxy is a wrapper for an Actor. Proxies are simple, data oriented classes that have two main jobs. First, they provide a common, uniform class that the editor understands. Second, they know everything about the Actor they wrap, especially the properties. Since the Actor classes can be anything, the editor must be protected from the internal workings of any one object. To make this happen each Actor must have a corresponding Proxy class that allows access to the internal properties. In essence, the ActorProxy is a conduit for actors to communicate their behavior to external applications or to ensure a consistent communications protocol among actors.

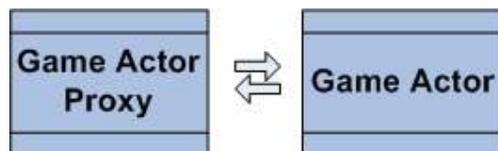


Figure 6.1.2- Game Actor and Game Actor Proxy

Messages and Actors and Components

To manage our game, the GM does three main things. The first, and most important, is the managing of messages: more precisely, Game Messages. Game Messages are what we are going to use to communicate between Actors, Components, and the GM. Messages can be used to communicate just about anything: updates to the player’s vehicle; the event for becoming a member of a cabstand; the fact that a player just entered the world (or leave) etc. Messages communicate all the behavior and state changes within the system, but messages don’t send themselves. The GM does that – it receives and routes messages to and from actors and components.

The second job of the GM is to hold onto ALL actors within the system. This includes both regular actors and the new GameActor . It tracks actors that we have created in code or in our map . The GM tells actors to process messages, makes sure they get in and out of the scene at the right times, and even tells them to ‘tick’ so they can do stuff.

The third job of the GM is to support GMComponents. What’s a component? A component is basically a high-level object that works with Game Messages. Game Actors do process messages; however, you will often have lots of instances of a particular actor in the game. Components are typically much higher level. They do things like networking behavior, game rules enforcement, logging. Whereas game actors only receive specific messages they are interested in, GM Components receive all messages.

What is a Game Manager Component?

In its simplest form, a GMComponent is an object managed by the Game Manager which can process and send messages. Unlike GameActors, GMComponents receive every message in the system. Components typically provide high-level system wide behaviors, but they can do pretty much anything you want. GM Components are the primary way to add custom behavior to the Game Manager.

Since components get all messages in the system, they have the opportunity to know about all actors and everything that happens! You can build simple components that wait for specific messages or listen for the keyboard/mouse. You can have complex components that hold onto large lists of actors and use hierarchies of helper objects. Components are the extendable architecture for adding important behaviors to your game.

What is a Game Message?

Game messages are simply the way actors and components communicate with each other. Messages are typically used for sending data (ex: property changes) or behavior (requests or commands). The following diagram shows a high level overview of the flow of messages to/from Game Actors, the Game Manager, and Game Manager Components.

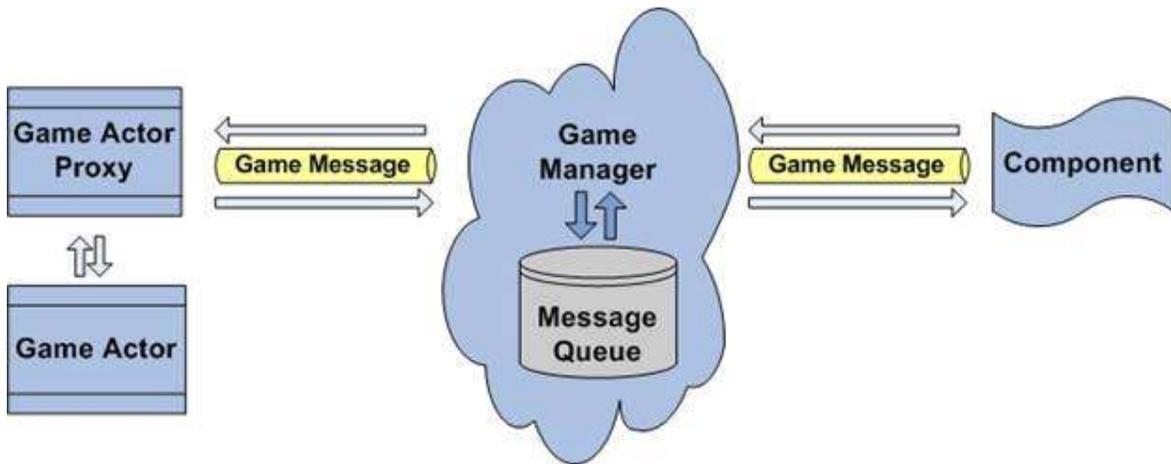


Figure 6.1.3-Game Message Diagram

There are a few basic methods here:

1. SendMessage() on GameManager
2. SendNetworkMessage() on GameManager
3. Invokables on GameActorProxy
4. ProcessMessage on GMComponent
5. DispatchNetworkMessage on GMComponent

The Game Manager has two primary mechanisms for handling messages: **SendMessage()** and **SendNetworkMessage()**. Hopefully, these are mostly straightforward. To send a message from anywhere in your game, you call SendMessage() on the GameManager. To send a message that you want to go out over the network, you call SendNetworkMessage() on the GameManager. Typically, you want to call SendMessage(). Only a few components should concern themselves with the network version.

The Components also have two methods: ProcessMessage() and DispatchNetworkMessage(). Components receive all messages. So, you override ProcessMessage() to receive normal messages. Then, if you are some sort of network

component, you override `DispatchNetworkMessage()` to receive messages that are ready to be sent over the network. Typically the `RulesComponent` is responsible for taking normal messages and resending them as network messages.

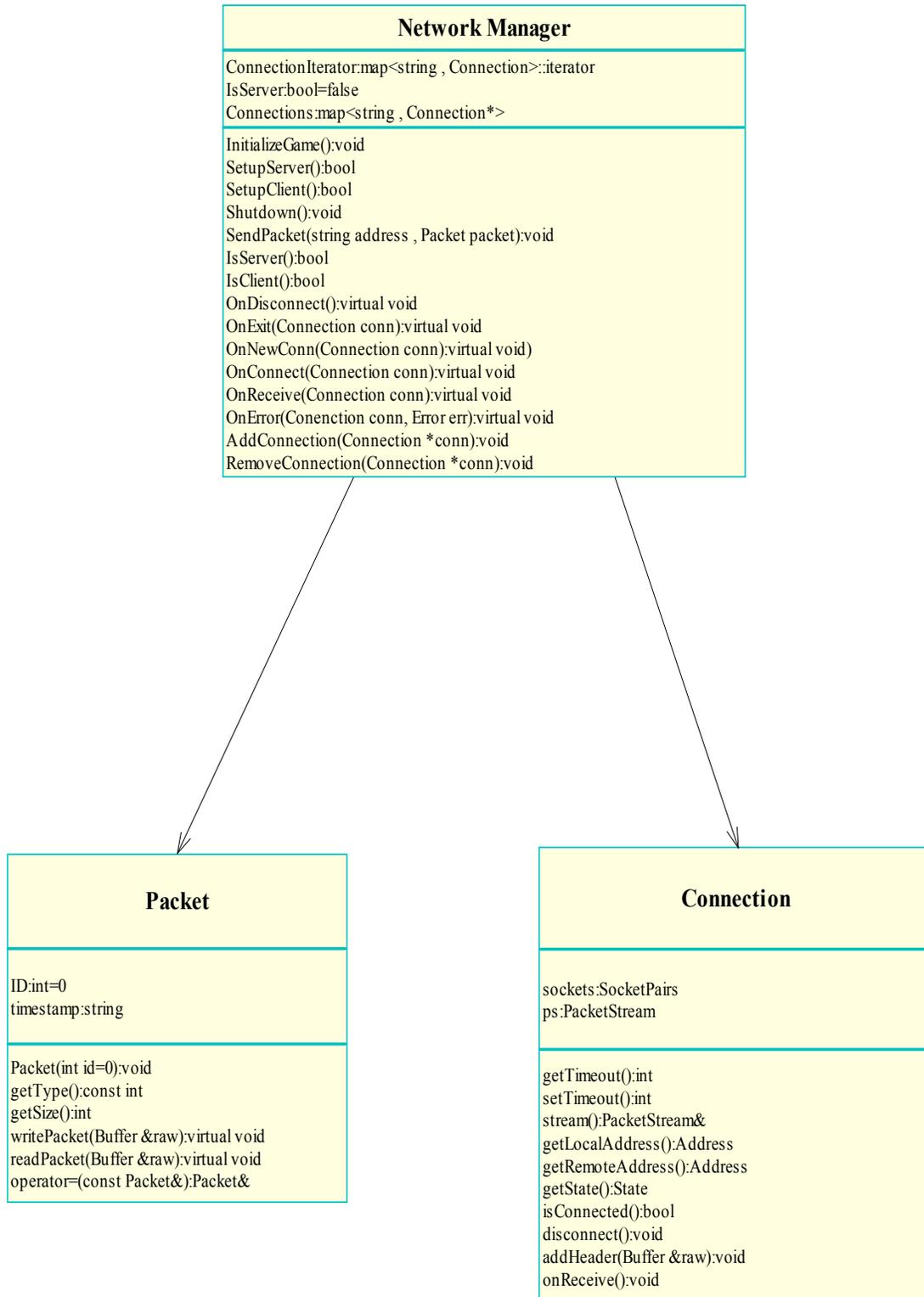
For actors it is a little different. Actors never receive network messages, they only get normal messages. Since there are often LOTS of actors in the system, we don't want to send all messages to all actors. Instead, actors register to receive certain types of messages by using `Invokables`. There are three default invokables on `GameActor`: `TickLocal()`, `TickRemote()`, and `ProcessMessage()`. `TickLocal()` is registered to take messages from local components and actors. `TickRemote()` is registered to be informed from other online clients. And `ProcessMessage()` is used to process the messages.

About our game

Since we have tried to describe the Game Manager and the logic behind it, it is time to embed our game objects, data models, message types and etc. to this structure. The first thing to do is defining our actors. The most important actor is the players' vehicles. It is the most dynamic object in the game world. So it would be defined as a Game Actor. And all the other dynamic objects such as traffic, customer, walking humans on the streets, flying planes, birds and etc. should be defined as Game Actors too. A detailed class diagrams will be supplied in the following sections. Then we should define our environment-mainly static objects that will not move or change during the game such as buildings, trees, lamps, some type of lights, garbage containers. So they should be regular actors. Then the components including user input, may be logging, and also networking have to be considered here. The user input is handled in this component. The network component gets and sends data packages in this part also. In our messages component there will be message types and the related event names such as `EnteredWorld()-EventName("EnterWorld")`, `StopCar()-Event("StopCar")`, `GetCustomer("GetCustomer")`, `ActorUpdated("Update")` and etc. These messages are sent through the actors and components and also through the network to make the required change by triggering related events. For example, when a new packet is come to the component, it passes the message to Game Manager via a proxy. The Game Manager then delivers this message to both all components and to all game actors that has already been registered to take this type of message. The registration is important to make the messaging traffic less. Then the message is processed by the taken components and the required update is applied.

Although we have not show in the Game Manager figure above, physics, sound and AI is also a part in the Game Manager. It controls all of them but we have to develop the parts and have to adapt to our game. We have to deal with physics engine to define a good and realistic car physics and deal with sound engine for both car sounds and other environmental and background or menu musics. And also we have to consider AI for the traffic, customer, walking humans and etc. and implement them efficiently. The detailed design of how we design our classes by using the existing ones in Delta3D is shown in the following sections more clearly.

6.2 NETWORK STRUCTURE



The network manager is a class to control the network and connections. It can be used for both client and server by calling the SetupClient() and SetupServer() respectively. In the server side, all the active connections are hold in a map with a host address and a connection pointer. And also there is a connection iterator in the server side which is useful to iterate over all connections. And isServer is a boolean attribute simply to indicate whether it is a server or not. The network managers work together with game manager. Shutdown() closes the network manager and also all the connections. OnExit, OnNewConn, OnReceive and OnError are events that is triggered when a remote connection disconnected successfully, a new connection is setup, a packet is received from that connection and an error occurred in connection respectively. SendPacket() sends the specified packet to the given host. And a network manager can add or remove connections by AddConection() and RemoveConnection() members.

The packet class constructs the base class for our datas that will be sent through the network. It has an id to define the type of the packet between the registered packet types. Id is unique to each type of packet. Timestamp attribute is required to order the received packets. Because in UDP protocol it is not guaranteed that the packets will arrive in order with it is sended. So, for example for a position packet we have to update the car position only if the received packet's timestamp is newer than the last packet that has updated the position. getSize() function returns the size of the packet in terms of number of the bytes it has. writePacket() writes the current packet to buffer to be sent and readPacket() reads the data that is currently on the buffer to this packet. Operator= is just to make the copying of packets easier.

The connection class is used to define and open a connection between client and server. It has methods just used to setup a connection and change or get the variables. The SocketPair is also a class handling the sockets to be listened and to be sended data over. SetTimeout() and getTimeout() sets and gets the timeout for that connection. getState() returns the state of the conenction indicating whether it is ReadyToConnect, Connecting, Connected, Disconnecting, Disconnected. Disconnect() closes the connection. onReceive() is a registered event that is triggered when there a packet is read from that connection.

Some of our packets:

PositionPacket:Packet
xyz:osg::Vec3 hpr:osg::Vec3 ownerID:const &string ID:static const int
PositionPacket():void ~PositionPacket():virtual void getSize():virtual void writePacket(Buffer &raw):virtual void readPacket(Buffer &raw):virtual void

This type packet is used to refresh the all players positions in the scene and need to be sended in every frame for an UDP protocol. ID is the packet type which is a static object that is created when the class is constructed. xyz is the position vector and hpr is the rotation vector of the car in three directions having owner whose id is ownerID.

PlayerQuitPacket:Packet

ownerID:const &string
ID:static const int

PlayerQuitPacket():void
~PlayerQuitPacket():virtual void
getSize():virtual void
writePacket(Buffer &raw):virtual void
readPacket(Buffer &raw):virtual void

We use this type of packet to understand that a player has quitted game. So we have to remove this player in all clients screen. The only thing we should send is the playerid. So we can search through the actors and remove it from the scene.

NewPlayerPacket:Packet

ownerID:const &string
ID:static const int

NewPlayertPacket():void
~NewPlayerPacket():virtual void
getSize():virtual void
writePacket(Buffer &raw):virtual void
readPacket(Buffer &raw):virtual void

This packet is used to inform every client that a new user has logged in, so create a user in your user map. Actually we donot a position packet with this packet. Because immediately after creating the user, the position packet will be sent to participant clients.

PlayerMessagePacket:Packet

nick:string
message:string
ownerID:const &string
ID:static const int

PositionPacket():void
~PositionPacket():virtual void
getSize():virtual void
writePacket(Buffer &raw):virtual void
readPacket(Buffer &raw):virtual void

This is the chat packet. When user enters the message on the screen and press enter, his messahe is sent to all clients. Nick is the nickname of the player that is shown on the ckhat screen and other menus, message is the entered message, and id shows the owner of the message

CreateTrafficVehiclePacket:Packet
type:integer ID:static const int trafficID:int
CreateTrafficVehiclePacket():void ~CreateTrafficVehiclePacket():virtual void getSize():virtual void writePacket(Buffer &raw):virtual void readPacket(Buffer &raw):virtual void

DestroyTrafficVehiclePacket:Packet
ID:static const int trafficID:int
DestroyTrafficVehiclePacket():void ~DestroyTrafficVehiclePacket():virtual void getSize():virtual void writePacket(Buffer &raw):virtual void readPacket(Buffer &raw):virtual void

TrafficVehiclePositionPacket:Packet
xyz:osg::vec3 hrs:osg::vec3 ID:static const int trafficID:int
TrafficVehiclePositionPacket():void ~TrafficVehiclePositionPacket():virtual void getSize():virtual void writePacket(Buffer &raw):virtual void readPacket(Buffer &raw):virtual void

These three packets are created to generate the traffic on the roads. Type is an integer value indicating whether the vehicle is a car, a bicycle, a truck etc. trafficID is unique to identify that vehicle. PositionPacket and Destroy packets are same as the player's except it uses trafficID to identify the car other than ownerID.

CreateCustomerPacket:Packet
type:integer ID:static const int customerID:int
CreateCustomerPacket():void ~CreateCustomerPacket():virtual void getSize():virtual void writePacket(Buffer &raw):virtual void readPacket(Buffer &raw):virtual void

DestroyCustomerPacket:Packet
ID:static const int customerID:int
DestroyCustomerPacket():void ~DestroyCustomerPacket():virtual void getSize():virtual void writePacket(Buffer &raw):virtual void readPacket(Buffer &raw):virtual void

CustomerPositionPacket:Packet
xyz:osg::vec3 hrs:osg::vec3 ID:static const int customerID:int
CustomerPositionPacket():void ~CustomerPositionPacket():virtual void getSize():virtual void writePacket(Buffer &raw):virtual void readPacket(Buffer &raw):virtual void

Since the customers should be automatically generated by server in order to be seen same by every player, the customer creation, deletion and position packets have to be sent periodically. By CreateCustomerPacket we create a customer having a type such as man, child, woman, girl, etc. and having id

customerID. The deletion and position is similar to above packets.

6.3 PHYSICS ENGINE

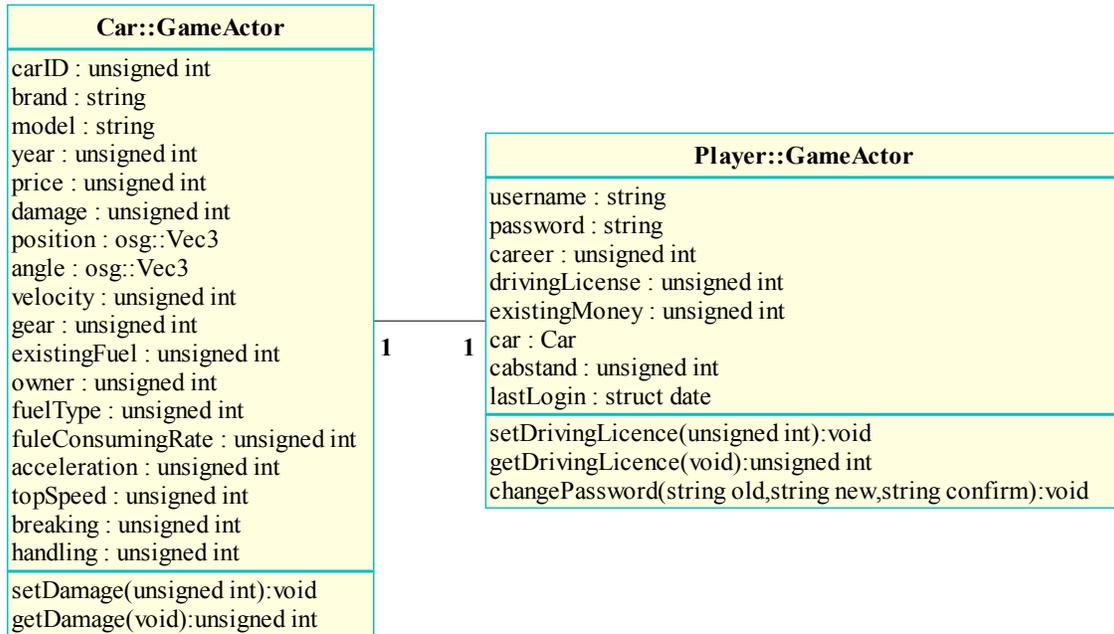
dtCore::Physical
attribute:Type = initialValue inertiaTensor:osg::Matrix centerOfGravity:osg::Vec3 mass:float isDynamicsEnabled:bool dBodyID:int
SetBodyID(dBodyID) : void GetBodyID(void) : dBodyID EnableDynamics(bool) : void DynamicsEnabled(void) : bool SetMass(const dMass *) : void GetMass(dMass *) : void SetMass(float) : void GetMass(void) : float SetCenterOfGravity(const osg::Vec3 &) : void GetCenterOfGravity(osg::Vec3 &) : void SetInertiaTensor(const osg::Matrix &) : void GetInertiaTensor(osg::Matrix &) : void PostPhysicsStepUpdate(void) : virtual void FilterContact(dContact *, Transformable *) : virtual bool

dtCore::Scene
x : float y : float z : float vec : osg::Vec3 gravity : osg::Vec3 data : MessageData * stepSize:double
AddDrawable(dtCore::DeltaDrawable *) : void RemoveDrawable(dtCore::DeltaDrawable *) : void RemoveAllDrawables(void) : void SetGravity(const osg::Vec3 &) : void SetGravity(float, float, float) : void GetGravity(float *,float *,float *) : void GetGravity(osg::Vec3 &) : void OnMessage(MessageData *) : virtual void SetUserCollisionCallback(dNearCallback *func , void *data=0):void GetPhysicsStepSize():double SetPhysicsStepSize(double stepSize=0):void RegisterPhysical(Physical *physical):void RegisterCollidable(Transformable *collidable):void UnRegisterPhysical(Physical *physical):void UnRegisterCollidable(Transformable *collidable):void

The dtCore:Physical class demonstrates the Delta3D's physics engine which we will use in our game to handle mainly collisions and other physical events. Since all objects in the scene is inherited from this class, we can enable physics of every element. The main method here is the EnableDynamics() method. It enables or disables dynamics for the object. SetMass() and GetMass() sets and gets the object's mass. BodyId is the ODE body identifier associated with this object. Centerofgravity is the gravity center of the object. PostPhysicsStateUpdate() updates the state of the object just after a physical simulation step. SetMass(const dMass*) sets the ODE mass parameters of the object.

We have written only the Physical related methods of the Scene class of dtCore. With AddDrawable and RemoveDrawable we can decide which objects we want to render. The gravity determines the gravity of the scene. OnMessage() performs collision detection and update physics. An object can be registered and unregistered as physical to scene. And we can also add collidable objects to scene. The engine will only find the collision with the registered collidable physical objects.

6.4 PLAYER RELATED STRUCTURES



Player and Car classes are inherited from the GameActor class of the Delta3D. GameActor class is designed for non-static objects on the game. GameActor objects are directed by the game manager.

Player class is used for holding the data and actions of the player. Since the player will have one car, for every Player object there will be a Car object if the player has bought a car.

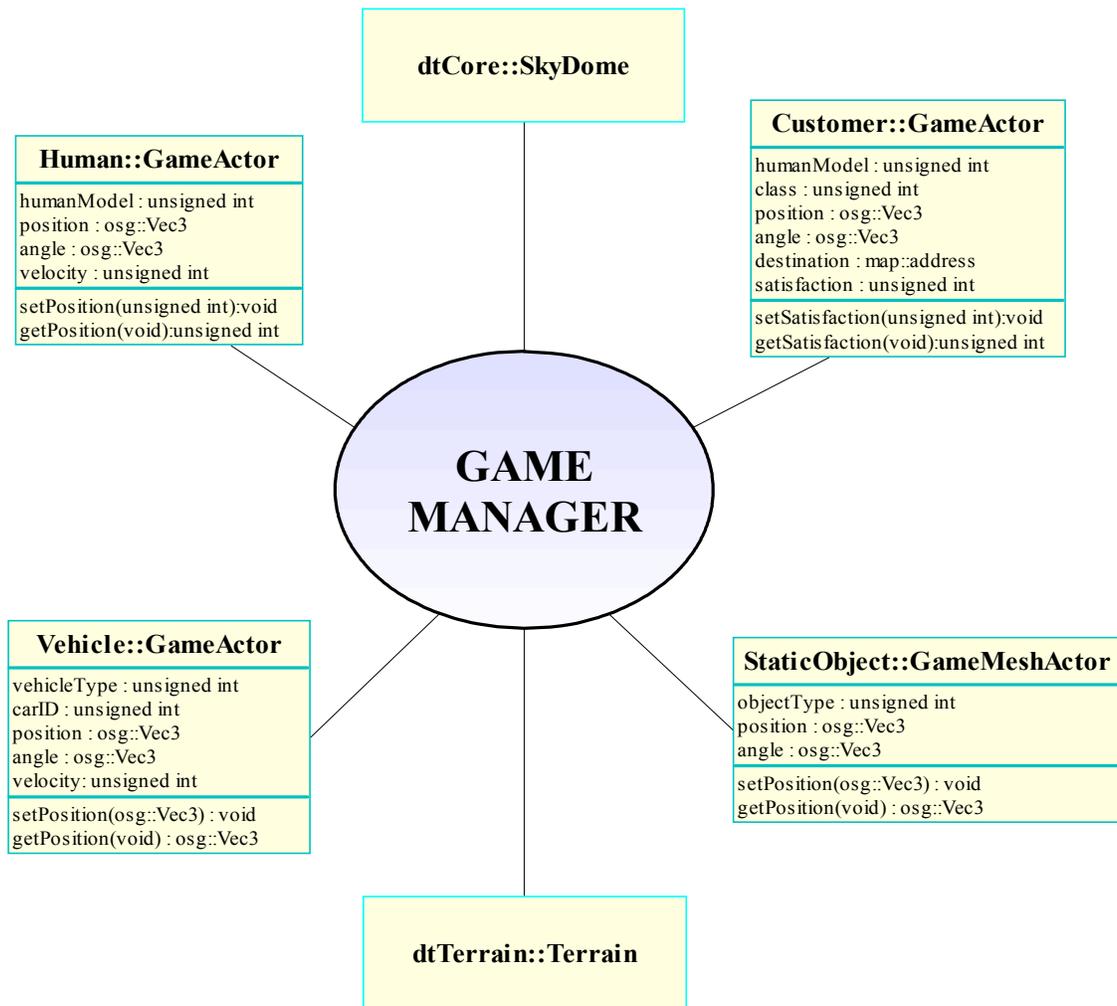
Since attribute names are generally self descriptive, there is no need to explain each one explicitly. Career attribute keeps the career point of the player, which is an integer value between 0 and 100. drivingLicense field keeps the driving license point of the player which is also between 0/100. car attribute is an object of Car class, which holds properties of the car for the player, and which holds needed functions to manipulate a car. Cabstand field holds the id of the cabstand that the player is a member of. If player is not a member of any cabstand, this field will have value 0.

Player class has ordinary set and get functions for the attributes that it has, which are not written on the diagram so as not to repeat the info. In addition to set/get methods, there exists a changePassword method, which changes the password with making confirmation test.

Car class is designed for manipulating car of the player, namely the taxi. Each car object has a unique carID. Damage ratio of the car is determined by the damage field, which has integer values between 0-100. 0 indicates no damage and as damage increases, integer value increases up to 100. Position and angle of the car is kept at related fields as Vec3 objects which is inherited from osg class. Vec3 class has 3 fields, keeping x, y, z values for position and alpha, beta, gamma values for angle attributes. fuelType attribute keeps integer value which are constants defined for indicating the type of the fuel that the car is consuming, namely super fuel, normal fuel, diesel, or LPG. Properties that are unique to car models such as fuel consuming rate, acceleration, breaking, handling, and top speed are also kept in Car class.

Car class has standard set/get functions for needed attributes, which are not shown on the diagram so as not to repeat info.

6.5 ENVIRONMENT RELATED STRUCTURES



Classes defined above are used for the objects that are at the environment and manipulated by either CPU or remote player, not by the player.

StaticObject class is designed for manipulating the static objects in the environment, with which player doesn't have any interaction. Namely these are trees, buildings, traffic signs, traffic jams, public gardens and etc. Type of the object is hold at objectType field, which are defined explicitly with constant ids. As usual, this class has set/get methods for position and angle attributes.

Human class is designed for manipulating the people in the environment. Human objects will be of several types, for example an old lady, a business man, a boy, a girl, a woman etc. This types will be kept at humanModel attribute. Human models will be able to move in the environment, with character animation. Character will have velocity, which will have three main degrees, stopping in which velocity is equal to zero, walking in which velocity will be 3 and running in which velocity will take value 6. Angle and position values are kept and changed from related fields as it is done in StaticObject class.

Vehicle class is designed for the cars that are in the environment. These cars may be the city traffic or the taxis that are being driven by other players. If the vehicle is the car of a remote player, vehicleType attribute will be set to 0 and carID field will keep the car id of that player, thus make it possible to take the needed information about the car, namely its model,

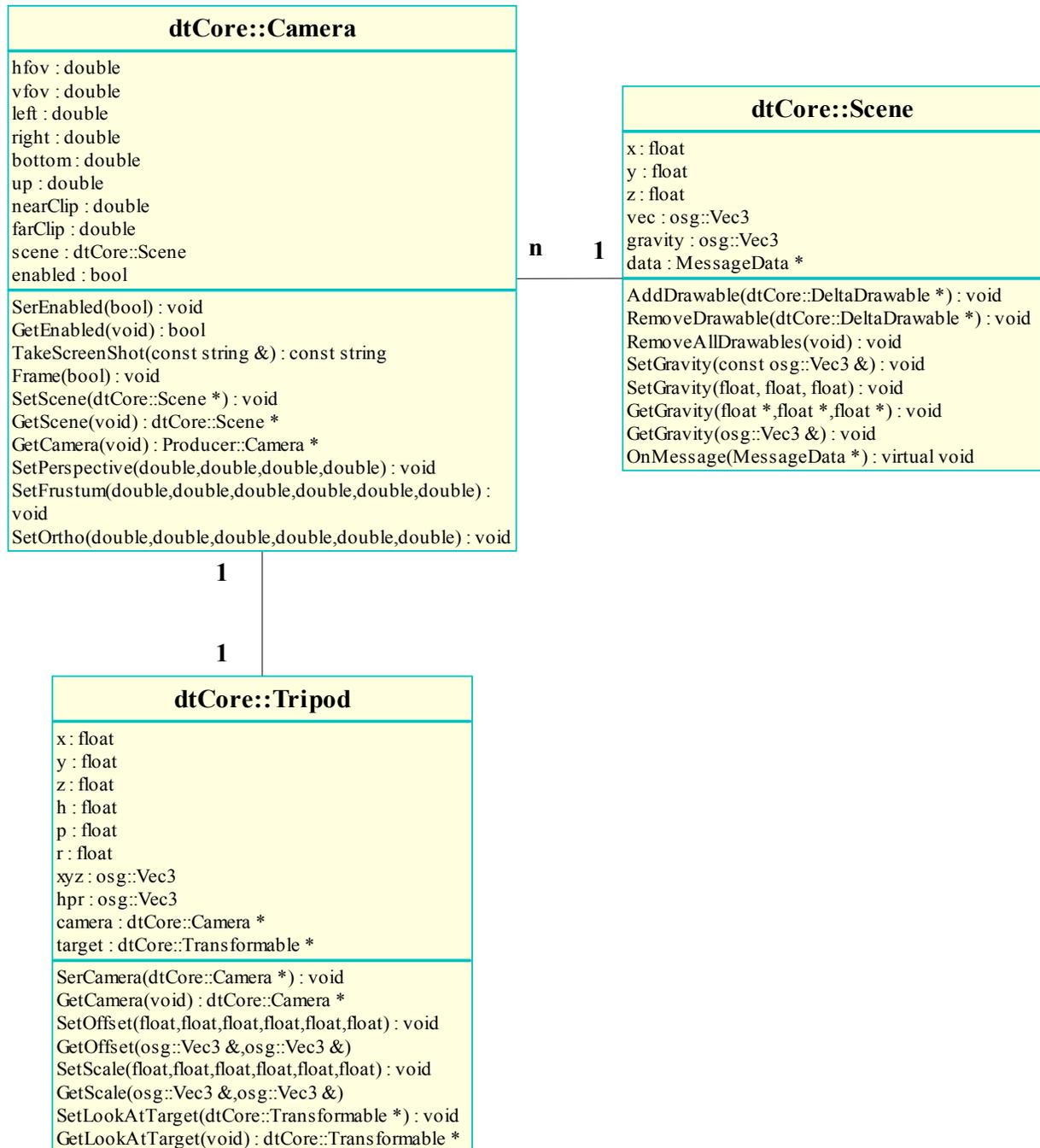
brand etc. If the vehicle is not controlled by a remote player, namely if it is a member of city traffic, vehicleType field will keep the type of that object and carID field will be NULL. For traffic there will be a few different type of automobiles, a truck model, and a bus model, which will be enough for our project.

Customer class is designed for manipulating the customers that will be served by the taxi. This class also has a humanModel field as it is in Human class. Additionally this class has a class field which indicates the class of the customer, namely a rich person will not want to get on an old or unqualified taxi, near that a rich person will give extra money to a taxi driver if s/he is satisfied from the travel. Satisfaction data is kept on satisfaction attribute. Also destination field keeps the address of the destination point in address type that is defined in map class. Since customer will wait fro the taxi, it will not have any movement, so velocity field is dropped. As it had been in other classes this class also has set/get methods which are not stated explicitly.

We will use SkyDome class from the Delta3D's dtCore library. This class makes it possible to generate a sky dome by adding it to environment and setting the time of the day. Lighting is adjusted according to this virtual time stamp. Also it is possible to generate moving clouds on the sky by the help of this class.

Terrain class from dtTerrain library of Delta3D will be used for generating the terrain. This class makes it possible to load a terrain to the environment with giving the frequency of hills. There exists a special program STAGE that comes with Delta3D package, with which it is possible to generate environment and adjust the terrain. We will also use this program for generating our environment.

6.6 CAMERA RELATED STRUCTURES

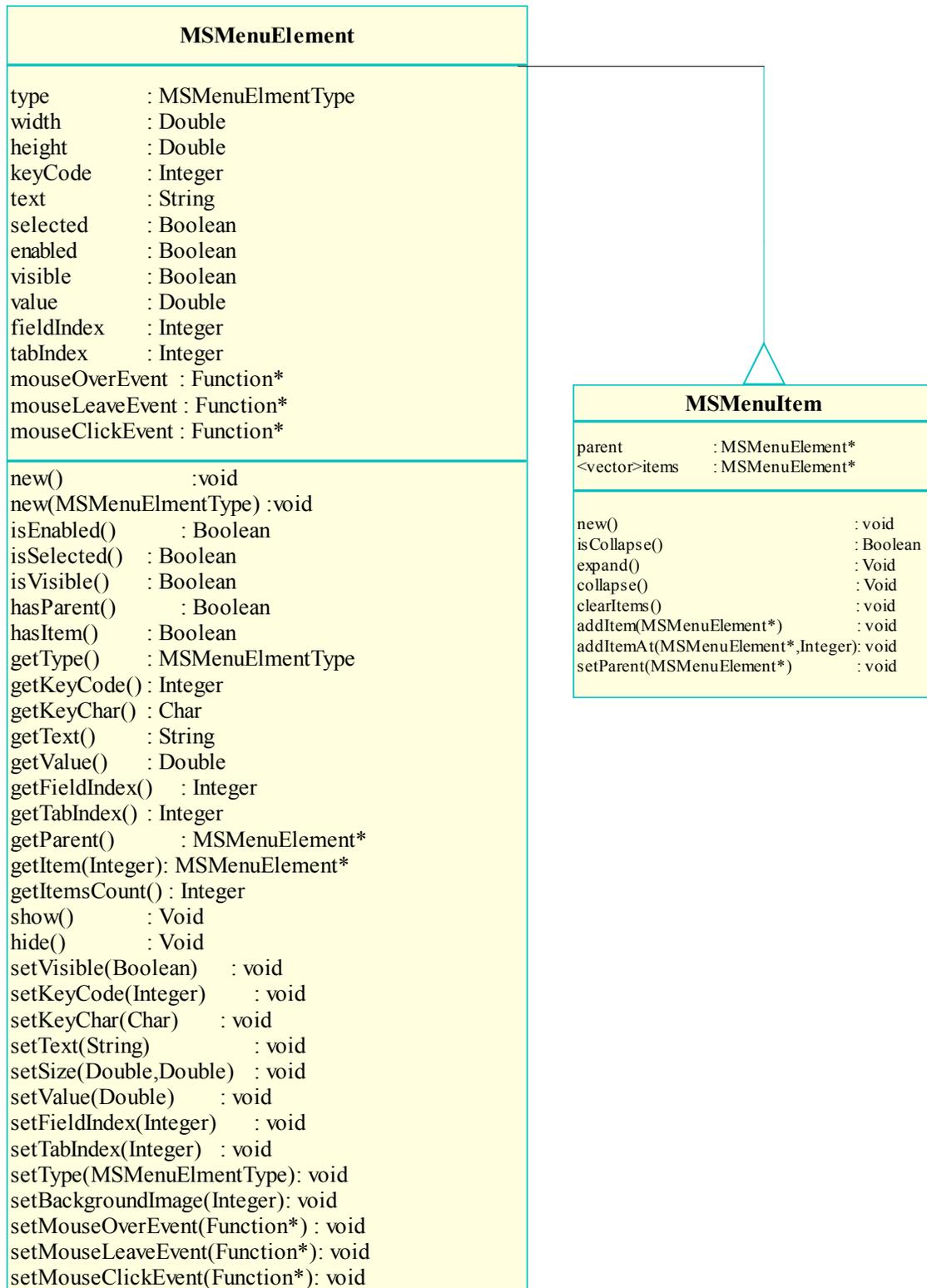


We will use Scene class as the main class for all objects that we want to draw in the scene. For making an object drawn to scene, after defining a scene, we need to add the object to the scene as drawable. This is achieved by calling AddDrawable function. With same logic, object can be removed from the scene by RemoveDrawable and RemoveAllDrawables functions. Gravity can be set and predefined gravity value can be get by SetGravity and GetGravity functions. By the help of OnMessage function, we can send message to physical class which handles collision detections. There are also many other function related to collision detection in Scene class which are explained in physical engine part of the report.

We use camera class so as to define a viewing point to our scene. We can enable/disable a camera with SetEnable and SetDisable functions. We can take the screen shot with TakeScreenShot function. Frame function makes screen redraws the screen. SetScene function sets the scene in which the camera will be used. And GetScene function gets the value of the scene that the camera has been assigned to. GetCamera function gets a non-const handle to the underlying Producer::Camera. SetPerspective, SetFrustum and SetOrtho functions sets the indicated projections to the camera respectively with the given values.

We use Tripod class for attaching a camera to a transformable such as the taxi. By the help of this class, we can move the view as the car moves forward. After defining a tripod object, we can attach a camera to a tripod by SetCamera function. GetCamera function returns the camera that was previously attached to a tripod. By the help of SetOffset function, we can set the distance between camera and tripod in x, y and z coordinates. By this way, camera comes after the car and car is also seen on the screen, which is the case in all car racing games. GetOffset function returns predefined offset value of a camera with respect to a tripod. SetScale function sets the per-frame lag for each degree of freedom. GetScale function return a predefined value. SetLookAtTarget functions makes tripod to point at some target which is in type Transformable. GetLookTarget function returns the value of the target Transformable object that is pointed by tripod.

6.7 MENU STRUCTURE



MSMenuElement Methods

MSMenuElement class is designed for some basic interface elements such as;MNButton,MNTextBox,MNLabel,MNRadioButton,MNImage.

This class has some public methods to check kinds of states of an element. Functions like isEnabled,isSelected,isVisible,hasParent,hasItem checks enableity, visibility,selected or whether it has parent, child or not. MSMenuElmentType of instancecan be get or set by using getType, setType method. getKeyCode,getKeyChar methods returns key code/char of instance that is used for rising event on that key pressed and thanks to setKeyCode, setKeyChar their values can be set. It also can be added some mouse events. Using setMouseOverEvent we can set MouseOverEvent funtion that triggers when mouse is over on it. And using setMouseLeaveEvent we can set MouseLeaveEvent funtion that triggers when mouse leaves it. MouseClickEvent funtion also can be set which triggers when left mouse is clicked on it. Background image of some elemnets can be set with function setBackgroundImage.

MSMenuItem Methods

MSMenuItem class is design in order to make it possible construct easy controllable with high funtionality menu. This class inherited from MSMenuElement so we can easily use lots of public methods of MSMenuElement. Besides, with this class we can construct hierarchical menu and we can control it. For example; expand function expands and colllapse function collapses the menu. New item can be added recursively wihtout depth limitation and also they can be removed. In addition to control, we can check kid of states of MSMenuItem like isCollapsed method.

6.8 SOUND ENGINE

MSSoundEngine	
volume	: Double
repeat	: Boolean
random	: Boolean
<Vector> items	: MSSound
stopAll()	: Void
stopAt(Integer)	: Void
pauseAll()	: Void
pauseAt(Integer)	: Void
resumeAll()	: Void
resumeAt(Integer)	: Void
playAt(Item)	: Void
playNext()	: Void
getCurrentMusics()	: <Vector> MSSound
setRandom(Boolean)	: Void
setVolume(Double)	: Void
increaseVolume(Double)	: Void
decreaseVolume(Double)	: Void
addMSSound(MSSound)	: Void

MSSound	
musicID	: Integer
volume	: Double
startAt	: Double
status	: MSSoundStatus
repeat	: Boolean
random	: Boolean
musicFinishedEvent	: Function*
stop()	: Void
pause()	: Void
resume()	: Void
play()	: Void
play(Double)	: Void
jumpTo(Double)	: Void
getStatus()	: Void
setMusic(Integer)	: Void
setVolume(Double)	: Void
increaseVolume(Double)	: Void
decreaseVolume(Double)	: Void
setMusicFinishedEvent(Function*)	: Void

MSSound

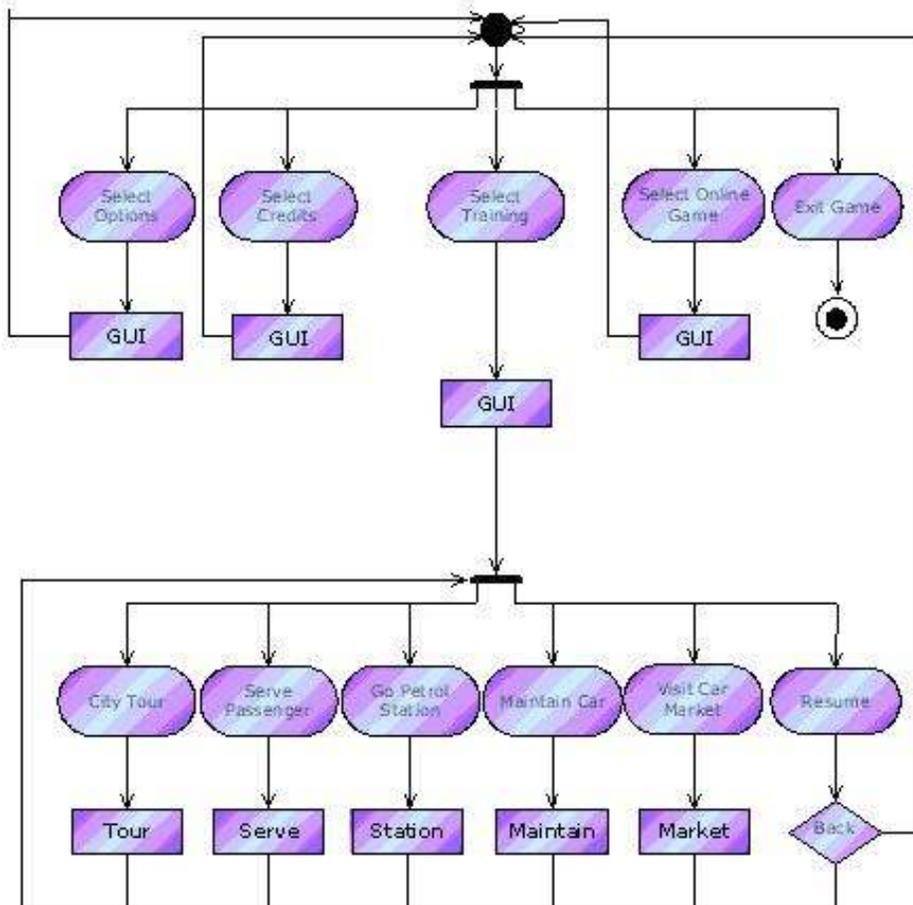
MSSound class deals with a sound stream. This stream can be music or a sound. We can control sound stream with methods. These are mainly, play method to play sound from starting point or any position of sound. StartAt parameter must be between 0 and 100(it can be thought as percentage). We can pause or stop playing sound and after a sound stream paused, it can resume. increaseVolume,decreaseVolume functions help us to control volume of the sound. This class also allow us to add MusicFinishedEvent listener in order to raise event when sound finished. Using jumpTo function we can jump to any position of the sound.

MSSoundEngine

MSSoundEngine class is designed to control more than one MSSound at same time. Although MSSound is a public property,Considering usability this class includes some method to control MSSound items without accessing MSSound items. For example; stopAt, pauseAt, resumeAt, playAt functions can be used instead of items[i].*. And this class allows to add sound item dynamically using addMSSound. And we can set, increase, decrease global volume of sound. Real sound is calculated by multiplying local volume and global volume.

7. ACTIVITY DIAGRAMS

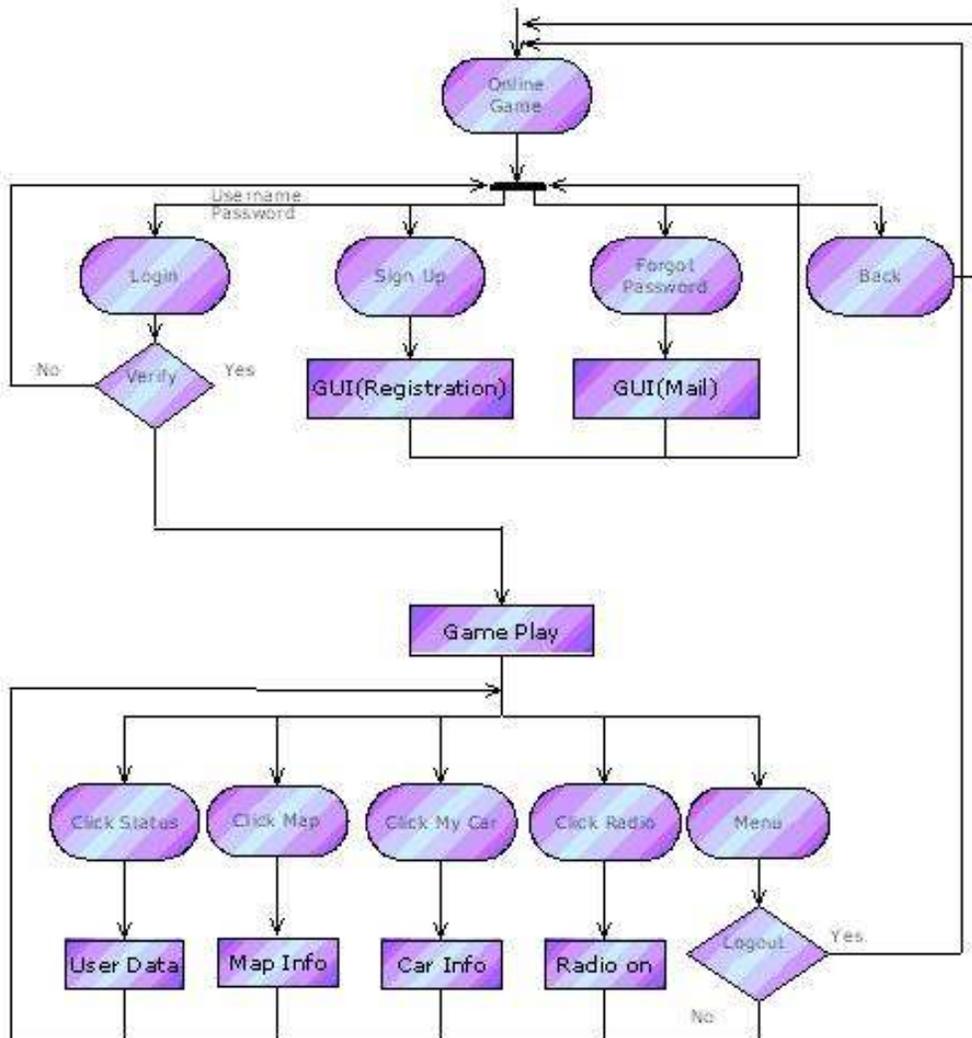
Training Activity Diagram



For training diagram we have 5 activities originated from the starting condition. These are select training, select online game, select options, select credits and click exit game. Select options and select credits returns to the initial condition after implementing their activities. On the other hand, exit game implements exit game and finishes the activities.

Select training and select new online game passes another condition after their activities. Online game condition is explained in diagram2 part. 6 activities originate from select training condition. These are; city tour, serve passengers, go filling station, maintain car, visit car market. After their activities, all these buttons return to the condition from which they are originated. Click resume button behaves according to a condition. If it creates the “resume the game” activity, this return to the condition from which the click resume button activity is originated. Otherwise it returns to the starting point.

Activity Diagram 2

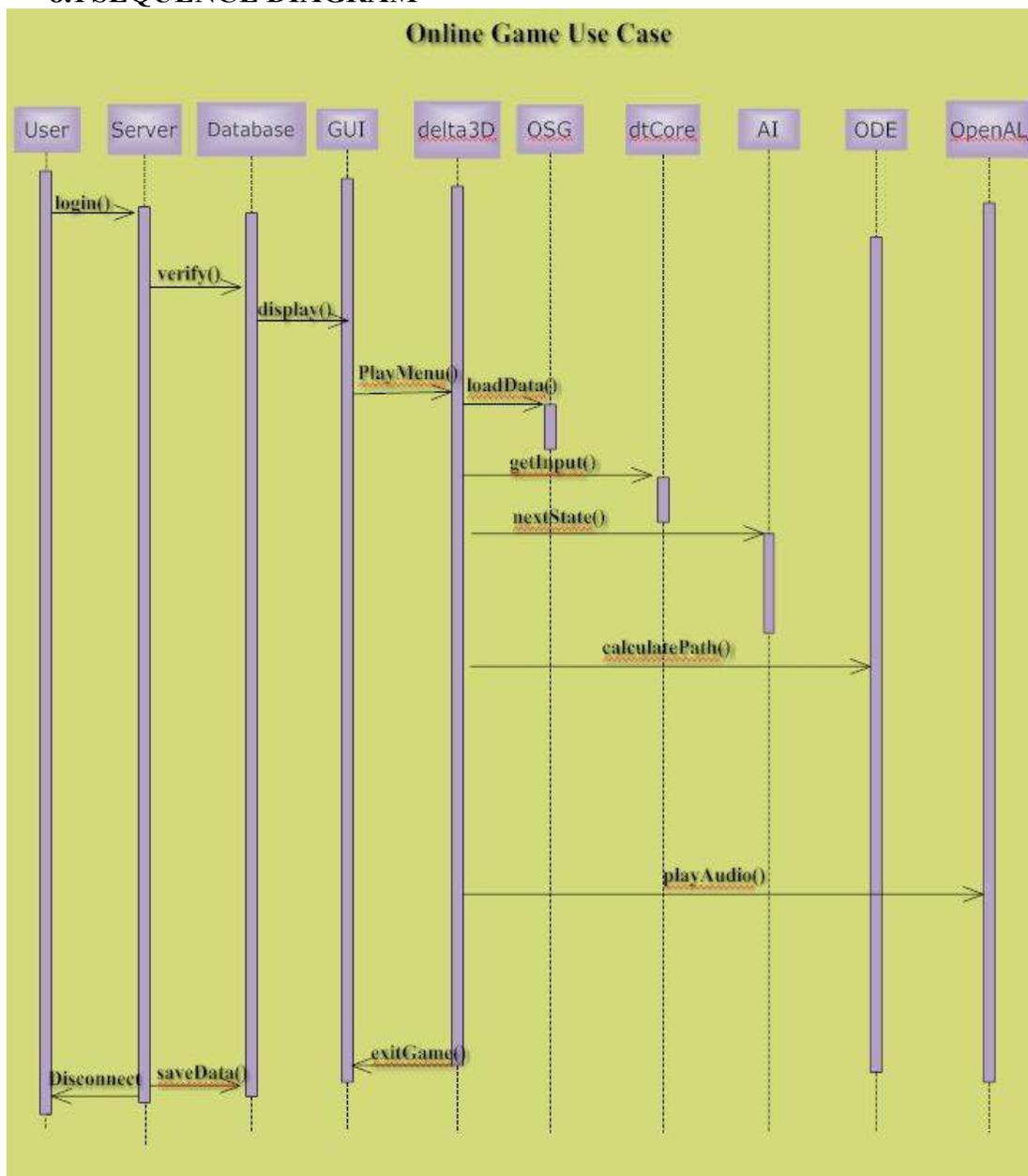


For online game diagram we have 4 activities originated from the starting condition. These are login, sign up, forgot password and back buttons. Sign up and forgot password returns to the initial condition after implementing their activities. On the other hand, back button implements exit and returns to the condition from which it is originated.

Select login passes another condition after verifying that player username and password are correct. 5 activities originates from online game condition. These are; click status, click radio, click my car, click map and menu buttons. After their activities, all these buttons except menu return to the condition from which they are originated. Click menu button behaves according to a condition. If player logs out, activity returns to the starting condition. Otherwise it returns to the condition from which the menu button activity is originated.

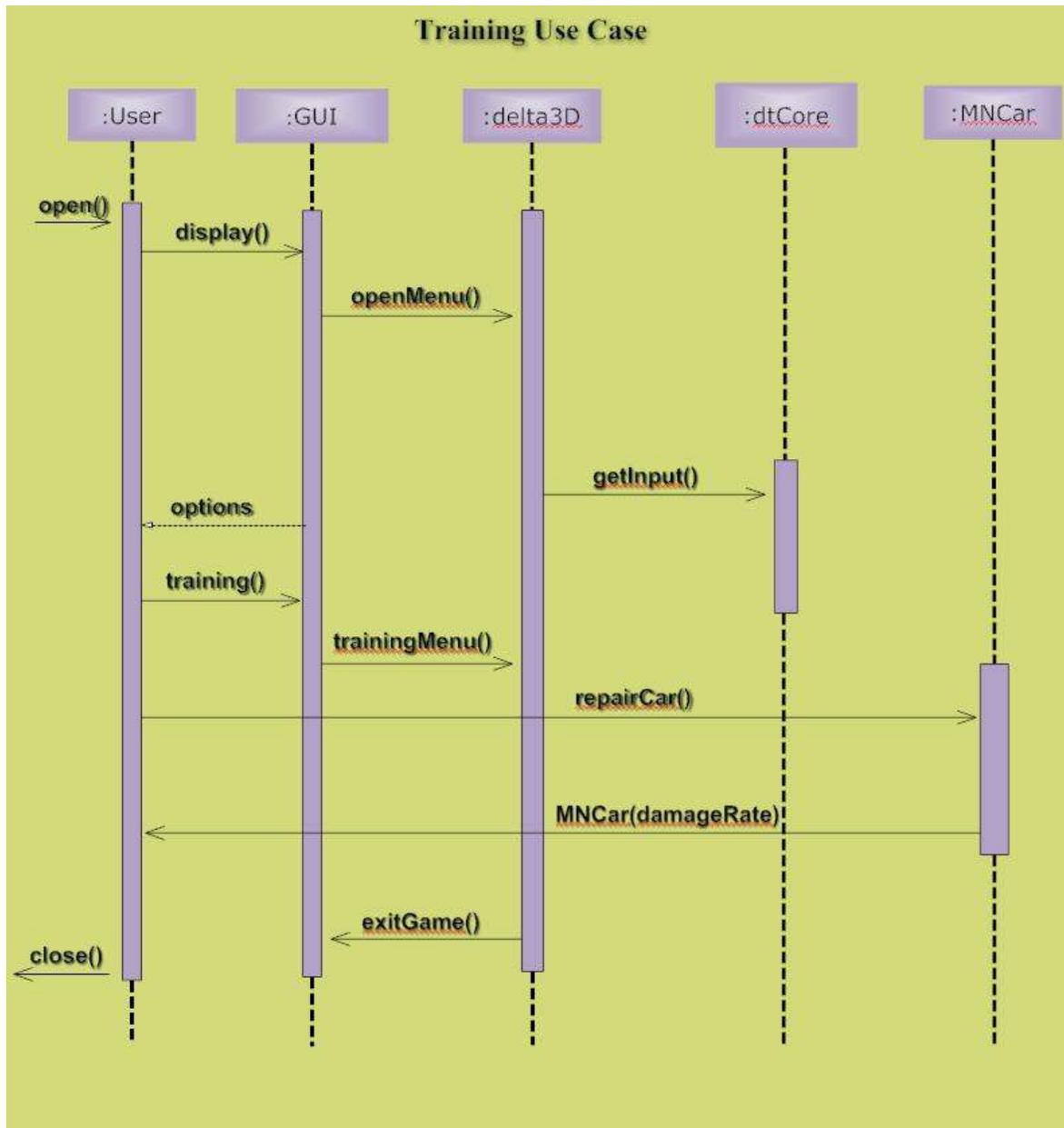
8. UML

8.1 SEQUENCE DIAGRAM



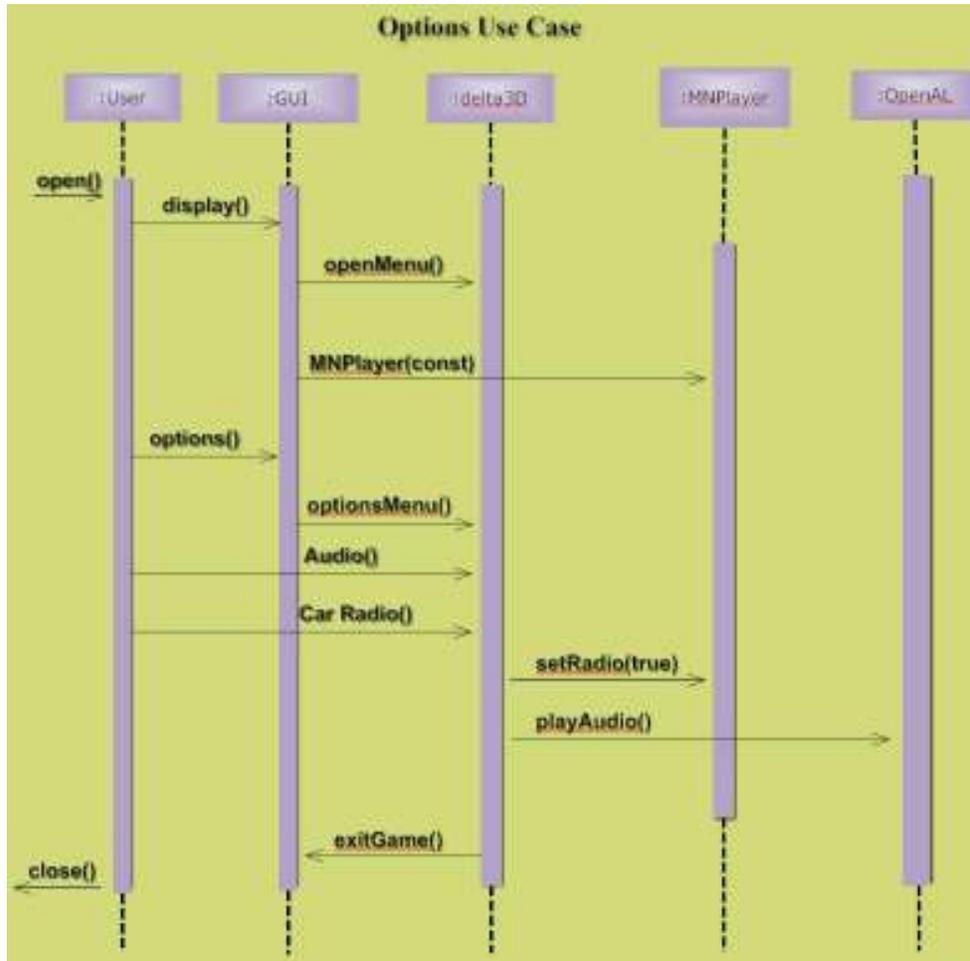
We represented the time-method sequence relationship in this diagram.

- ✚ At the beginning, the user must login the system, so user enter his username and password. At this stage user object sends login() method to server for verifying.
- ✚ Server object starts its lifeline when user sends his login info. After connecting database and verifying user account info, server make connection with the game and player and maintain it unless user exit the game.
- ✚ After confirming user password, server sends display() method to GUI.
- ✚ GUI object is related to the Delta3D game engine via openPlayMenu() method.
- ✚ Delta 3D engine is core of the system. It provide bridge between player and system. After user enter the game, engine sends loadData() method to Open Scene Graph object. This object loads necessary information regarding to game (car models, city map, textures etc...) and system waits for user to do action.
- ✚ If user moves the car system must do corresponding actions. First of all system should ensure that inputs are available. For this purpose delta engine sends getInput() method to dtCore class. As a result this class provides necessary functionality.
- ✚ After getting input parameters computer must calculate these to move car from one place to other. At this point delta engine sends calculatePath() method to Open Dynamics Engine.
- ✚ At this point server must provide some future states. So server sends nextState () method to AI engine.
- ✚ Since visual and audio effects provide entertainment, engine sends renderDisplay () and playAudio() methods to Graphics and OpenAL objects.
- ✚ System processes these methods iteratively and if user play exit button then GUI sends exitGame() method to server. Servers send users info to database via saveData() method call. Then server sends disconnect() method to user.



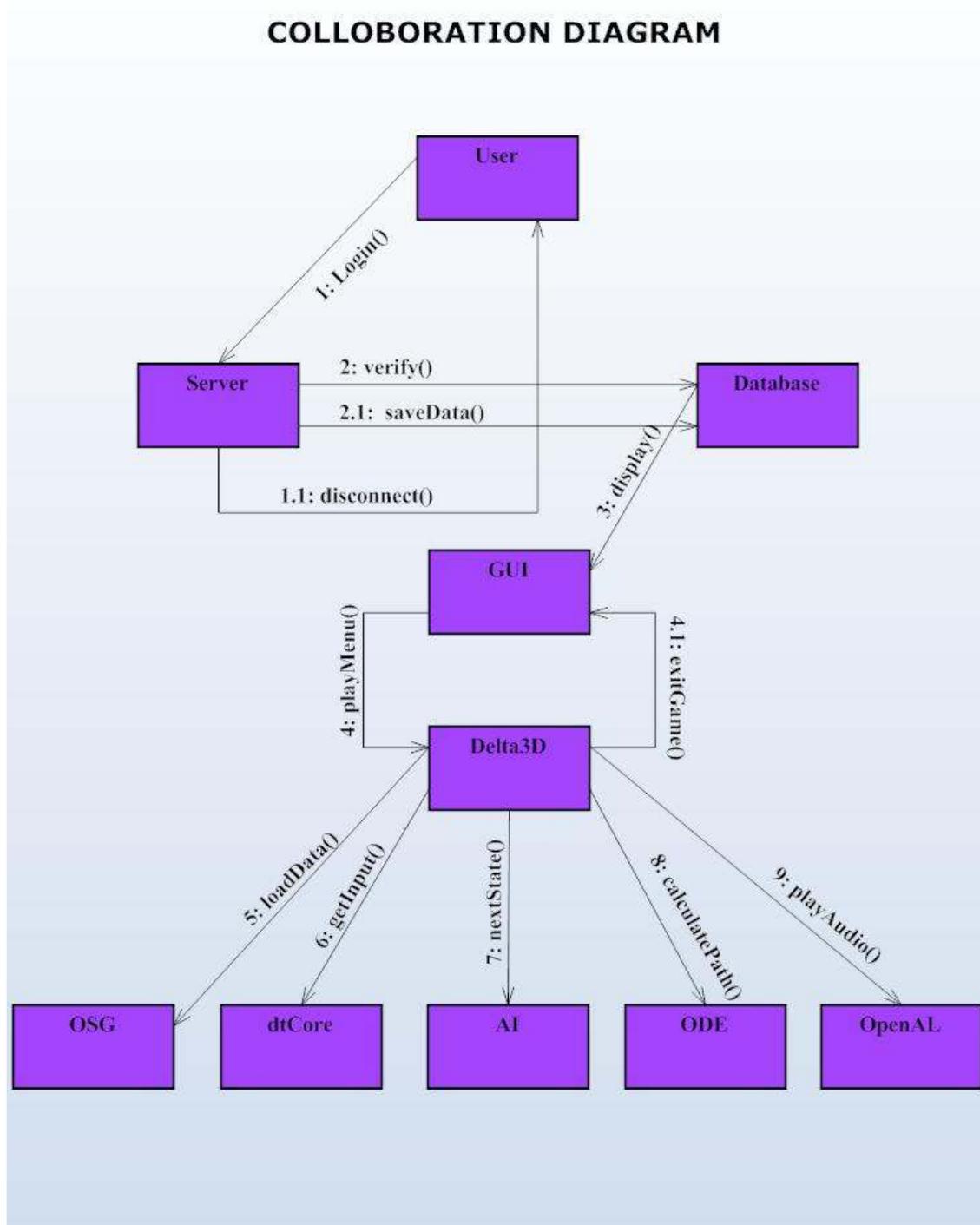
- ✚ We represented the time-method sequence relationship in this diagram.
- ✚ At the beginning, the user must login the system, so user enter his username and password. At this stage user object sends login() method to server for verifying.
- ✚ After confirming user password, server sends display() method to GUI.
- ✚ GUI object offers some options to player via openMenu() message to delta3D.
- ✚ Delta 3D engine is core of the system. It provide bridge between player and system.
- ✚ If user selects an option system must do corresponding actions. First of all system should ensure that inputs are available. For this purpose delta engine sends getInput() method to dtCore class. As a result this class provides necessary functionality.
- ✚ After providing system to get input parameters, GUI provides some options to user (menu)

- ✚ User then choose action training and GUI opens another screen via trainingMenu function call to delta3D.
- ✚ User can then send repairCar() message to the MNCar object. This object gives some parameters to the MNCar constructor and returnsto the user
- ✚ System processes these methods iteratively and if user play exit button then GUI sends exitGame() method to server. Servers send users info to database via saveData() method call. Then server sends disconnect() method to user.



- ✚ We represented the time-method sequence relationship in this diagram.
- ✚ At the beginning, the user must login the system, so user enter his username and password. At this stage user object sends login() method to server for verifying.
- ✚ After confirming user password, server sends display() method to GUI.
- ✚ GUI object offers some options to player via openMenu() message to delta3D.
- ✚ GUI instantiate MNPlayer object via constructor call to MNPlayer Object.
- ✚ User then selects the Options menu and GUI sends optionsMenu() message to delta3D.
- ✚ User then choose Audio() and CarRadio() options and delta3D object sends setRadio() and playAudio() messages to OpenAL object.
- ✚ System processes these methods iteratively and if user play exit button then GUI sends exitGame() method to server. Servers send users info to database via saveData() method call. Then server sends disconnect() method to user.

8.2 COLLABORATION DIAGRAM



The relationships of the classes are as below:

User Class:

- ✚ Server via login() and disconnect()

Server Class:

- ✚ Database via confirmUser() and saveData()
- ✚ GUI via display() and exitGame()

GUI Class:

- ✚ Delta 3D engine via openPlayMenu()

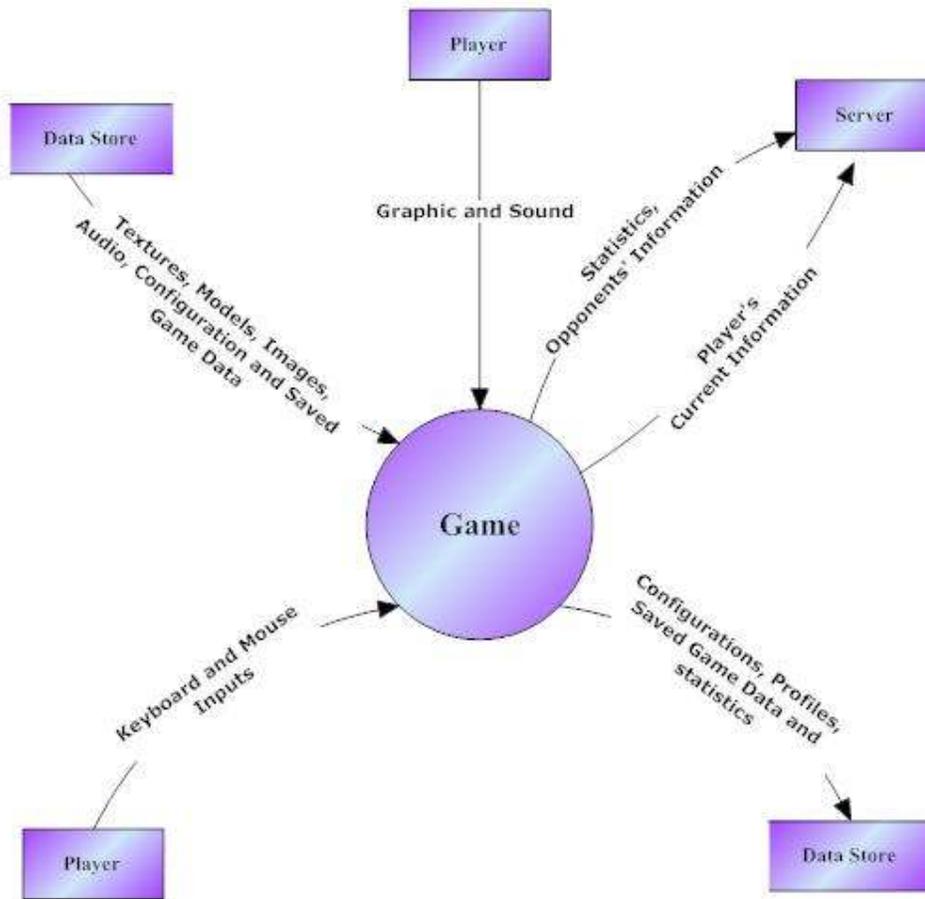
Delta 3D Engine Class

- ✚ Open Scene Graph via loadData()
- ✚ dtCore via getInput()
- ✚ Open Dynamics Engine via calculatePath()
- ✚ AI engine via nextState()
- ✚ Graphics via render()
- ✚ OpenAL via playAudio()

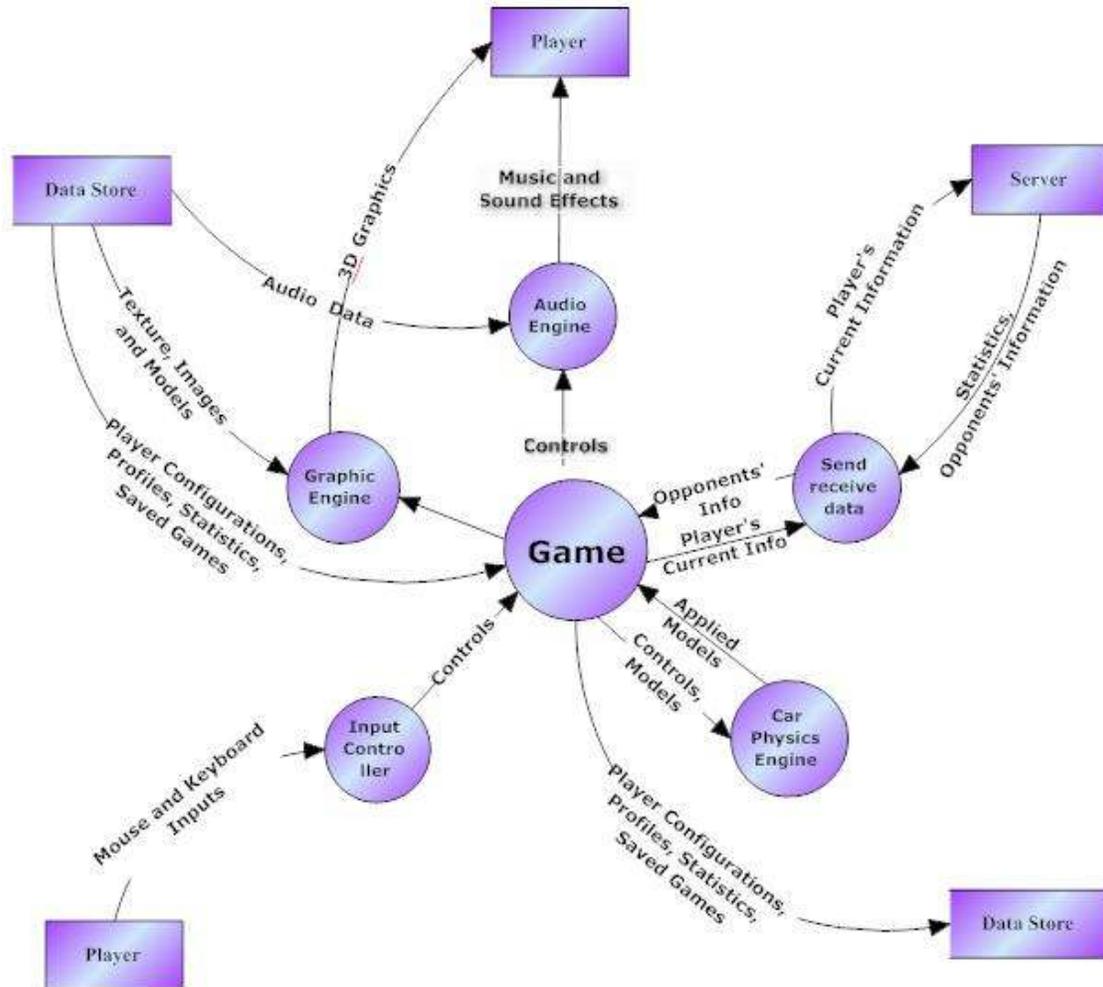
8.3 DATA FLOW DIAGRAM

In this section, the system is represented with a Data Flow Diagram (DFD), which depicts information flow and the transforms that are applied as data move from input to output. Input resources to the system are mainly the keyboard, mouse, texture, models, images, audio are main. Outputs of the system are mainly graphic, sound.

8.3.1 DFD Level 0



8.3.2 DFD Level 1



The system is composed of six sub modules and the data flows among them.

Input Controller is responsible to invoke suitable events according to keyboard mouse inputs

Load/Save module manages the load/save operations. It is also responsible for loading a new map. To do these, it reads information from database or from files to the Game Engine. On the other hand, it writes the game save data to files and to database.

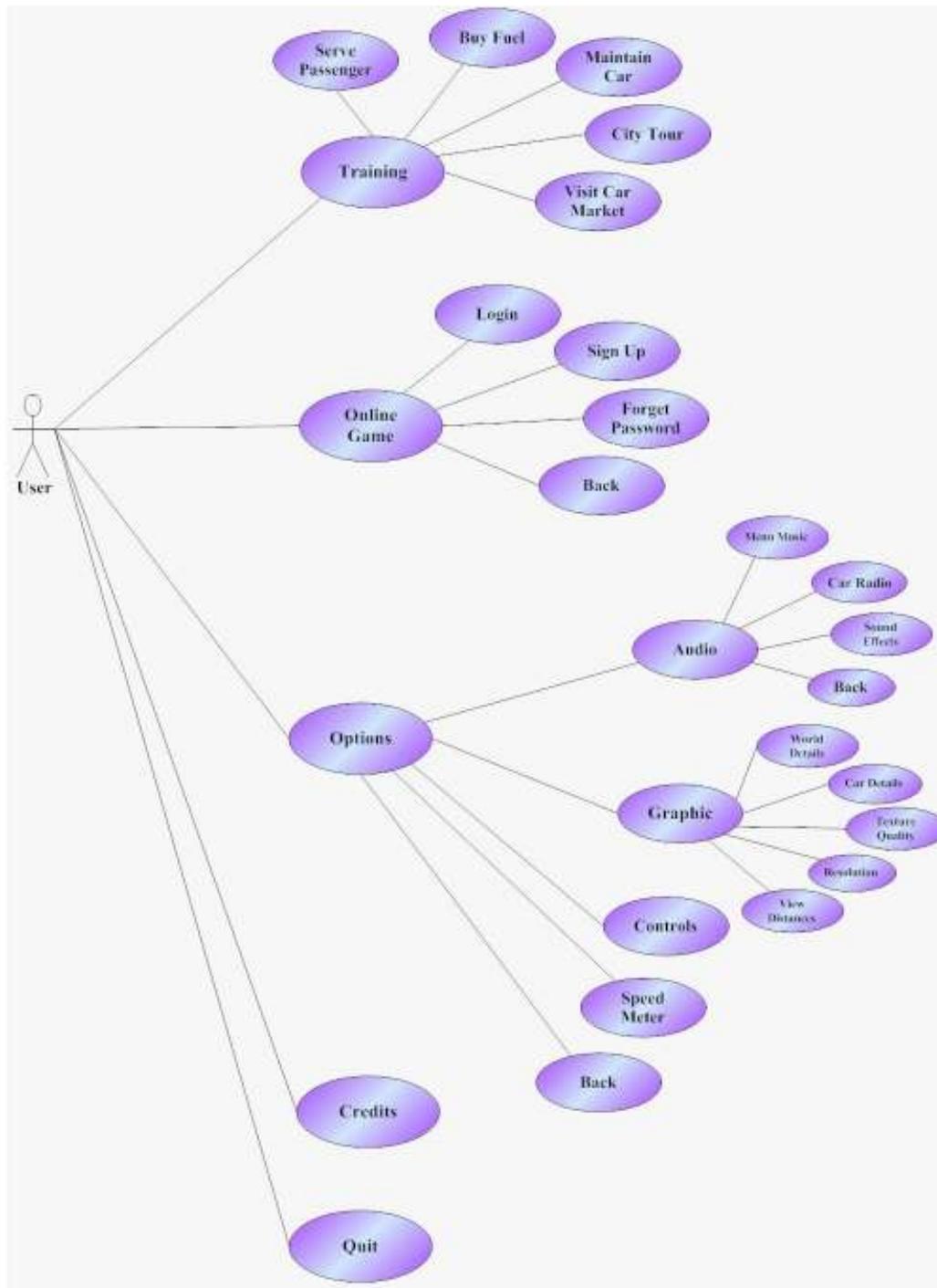
Graphic Engine manipulates and transmits the visual data that it receives from the central Game Engine to the monitor.

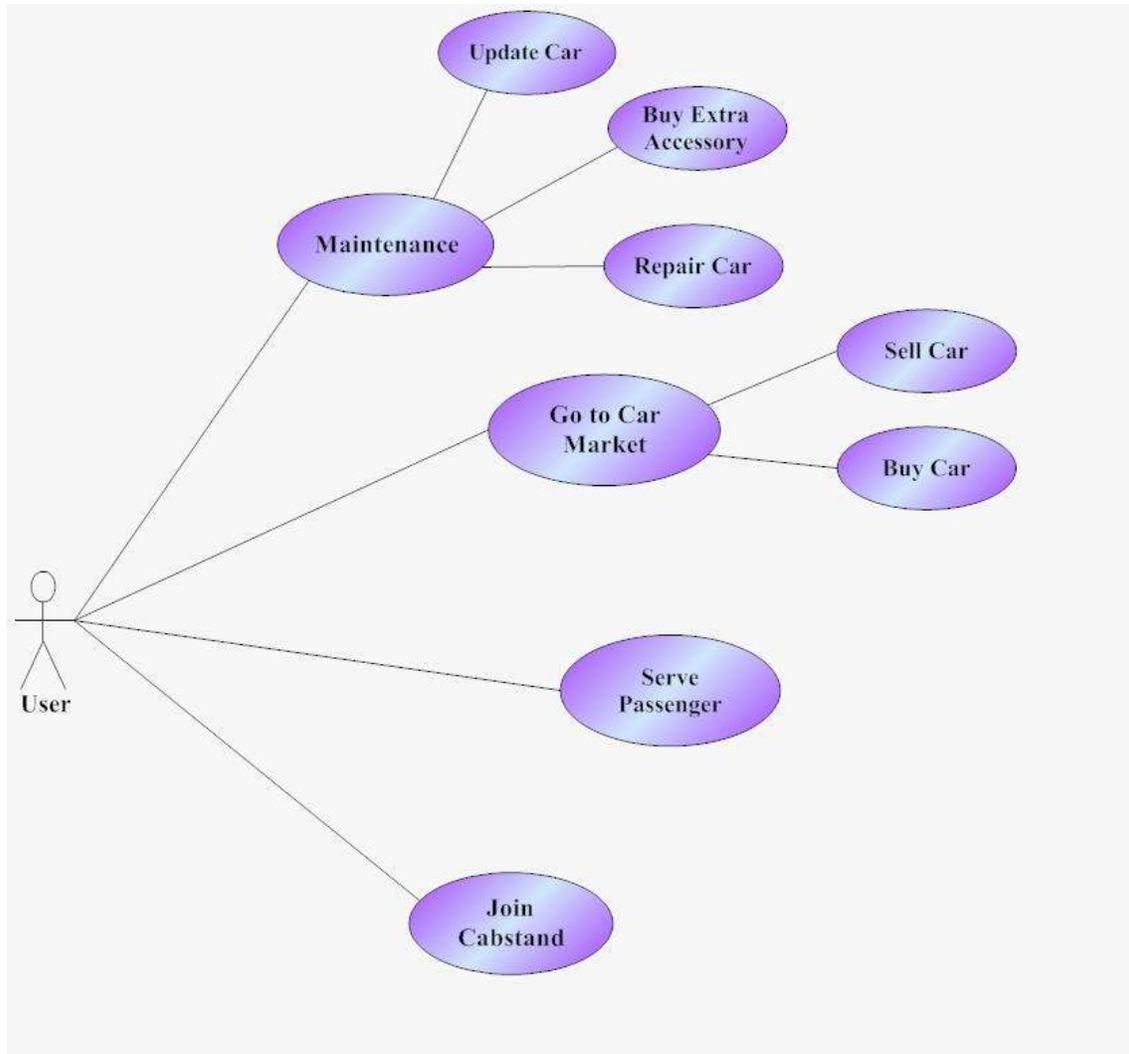
Sound Generator serves the requests of Game Engine and transmits the sound read directly from file to the speakers.

Physical Engine is responsible for detecting the collisions of cars and deciding movement of car.

Network Engine send information, received from Game Engine in a predefined stream format, to the server and gets information from server such as; traffic information, saved game.

8.4 USE CASE DIAGRAM

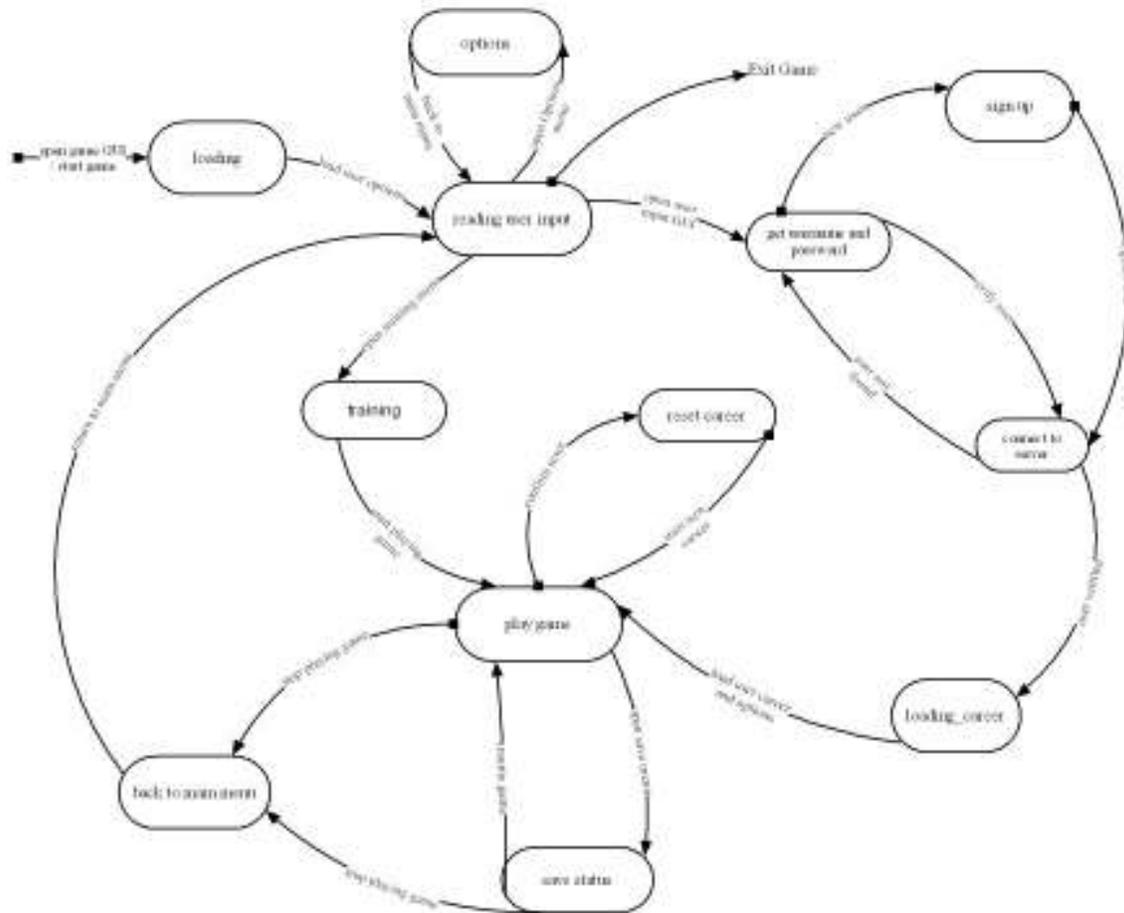




Our main aim here is to determine actors and their activities. Use case diagrams based on there actor group.

Maintenance	User can configure the car. By increasing comfort of the car he attract much customer.
Repair Car	Repair damages
Update Car	Change parts of car with new ones
Buy Extra Accessory	Buy extra accessory like coffee machine
Go to Car Market	User can go car market to sell his/her car or buy a new car.
Sell Car	Sell his/her car
Buy New Car	In order to buy a new car, he/she has to sell his/her car.
Serve Passenger	User can make money by serving passenger. Money depends on his/her quality of service.
Join Cabstand	User also can join cabstand. In this he/she can make much money.

8.5 STATE TRANSITION DIAGRAM



- **loading:** This is the state when user first run the game. The saved options are loaded according to the selected username. When loading is completed, this state is directed to the read user input state.

- **reading user input:** This is the state in which the main menu is displayed. User can select the options from menu, then options state becomes active. If the training item is clicked, the state is changed to the training state. The last thing that user can do in this state is to connect to server. Then he is directed to the get username and password state.

- **get username and password:** When user wants to play online and continues or starts to a new career, he is firstly asked for the username and password. If he is a new to the game and has no career, he goes to sign up state and register. If he is already a member, he enters his info and send server for verification.

- **connect to server:** This is the state where the user is verified. If the username and corresponding password are correct, this state is changed to loading career state.

Otherwise the user is asked again to enter the username and password correctly at get username and password state.

- **loading career:** If the user is verified by server, his status which is saved at database, is loaded with all options. Then immediately the play state comes.

- **play game:** This is the main state of the game. The user continues his career at this state. There are transitions to other states in this one.

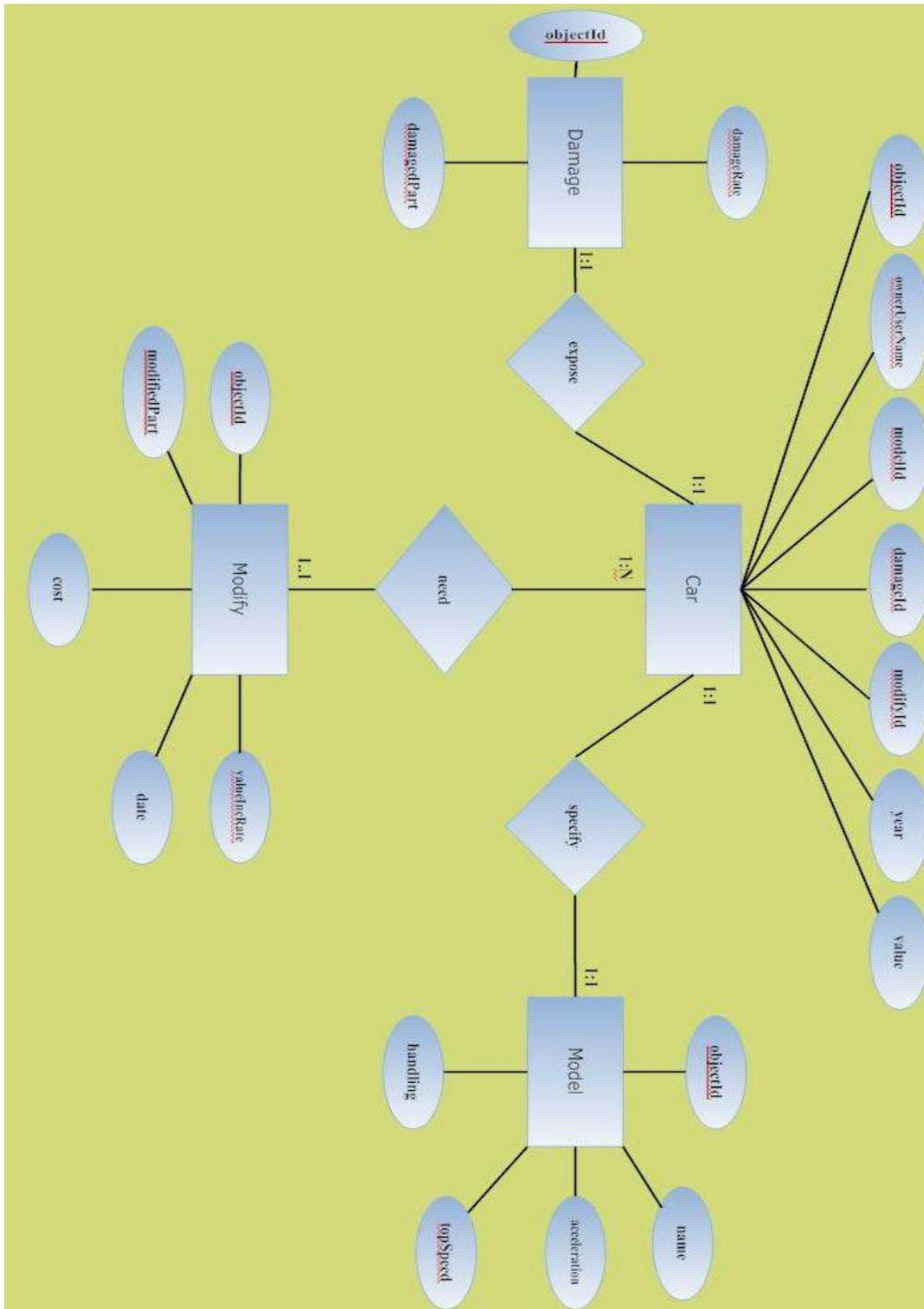
- **reset career:** If user wants to reset his career due to some bad conditions, he is directed to this state. Then he becomes a new user.

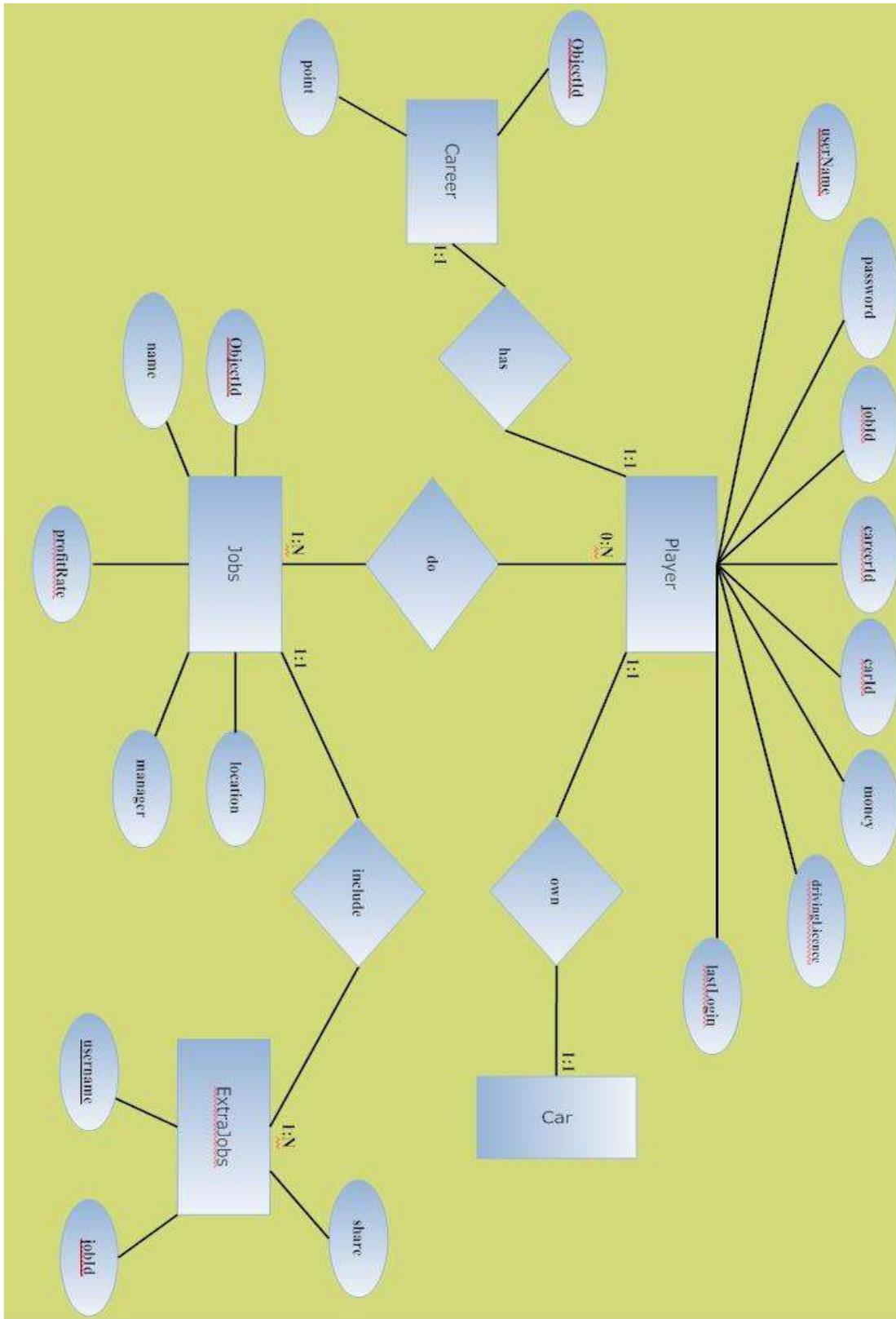
- **save status:** This is the state in which the career status at that time is saved to the user account in database. While loading the career, the last save is valid.

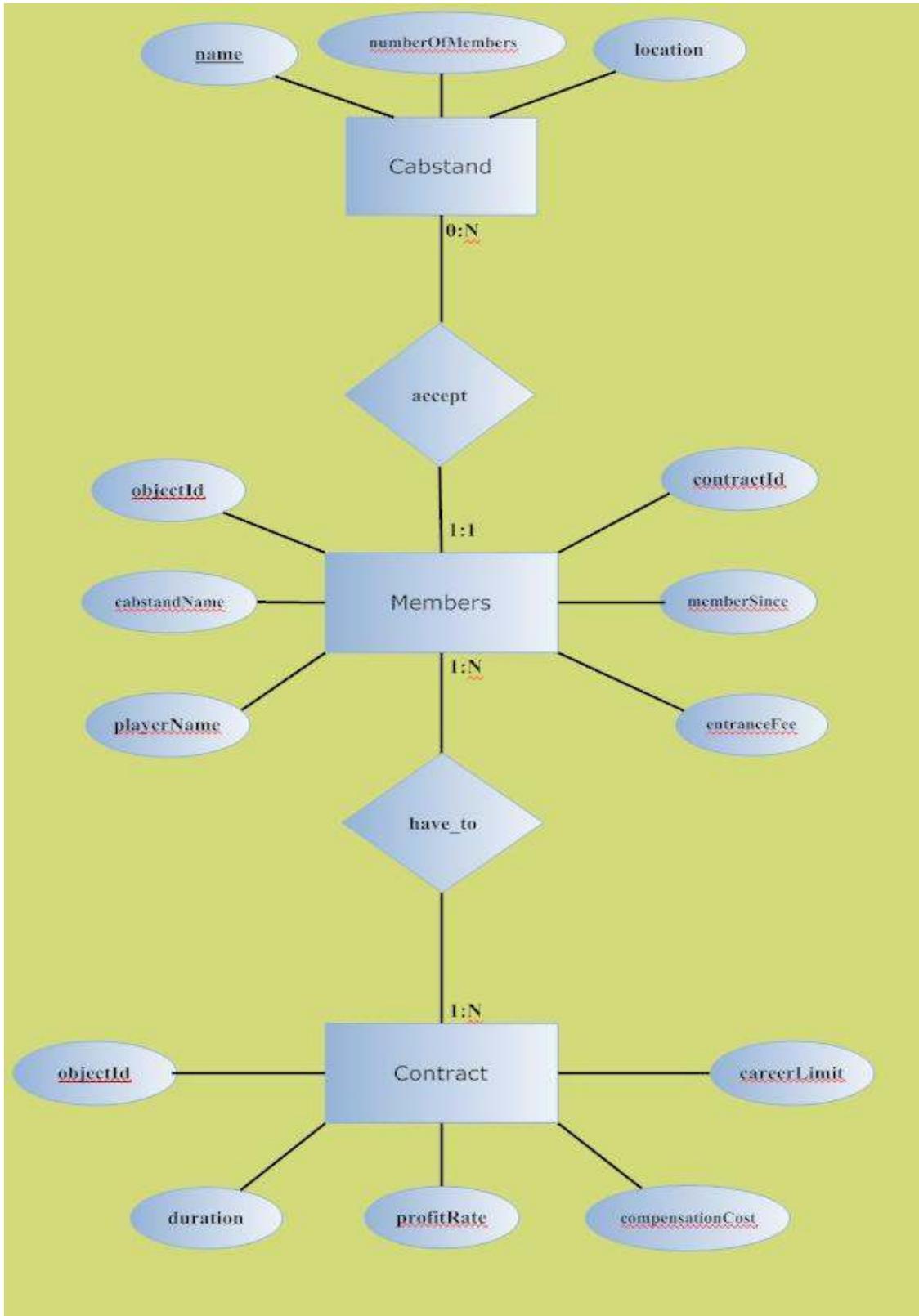
- **back to main menu:** To disconnect from server and return to the main menu, this state is used. Transition to this state is possible either from play game or from save status states.

- **training:** This is the exploring game and learning the basics state. After choosing which part to play, user can play the game in a single player mode.

9 DATABASE DESIGN







CAREER

Attribute Name	Attribute Type	Explanation
objectId	Int(12)	This integer holds the career id of the player
point	Int(12)	This integer holds the amount of point that player has

Key: objectId

PLAYER

Attribute Name	Attribute Type	Explanation
username	Varchar(16)	A unique string defined by the user to access the system
password	Varchar(16)	A string defined by the user to login the system with the username
jobId	Int(12)	This integer holds the job_id of the player
careerId	Int(12)	This integer holds the career_id of the player
carId	Int(12)	This integer holds the car_id of the player
money	Int(12)	This integer holds the amount of money that player has currently
drivingLicense	Varchar(2)	This string holds the type of license of player
lastLogin	Datetime	This value holds the last date of player login

Key: username

Foreign Key: jobId, careerId, carId

CAR

Attribute Name	Attribute Type	Explanation
objectId	Int(12)	This integer holds the id of car
ownerUserName	Varchar(16)	This string holds the username of the player
modelId	Int(12)	This integer holds the model_id of the car
damageId	Int(12)	This integer holds the damage_id of the car.
modifyId	Int(12)	This integer holds the modify_id of the car.
year	Datetime	This value shows the production year of the car
value	Int(12)	This integer holds the total cost of the car

Key: objectId

Foreign Key: modelId,damageId,modifyId

MODEL

Attribute Name	Attribute Type	Explanation
objectId	Int(12)	This integer holds the id of car
name	Varchar(16)	This string holds the name of the car
acceleration	Int(4)	This integer holds the amount of acceleration of the car
handling	Int(4)	This integer holds the amount of handling of the car
topSpeed	Int(4)	This integer keeps the top speed of the car

Key: objectId

Foreign Key: modelId,damageId,modifyId

MODIFY

Attribute Name	Attribute Type	Explanation
objectId	Int(12)	This integer holds the modify id of the car
modifiedPart	Varchar(12)	This string holds the part on which the change will be applied
cost	Int(12)	This integer holds the total amount when there is a modification
date	Datetime	This vale holds the date of modification
valueIncRate	Currency	This value represents the percentage of modification.

Key: objectId,modifiedPart

DAMAGE

Attribute Name	Attribute Type	Explanation
objectId	Int(12)	This integer holds the damage id of the car
damageRate	Currency	This value represents the percentage of damage
damagedPart	Varchar(12)	This string holds the name of the part of the car.

Key: objectId

JOBS

Attribute Name	Attribute Type	Explanation
objectId	Int(12)	This integer holds the id of job
name	Varchar(16)	This string holds the name of the job
location	Varchar(16)	This string holds the location of the job owner
manager	Varchar(16)	This string holds the username of the job owner
profitRate	Currency	This value represents the percentage of profit

Key: objectId

EXTRAJOBS

Attribute Name	Attribute Type	Explanation
username	Varchar(16)	This string holds the username of the job owner
jobId	Int(12)	This integer holds the id of job
share	Int(12)	This integer holds the total amount share

Key: username,jobId

CABSTAND

Attribute Name	Attribute Type	Explanation
name	Varchar(16)	This string holds the name of the cabstand
numberOfMembers	Int(12)	This integer holds the players member of this cabstand
location	Varchar(16)	This string holds the location of the cabstand

Key: username,jobId

CONTRACT

Attribute Name	Attribute Type	Explanation
objectId	Int(12)	This integer holds id of the contract

duration	Datetime	This shows the period in which the contract is valid
profitRate	Currency	This value represents the percentage of profit
compensationCost	Int(12)	This integer holds the money which the players must pay when they leave the cabstand
careerLimit	Int(12)	This integer holds the maximum amount of career

Key: objectId

MEMBERS

Attribute Name	Attribute Type	Explanation
objectId	Int(12)	This integer holds the id of the supplier
contractId	Int(12)	This string holds the company of the supplier
cabstandName	Varchar(16)	This string holds the name of the cabstand
playerName	Varchar(16)	This string holds the name of the player
memberSince	Datetime	This shows the period in which the player has been a member
entranceFee	Int(12)	This integer holds the money which the players must pay when they enter the cabstand

Key: objectId

Foreign Key: contractId

10.SYNTAX SPECIFICATION

As every software development has its own syntax, we decided to determine our own syntax specification for our project. Since “The TAXI” is a large project, having programming guidelines from the beginning will be useful for us. This is a vital issue, because it will be impossible to understand the code for others who are not the author of the code. As the total lines will grow, it may become impossible to debug the code unless we obey a common syntax.

Most of the rules we stated below are the general rules that are being widely used by the coders all over the world. Here we are restating the ones that we will use in our project.

Our team will try to stick to the guidelines as much as possible, which are mentioned below.

10.1DATABASE NAMING CONVENTIONS

Names of entities in our database have capital letters at the beginning of each word forming it. For example, the entity holding accounts of the players has the name “PlayerList”.

The names of the attributes of the tables have an underscore (“_”) between the words forming it, and are in lowercase. For example, the attributes of the “PlayerList” table may be as follows: “user_name” and “career_id”.

10.2 FILE NAMING CONVENTIONS

Names of the files containing class definitions will have the same name as the class. Documentation related file names will be initiated with “Doc_” prefix. And the implementation files will be classified according to the module they are belonging to. They will be given clear names describing their content and prefixed with the module they are in.

10.3 CLASSES

Classes will be declared in the following order: private data members and member functions, protected data members and member functions, and public data members (if any) and member functions. We will try not to use public variables as much as possible. We will try to write global variable free code, since this causes serious conflicts between different coders. We will use set and get methods instead. The names of the classes will be unique and will start with “C_”.

10.4 METHOD AND FUNCTION DEFINITIONS

All the method and function definitions will be preceded with the following lines of comments:

```
//-----  
// Description: Description of the method/function  
// Prototype : <return type> <function name> (<parameter type> <parameter name>...)  
// Parameters : <parameter name> <parameter description>  
// Return : <return value>< return value description>  
// Exception : <exception> <description of exception (when it is thrown)>  
//-----
```

The functions will have meaningful names in order to be understood easily. First letter of the function will be small letters; all other letters are also small letters except the first letters of the words. No underscores will be used. For example, “getPlayerName()” is a valid function name for our project.

10.5 GENERAL CODING PRINCIPLES

- ✚ We will try to write as self explanatory code as possible. Since writing tricky but short codes makes code impossible to understand even for the author of the code after some time, we prefer self explanatory but long code.
- ✚ When debugging the code, if a correction is need, instead of quick solutions, we prefer more clever solutions that are written after long thoughts. This is because most of the time short and dirty solutions add two bugs to code when solving just one.
- ✚ We will use the CVS account that will be supplied to us by the department as frequently as possible. Since four people working on the same project may corrupt each others code, CVS may become a lifesaver for us in some point.

10.6 VARIABLE NAMING CONVENTIONS

In our project, the variable names will have some additional name prefixes that indicate the scope of the variable. A global variable will be prefixed with “g_” if exists (we hope it won’t), a static member variable will be prefixed with “s_”.

10.7 COMMENTING

Commenting of code is essential for later understanding and maintenance and is thus mandatory in our source code. However, it is not efficient to over-comment the code. Only the parts that cannot be immediately understood by looking at the code should be commented.

- ✚ Complicated calculations and operations should always be commented by adding a multi-line comment block with detailed descriptions.
- ✚ All classes must have a brief and detailed description that is usually put right in front of the class declaration in the header file.
- ✚ All member functions that are part of the class must have a brief description, if applicable. This description must be put just before the method definition in the implementation file.
- ✚ The documentation for public member variables or static class variables can only contain a brief description and need not have a detailed description.
- ✚ All source files must have a header like the following at the beginning of the file:

```
//-----  
// The TAXI  
// File: File Name  
// Author: Author's Name  
// Version: x.y  
// Edited: Date(dd.mm.yy) , Hour(hh:mm)  
// Comment: Comments  
//-----  
//-----  
// Long comments which need more than one line use  
// this block comment style  
//-----
```

11.GANTT CHART

