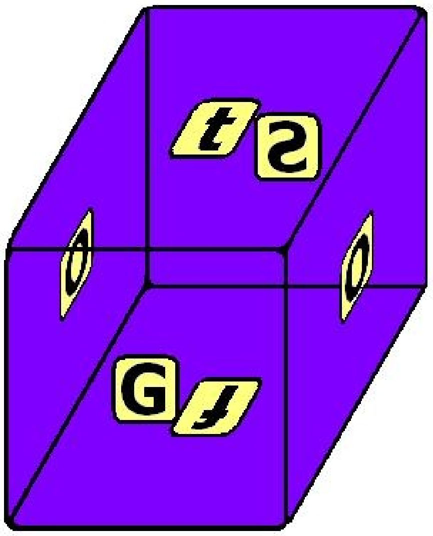
CENG 490
Senior Project

A 3D – Massively Multiplayer Online Game **"The Ma3e"**

Final Design Report

by



Ömer Akyüz e1347079
Önder Babur e1347186
Süleyman Cincioğlu e1347277
Güneş Aluç e1462670

Table of Contents

1. Introduction

1.1. Motivation

For our senior project, we will be implementing a 3D Massively Multi-player Online Game that is capable of simultaneously supporting hundreds of online players. The current Massively Multi player Online Game (MMOG) market is dominated by Role Playing Game (RPG) and First-Person Shooter (FPS) games. There are also a few examples of strategy games in this field. Since implementing multi player online games in a 3D environment is a difficult task, the examples of 3D MMOG games are limited, most of them being commercial. The reason why MMOGs have become so popular is that a person can play this game with other people and even his/her friends interactively. The 3D feature is an add-on to the environment. In this market, creative ideas are more likely to survive since they have a capability of leaving a stronger impact. We believe that our idea, whose details are provided below, is innovative in this respect, especially regarding the standard approaches.

1.2. Project Definition

The game to be implemented in this project should be capable of supporting more than a hundred online players simultaneously. This implies that a strong network architecture is essential for a working product. Thus, before proceeding with any design procedure, an in-depth analysis of existing architectures is essential.

Furthermore, the online players in the game are expected to interact with each other. This is actually the key point in producing a game that can attract a large number of people. For this purpose, the game should provide facilities in which one player's action should affect others'. However, the desired condition is that the scenario or the concept of the game **forces** the players to interact. Although, it is true that the server will suffer from this functionality, as it is a necessity, it cannot be neglected.

The game will be played in a 3D atmosphere that is rendered as realistic as possible. A good network architecture that separates graphics rendering from the game engine, or equivalently in this case the client module from the server, allows high-quality 3D graphics to be created and to be played without much of a network concern.

Although our initial view is that its integration to our game scenario will be difficult, the role of Artificial Intelligence cannot be neglected. Unlike in strategy games, our game – whose scenario details are given next – does not contain a solid set of rules that can guide the AI unit in resolving its search tree. The game to be implemented is mainly based on puzzle solving and for the AI player to become a part of it, some relations should exist between the puzzles. In reality they do not and this is the difficult part. To overcome this difficulty, the puzzles will be associated with each other by ontology mappings. Such ontological mappings should aid us in the development and deployment of additional puzzles since they will set out rules by which both the puzzle designer and the programmer should abide.

1.3. Overview of the Project Scope

For this project, our goal is to implement an adventure game from the first person's point of view. The game scenario is based on the movie "The Cube[1]" in which the actors are trying to get out of a very large cube shaped building consisting of n*n*n – k rooms. The rooms are also cube shaped and they are similar in view. Furthermore, on regular intervals, the rooms in the construction change their absolute position inside the cube.

The primary goal of the players in the game is to eventually get out of the cube. This, as one might guess, is not an easy task. The rooms are filled with tricks, booby-traps and several puzzles to solve. Even if the players are very close to the exit location, just before they move out of the room, that room's absolute position might change. In our opinion, this will add some flavor into the game.

To safely move from one room to the other, the players need to collaboratively solve the predefined puzzles. It is different from the original movie in this respect. After all, in the movie, the actors try to avoid the booby-traps by solving the puzzles. On the contrary, in our scenario, the players unlock the doors by solving the puzzles. Of course, not all actions performed by the players lead to a solution. Additionally sometimes the actions that are performed might harm the players, thus they need to be very careful.

The puzzles can be solved directly or indirectly interacting with the objects in the environment. Therefore, exchange of ideas between the players becomes very crucial at this point. By direct interaction we mean actions such as clicking on a wall item, using an object from the object inventory and various combinations of each. Indirect interaction refers to the personal experience gained in interacting with the environment objects. For example, there might be a written text on the wall that has clues as to how the puzzle for that room (or for some other room) can be solved. When a puzzle is solved, the players will be able to move to a different room. Although the players may prefer moving in groups, such a restriction is not imposed. Therefore, they might leave others behind. When they move inside the cube, the players will either encounter rooms with other players or some empty rooms.

As in almost every game, there will be some artificial intelligence present. However it will be implemented in a different way. In almost every occupied room there will be some (one or two, usually one) AI players present. They will mainly be moving with the group and performing actions together with them. The key role of the AI player will be to assist the other players in solving the puzzles. The AI players will usually have some background experience. After all, they will have moved with different people that have solved various puzzles. As the puzzles are solved, the AI players will learn from the actions that led to the solution. For this purpose, several predicate clauses will be associated with each of the actions that can be performed. As the AI player's search tree grows wider and deeper, it will be able to come up with better suggestions as to how the puzzle in the current room can be solved. Therefore, it will assess the objects and the actions that can be performed on these objects and later on evaluate its search tree to find if anything similar matches. The AI player can make a suggestion through the chat functionality or by answering questions asked by other players, a technique

1 Cube. Natali, Vincenzo. Movie. http://www.imdb.com/title/tt0123755/

known as “interrogation”.

1.4. Goals and Challenges

Within the scope of the project the general goals can be classified as follows:

- Maintaining the game-state consistent,
- Ordering the events,
- Increasing interactive responsiveness,
- Providing realistic rendering of 3D components,
- Integrating artificial intelligence into the game.

The above functionalities can be used in setting up the roadmap for developing the MMOG for the senior project. Other goals are also present, but we will just provide them as challenges. In our design we will try to leave some open doors to the following items:

- Displaying animations,
- Scheduling computations across players,
- Providing authentication/authorization mechanisms,
- Implementing a cheat-proof design.

1.5. Comments on the Final Design Procedure

1.5.1. Work from the Previous Phase

With our initial design – whose details are thoroughly explained in the following sections of the report – we have covered mainly the basic goals. Our architecture allows:

- The game-state to be kept consistent and it contains modules for the ordering of various events.
- Interactive responsiveness to be increased by dividing the processing on the client application into separate modules each of which with distinct duties. The network layer, the game engine, and the graphics processing layers are all separated, with minimal interfaces with each other. In our opinion this should provide us a good deal of flexibility when it comes to tailoring the product to our needs.
- Realistic 3D images to be rendered. There seems to be some problems with rendering too many components, but they are planned to be resolved before the prototype demonstration.

Currently we are facing some problems with artificial intelligence design and implementation. Therefore what we have done was to include the AI in our design the way we have predicted it to be and the way our research data forced us to do. The relevant components of our system are associated with our AI components so that we do not face integration problems in the future. The interfaces are again kept minimal and abstract so that modification costs are minimized.

The challenges such as displaying level-end animations, scheduling computation across players to distribute workload, authentication/authorization mechanisms and a cheat-proof design were not dealt with in the initial design. On the other hand our approach to each of those items will be:

- Level-end animations can be defined through a set of static rules. Animation processing can completely be done on the client application. All that the client needs to do is to obtain these static rules at start-up. Other static information is already being sent to the client. Therefore, including animation data will not cause too much of a disturbance. Once the animation data is loaded, the client should be ready to process level-end animations. The graphics engine is modular enough to support animations. It is already intended to support some in-game animations.
- Instead of trying to schedule computations across players, one might consider dividing the server into a group of servers and scheduling the computations across each member of the group. Implementing servers with different functionalities could be one approach or implementing servers that serve a smaller subset of the whole "game world" could be another. If necessary, we are planning to take the second approach. That would definitely increase the number of concurrent players in the game. Although in that case there would exist synchronization problems, they can be solved by using a global database that is accessible by each member of the server group. In our initial design, we have already included the interface for database connections. Based on our design, it does not make too much difference if the Server instance processes the whole world data or a subset of it. That is one of the major advantages of our scenario – that we have distinct rooms in the Cube –.
- Authentication / authorization mechanisms are not yet included in the initial design. To implement such a feature, one method would be to use public/private key encryptions. Fortunately, such an add-on will not cause costly modifications to be made. The communications are done via the Message objects that are an abstract representation of what is being sent and received by the connected nodes. Authentication and authorization can be implemented just by changing the structure of the message object to include key encryptions.
- A cheat-proof design is too much of a challenge for us considering requirements prior to that, therefore it will be neglected for the rest of the project.

1.5.2. Recent Progress

During the final design phase of our project, we have enhanced our initial design both by resolving errors from the previous phase and by adding the relevant detailed design and implementation parts of some additional components to the report. Consequently, the following tasks have been accomplished:

- Based on the first-hand experience obtained from the prototype implementation, the interfaces that realize the integration between the two major components of the game: the Network Backbone and the Game Engine is enhanced.
- The Game Engine is made more modular by describing everything as an event: input received from the keyboard, a request to join the game, player-object interactions, etc.
- The responsibility of the "Message Encoder" and the "Message Decoder" classes were

enhanced. In other words, now, they do not only deal with the core messages that are transmitted during game-play but also with chat messages and serial object instance messages.

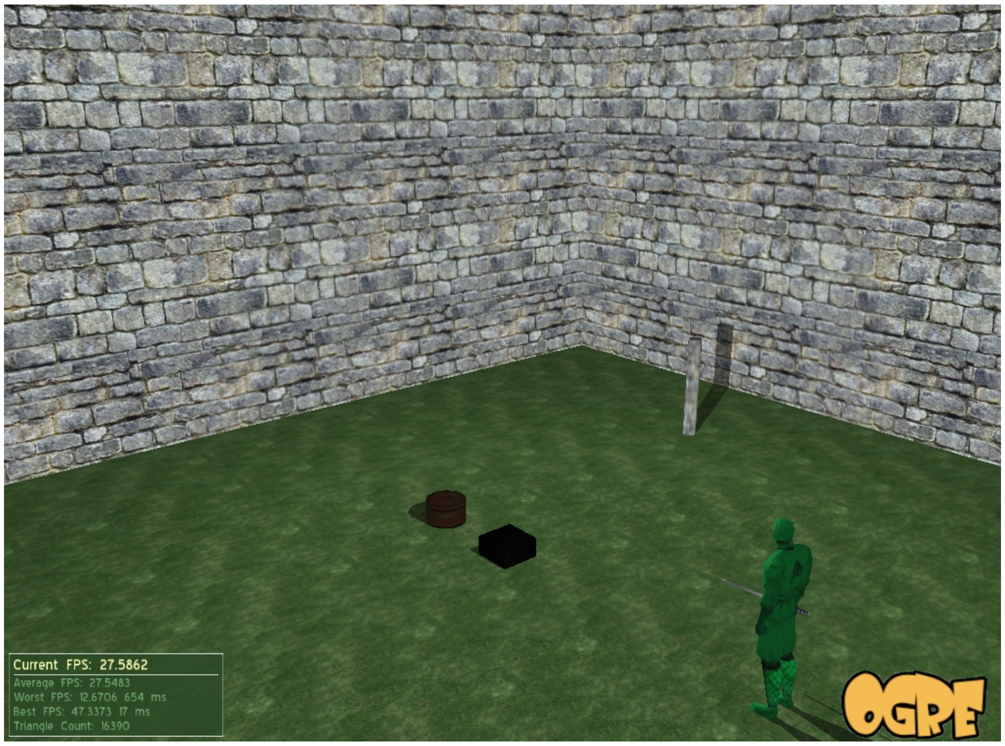
- The Graphics Engine is made more modular by introducing the "Animation Engine" module. It is capable of simulating the events both introduced by the player itself via I/O device input and the events that are received from the server describing the actions performed by other players.
- The Chat Module -one of the key components in player-player interactions- is introduced. Its design and implementation details are expressed via several UML diagrams and verbal descriptions.
- The roadmap for integrating AI into the game: how ontologies will be utilized in deduction and resolution of rules and how the clients will be notified of these deductions is explained.
- Based on the feedback received from our advisors, a technique for the deployment of the puzzles into the game is developed. It is based on expressing the events involved in a puzzle as a chain of rules and making use of the ontologies upon which the puzzles are developed.

1.6. Current Status in Prototype Implementation

The fact that we have started the implementing our prototype, we were able to see what issues there were to resolve for the initial design of our project. The implementation evolved in two distinct disciplines: Network and Graphics. The work accomplished so far can be summarized as follows:

For the Network core, a similar architecture involving the "Channel", "Connection", "Stub" and "Skeleton" concepts was implemented. Although a parser for every type of message was not written, the available model implements the encoding and decoding of movement messages. Whenever a player moves in the room, that information is encoded via a message construct and sent over the network for transmission. The server is responsible for the distribution of that message to every other player in the same room. With the implementation, we had the chance to see the possible consequences of network lags and overloads and could modify our design accordingly.

On the other hand, we have built upon our existing Graphics implementation and enabled some animation (walk, jump, crawl, etc.). Whenever a player interacts with the environment via the I/O devices, that information is sent to the "Animation Engine" and to the network backbone. This way, a player can see the movement of other players in the room.



```
Current FPS: 27.5862
Average FPS: 27.5483
Worst FPS: 12.6706 654 ms
Best FPS: 47.3373 17 ms
Triangle Count: 16390
```

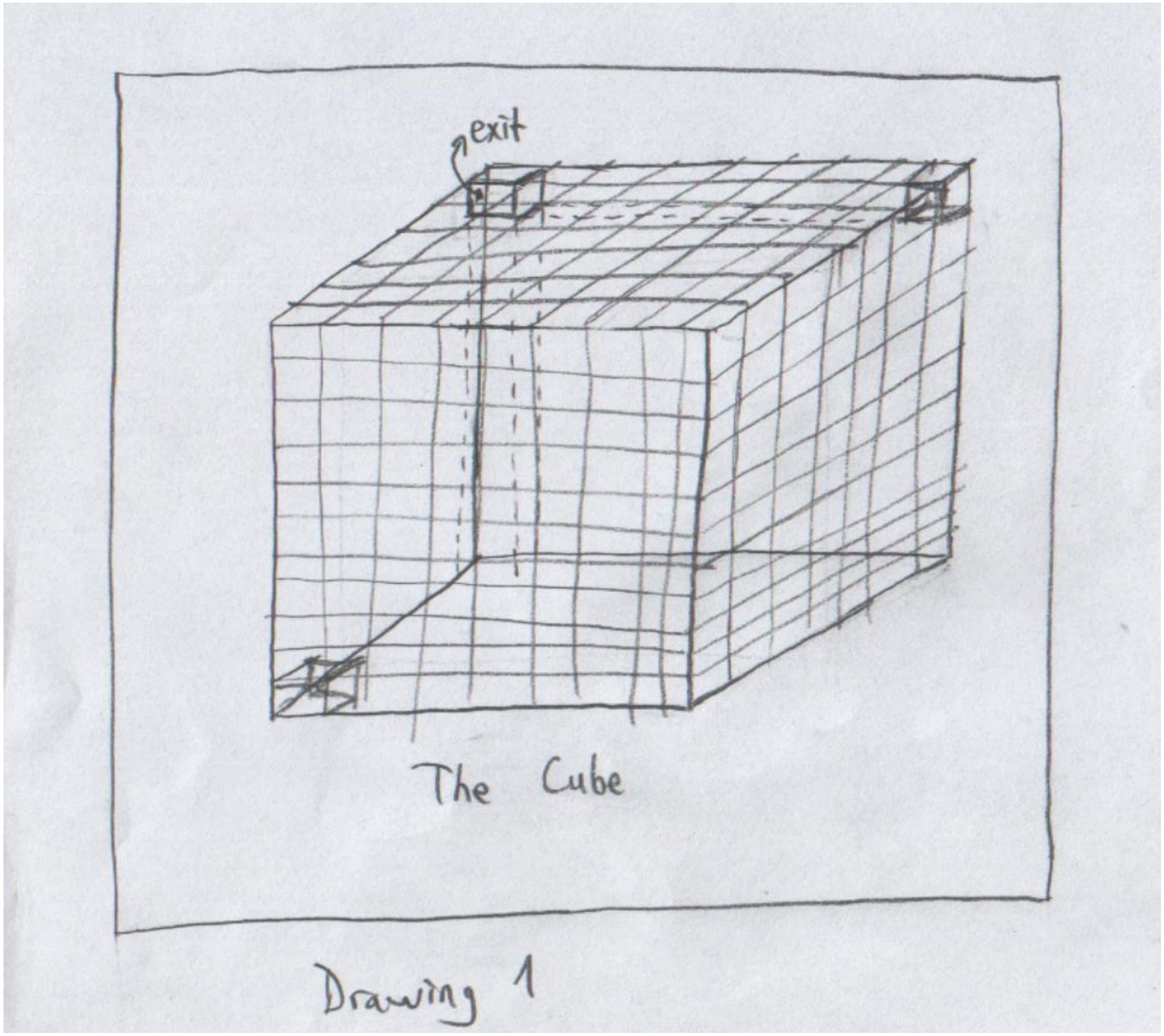
A scene from prototype implementation

2. Game Play and Story

In this section of our initial design report I will try to tell the details about the story and game play of the game. In order to make everything clearer, some explanatory drawings are also added. But first of all, I have to tell you that our game is a bit different than the usual FPS(First Person Sight) adventure games. Generally, in usual FPS adventure games, there is a predefined scenario, and all the events in the game flows through this predefined end case. Mostly, the events in the game are separated into levels and at least between levels the player can save the game and then continue from its last saved position. Game flow is generally incremental in these games. However, for our game the concepts of a usual FPS game do not apply. Our game is also a FPS adventure game but its being massively multi player online game changes its nature a bit. Since a lot of human players will be in the game, there will not be a scenario which is based on the main character as in the usual FPS games. Every player in the game must have an equal share in the story of the game. Otherwise, there will be conflicts between the players. So we have chosen a simple but an innovative story which does not have an incremental, predefined flow. The general story of the game is discussed in section 2.1 below. The drawings will help you understand the blurry parts of the story.

2.1. General Story of the Game

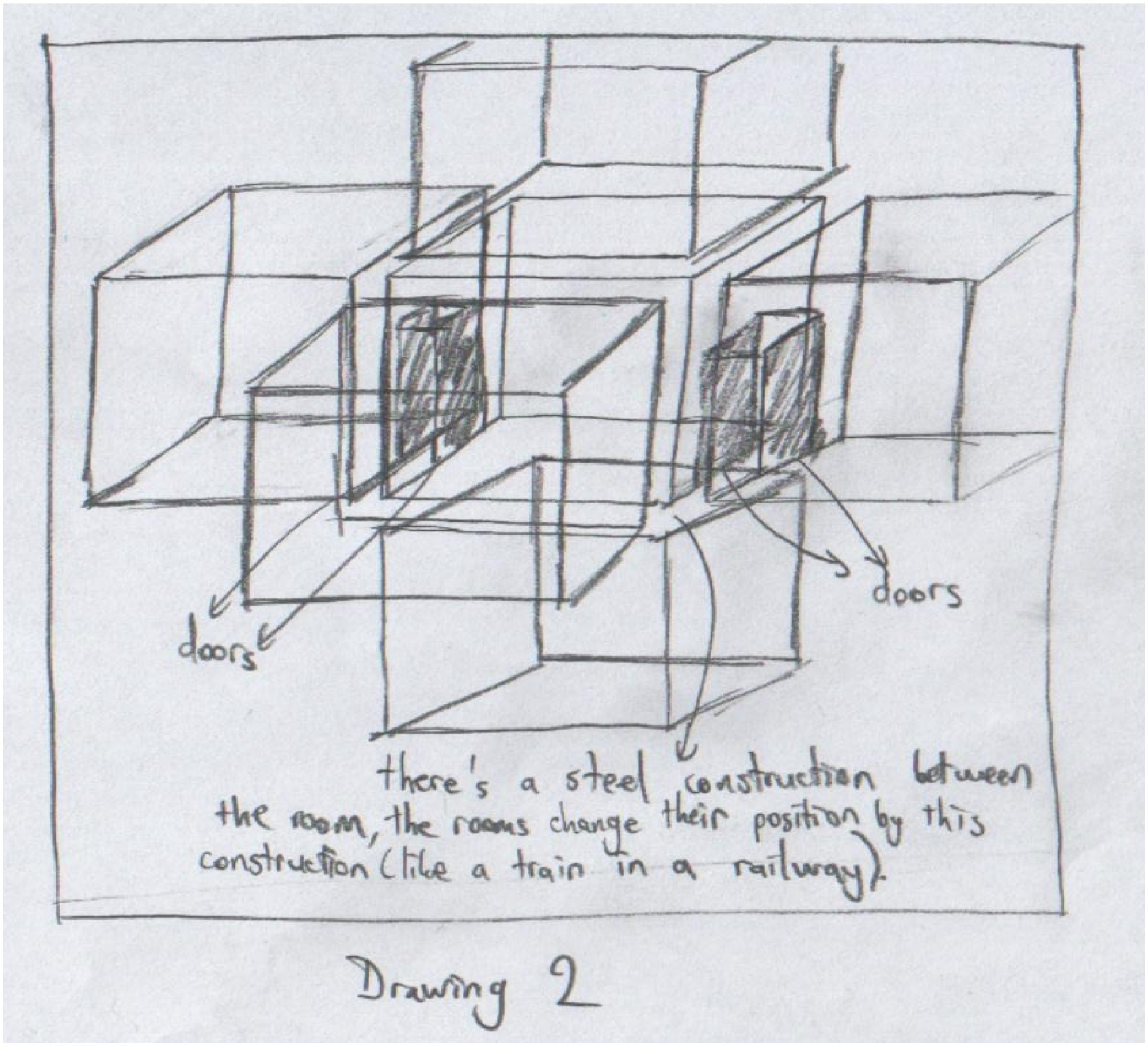
In this part I will try to explain the general story and game flow of the game. Our game is based on a thrilling Canadian movie 'The Cube'. The main idea of the game is very much alike with the movie 'The Cube', to get out of a smartly designed cube. Our game's story starts before the player's get into the game, where 4 crazy but absolutely very smart scientists wants to feel the emotion of controlling people and to experiment the survivor nature of the human. Actually they want to feel like God for sometime; to create a universe for at least some people and see what happens when these people get into this universe. For time being lets call these league of extraordinary smart but crazy scientists' team 'GOSO'. After sleepless nights and spending millions of dollars GOSO started the design of an evil universe for their experiment. This universe had to be very smartly designed and GOSO don't intend to let their humans escape so quickly from this universe, but also they are merciful enough to let the most talented and smart humans to get out of this hell-like universe. After many years, GOSO finished their universe, it was a cube consisting of many rooms which are connected to each other. But these rooms are not stable, they are constantly changing their positions in the cube, nobody knows the exact number of the rooms in the cube, because GOSO decided to make their masterpiece design better by adding the functionality to add and remove rooms from the cube whenever they want. Did I say that they were totally crazy? The cube has only one entrance and exit and this one way exit is not stable since the rooms are not stable too. The room which leads to exit changes every time, there may even does not exist a room in the place of exit for some time, but be patient, the time will come when a room come to that place.



Drawing 1

GOSO built this hell-like cube underworld, only the entrance(which is also the exit) is above the soil. So as you understand it is impossible to get out of the cube without finding this one exit. The players are supposed to go from one room to another until they find the exit. But it isn't as easy as it sounds. Most of the rooms contain puzzles and traps which makes everything more complex and harder for the players. Most of the time the doors are locked and can not be unlocked without solving the puzzle of that room. Puzzles can be harder or easier, there are even some puzzles that can only be solved by using some items which are found in some other rooms. Solving the puzzles will be the most enjoyable part of our game, we hope. There are also some deadly traps in the game. GOSO put these obstacles to see the survivor nature of the players. Luck can play a very important role in the game. A player may choose not to go to any other room after solving the puzzle in the room and unlocking some doors of the room, instead he may choose to wait until the room comes to exit and walk away to the fresh air of the outside world. But what if the next room you didn't enter was the exit room. One can get so mad that he can not resist trying the next room. Yes, GOSO wins again, they want to get you mad. You may get mad while going one room to another, this cube is like an endless maze, since it always changes.

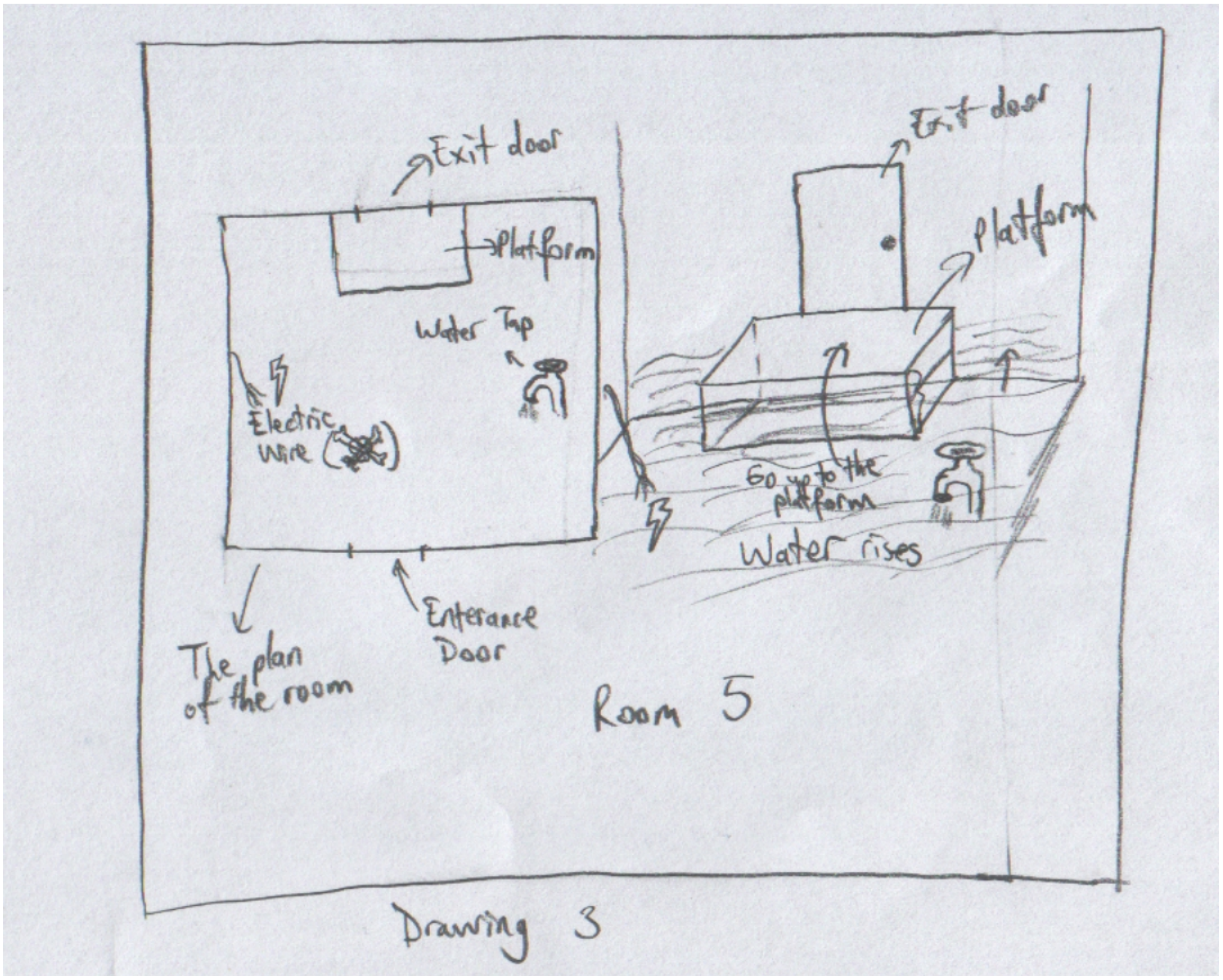
The story has some mysterious parts. The players start in a room in the cube. None of the players knows how he came to that room or who got him to that room. And probably they wont have a chance to learn this for their whole life. In a room generally there are 5 or 6 players. But a group of some other 5 or 6 players may also come to the room they are in. The core of the game story is the puzzles and traps of the rooms. So I will explain some of the puzzles and traps to give you some idea.


doors
doors
there's a steel construction between the room, the rooms change their position by this construction (like a train in a railway).
Drawing 2

Drawing 2

2.2. Puzzle1

In most of the rooms there are puzzles to be solved in order to get out of the room. In this puzzle there are two doors in the room; one of them is the door which the players have entered, the other one is locked. The locked door is a bit higher from the ground, there is platform in front of this exit door which is half a meter high(see the drawing below). The dimensions of the platform is also given below. There is a tap in the room, from which water comes when it is opened. The tap can also be closed to stop the water. There is also a high voltage open electric wire in the room , which is on the wall and 25 cm high from the ground. To touch the wire is deadly, not recommended. When the tap is opened the water starts to fill the room. The players can get on the platform while the water is flowing. When the water fills the room 25 cm, the wire touches the water and players who are not on the platform dies because of the electric shock. When the water fills the room to the level of platform(50 cm), the water stops filling the room and the door is unlocked. The players are free to get out of the room. If the tap is closed before the water level reaches to the platform, nothing happens.



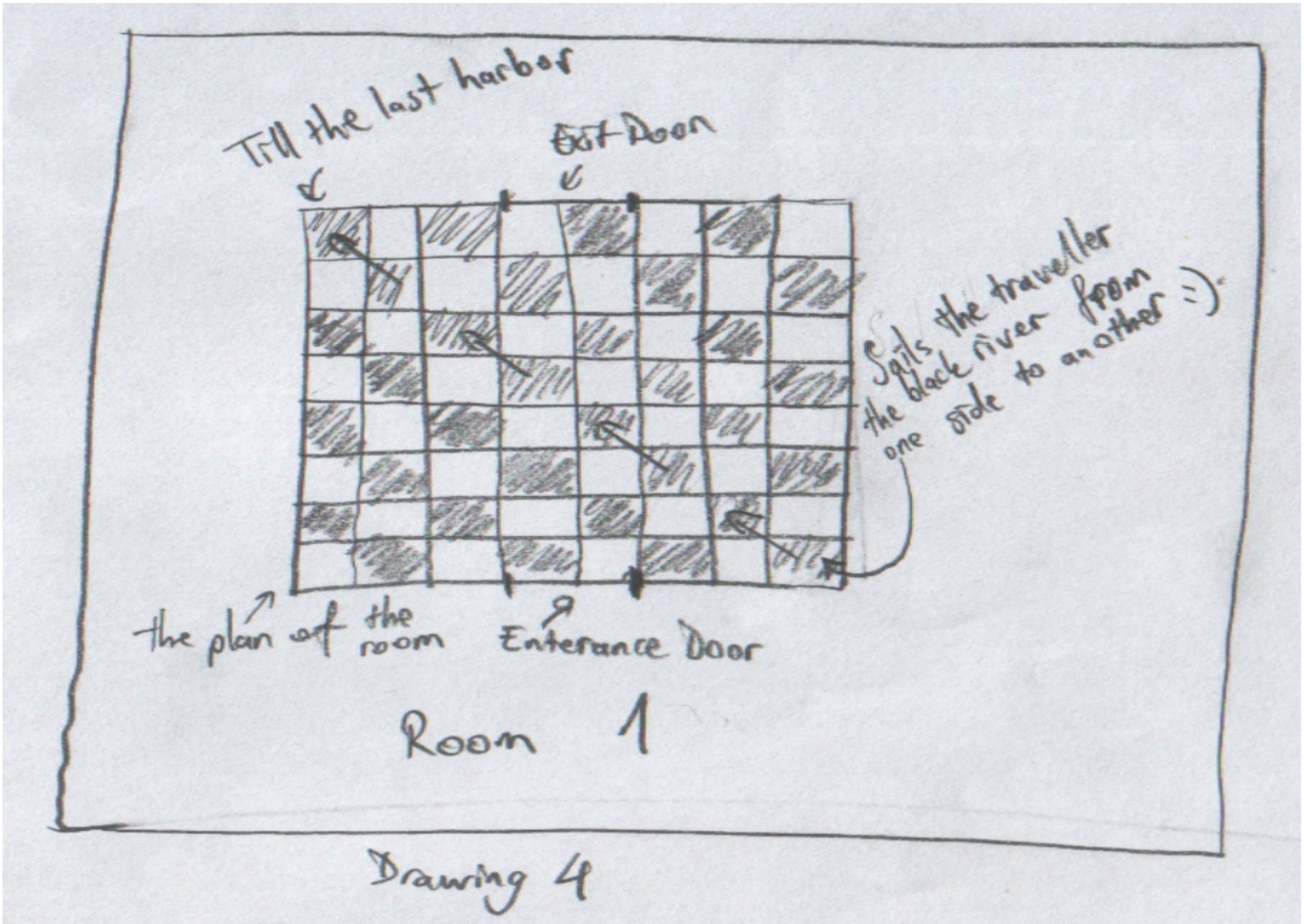
Drawing 3

2.3. Puzzle2

In this puzzle there are two doors in the room; one of them is the door which the players have entered, the other one is locked. The floor is like a chessboard. There are 32 white and 32 black squares in the floor. And there is poem written above the exit door:

> Sails the traveller the black river from one side to the other
> Can not see the end of his journey until the last harbor
> But must cross the black river with patience from one corner to the other
> Without disturbing the calmness of the river

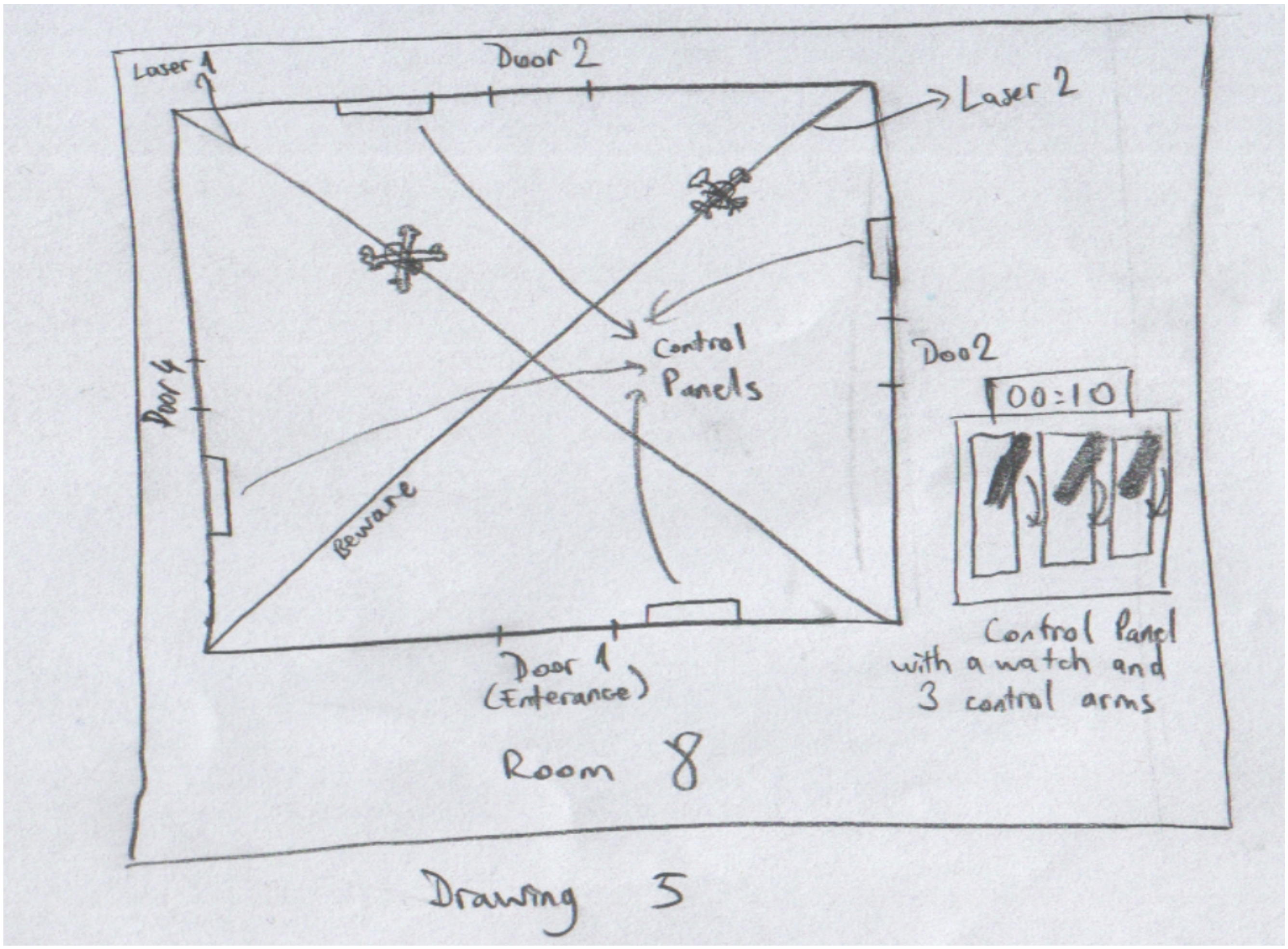
This poem is the key tip to solve the puzzle. The locked door can only be unlocked by walking on the 8 black squares from lower right corner to upper left corner (this is our black river). All these 8 black squares must be walked step by step and without interrupted with another square, which means if someone also walks from one square to other square, the chain is interrupted(the calmness of the river is disturbed). Everyone must stand still till someone crosses all the 8 squares from one corner to the another. If the chain is completed without interruption the door is unlocked. The players are free to get out of the room.



Drawing 4

2.4. Puzzle3

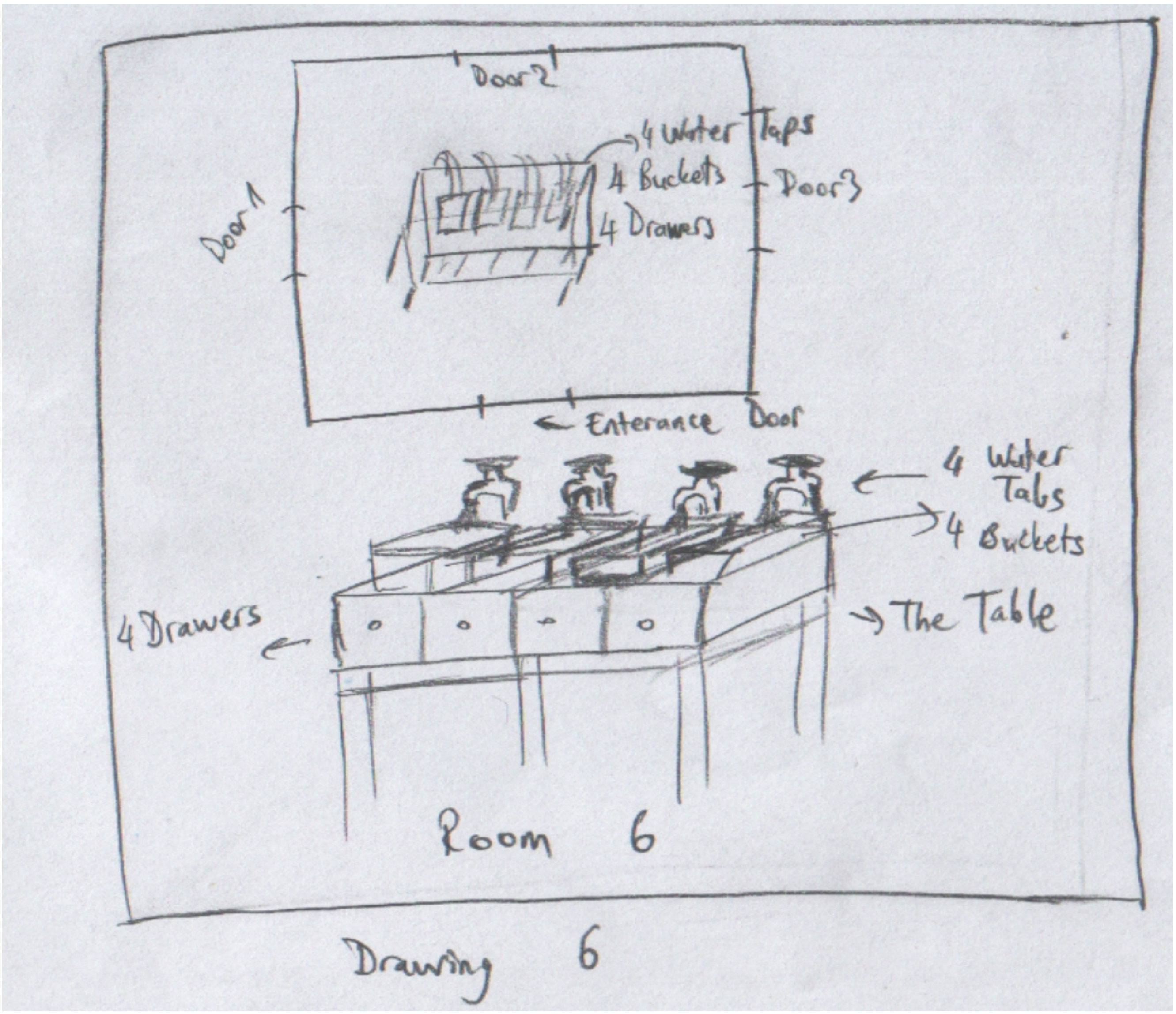
In this puzzle there are 4 doors in the room; one of them is entrance. When all players get into the room all doors are locked including the entrance door. the lights are on in the room at first, there is light button in the wall, when it is turned off the room gets dark. There two laser walls in the room each of them passes from one corner of the room to the opposite corner. These lasers can only be seen in the dark. And interaction with these lasers is fatal. There are 4 panels in the walls, each consists of 3 arms all of which are up and a LED screen watch which is set to 10 seconds. The first arm closes the first laser wall, the second arm closes the second laser wall, and the third one unlocks the door which is at right of the panel. Each time only one arm in the room can be pulled down and whenever an arm is pulled down the watch is activated and starts to count from 10 to 0. When it reaches 0 the arm go backs to its position and everything is reset in the room. So each time only one action can be made. In order to get out of the room, first or second arm must be pulled down to close the related laser wall. Then players must get to the right of the panel, but one of them must be left behind to unlock the door. The last one sacrifices himself for the team. One player can not get out of this room, since each door can only be unlocked for 10 seconds. Others are free to get out of the room whenever the door is unlocked.



Drawing 5

2.5. Puzzle4

In this puzzle there are 4 doors in the room; one of them is entrance. There is a desk and 4 water taps in the room. There are also 4 buckets in front of the taps(on the desk) to be filled and 4 drawers in the desk which are all locked. The doors are also locked. The taps can be opened to fill the bucket. Whenever a bucket is filled, the related drawer is unlocked. But other buckets must all be empty. The buckets can be unloaded and taps can be closed as well. When each bucket is filled on its own and the related drawer is unlocked, you find a key in the drawer. The key unlocks the related door. The key you find in the first drawer opens the first door, the second key opens the second door, the third one opens the third door, and the fourth key is a joker key to be used in another room, it is not used in this room. So you better keep it. Once a player opens a door with its key he is free to get out of the room.



Drawing 6

2.6. Game Play

This section briefly summarizes the general game play within our game. Details of the following can be seen in the requirements specification documentation.

Player is provided with the main menu when he first enters the game. At this state, he can customize the provided game options and continue his game in the cube. But to continue does not mean to continue from the last room he was in. Our game does not have save and load functionality. If you leave the game you can not continue from where you left. You start in a random room. Also to save and load game in our game is also meaningless. Because the rooms are not stable, their positions are always changing, so saving the game does not mean anything. If the player dies in the game, he does not die forever but could not get into the game for several time(we didn't decide the interval but it can be several hours). In the game, player takes place as a first person observer and at the same time he can monitor some game related information like player health and inventory. Detailed inventory will be provided whenever requested by the player. During the game, the player can interact with the objects and other players. The player can wander within the room, while picking up or using a variety of objects. Moreover, the player can talk with the other human or AI players private or public.

While in the game, the player can escape to the paused game menu. At this state, he can customize the options, view his elapsed time, resume the game or leave the game.

All the game is based on getting out of a crazy cube consisting of moving rooms. Rooms will change their positions every half an hour. And most of the rooms has puzzles and traps. There will also be Hall Of Fame section, displaying the most valuable(most lucky) players who succeeded in getting out of the cube in minimum time.

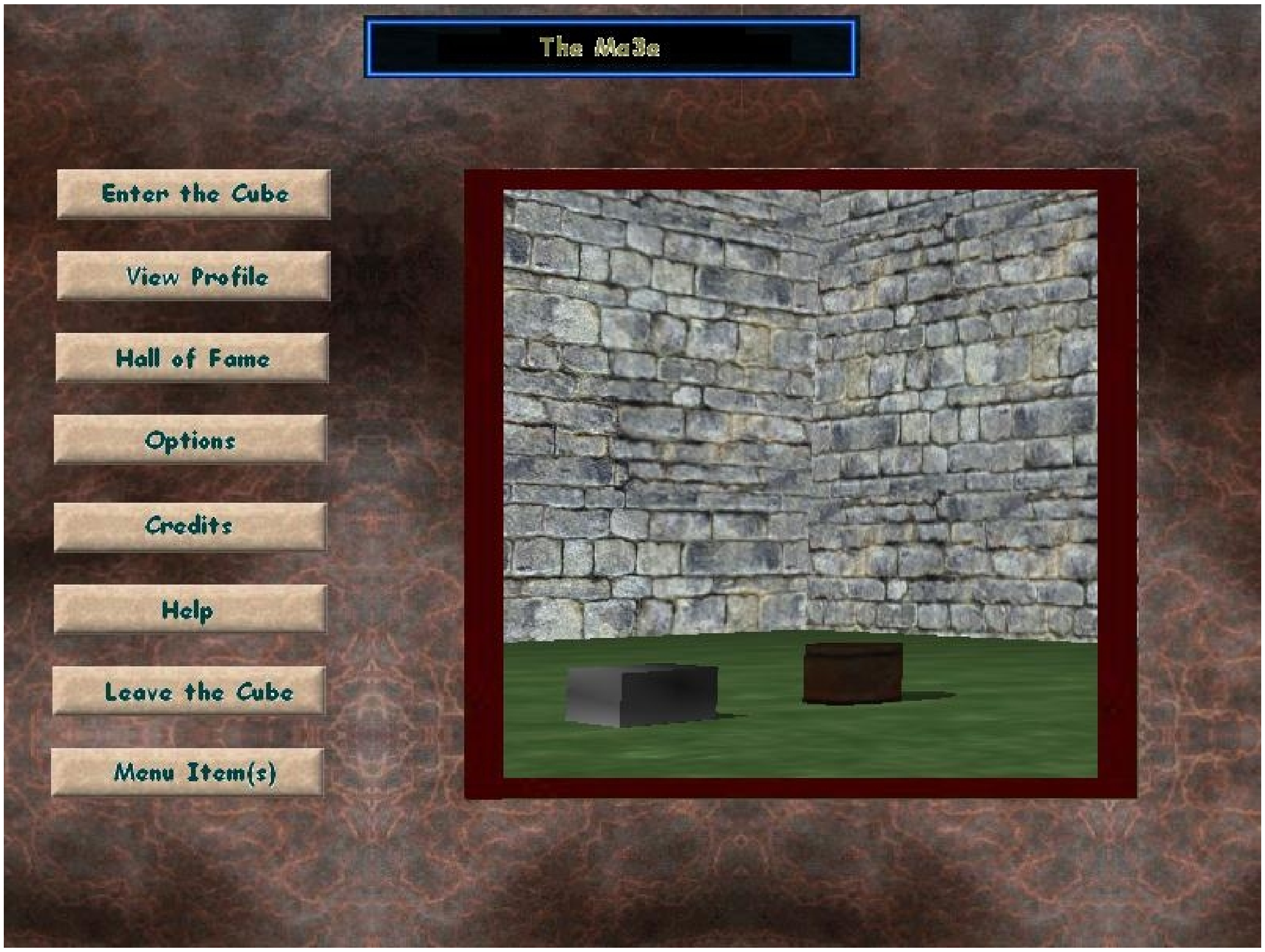
3. Interface Design

The following subsections explain the general interface designs for our game. The transitions between these interfaces are defined previously in the Game Play section of this document and in the Requirements Specification Document. Therefore we will not mention any details about them again.

3.1. Game Menu Design

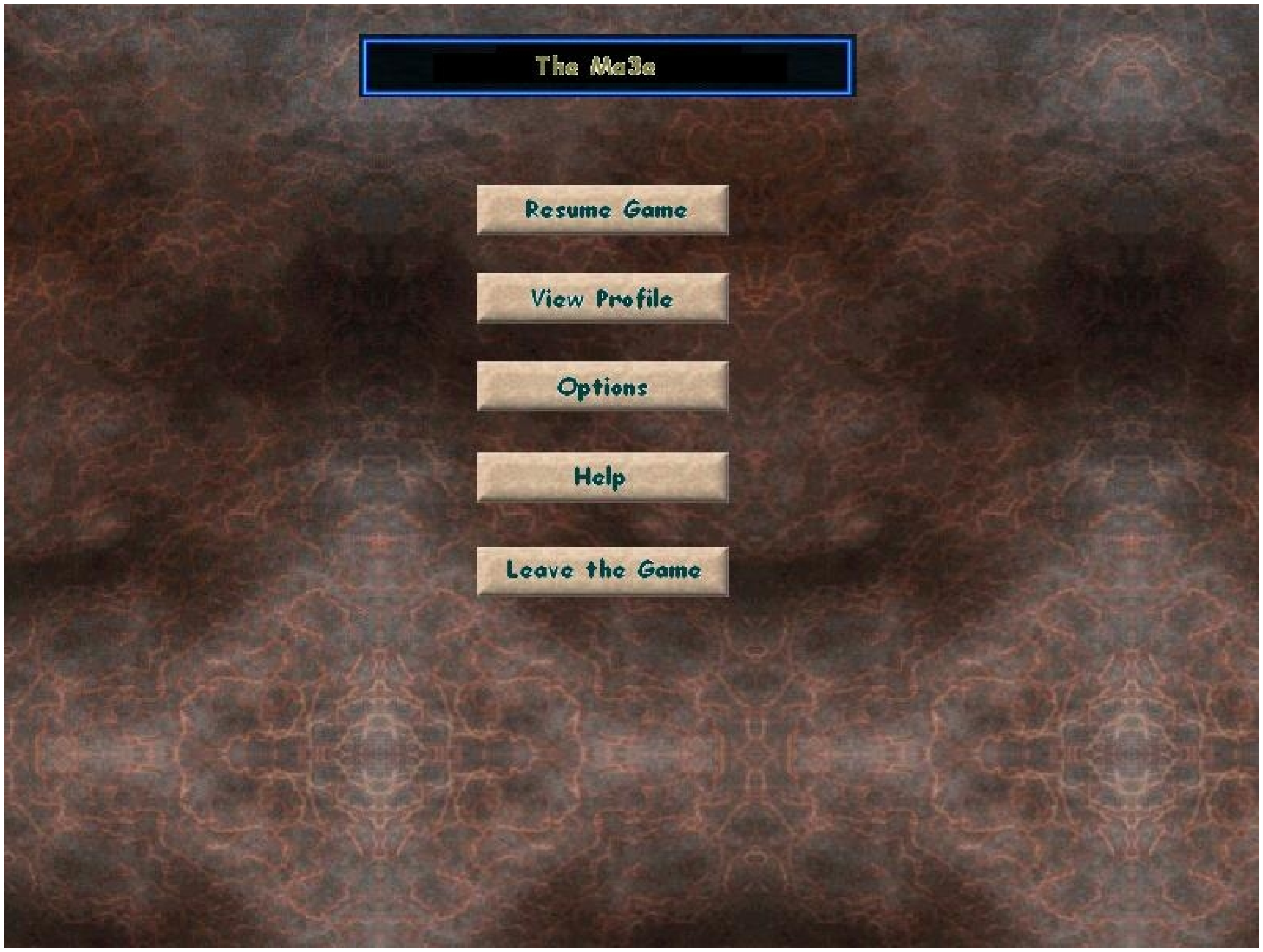
As stated in the requirement specification report, our game has two different types of menu: main menu and paused game menu. Each of them has some parts in common, but they have mainly a different interface.

Below you can see the Main Menu screen. At the top you can see the game's title. At the right part of the interface contains some introductory part for the game; these maybe some introductory text, some screen-shots or a video. At the left part of the screen you can access the menu items. The details of the menu items are stated in the Requirements Specification Document of the Analysis Report. These menu items are Enter The Cube(makes you enter the game), View Profile(gives the profile information of the player, elapsed time etc.), Hall of Fame(lists the most valuable players), Options(you can adjust the options of the game via this), Credits(gives information about the creators of the game), Help(gives helping information about the game play of the game, controls etc.), Leave The Cube(quit the game). Menu Item(*) is an optional button, if we want to add some new menu item to our menu we can add a new item there.



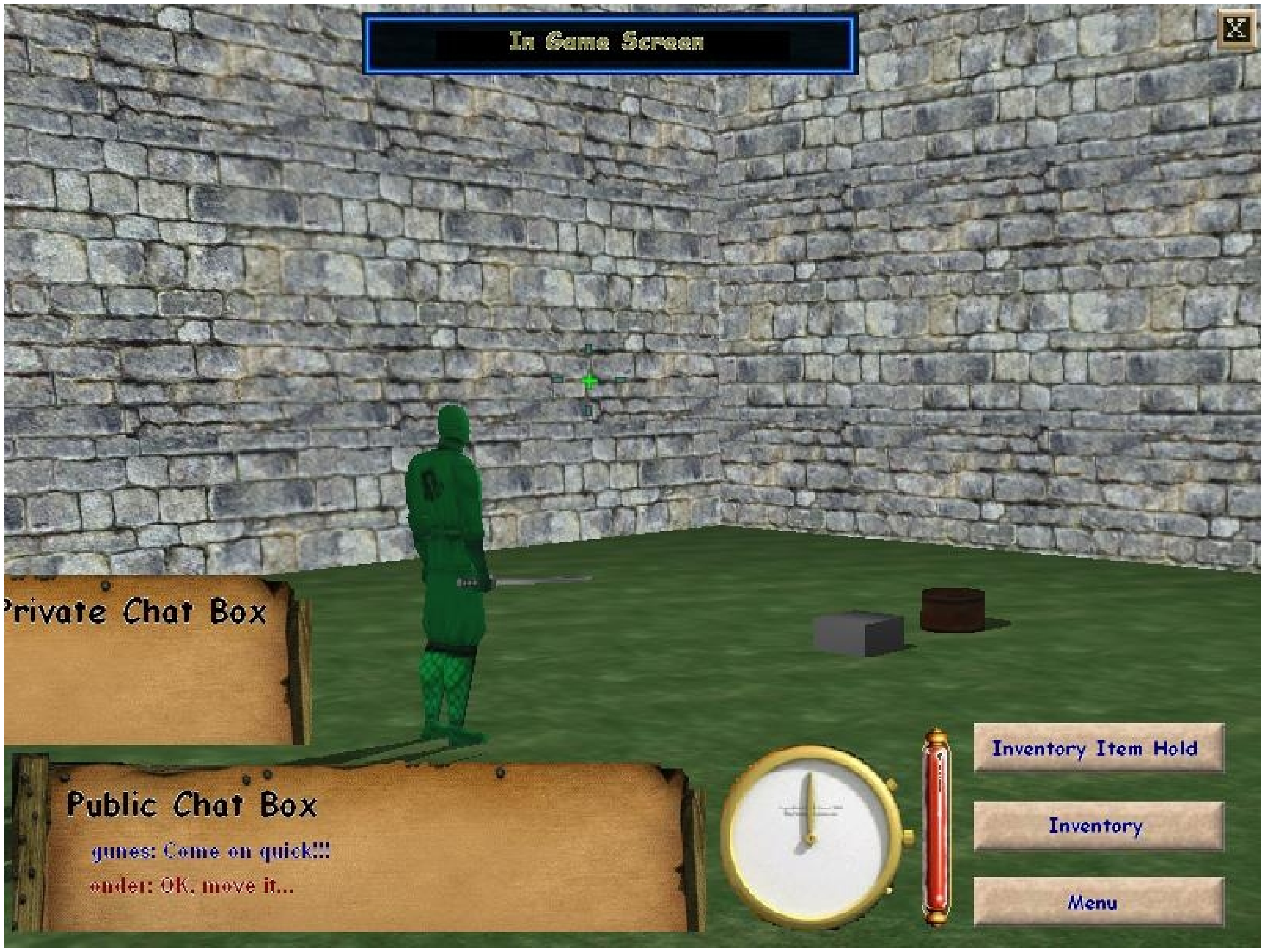
Main Menu Interface

Below you can see the Paused Game Menu screen. It appears in the middle of the game screen(we may change this part). At the top you can see the game's title. The details of the menu items are stated in the Requirements Specification Document of the Analysis Report. These menu items are Resume Game(you go back to the game), View Profile(gives the profile information of the player, elapsed time etc.), Options(you can adjust the options of the game via this), Help (gives helping information about the game play of the game, controls etc.), Leave The Cube(quit the game).



Paused Menu Interface

3.2. In Game Screen Design



gunes: Come on quick!!!
ender: OK, move it...

In Game Screen Interface

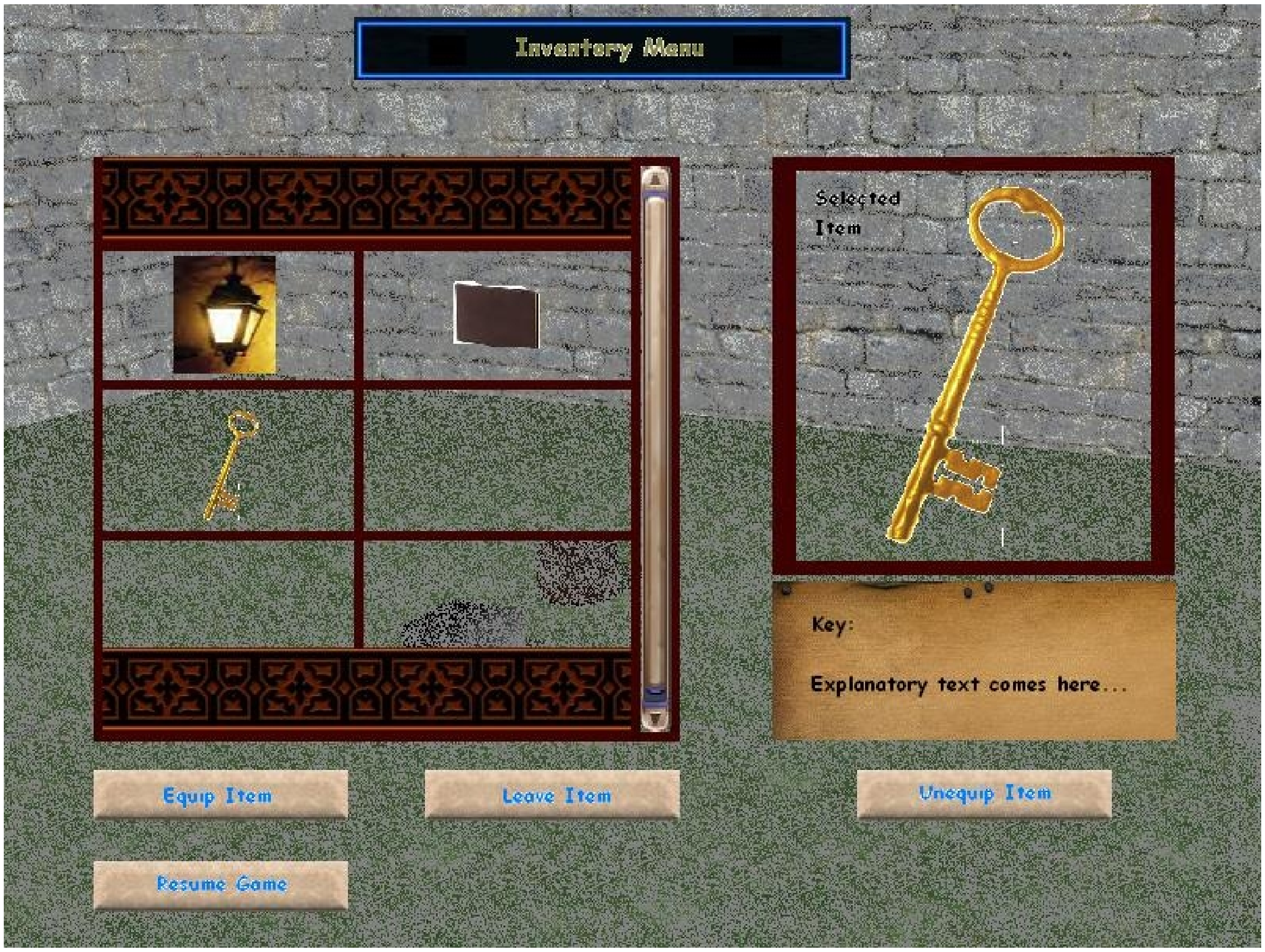
The above figure illustrates the In Game Screen design of our game. The current view of the player will be rendered to the full screen and the other content which can be named as heads up display are rendered onto this view.

The components of the heads up display which are located at the left shows Public and Private Chat Boxes. Public Chat Box is displayed continuously during the game and updated as soon as a change occurs. Private Chat Box is optional and it only appears when two players in the game talk privately.

The components of the heads up display which are located at the right shows the current health amount of the character, the inventory item which is currently hold by the character, an inventory access button by which you can go to the inventory and a menu access button by which you can ago to the Paused Game Menu.

At this screen the cross-hair is a very important component. This cross-hair always displayed at the middle of the screen. It can be used for inspecting an object, picking up an object or even using an object with another object. For each of these actions, this cross-hair changes its shape in order to express the possible action to the player.

3.3. Inventory Menu Design



Inventory Interface

The above figure illustrates the Inventory Menu design of our game. In the game when the player requests to view the character's inventory, this menu will be displayed. At the left hand side of this menu the inventory items that are in the inventory of the character are displayed as list. This list will include each item rendered at a small scale. Obviously, the list can have a number of items more than that can fit in the screen, so a scroll box is supplied just at the right of the list. The player can navigate through the inventory items via keyboard or mouse. Below there are Equip Item(to equip the selected item) and Leave Item(to leave the item to the ground) buttons. During this navigation, current inventory item will have a thicker border and it will be rendered at a larger scale at the right hand side of the screen. Below this larger render, name of the item and a brief explanatory text about the item will be displayed. This explanatory text will be optional which means that it may or may not exist according to the current item. Below there is a Un-equip Item button(to un-equip the current equipped item). At the very below There is Resume Game button to go back to the game.

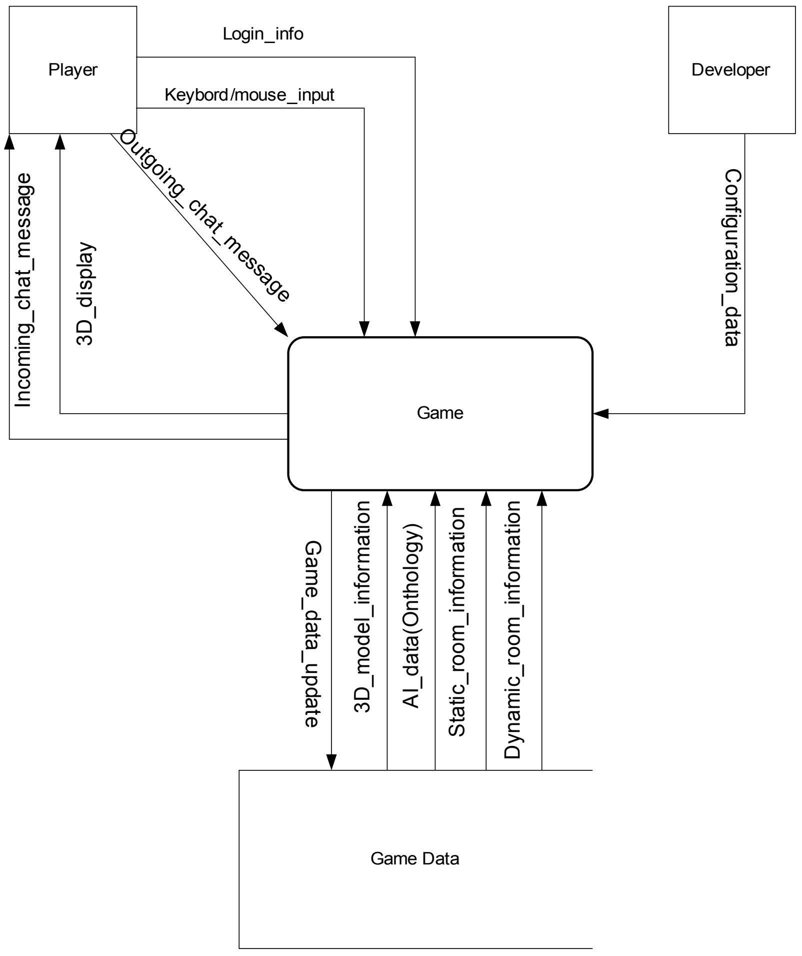
4. Game Architecture

4.1. High-Level Operation of the System

Our system consists of separate but highly interoperating modules, as can be seen from the following diagrams. The abstraction of related subsystems recognized from data flow diagrams into modules has led to a well-designed architecture adopted for the project, which is clearly explained in the following section.

4.1.1. Abstract Data Flow View (Level:0 DFD)

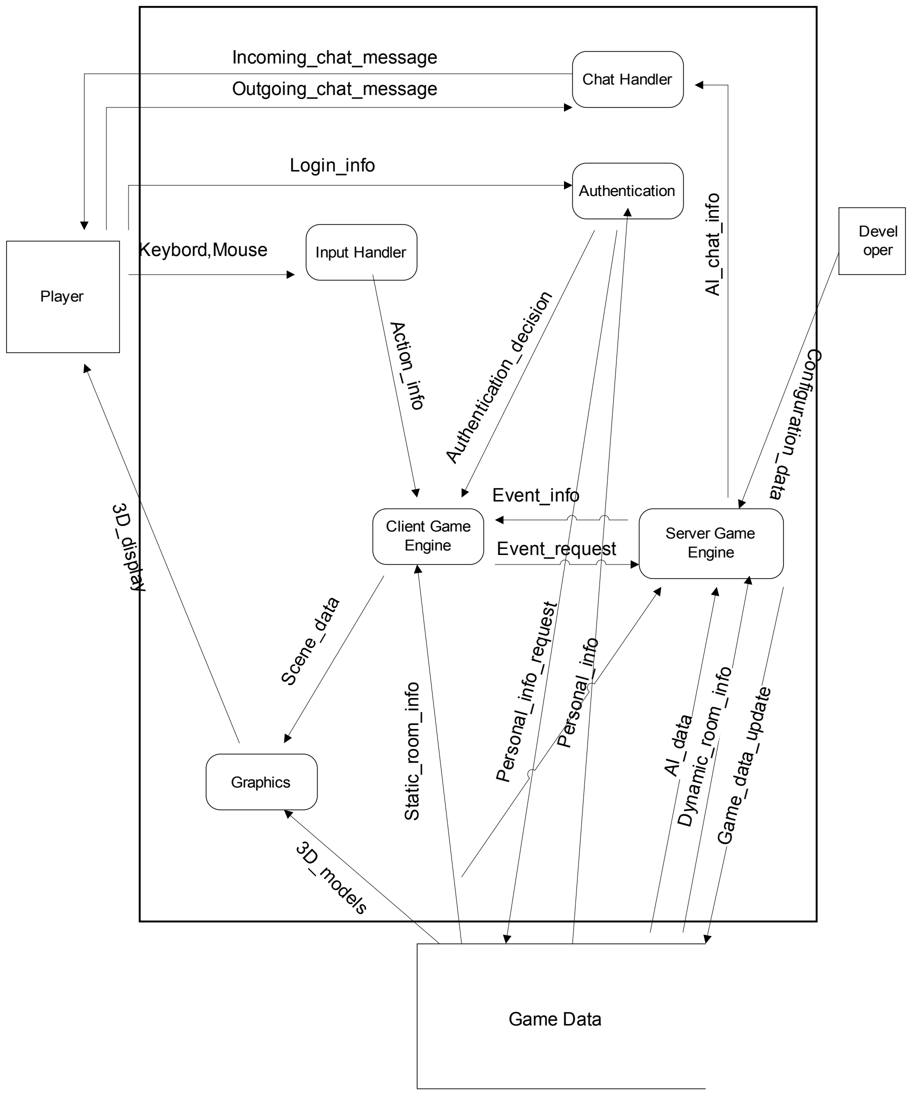
Level 0 DFD is the system in the highest level. The whole software is referred as ‘Game’ process with which player and developer interacts with. All the necessary game data is stored in the game data repository. Player interaction involves keyboard/mouse and chat message inputs, and display and chat message outputs. Game continuously queries 3D model information, AI data, static/dynamic room information and updates the necessary game data.



Level 0

4.1.2. Game-Core Data Flow View (Level:1 DFD “Game”)

Level 1 DFD indicates the main functions of the game. Player provides the game with login info, which is authenticated and matched with the corresponding personal info of the player. This information is then passed to the Client Game Engine. Input Handler is the layer where player inputs are processed and converted into action information that is evaluated in Client Game Engine, and with the necessary room data from the repository, is combined to scene data. This is combined by Graphics module with 3D model data to construct the 3D scene to be displayed to the player. The actions performed by the player is transformed into events and sent to Server Game Engine by Client Game Engine, which also receives the incoming events from the Server Game Engine. Server Game Engine is the central part where events from clients are processed and distributed, highly using the Game Data Repository as well as the configuration data from Developer. There exists a further process of Chat Handler, which handles the chat message traffic among the human and AI players.



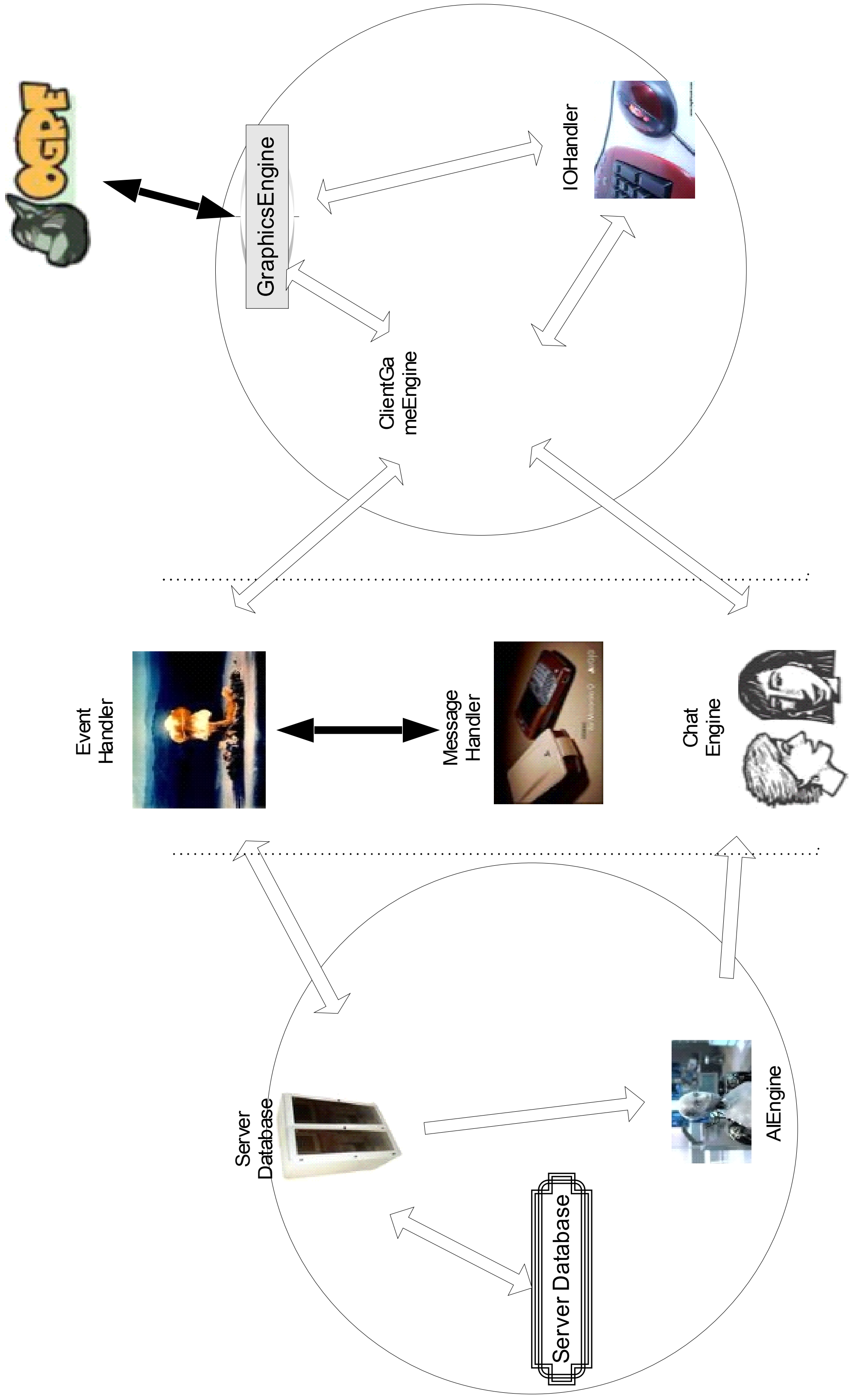
Level1 Game

4.2. Overall Architecture

The project is compartmentalized into packages, each of which consists of the classes with related functionality. The architecture of the project demonstrates a refined view of the system. Cohesion and coupling is aimed to be intensively employed to maximize modularity. This provides a good initial design with good opportunity to better express and even further extend the project functionality in the final design.

The game consists of two main parts, server and client game engines as the managers of the modules. Server game engine is responsible for database operations and event distribution, which minimizes the share of processing in the server side, allowing it to support large number of clients. Client game engine generates and receives events and holds a reflection of the game state on the server side, while an initial database connection for the static data to be loaded is also regarded. The interface between server and clients is purely "Event"s, but in the form of "Message"s. All the data necessary for a game state change is encapsulated within an event, which is encoded into a Message to be later decoded by the recipients. Graphical user interface and 3d display is achieved through Graphics Engine, which solely renders everything defined in a room in the client. Graphics Engine stands on top of the OGRE3D engine, as the higher level for 3d display. User input is handled through a IOHandler module, dealing with not only keyboard and mouse inputs for object/player interaction and 3d steering, but also text input for chatting among the players. Finally, AI Engine is the module that manages all the AI players in the game, possibly distributed among many rooms. Connected to directly Server Game Engine, which means being server-side only, AI players interact with the human players through the chat engine by sending them chat messages. The chat messaging facility is provided for the human players as well.

The interaction between the two major components, namely the Client and the Server, is achieved via the exchange of "Message"s. Coming up with an extensible, modular and yet lightweight architecture for the exchange of such "Message" objects is the key principle that will determine whether the outcome of this project is regarded as a **Massively Multi-player** Online Game but not just a Multi-player Online Game. A more detailed description of the underlying messaging architecture can be found in the subsequent sections.



Overall Game Architecture

4.3. Message Types

4.3.1. Overview

The format of the messages that are to be sent between the clients and the server are provided below. The table gives a grammatical description of the various types of messages. The full syntax that is to be used in the implementation will be provided later on.

Type	Priority	Field	Field	Field	Field	Field
synchronization	1	timestamp	player_id	state	current_position	current_direction
movement	2	timestamp	player_id	movement_type	to_position	{current_position}
direction change	2	timestamp	player_id	to_direction	{current_direction}	
interaction	3	timestamp	player_id	object_id	action_id	parameter_list
object-object interaction	3	timestamp	object_id	object_id	action_id	parameter_list
player-system interaction	3	timestamp	player_id	action_id	parameter_list	

Table - Messages

{}: represents that the use of the enclosed field is optional within the message. Message encoders and decoders are responsible for correctly implementing this scheme.

In addition to the messages described above, some other messages such as chat messages and serialized object instances are transmitted over the network backbone. The use cases in which the latter messages are involved will be discussed in the following sections.

4.3.2. Message Priorities

Whenever a player is involved in an interaction, other clients are notified of it via the exchange of messages. Delays and losses are inevitable if especially there is some network overload. Assigning priorities to messages and implementing the necessary enforcements is a step taken towards achieving game-state consistency.

As depicted on Table-Messages, synchronization messages have the highest priority. The reason of choice is simple. The message is used to synchronize the position and direction of a player. It should have a higher priority than simply a movement message or an interaction message. Similarly, the player would be more annoyed if he/she sees other players at wrong positions than if there is some delay in perceiving an interaction with an object. Consequently, interaction messages have the lowest priority.

Inevitably, the "Event Orderer" takes into account both the timestamp and the priority of messages when actually performing the ordering and validation. Algorithms for combining the two parameters will be deduced on a trial-and-error basis, during the implementation phase.

4.3.3. Description of the Fields Used

timestamp: a field that is used mainly by the "Event Orderer" during validation and ordering of events. Over a large network, there is no guarantee that messages will be received by the order they are sent. Thus, a mechanism for putting them in the correct order is essential. Before generating the desired message, the "Message Encoder" consults a trusted, synchronized timing authority for appending the current time value to the message.

player_id: In our model, events are mainly player-driven; that is most of the events are caused by the interaction of a player with other players, or a player with an object in the environment. The "player_id" field represents the unique identifier of the player involved in the event.

object_id: The "object_id" field aids the relevant authorities (such as the "Message Decoder") for identifying the object (or the subject in an object-object interaction) of the event that is generated during a player-object or an object-object interaction.

action_id: Unique identifiers for the actions that are invoked in any type of interaction are embedded in the relevant messages for guiding the authorities in simulating the action. Details of this scenario will be discussed shortly.

parameter_list: There may be cases when the invocation of an action requires several parameters to be passed as arguments. One can think of interaction messages as the basis for implementing remote procedure calls in our model. A thorough explanation can be found in the following section.

state/movement_type: Both the "state" and the "movement_type" fields practically resemble the same information but in different contexts. A player is allowed to make a movement of any of the following types (the list can easily be extended in the upcoming stages of the project):

- STILL,
- WALK,
- RUN,
- JUMP,
- CRAWL.

The "movement_type" field indicates the type of the movement that the player is currently performing. On the other hand, when this information is represented in a synchronization message, it is simply called "state".

current_position: the current physical location of the player is indicated via this field.

to_position: the physical location to which the player intends to make his/her movement is represented via this field.

current_direction: the most up-to-date direction, represented as a combination of the look-at-vector and the up-vector of the player, is indicated via this field.

to_direction: When a player wishes to change his/her current direction without making a movement (look up, look down, turn left, etc.) that information is contained in the message under the “to_direction” field.

4.3.4. Explanations of the Messages Used

- Synchronization Message:

  These messages are used whenever conflicts arise and player positions, orientations need to be synchronized. The message starts with the type tag and continues with the id of the player which needs synchronization. The state { UNDEFINED, STILL, WALK, RUN, JUMP, CRAWL }, the position and the player's direction are all supplied within this message. As in all cases, the message is timestamped with a trusted authority.

- Movement Message:

  The message indicates a change of position on a player. The player with *player_id* moves with type { UNDEFINED, STILL, WALK, RUN, JUMP, CRAWL } to the indicated position. The player's current position can also be supplied when necessary. That information is for synchronization purposes.

- Direction Change Message:

  The player indicated by *player_id* changes his/her direction from *current_direction* to *to_direction*. The *current_direction* field is optional and may be supplied for synchronization purposes.

- Interaction Message:

  Interaction messages are more generic than the others and they can be used for various purposes. However, the intended purpose of this type of messages is to let the users interact with objects in the environment. Objects have several functions that can be invoked in order to change their properties and also the properties of the environment. These functions can be generic, in the worst case taking various number of parameters whose types may also be varying. To overcome such difficulties, in our design, modules have been included such that an interaction message can be processed as a remote function invocation. The message starts with its type, containing fields such as player_id, object_id, action_id and a list of arguments.

- Object-Object Interaction Message:

  A restricted subset of interaction messages.

  Although an interaction is usually player-driven, certain scenarios require objects in an environment to be the initiators themselves. For instance, in one of our scenarios, water filling out in a room – when it reaches a certain level – triggers the uninsulated electric cable to cause an electro-shock on the players that are in contact. In this case, the water object itself triggers an action on the electric cable. The only way that every player in the room is notified of this event is via the exchange of messages, namely the object-object interaction message.

- Player-System Interaction Message:

  A restricted subset of interaction messages.

  Several interactions, which can be considered outside the core functionalities but are absolutely essential for a smooth game-play, need also be represented as messages and sent over the network. Such interactions are usually between a player and the system itself, as listed below:

  - A player requests permission to join the game;
  - A player is given the permission to join the game in a certain room, other players in the room must be notified of this event;
  - A player quits the game on his/her own will, other players should immediately be notified;
  - A player dies and is forced to exit, other players in the room are also notified.
  - Players in the room have completed the puzzle and request to move into a different room. Players in the neighboring room need to be notified.

  The "player-system interaction" message acts as a container for the information that is exchanged between the clients and the server in case any of the aforementioned events occur.

- Serialized Object Instance Message:

  When the player is admitted into the game, some static as well as dynamic data associated with the room in which the player appears must be supplied to the player's machine. Such information is again transmitted over the same network backbone, by means of messages. However, the message format is slightly different. The static data includes information such as the dimensions of the room, the contained objects and their relative positions. For the proper transmission of this static data, a mechanism is defined: any object has to implement our Serializable interface, and only then it can be transmitted through our Serialized Object Instance message. This flexibility is provided for future freedom of sending any object, but for that being we have only defined Room class to be Serializable, so that it can be sent for client game loading/initialization at the beginning of the game.

- Chat Message:

Chat Messages are defined as specialized messages for text transmission. The message mostly consists of the text message being sent, only with the exception of the extra part of sender id. {player_id | textcontent} message is then resolved at the client side, so that the id of the sender is matched against player name and displayed in front of the text content.

4.3.5. Achieving Game-State Consistency in a Chain of Interactions

The fact that player-object interactions are distributed to other clients in the form of interaction messages that are transmitted over a loaded network and that consequently the change-of-state of objects in each client machine is achieved separately in an event-driven manner creates a major synchronization problem.

For justification, consider the following scenario:

- Player A-E reside in the same room.
- There is a chest in the room. The chest can be opened, if closed; and various items can be picked out of it, only after it is opened.
- Player A opens the chest. The relevant interaction message is sent to the server for distribution to the other clients.
- The server processes the information, makes the implied change in its own global game state repository and sends back the message to the Player B-E.
- For some reason, the messages are lost on the way to two of the recipients (assume these to be Player C, Player D).

At this point, there is a major game-state inconsistency. Player C and Player D can only issue an "open-chest" action while Player A, Player B and Player E can pick items from the chest. If Player A now picks an item from the chest, both Player C and Player D will be confused on what to do, since the state of their own instances of the chest object do not allow such an action.

There are various schemas that can be implemented to tackle this problem:

1. Make sure that the effects are not visible in either client until every client receives the notification. In other words, Player A will not be able to pick-up an item from the chest until Players A-B are fully notified of the "open-chest" event.
2. On regular intervals, send various object-state-synchronization messages to every subscribed client.
3. Define action chains for the objects and issue the "Event Orderer" to resolve ambiguities. If Player C receives a message such as "Player A picks up Item X from the chest" then its "Event Orderer" realizes this by simply understanding that there was a problem with its own execution of actions and simply assumes the "open-chest" action to be completed.

4.4. Puzzle Deployment

One of the main challenges that face the developers of the project is to realistically implement the puzzles that have been designed up to now (and that will be designed in the future). Coming up with an abstract model to represent the puzzles would definitely be a time-saving accomplishment. Although, the modules and data structures that deal with the realization of the puzzles (Rule class; Action, Object entities etc.) exist; hierarchically, there does not yet exist a structure that encapsulates them all. Based on the feedback we received from our advisors, we have decided to include in our report, an initial view on how we will tackle the problem. In our opinion, the best solution would be to use a human-understandable – yet machine processable – resource to represent (and tie together) the building blocks of a puzzle. That way, developers need only work with such resources, which can later on be supplied to the system for further processing.

The resource is to be used by three of the major components of the game, namely: the Game Engine, the Graphics Engine and the AI Engine. The Game Engine will utilize it to simulate the effects of the action chains described by the puzzle. The AI Engine will work with the rules by which each of these actions are associated. Furthermore, the Graphics Engine will simulate the environment by associating the metadata included in the resource with some predefined object models.

Before going into the details of how the proposed resource can be represented, let us briefly go over each of the key concepts that are used throughout this section. For our case study, we will work with the following puzzle:

- There is a tap in the room. The tap can only be turned on. Once it is turned on, it will not be possible to turn it off.
- If the tap is turned on, water starts filling the room out.
- When water reaches a certain level, the uninsulated cable causes an electric-shock on anyone that is intact with the water.

Based on the given puzzle, the **objects** involved in the scenario can be identified as:

- Tap
- Water
- Uninsulated electric cable

Furthermore, the following **actions** are defined on each of the objects:

- Tap: turnOn()
- Water: fillOut()
- Uninsulated electric cable: createShock()

As it is observed, an action usually leads to another in a “cause-effect” fashion. **Rules** define how these actions are associated:

● Tap: turnOn()	<leads to>	Water: fillOut()

- Water: fillOut() + <condition>   <leads to>   Cable: createShock()

In our model, every object will have a **state**. Actions defined on an object will either cause a change of state on that object and/or on some other object(s) in the environment. The Game Engine and the Graphics Engine will view the state of the objects and do the simulation accordingly.

Unfortunately, the way in which these states are processed by the relevant engines will have to be hard-coded. However, if somehow the set of all states is kept minimal, then the amount of work will be reduced.

- ✔ The design choice we have made here is to use ontologies to describe various object classes and the possible sets of actions allowed. Furthermore, rules will be defined over various classes in these ontologies rather than individual class instances. If the puzzle designer and the developer restricts himself/herself with the definitions in these ontologies (and use as small extensions as possible) then the Game Engine, the Graphics Engine and the AI Engine can simply use the information provided in these ontologies for processing. Now, the puzzle designer can add new objects and actions by simply associating them with the predefined ontologies. That way the process of integrating new puzzles into the game and processing these puzzles will – to some extent – be decoupled.

The problem now is to describe an individual puzzle instance as a combination of predefined objects, actions and rules. One possible solution would be to use XML. From now on, the abstract language constructed will be referred to as X-PML (eXtensible Puzzle Modeling Language). X-PML will consist of two major sections: one that contains information for the Graphics Engine to process and the other for the Game Engine and the AI Engine. The first part would give information on the models used for the objects, their relative positions and some major properties. On the other hand the second part will reference the rules that associate actions which are defined over a set of objects. This approach is also reflected on the Database Design.

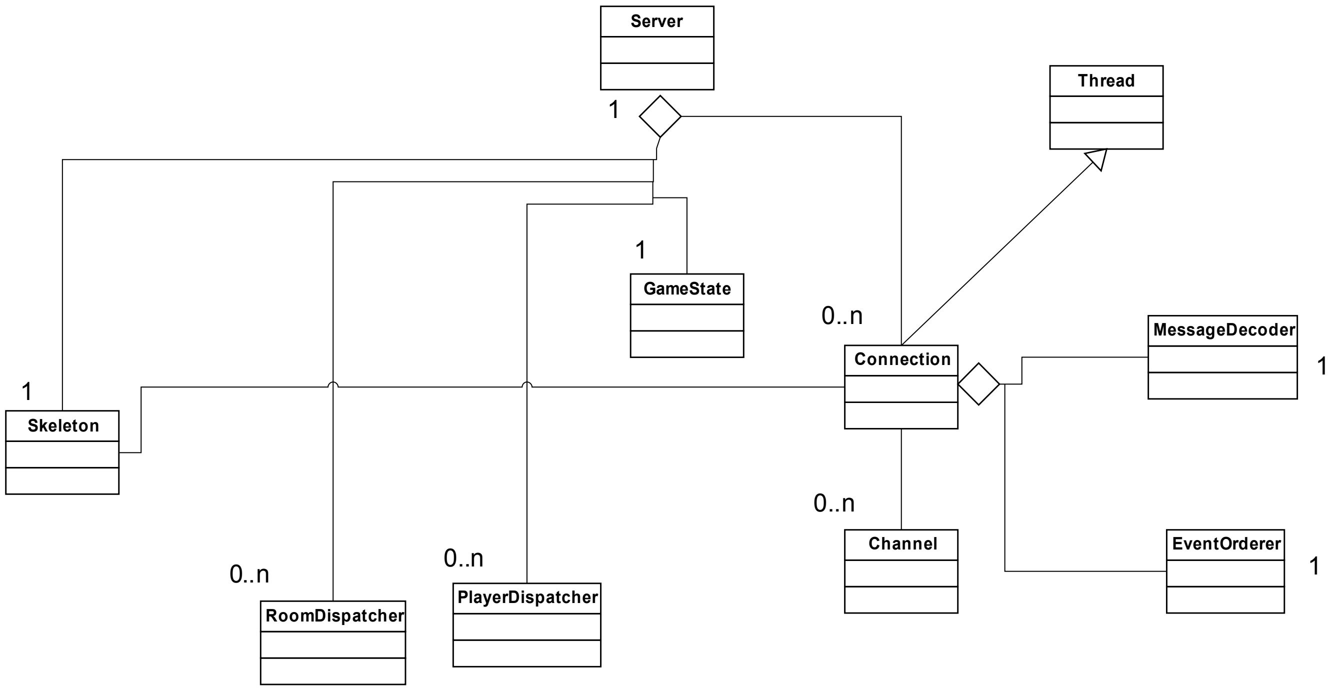
This is a draft idea as to how the puzzle deployment problem can be solved hierarchically. The work is to be extended and refined before the full implementation starts.

5. Detailed Design

We have sought to have a solid design, with as much compartmentalization as possible. The main modules, each encapsulated as "engine"s, prove to be highly interrelated yet mostly independent, granting the process a strong sense of modularity.

Our system is presented according to both static and dynamic views in the following subsections.5.1. Structural Design

5.1.1. Server-Side Network Backbone



Purpose:

It is a layer of abstraction on top of the network connectivity between the clients and the server. Eventually, everything can be viewed as an "Event" by the application layer of the server-side. This provides a great deal of flexibility in the amount of information that can be exchanged. Furthermore, it reduces the workload of the server-side game engine and leads to a loosely-coupled client-server architecture.

Abstract:

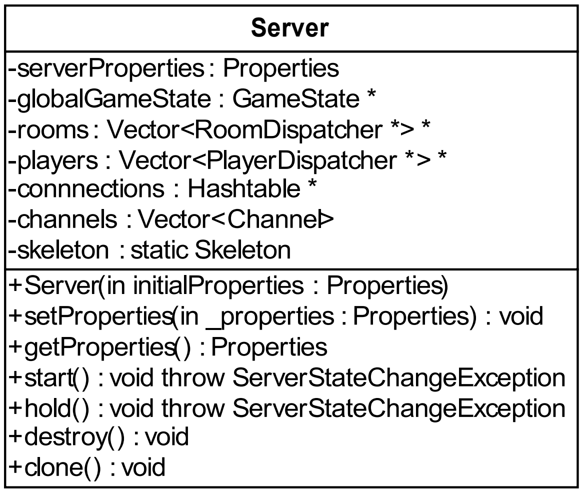
As you would remember, the game to be implemented in this project – roughly speaking – consists of various clients, a server and an architecture that serves as an interface for the underlying connections. Both the client and the server has the relevant modules to provide this connectivity. In other words, the application layer sits on top of the classes that deal with all the connections and resolve conflicts when necessary.

The package named "Server-Side Network Backbone" consists of the modules that support these functionalities. In other words, it is the server-side correspondent of the connectivity engine. When viewed as a whole, this package supports the following functionalities:

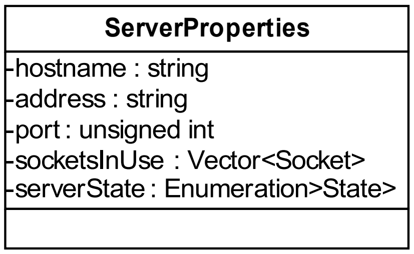
- provide a means of a seamless connection by abstracting out some networking constructs,
- send/receive serialized message instances as a means of connection with the clients,
- enable multi-channeled connections,
- implement the publisher-subscriber model by logically grouping the connected clients,
- encode/decode the serial instances into abstract "Message" entities,
- validate and order the messages,
- based on a sequence of valid messages generate "Event"s that make up the core of game-state changes,
- put the resolved information in the relevant data structures.

The following sub-sections give detailed information on how each of these functionalities are implemented. Various modules, the data structures used and the interaction between each module is thoroughly discussed.

5.1.1.1. Server



```
Server
-serverProperties : Properties
-globalGameState : GameState *
-rooms : Vector<RoomDispatcher *> *
-players : Vector<PlayerDispatcher *> *
-connnections : Hashtable *
-channels : Vector<Channel>
-skeleton : static Skeleton
+Server(in initialProperties : Properties)
+setProperties(in _properties : Properties) : void
+getProperties() : Properties
+start() : void throw ServerStateChangeException
+hold() : void throw ServerStateChangeException
+destroy() : void
+clone() : void
```



```
ServerProperties
-hostname : string
-address : string
-port : unsigned int
-socketsInUse : Vector<Socket>
-serverState : Enumeration>State>
```

This class makes up the core of the server to be implemented in the project. In the initial design report, a client-server architecture that supports the publisher-subscriber model is tackled with. Currently, there exists a single server for the whole system - the main reason being synchronization issues. If during the prototype implementation it is observed that a different architecture is necessary (a layered structure of server processes, a group of servers with different responsibilities, etc.) the current design will accordingly be modified. Here, the main criterion to consider would be whether or not the server supports a reasonable number of concurrent players.

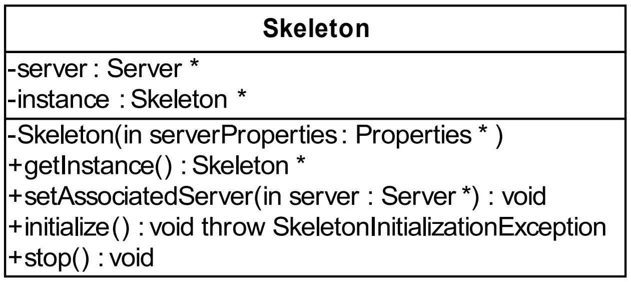
The "Server" class contains instances of classes that are responsible for storing dynamic attributes related to the game play. Such information includes but is not limited to the position and orientation of the players in a room, puzzles solved so far, positions of the rooms inside the cube, etc. GameState, RoomDispatcher, PlayerDispatcher are examples of such dynamic-attribute-storing classes.

Physical properties of the server are also stored in this class, namely by the serverProperties attribute. It contains information such as the network address, the host name used, etc.

The core functionality of the server is implemented by the Skeleton instance. The “skeleton” is responsible for listening to the port(s) to which the server is bound. Whenever a new connection request arrives from a client, the skeleton directs the processing to a newly generated thread whose details will be discussed shortly.

Inevitably, the server contains methods that help in starting, holding and destroying the server application. Furthermore, a clone() method is included in the design. Although, we do not believe that we will implement the functionality of moving the server to a different location while the game is in play, we still included this functionality in the design just so that it is not difficult to turn back if we change our mind in the near future.

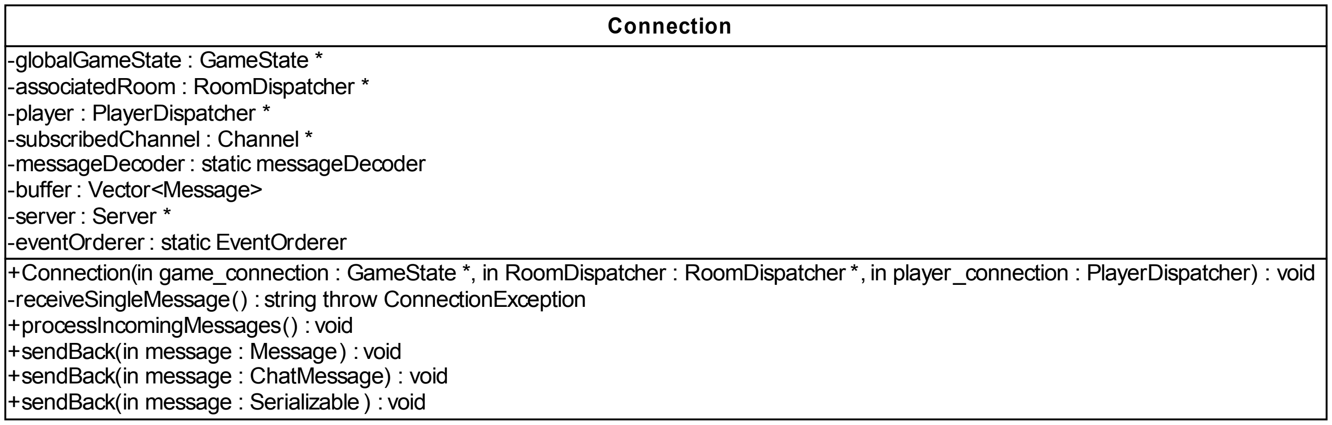
5.1.1.2. Skeleton



```
Skeleton
-server : Server *
-instance : Skeleton *
-Skeleton(in serverProperties : Properties * )
+getInstance() : Skeleton *
+setAssociatedServer(in server : Server *) : void
+initialize() : void throw SkeletonInitializationException
+stop() : void
```

The skeleton, as mentioned earlier, is the first point to which a connection request is sent. The client machine, based on its properties, is processed and the new player is added to one of the rooms. Furthermore a “Connection” instance is generated whose responsibility is to deal with any of the future messages sent back and forth between the client and the server. The “Skeleton” class should be implemented as a singleton, since the Server should contain only a single instance of “Skeleton”. Before its final version and after some implementation, this design decision will again be evaluated by all of the group members.

5.1.1.3. Connection



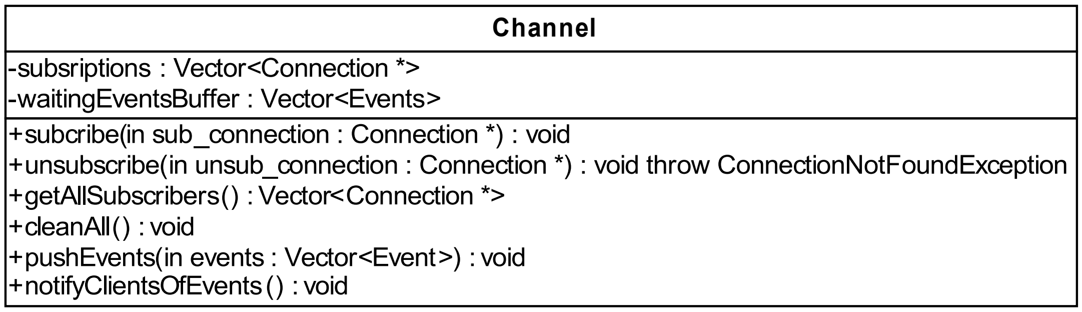
```
Connection
-globalGameState : GameState *
-associatedRoom : RoomDispatcher *
-player : PlayerDispatcher *
-subscribedChannel : Channel *
-messageDecoder : static messageDecoder
-buffer : Vector<Message>
-server : Server *
-eventOrderer : static EventOrderer
+Connection(in game_connection : GameState *, in RoomDispatcher : RoomDispatcher *, in player_connection : PlayerDispatcher) : void
-receiveSingleMessage() : string throw ConnectionException
+processIncomingMessages() : void
+sendBack(in message : Message) : void
+sendBack(in message : ChatMessage) : void
+sendBack(in message : Serializable ) : void
```

This class, which is a subclass of the “Thread” class, deals with all connections with the client. It provides a means to implementing the server multi-threaded, thus increasing overall efficiency. Furthermore, it acts as a buffer before the messages can fully be processed as “events”. By definition, an “event” is the minimal yet sufficient description of how the current world state (or game state) should be changed so that the game playing can be simulated. As mentioned in the Requirements Analysis Report, messages will not be received by the order in which they are sent. In many cases, further processing – such as message ordering – is necessary before they can be converted to events that faithfully reflect the players interaction. For these purposes, instances of “EventOrderer” and “MessageDecoder” classes are contained within this class.

It is also possible to see the effects of the publisher-subscriber model within this class. The subscribedChannel attribute, which is an instance of the “Channel” class, servers this purpose. The main idea here is that each time a client makes connection to the server, the client is subscribed to one of the “Channel”s depending on the Room to which the Player is instantiated in. In other words, there is a corresponding “Channel” entity for each of the Room in the Cube. That way, whenever something happens within a Room, changes can be made effective to every other player in that Room.

Effectively, changes should be made on the instances that store information about the dynamic variables representing the Game State (GameState, RoomDispatcher, PlayerDispatcher). These changes are made in regular intervals: after several messages are grouped together, checked for consistencies and ordered according to their semantics, as discussed in the previous paragraphs.

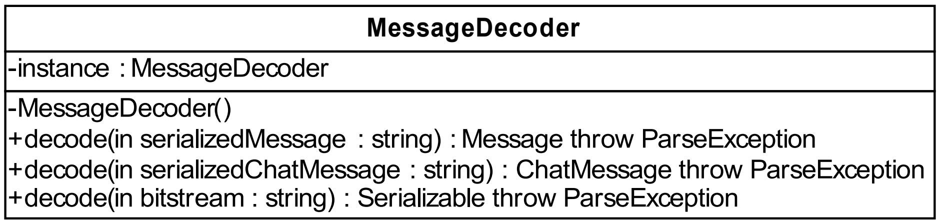
5.1.1.4. Channel



```
Channel
-subsriptions : Vector<Connection *>
-waitingEventsBuffer : Vector<Events>
+subcribe(in sub_connection : Connection *) : void
+unsubscribe(in unsub_connection : Connection *) : void throw ConnectionNotFoundException
+getAllSubscribers() : Vector<Connection *>
+cleanAll() : void
+pushEvents(in events : Vector<Event>) : void
+notifyClientsOfEvents() : void
```

It basically provides an interface for clients to subscribe as well as methods to make subscribed clients be notified on Game State updates. Events can be pushed in by the pushEvents () function call.

5.1.1.5. Message Decoder



```
MessageDecoder
-instance : MessageDecoder
-MessageDecoder()
+decode(in serializedMessage : string) : Message throw ParseException
+decode(in serializedChatMessage : string) : ChatMessage throw ParseException
+decode(in bitstream : string) : Serializable throw ParseException
```

The “Message Decoder” is designed based on the singleton design-pattern. In theory, the server should contain only one instance. Both the prototype implementation and the discussions with the group members confirm this design decision.

The class contains the decode() function that converts a serial message into a Message instance.

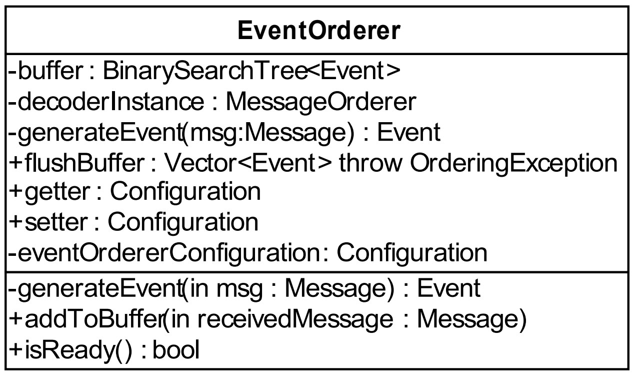
After the initial design phase, with the inclusion of the chat and the static/dynamic game-state loading functionalities, the responsibility of the “Message Decoder” module has changed significantly.

Previously, the module only resolved messages that represented movements and interactions of the players. However, currently it supports the exchange of:

- movement/interaction messages,
- serialized game-state representation instances (during initial loading, for instance when a player requests the necessary data to join a room),
- chat messages.

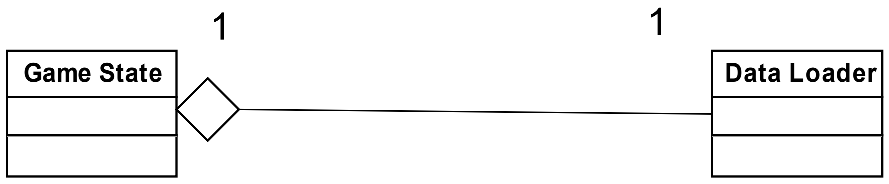
The three different versions of the “decode” message serve this purpose.

5.1.1.6 Event Orderer

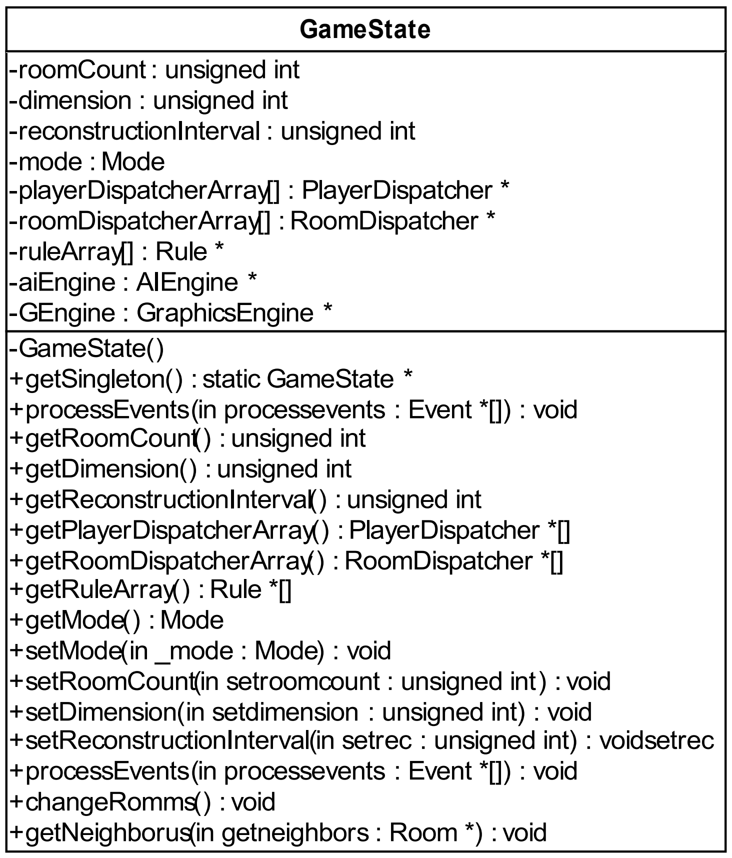

EventOrderer
-buffer : BinarySearchTree<Event>
-decoderInstance : MessageOrderer
-generateEvent(msg:Message) : Event
+flushBuffer : Vector<Event> throw OrderingException
+getter : Configuration
+setter : Configuration
-eventOrdererConfiguration: Configuration
-generateEvent(in msg : Message) : Event
+addToBuffer(in receivedMessage : Message)
+isReady() : bool

Incoming messages should be ordered and synchronized before they can faithfully be converted to a list of events. Messages are atomic and events may contain semantics provided by multiple messages altogether. This will be one of the most difficult parts to implement, as such processing requires complex algorithms to be implemented. For this reason, we have made the class as abstract as possible to allow future modifications. An instance of this class can be configured via supplying a "Configuration" instance to it. The "Configuration" class is yet not to be implemented so a default set of rules will be hard-coded in the "Event Orderer" class. The role of the functions such as "addToBuffer(), flushBuffer() and isReady()" is to control the "Event Orderer".

5.1.2. Server-Side Game Engine



5.1.2.1. Game State

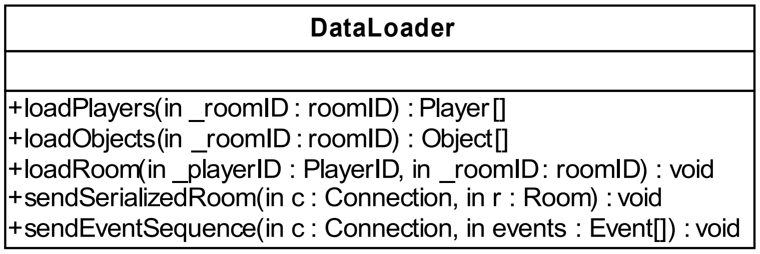


```
GameState
-roomCount : unsigned int
-dimension : unsigned int
-reconstructionInterval : unsigned int
-mode : Mode
-playerDispatcherArray[] : PlayerDispatcher *
-roomDispatcherArray[] : RoomDispatcher *
-ruleArray[] : Rule *
-aiEngine : AIEngine *
-GEngine : GraphicsEngine *
-GameState()
+getSingleton() : static GameState *
+processEvents(in processevents : Event *[]) : void
+getRoomCount() : unsigned int
+getDimension() : unsigned int
+getReconstructionInterval() : unsigned int
+getPlayerDispatcherArray() : PlayerDispatcher *[]
+getRoomDispatcherArray() : RoomDispatcher *[]
+getRuleArray() : Rule *[]
+getMode() : Mode
+setMode(in _mode : Mode) : void
+setRoomCount(in setroomcount : unsigned int) : void
+setDimension(in setdimension : unsigned int) : void
+setReconstructionInterval(in setrec : unsigned int) : voidsetrec
+processEvents(in processevents : Event *[]) : void
+changeRomms() : void
+getNeighborus(in getneighbors : Room *) : void
```

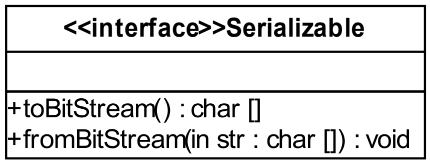
Purpose: Game State class is the core class of the server side game engine. It handles all the changes and interactions in the server side. It is a singleton class, so that there can be only one Game State instance.

Abstract: Our Game State class contains some static information about the cube, for example dimension and number of rooms. Our cube changes the rooms positions in a predefined interval(reconstruction interval) with a function called changeRooms(). It takes all the events from the network and process them with processEvents(Events *[]) function. After processing all the events the consequences of that events are sent to the players in that related room correctly also as an event. It manages all of these by using the instances of classes PlayerDispatcher and RoomDispatcher which handles the dynamic parts of the Player and Room classes. This class mainly deals with the general parts of the cube and the interaction among the players.

5.1.2.2. Data Loader

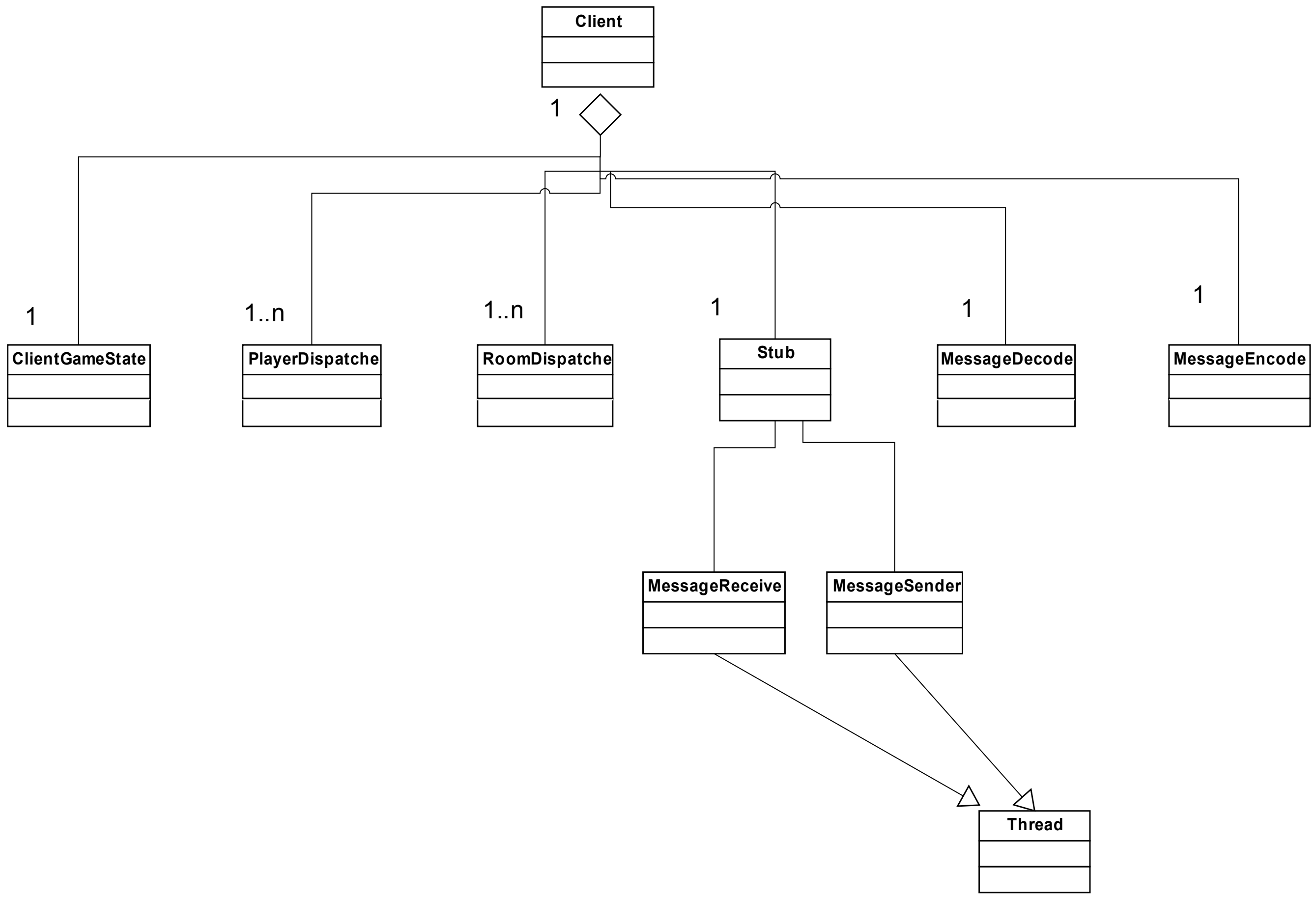

DataLoader
+loadPlayers(in _roomID : roomID) : Player[]
+loadObjects(in _roomID : roomID) : Object[]
+loadRoom(in _playerID : PlayerID, in _roomID: roomID) : void
+sendSerializedRoom(in c : Connection, in r : Room) : void
+sendEventSequence(in c : Connection, in events : Event[]) : void

DataLoader is the class responsible for the serving the client necessary information for initializing the game state. As the client requests to enter the game, it requests the static data (room information, dimension, list of objects, etc) and dynamic data (players in the room, object states, etc) . loadRoom() method handles the database connection to query the static room data, while sendSerializedRoom() method sends the retrieved Room object (which implements the Serializable interface) to the client. In the same manner, the player and object information retrieved from the database by loadPlayers() and loadObjects() methods is sent back to the client via sendEventSequence() methods in the form of <list of Events> (all the changes that have ever occurred in the room).


<<interface>>Serializable
+toBitStream() : char []
+fromBitStream(in str : char []) : void

Serializable is the interface that we have defined (inspired by the Java interface *Serializable*) to allow any object define its own methods toBitstream() and fromBitStream() to be sent over our network backbone. Our Room class implements this interface for loading the game, for instance.

5.1.3. Client-Side Network Backbone



Purpose:

It is a complementary package to the “Server-Side Network Backbone” that sits on the client. It provides a high-level means of abstraction for the Client Game Engine and the Graphics Engine when they need to interact with the server. Its overall architecture is similar to that of the “Server-Side Network Backbone”, however, it does not support multi-channeled connections.

Abstract:

Every client is connected to exactly one server. Therefore, all communications between a single client and the server can be handled via exactly one channel. Consequently, unlike the “Server-Side Network Backbone”, neither modules that implement the publisher-subscriber model nor constructs that enable multi-channeled communications are necessary.

Based on this criterium, the functionalities implemented by this package can be summarized as follows:

- provide a means of a seamless connection by abstracting out some networking constructs,

- send/receive serialized message instances as a means of connection with the server,
- encode/decode the serial instances into abstract “Message” entities,
- validate and order the messages,
- based on a sequence of valid messages generate “Event”s that make up the core of game-state changes,
- put the resolved information in the relevant data structures.

5.1.3.1. Client

Client
-clientID : string -clientProperties : Properties -hostProperties : Properties -clientGameState : ClientGameState * -rooms : Vector<RoomDispatcher *> * -players : Vector<Playerdispatcher *> * -stub : static Stub -encoder : static MessageEncoder -decoder : static MessageDecoder
+Client(in initialClientProperties : Properties, in hostProperties : Properties) +setClientProperties(in _properties : Properties) : void +getClientProperties() : Properties +setHostProperties(in _properties : Properties) : void +getHostProperties() : Properties +resolveHostProperties(in hostname : string, in physicalAddress : string) : Properties +openConnection() : void throw ConnectionException +closeConncetion() : void +loadStaticData() : void +loadDynamicData() : void

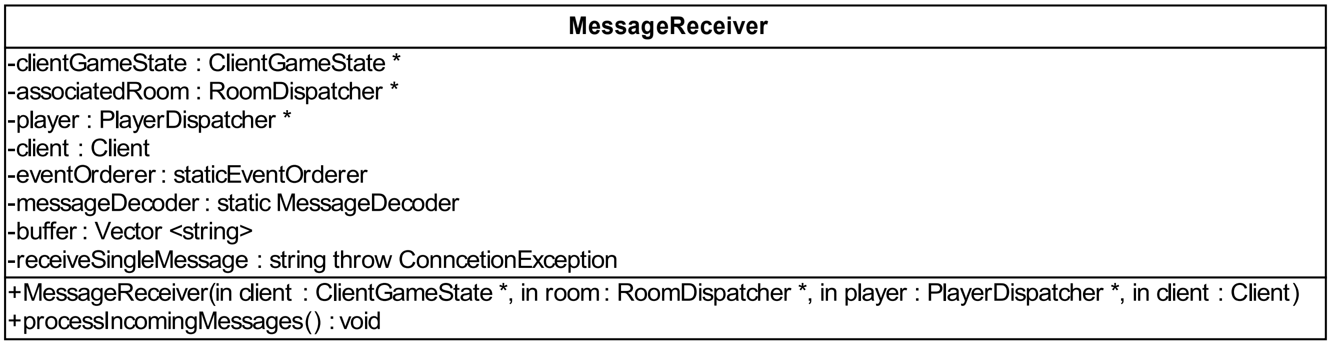
MessageSender
-client : Client * -encoder : static MessageEncoder -eventBuffer : Vector<Event>
+pushEvent(in lastEvent : Event) : void +sendOutEventsAsMessages() : void

Message Encoder
-instance : MessegaEncoder -MessageEncoder
+encode(in event : Event throw MessegaEncodeException) : Message +encode(in event : Event throw MessageEncodeException) : ChatMessage +encode(in event : Eventthrow MessageEncodeException) : Serializable

After the initial design phase, with the inclusion of the chat and the static/dynamic game-state loading functionalities, the responsibility of the "Message Encoder" module has changed significantly. Previously, the module only resolved messages that represented movements and interactions of the players. However, currently it supports the exchange of:

- movement/interaction messages,
- serialized game-state representation instances (during initial loading, for instance when a player requests the necessary data to join a room),
- chat messages.

The three different versions of the "decode" message serve this purpose.

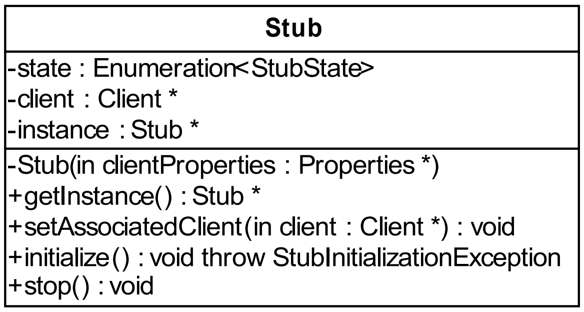


```
MessageReceiver
-clientGameState : ClientGameState *
-associatedRoom : RoomDispatcher *
-player : PlayerDispatcher *
-client : Client
-eventOrderer : staticEventOrderer
-messageDecoder : static MessageDecoder
-buffer : Vector <string>
-receiveSingleMessage : string throw ConncetionException
+MessageReceiver(in client : ClientGameState *, in room : RoomDispatcher *, in player : PlayerDispatcher *, in client : Client)
+processIncomingMessages() : void
```

Similar to the "Server" class, the "Client" is responsible for holding the necessary data structures, process modules and constructions related to the network connections. The client is defined by its unique id as well as its properties. Information such as the physical address on the network, ports used, etc. are all contained in the clientProperties attribute. The data structures that resemble dynamic game information relevant to the Room the Player is currently playing in are namely: ClientGameState, RoomDispatcher and PlayerDispatcher.

Actual network connections are handled by the "Stub", "MessageSender" and "MessageReceiver" instances respectively. Their details will be discussed shortly. The client first resolves the properties of the host to which it needs to connect. This functionality might be provided by a small scale registry/repository system, however, due to time restrictions imposed on the project, that is left out. For the time being, the host properties are set to default. The client then tries opening up a connection to the server machine and obtain static as well as dynamic game playing data [openConnection(), loadStaticData(), loadDynamicData()].

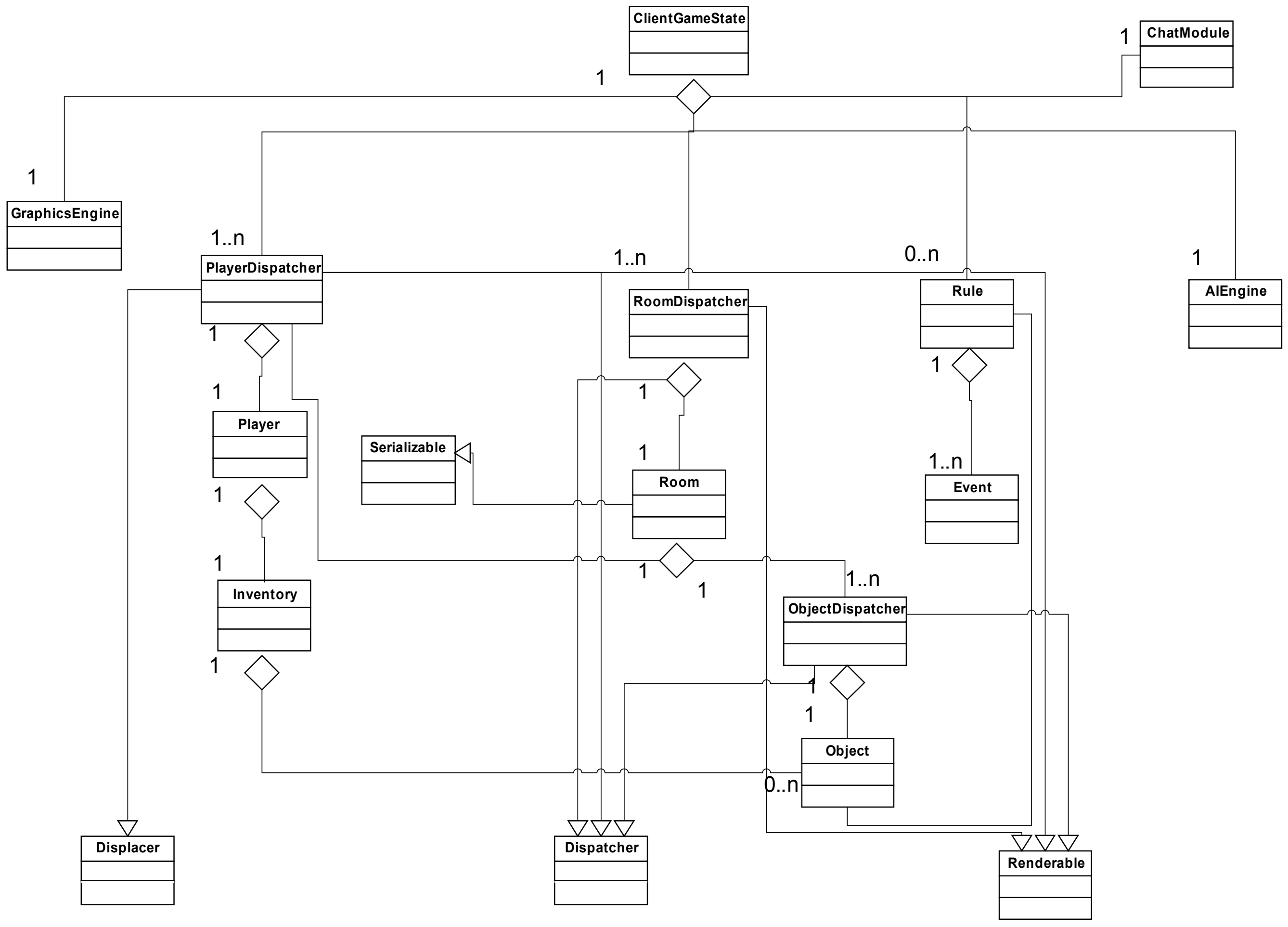
5.1.3.2. Stub



```
Stub
-state : Enumeration<StubState>
-client : Client *
-instance : Stub *
-Stub(in clientProperties : Properties *)
+getInstance() : Stub *
+setAssociatedClient(in client : Client *) : void
+initialize() : void throw StubInitializationException
+stop() : void
```

Upon initialization the stub class generates an instance of both MessageReceiver and MessageSender. These classes are implemented as two separate threads that resolve all the incoming and outgoing messages. These two classes have the responsibility of converting Message instances into a list of processable Event objects.

5.1.4. Client-Side Game Engine



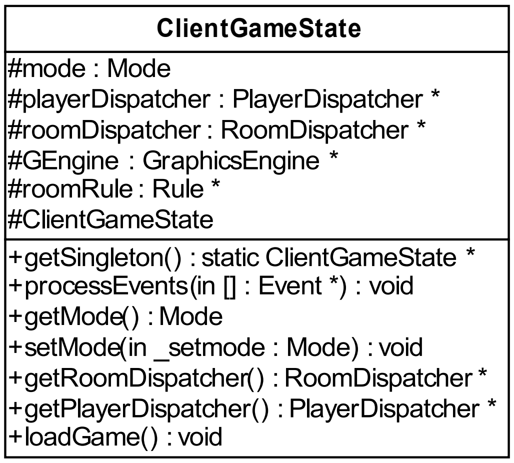
Purpose:

Client Game State is the main control structure of our client side game engine. Everything about the game in the client side is related with it.

Abstract:

It contains AI Engine, Graphics Engine, Chat Module, Player Dispatcher, Room Dispatcher and Rule in it. They are the cores of our game data . Player Dispatcher contains Player which also contains Inventory in it. Room Dispatcher contains Room in it which consists of Object Dispatcher containing Object in it. All our dispatchers are also renderable and player dispatcher is also a displacer. All these core entities work collobaretevly in the client side to make the game played properly.

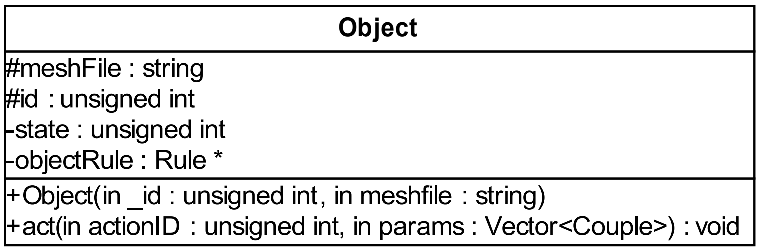
5.1.4.1. Client Game State



```
ClientGameState
#mode : Mode
#playerDispatcher : PlayerDispatcher *
#roomDispatcher : RoomDispatcher *
#GEngine : GraphicsEngine *
#roomRule : Rule *
#ClientGameState
+getSingleton() : static ClientGameState *
+processEvents(in [] : Event *) : void
+getMode() : Mode
+setMode(in _setmode : Mode) : void
+getRoomDispatcher() : RoomDispatcher *
+getPlayerDispatcher() : PlayerDispatcher *
+loadGame() : void
```

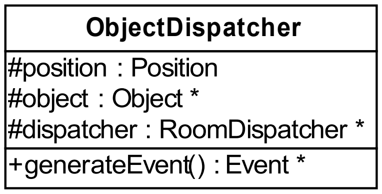
ClientGameState is the core game engine class of the client side. It handles everything in the client side. It includes the room dispatcher and player dispatcher inside to resolve everything and passing events. It runs in no_mode, game or menu mode as in the GameState which shows the current state of the game in the client's side.

5.1.4.2. Core Objects



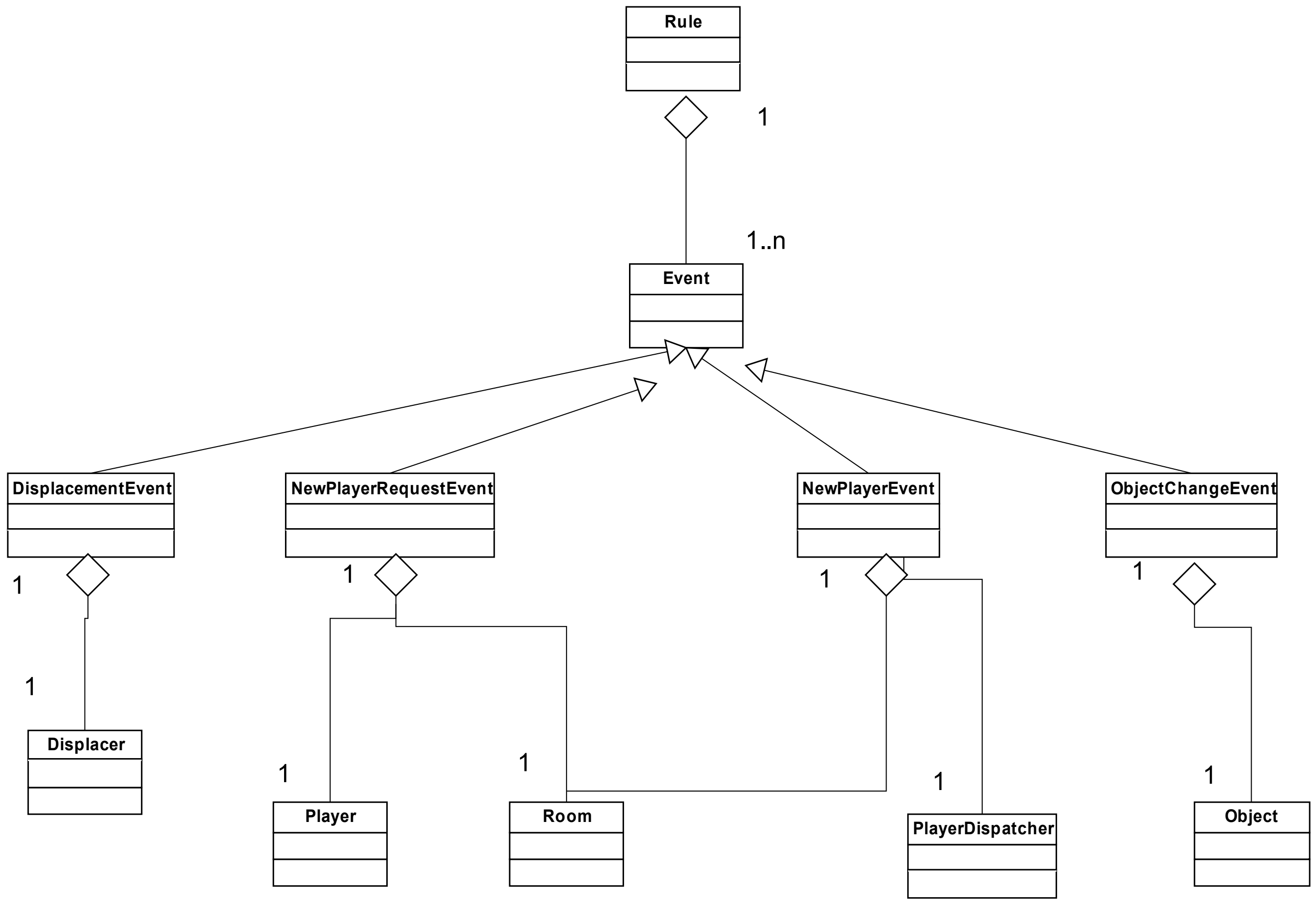
```
Object
#meshFile : string
#id : unsigned int
-state : unsigned int
-objectRule : Rule *
+Object(in _id : unsigned int, in meshfile : string)
+act(in actionID : unsigned int, in params : Vector<Couple>) : void
```

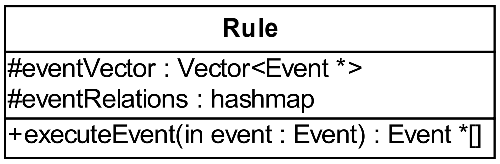
Our main entities in our game are the players, the rooms and the objects inside them. So we all defined them as classes. But we divided the static and the dynamic parts(the parts that changes often in the game play). We write Player, Object and Room classes as static classes and their Dispatcher classes as the dynamic classes. PlayerDispatcher, ObjectDispatcher and RoomDispatcher classes deals with the changing parts in the system. They all extend the Dispatcher and Renderable abstract classes which defines that the class deals with the dynamic data and these entities can be visualized by the Graphics Engine class.



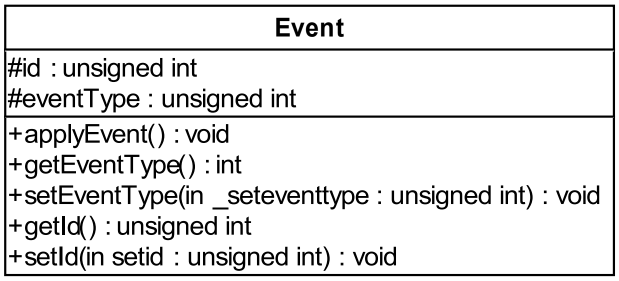
```
ObjectDispatcher
#position : Position
#object : Object *
#dispatcher : RoomDispatcher *
+generateEvent() : Event *
```

Object class contains mesh file, id , state and Rules of that object. Mesh file is the relevant information to render an object. Rules are the core parts of the object. Every object is defined with a rule. Rule contains the actions and event triggers of an object and defined in the Rule class. For example, to turn on the lights is an event and it results in another event which is illumination. Rules are defined like this; an event triggers another event or events. For every object at least one rule is defined. The rules will be checked in the ClientGameState for the validation of the actions done by using that objects. As you see Object class is needed only at loading of a room at first place, then every change or action done by using object is handled via ObjectDispatcher. ObjectDispatcher holds the instance of that object and the position information of that object. Everything about the object is handled by the generateEvent() method. Changes in the object are turned into an event by the ObjectChangeEvent class which is abstracted from the Event class. This class turns a change at object into an event so that network can resolve and then calls the applyEvent() function to execute the related event. Every dynamic thing in our system is resolved by using that technique: a change in the system for an entity is defined as a class and the event is then executed by using applyEvent() function.



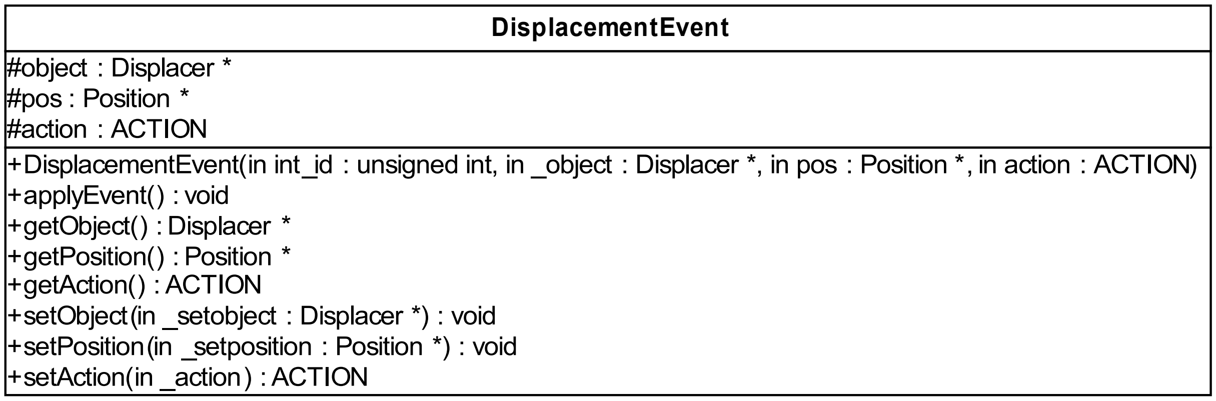


```
Rule
#eventVector : Vector<Event *>
#eventRelations : hashmap
+executeEvent(in event : Event) : Event *[]
```

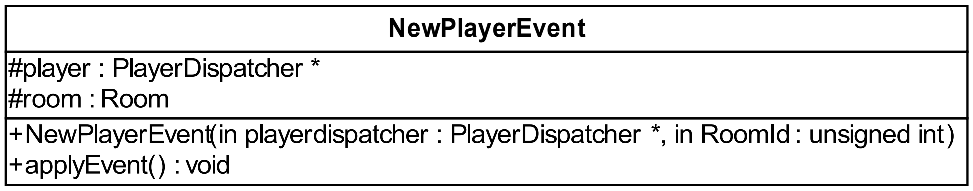


```
Event
#id : unsigned int
#eventType : unsigned int
+applyEvent() : void
+getEventType() : int
+setEventType(in _seteventtype : unsigned int) : void
+getId() : unsigned int
+setId(in setid : unsigned int) : void
```

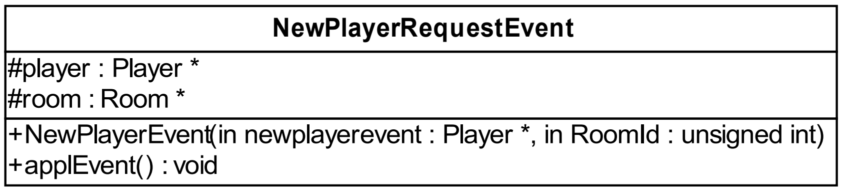
Event class is an abstract class and it is the core of our design. Every move in our system is handled by the event and message system. Many classes are derived from the Event class. For every action in the game we define an event class. below you can find more detailed information about the event classes.



```
DisplacementEvent
#object : Displacer *
#pos : Position *
#action : ACTION
+DisplacementEvent(in int_id : unsigned int, in _object : Displacer *, in pos : Position *, in action : ACTION)
+applyEvent() : void
+getObject() : Displacer *
+getPosition() : Position *
+getAction() : ACTION
+setObject(in _setobject : Displacer *) : void
+setPosition(in _setposition : Position *) : void
+setAction(in _action) : ACTION
```



```
NewPlayerEvent
#player : PlayerDispatcher *
#room : Room
+NewPlayerEvent(in playerdispatcher : PlayerDispatcher *, in RoomId : unsigned int)
+applyEvent() : void
```



```
NewPlayerRequestEvent
#player : Player *
#room : Room *
+NewPlayerEvent(in newplayerevent : Player *, in RoomId : unsigned int)
+applEvent() : void
```

GameExitRequestEvent

In the same manner as the above events – using our event expansion mechanism – a player’s request to exit the game is represented by this event. PlayerID is sent within the message (generated by the event) to the serve. The server receiving this request and validating it deletes the player from the room and then sends back a new event to all the players in the same room with the exiting player, to also delete this player from their room.

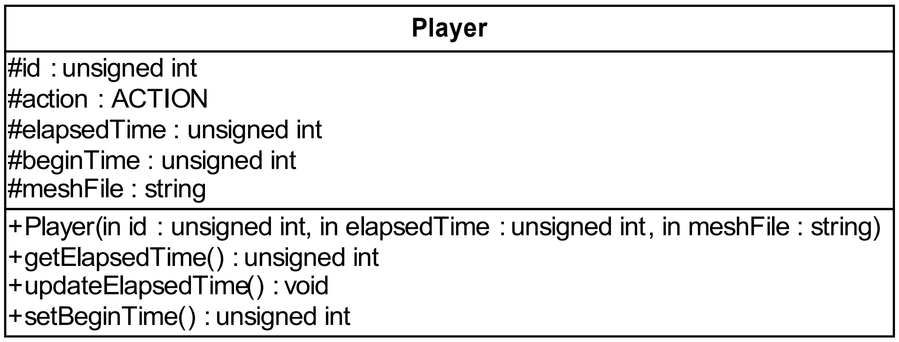
RoomExitEvent

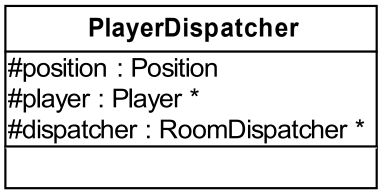
This event represents the exiting action of a player to be deleted from the room. All the information related to the corresponding player with the PlayerID contained within the event is deleted, updating the game state. Note that in the case of self-exit, client program simply terminates.

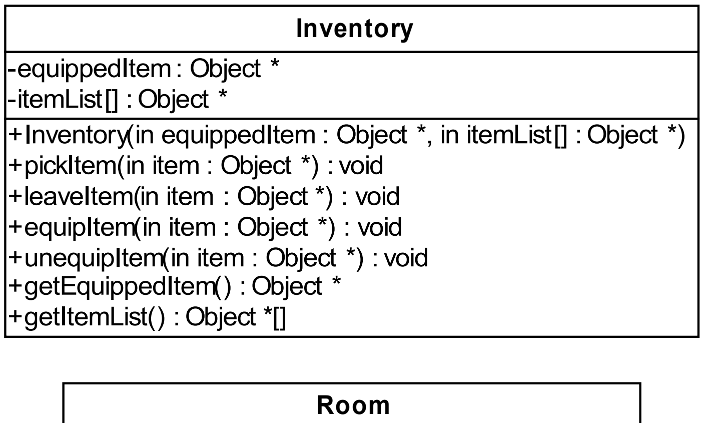
RoomChangeRequestEvent

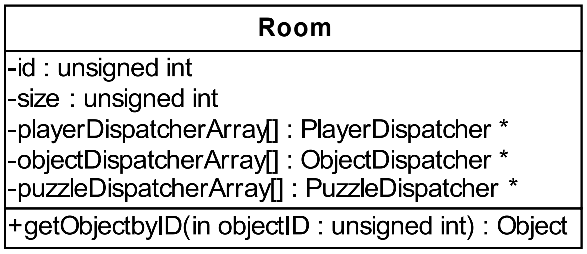
When a user completes all the requirements of the room, this event is thrown and sent to the server. Server is responsible to change the room information related to the player (with playerID). RoomExit event is sent to all the players in the old room, and NewPlayerEvent is sent to all the players in the new room, including the changing player.

Player class is the static class of the Player entity which contains the meshfile, player id, the action player is doing(still, walk, jump, crawl...) and elapsed Time (begin Time is used to calculate the elapsed Time). PlayerDispatcher holds the instance of the player and its position. This class is also abstracted from Displacer class, besides Dispatcher and Renderable. Displacer shows that this is entity can move from one place to another. generateEvent() handles event generation for that class. A very important event related with the player entity is creating a new player or adding an existing player to the game. For them we created two event abstracted classes: NewPlayerRequestEvent class which deals with the server's initialization of the player in the cube and the other is NewPlayerEvent sends the information of a new player in the room to everyone in the room. We used different attributes in these classes. NewPlayerRequestEvent uses Player instance directly since it creates a player from the beginning, NewPlayerEvent uses PlayerDispatcher since it has to apply all the changes that has happened in the room before the player comes into the room. The player has also an Inventory class. It consists of the object list that is in the inventory and the equipped item. You can pick, leave, equip or unequip items using related methods.

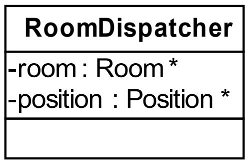

Player
#id : unsigned int
#action : ACTION
#elapsedTime : unsigned int
#beginTime : unsigned int
#meshFile : string
+Player(in id : unsigned int, in elapsedTime : unsigned int, in meshFile : string)
+getElapsedTime() : unsigned int
+updateElapsedTime() : void
+setBeginTime() : unsigned int

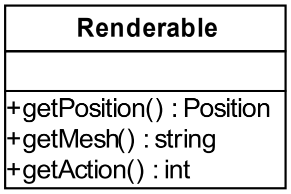


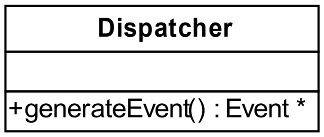

Inventory
-equippedItem : Object *
-itemList[] : Object *
+Inventory(in equippedItem : Object *, in itemList[] : Object *)
+pickItem(in item : Object *) : void
+leaveItem(in item : Object *) : void
+equipItem(in item : Object *) : void
+unequipItem(in item : Object *) : void
+getEquippedItem() : Object *
+getItemList() : Object *[]

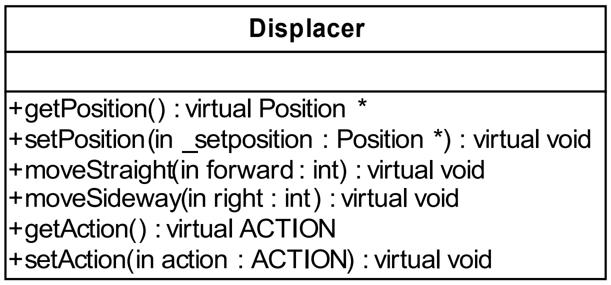

Room
-id : unsigned int
-size : unsigned int
-playerDispatcherArray[] : PlayerDispatcher *
-objectDispatcherArray[] : ObjectDispatcher *
-puzzleDispatcherArray[] : PuzzleDispatcher *
+getObjectbyID(in objectID : unsigned int) : Object

Room class is the static class of the Room entity which contains the size information and the player and object dispatcher arrays that are in the room. RoomDispatcher contains the position information of the entity. Room is one of the most important entities in our design because it also contains the other entities in it.

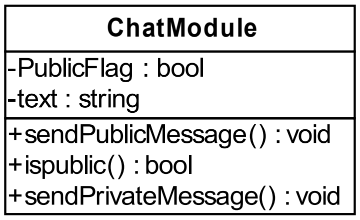




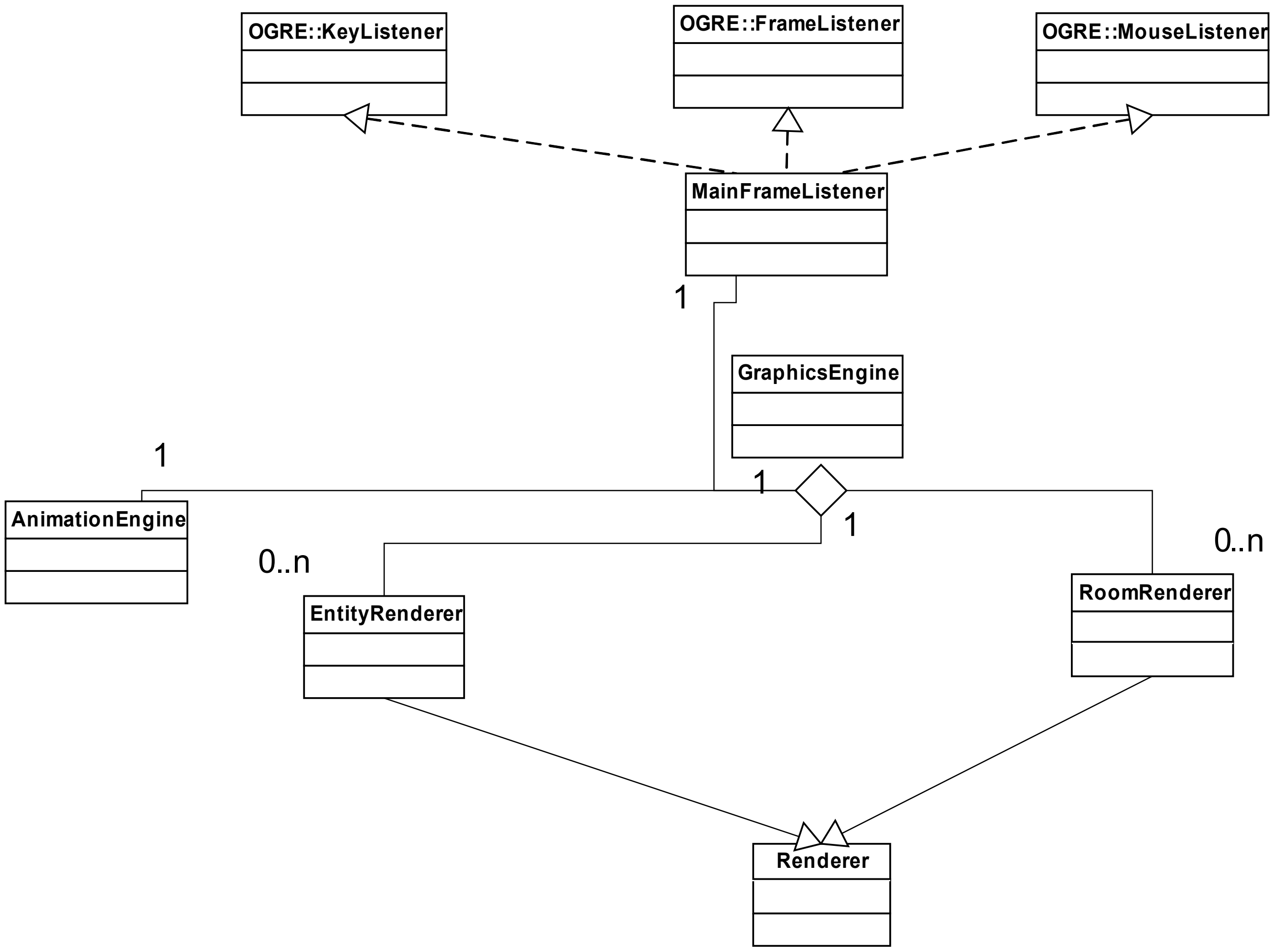



Displacer
+getPosition() : virtual Position *
+setPosition(in _setposition : Position *) : virtual void
+moveStraight(in forward : int) : virtual void
+moveSideway(in right : int) : virtual void
+getAction() : virtual ACTION
+setAction(in action : ACTION) : virtual void

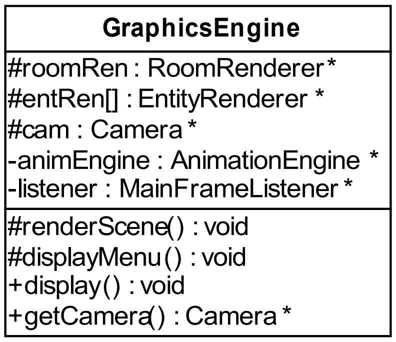
5.1.4.3. Chat Module

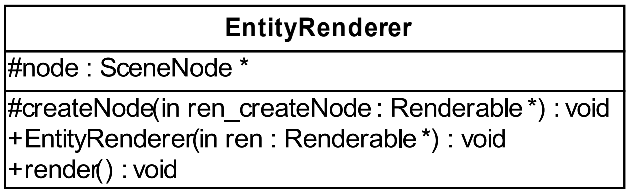


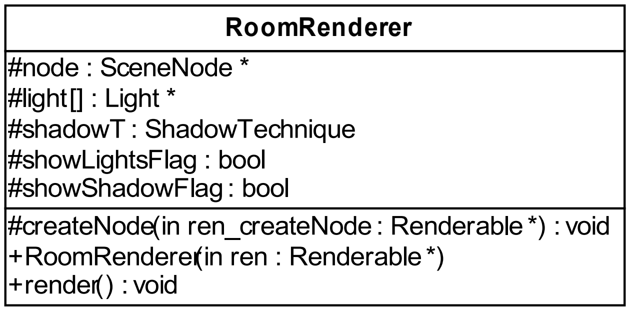
5.1.4.4. Graphics Engine



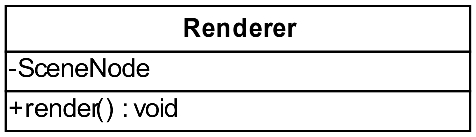
As the name implies, the "Graphics Engine" is the master class for rendering 3D graphics. It contains one AnimationEngine instance and one MyFrameListener instance for animation and controller parts. With control escalated to these parts, it maintains the proper rendering of the 3D scene represented by RoomRenderer vector viewed by perspective represented by Ogre::Camera instance. The render() method merely does this task, in coordination with each of the components (referenced in attributes).


GraphicsEngine
#roomRen : RoomRenderer *
#entRen[] : EntityRenderer *
#cam : Camera *
-animEngine : AnimationEngine *
-listener : MainFrameListener *
#renderScene() : void
#displayMenu() : void
+display() : void
+getCamera() : Camera *


EntityRenderer
#node : SceneNode *
#createNode(in ren_createNode : Renderable *) : void
+EntityRenderer(in ren : Renderable *) : void
+render() : void

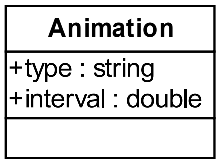

RoomRenderer
#node : SceneNode *
#light[] : Light *
#shadowT : ShadowTechnique
#showLightsFlag : bool
#showShadowFlag : bool
#createNode(in ren_createNode : Renderable *) : void
+RoomRenderer(in ren : Renderable *)
+render() : void

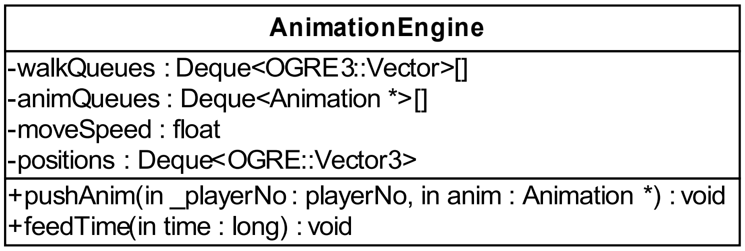
The "Renderer" class represents a single (but possibly composed of many) entity to be rendered by the graphics engine (denoted by Renderable interface). Holds the Ogre::SceneNode representation of the entity to directly render the node, which itself holds the actual Ogre::Entity object (internal representation) and the rest of the necessary information to display.



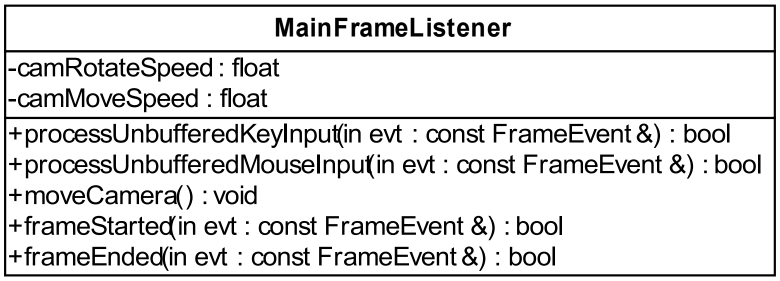
"Animation" class represents an animation instance. Holds the animation type and time interval for which it lasts (eg. "Idle" animation for 5 seconds). Those values have been encapsulated for their semantic integrity as well as future compatibility with complex animation representations.

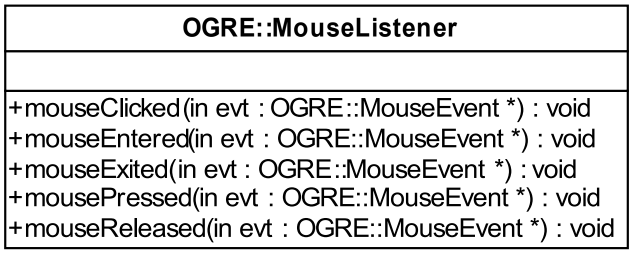
"Animation Engine" class represents the main responsible class for animation generation. Holding the Animation instances and position vectors for the corresponding objects, this class keeps track of animation states and manages animation. The feedTime() method gives the current time to the AnimationState objects (inner representation) for OGRE3D engine to realize the proper animation phases. In the case of movement, all players are expected to move at constant speed specified by the moveSpeed attribute.

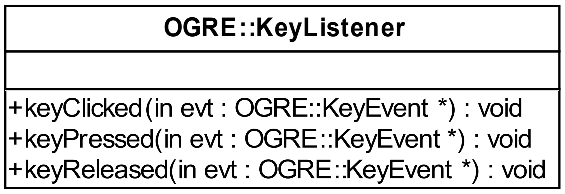



AnimationEngine
-walkQueues : Deque<OGRE3::Vector>[]
-animQueues : Deque<Animation *>[]
-moveSpeed : float
-positions : Deque<OGRE::Vector3>
+pushAnim(in _playerNo : playerNo, in anim : Animation *) : void
+feedTime(in time : long) : void

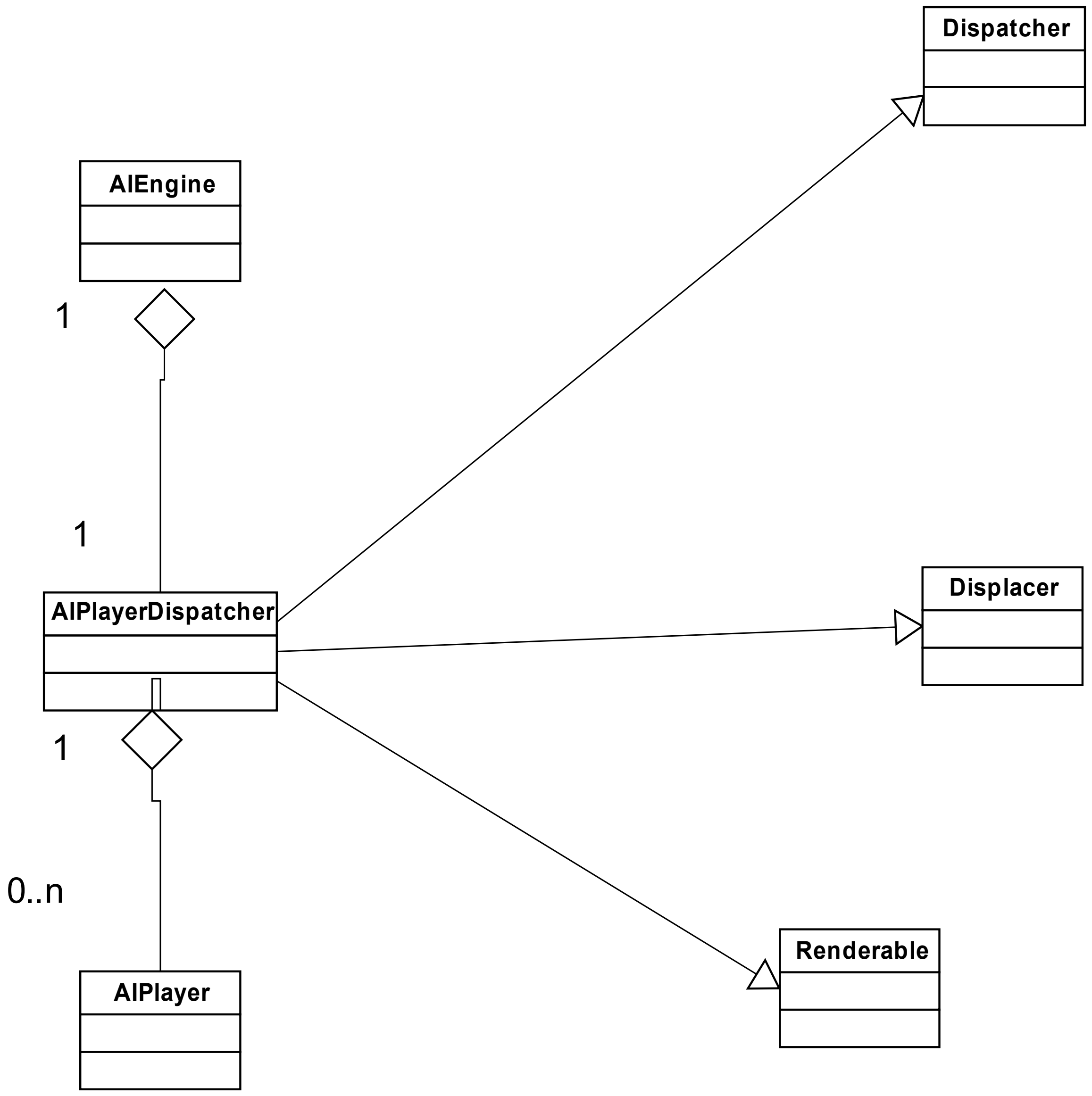
The "Main Frame Listener" class is the controller class which manages all the input into the program. Implementing Ogre::KeyListener and Ogre::MouseListener interfaces, it handles all the actions from keyboard and mouse. It also implements Ogre::FrameListener interfaces, so that it is active between frames. This leads to the capability of animation and unbuffered I/O control. Camera is expected to move and rotate with constant values specified by camRotateSpeed and camMoveSpeed attributes, realized by moveCamera() method for every relevant input.


MainFrameListener
-camRotateSpeed : float
-camMoveSpeed : float
+processUnbufferedKeyInput(in evt : const FrameEvent &) : bool
+processUnbufferedMouseInput(in evt : const FrameEvent &) : bool
+moveCamera() : void
+frameStarted(in evt : const FrameEvent &) : bool
+frameEnded(in evt : const FrameEvent &) : bool


OGRE::MouseListener
+mouseClicked(in evt : OGRE::MouseEvent *) : void
+mouseEntered(in evt : OGRE::MouseEvent *) : void
+mouseExited(in evt : OGRE::MouseEvent *) : void
+mousePressed(in evt : OGRE::MouseEvent *) : void
+mouseReleased(in evt : OGRE::MouseEvent *) : void

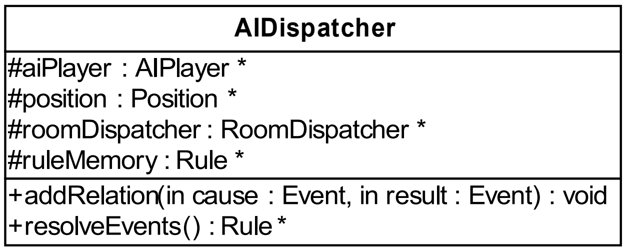

OGRE::KeyListener
+keyClicked(in evt : OGRE::KeyEvent *) : void
+keyPressed(in evt : OGRE::KeyEvent *) : void
+keyReleased(in evt : OGRE::KeyEvent *) : void

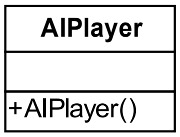
5.1.4.5. AI Engine



AI Engine class handles the behavior of the AI players in the game. AI players are defined in AIPlayer and AIPlayerDispatcher. The core of the AI in the game is creating a rule memory for the AI player by resolving events and adding rules to that list. We do that by using resolveEvents() and addRelation() functions.

AIEngine
#AIPlayerDispatcherVector : Vector<AIPlayerDispatcher *>
+updateAIMemory(in RoomId : int, in cause : Event, in result : Event) : void


AIDispatcher
#aiPlayer : AIPlayer *
#position : Position *
#roomDispatcher : RoomDispatcher *
#ruleMemory : Rule *
+addRelation(in cause : Event, in result : Event) : void
+resolveEvents() : Rule *



The game world is designed in order to allow proper integration of AI. These concepts, which are important in order to clarify the main working principles of AI, are as follows:

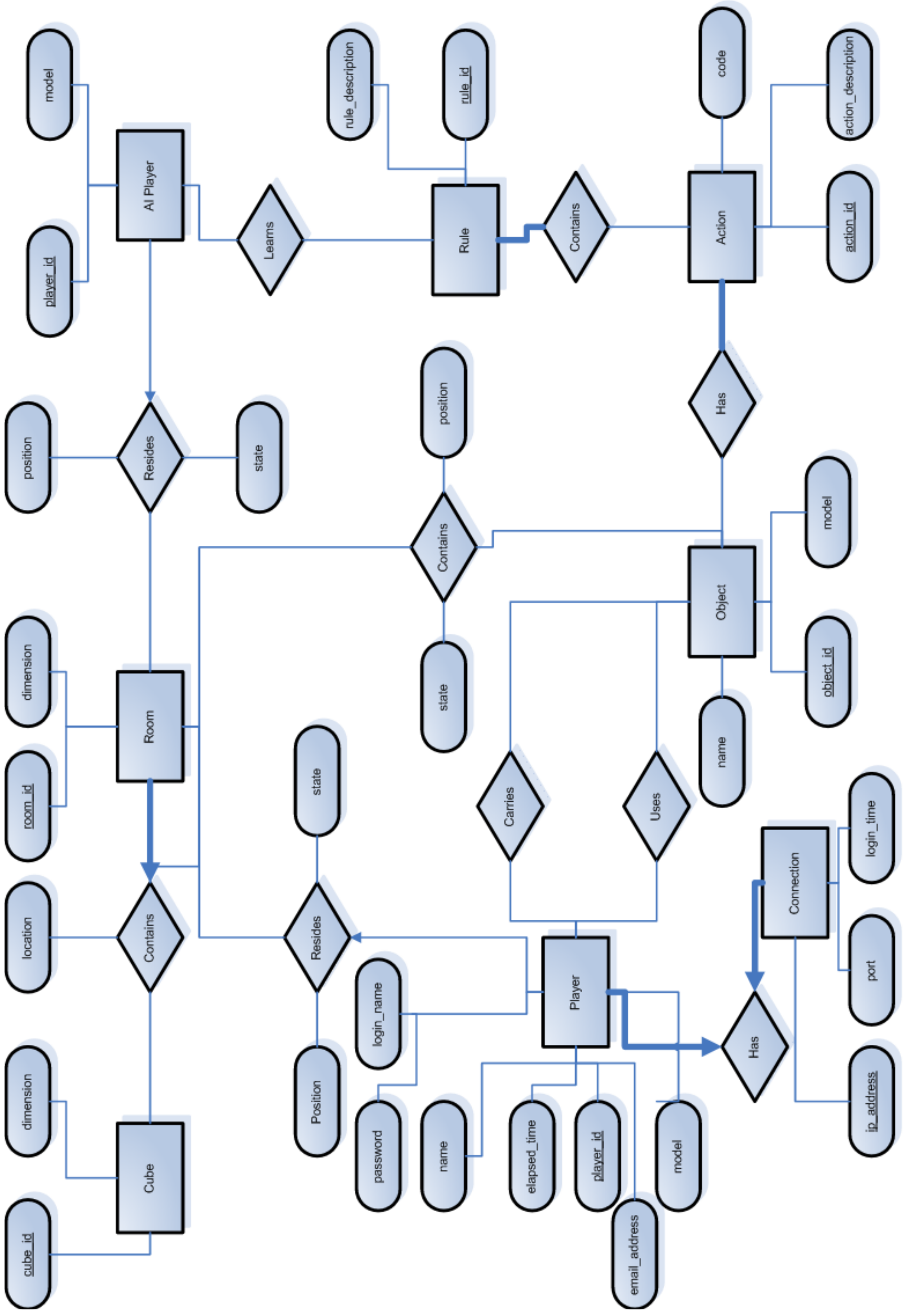
- Every object in the game has some enumeration of actions, which are well-defined and represented by their identifiers.
- Object actions are reflected systematically in our ontologies. These are to be used by the AI as the base knowledge of domain.
- Events resulting from a human-player action (which is also represented by an event) is stored in the Rule objects for each room. This way, game mechanics is embedded into these Rule objects in the form of Event-Event pairs, but also having the meaning of 'Event chains' intrinsically.

With these concepts as the underlying requirements, two main principles are adopted in our AI design: cheating and resolution. Cheating, which means the recording of the events with no effort, corresponds to some sort of witnessing mechanism. Resolution does the work of deducing some results given a set of relevant base predicates. Our AI is aimed to work in the following way:

- There will be players controlled by AI as the AI players, which consist of simple Player classes, with no controller functionality.
- AI engine will on top of AI players, controlling all the AI players in one command. All the events occurring in the game are reported to the AI engine.
- AI Engine assigns some memory (null as default) to all the AI players. As the events occur, engine writes the causing result and its result into the AI players' memory, provided that they are in the same room with the event.
- Once that AI players have some nonzero number of pairs in their memory, they try to use resolution algorithm to resolve the pairs and come up with suggestions that may lead to the solution of the puzzle (this is not guaranteed though).
- AI players will inform the human players through the chat facility, that is, they will send messages to report what they have found at regular intervals.

In short, since we define the set of cause-results of the game, 'AI gets to know the some portion of the actual rules applied to the room' and is capable of guessing events that will lead to the solution based on their prior knowledge. We try to give the AI players an artificial & simplistic sense of cognition.

5.1.5. Database Model



Our database lays in the server side of our architecture. It contains all the needed static information about the game and necessary dynamic information of gameplay. The main entities of our game are;

- The Cube
- Room
- Player
- Connection
- AI player
- Object
- Action
- Rule

We have only one huge cube in the game with dimensions we specify. It contains many rooms which are ordered through their room_id's. Of course their dimensions can be specified. The location of the room in the game changes about every half an hour, and we also write this location information to our database. This dynamic data will help us very much when placing a player to a room when he enters the game.

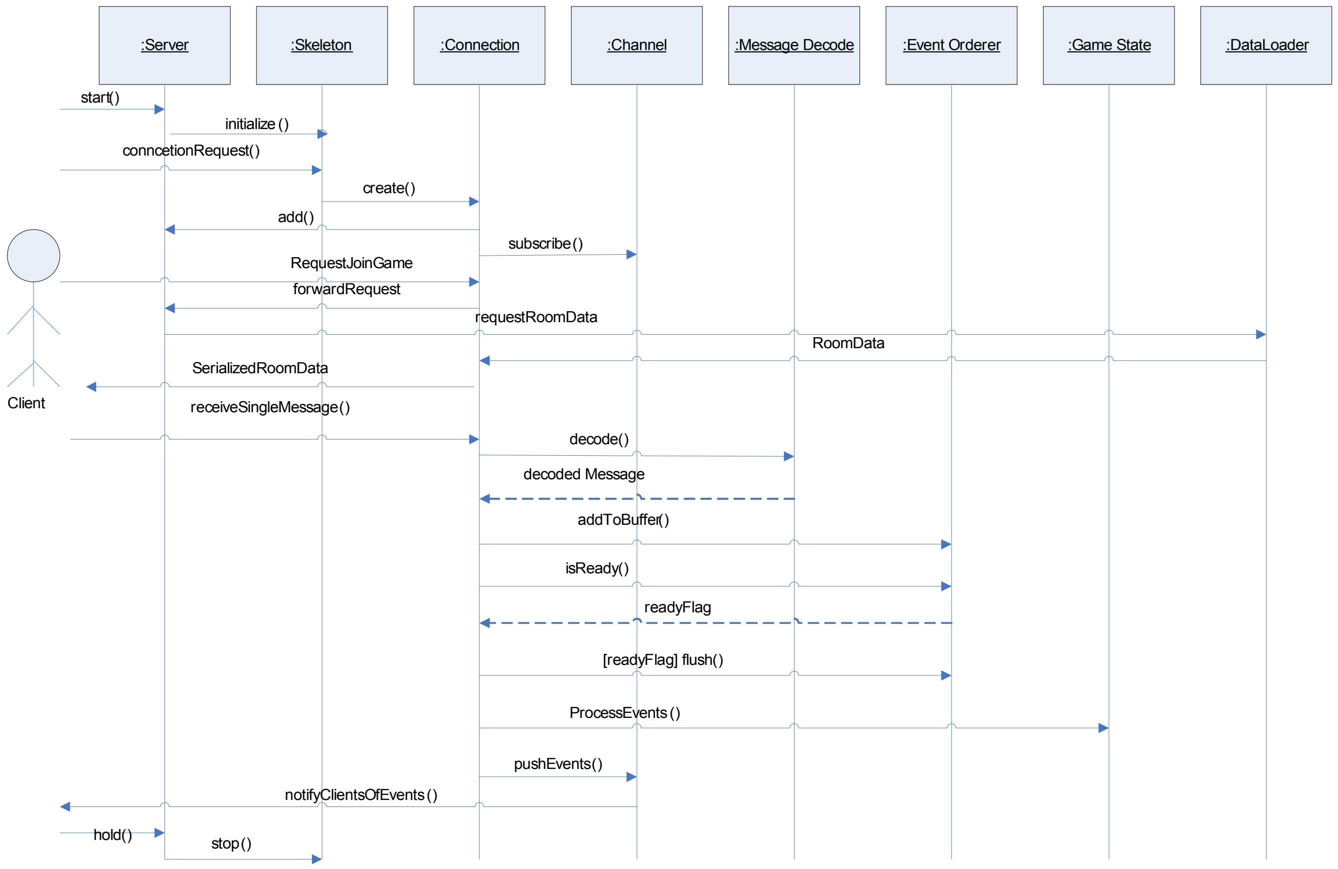
Rooms have players, AI players and objects in it. Players and AI players must stay in at most one room, while there are objects which are in many rooms. Players and AI players have very muc alike characteristics. Players are real players who have attributes such as player_id, name, login_name, email_address, elapsed time and the model name which is a file name ending with ".mesh" . Players has also a connection info which is one to one correspondant with the player. Connection has attributes such as ip_address, port and login_time. AI players have only three attributes player_id, name and model name. Players and AI players reside in the room at a specified position and in a specified position.

Objects are also inside the rooms in a specified position and state. They have object_id as primary key and name and model as their other attributes. Players can carry many or none objects in their inventory and also use many or none of them. Object may have actions defined for them and an action can be claimed by at least one object. These action are identified by their action_ids and they have also attributes such as action description and code which is a machine processable code to for game to execute the specified action.

There are also Rules which contains at least one action in it and has attributes such as rule_id and rule description. But not every action has to be involved in a rule. Actually this rule concept is the core of our gameplay. We will define every puzzle in the game in a format of list of actions. So at first we will define onthologies for the actions and store them at the database. Then we will define a chain of actions which leads to a result action. This result action is our puzzle result, which can be opening a door or maybe get wounded or even die. These rules can be learned by the AI players. AI players will have rule memory in the database.

5.2. Behavioural Design – Interaction Modeling

5.2.1. Server-Side Network Backbone: Initialization, Load-game and Event Processing Scenario



Server Side Sequence Diagram

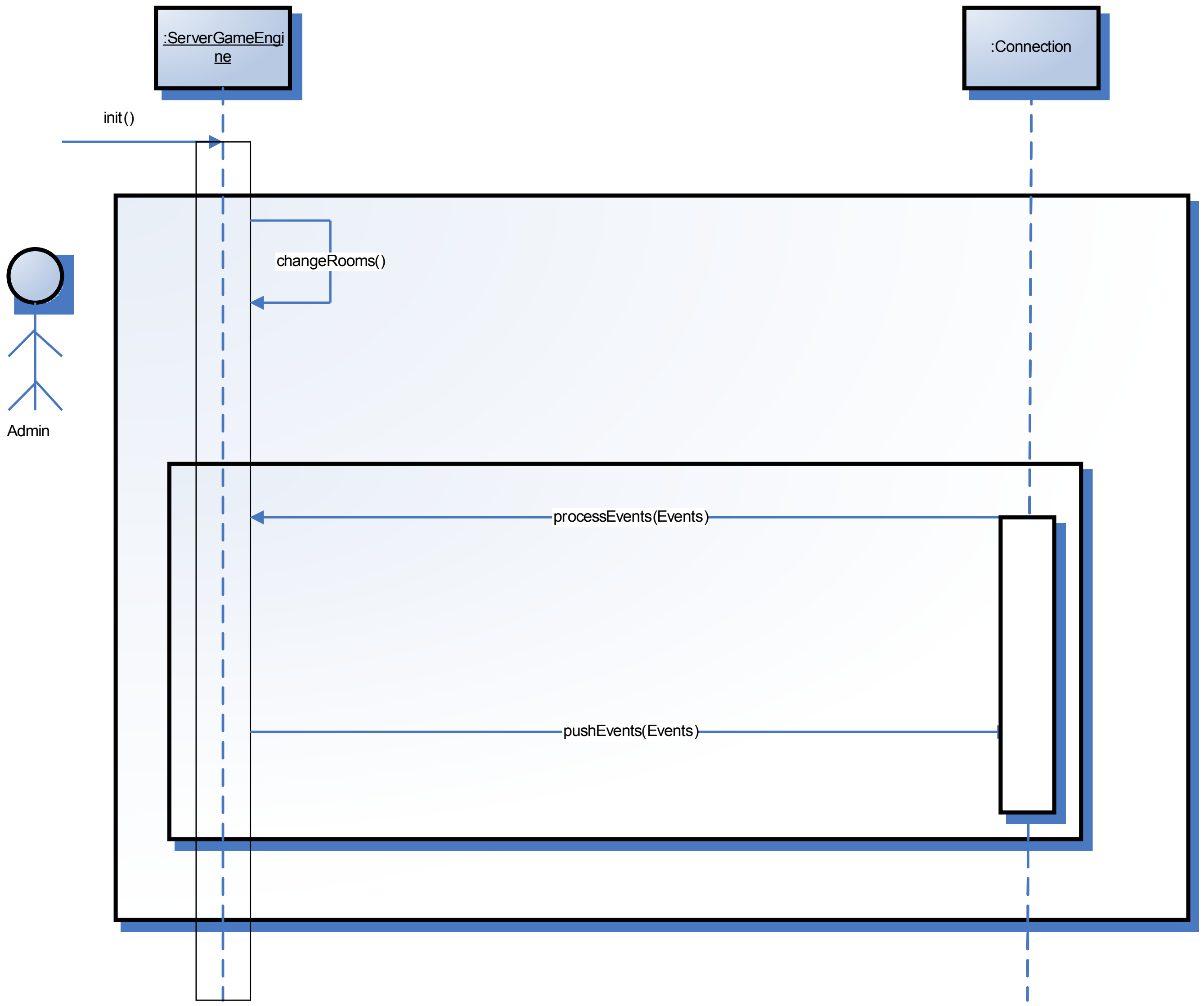
Although not all of the class-to-class interactions are provided here, the diagram provides the reader a thorough understanding of how the server behaves on incoming messages. The states through which the server goes, can be outlined as follows:

- The "Server" is initiated by some external entity. For the moment, we assume that this entity is the system administrator. [start()]
- The "Server" initializes its "Skeleton" so that it is able to receive messages from client applications. The "Skeleton" can be considered as the main point of entry for new clients to get involved in the game. [initialize()]
- The "Skeleton" sits idle until there is a new connection request. Whenever such an event occurs, the "Skeleton" generates a new instance of "Connection". This results in the generation

of a new thread and from that point on all of the interactions between the current client and the server are handled by that thread, namely that “Connection” instance. [create()]

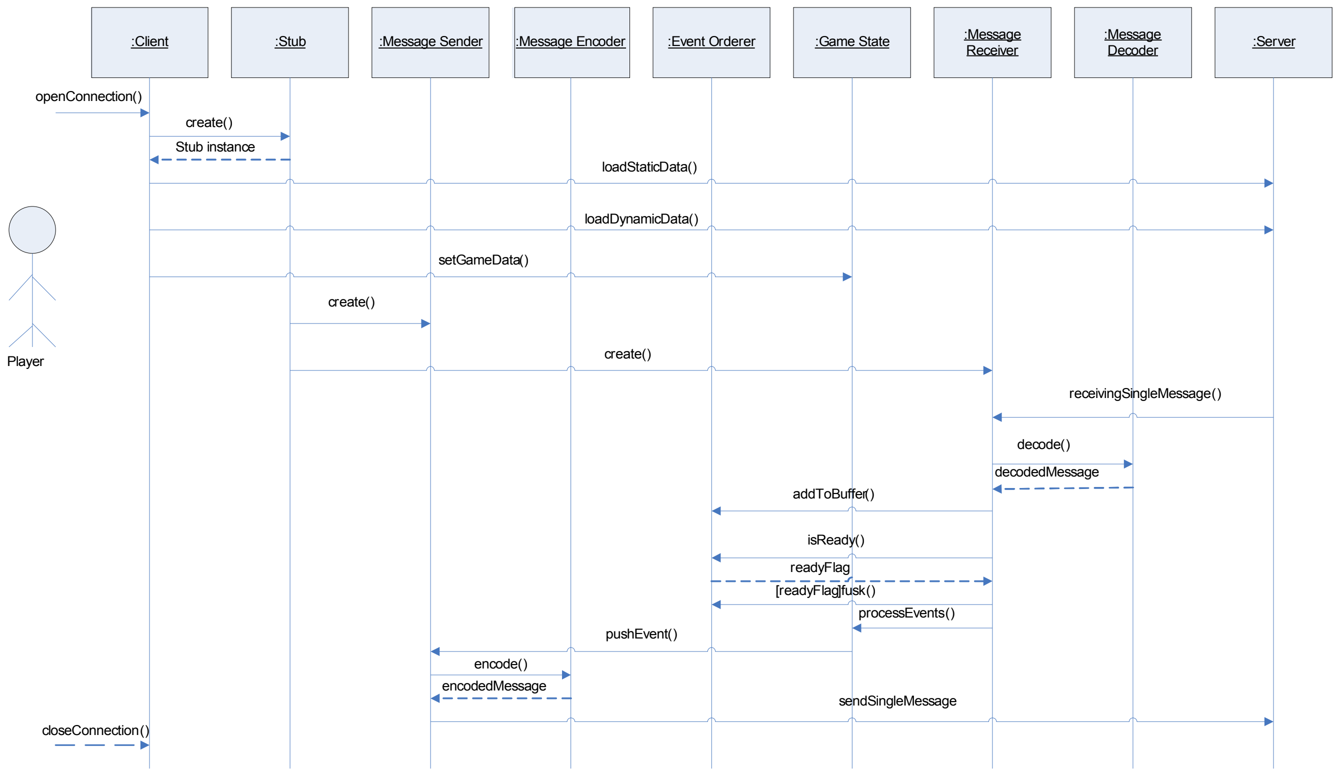
- It is a good programming practice that the “Connection” instance is visible from its generator, namely the “Skeleton”. For this purpose, the “Connection” adds itself to the list of all “Connections” in the “Skeleton”. [add()]
- The “Connection” thread must register itself with the appropriate “Channel” so that it is evaluated together with the other “Connection” instances in the same “Channel”. This is a must since the players in the same room should literally be viewing the same dynamic definitions regarding the room at hand. Maintaining synchronized game state data in various clients will be one of the major challenges in the project. [subscribe()]
- The “Connection” thread sits doing nothing until a message is received from the client.
- One type of a message that can be received is a request to join the game [requestJoinGame()]. The player should be assigned a room and more importantly that player should be sent the static/dynamic information regarding the room.
- The “Connection” thread forwards that request to the “Server” [forwardRequest()] which evaluates it and finds a suitable room to add the player into. Then it asks the “Data Loader” entity to supply the relevant information by means of its serialized instance [RoomData, SerializedRoomData].
- Whenever a message (relevant to a player's movement/interaction) is received though, it first needs to be decoded. The “Message Decoder” class comes in handy at this point. Messages are transmitted as raw byte-streams based on a predefined messaging format. “Message Decoder” converts them into Message objects processable by the server itself. [decode()]
- The decoded message is not yet ready to be processed. As mentioned in various parts of the report, messages need validation and ordering. For this purpose, the decoded message is sent to the “Event Orderer” unit for further processing. [addToBuffer()]
- There will come a time when the messages in the event queue can be interpreted to form a meaning. On that case, the “Event Orderer” will signal it to the “Connection” by sending its “ready” flag. [isReady()]
- Upon receiving the notification, the “Connection” thread is ready to send the list of ordered events both to the “Game Engine” [processEvents()] and to the subscribed “Channel” [pushEvents()].
- The “Channel” is responsible for notifying all its subscribers on the events pushed by the “Connection” [notifyClientsOfEvents()]. To be more specific on the issue, let us clarify it with an example. Suppose Player A is in Room X and there are 3 more Players in the room (Player B, Player C, Player D). If Player A makes a movement then all of the other Players (B – D) in the room must also be made aware of this change. The idea of a “Channel” aids us in implementing this concept. All the players in a single Room are subscribed to the same “Channel”. The “Channel” acts as a mediator whenever there is a change in the game state. Furthermore, whenever a message is sent to the server indicating that there is a change, the above procedure is automatically invoked.
- The server can be put on hold [hold()] or stopped [stop()].

5.2.2. Server-Side Game Engine: Event Receiving and Sending Scenario



The initialization of the server is when the game administrator calls the init() method of GameState (which is server-side). During the execution of the main loop, the positions of the rooms is changed in regular intervals through the self call of changeRooms() methods. The incoming events from the clients are also received, applied to the global game state and then distributed to all the related clients. This continues during the whole lifetime of the server.

5.2.3. Client-Side Network Backbone: Load-game and Message Receiving (encoding/decoding) Scenario



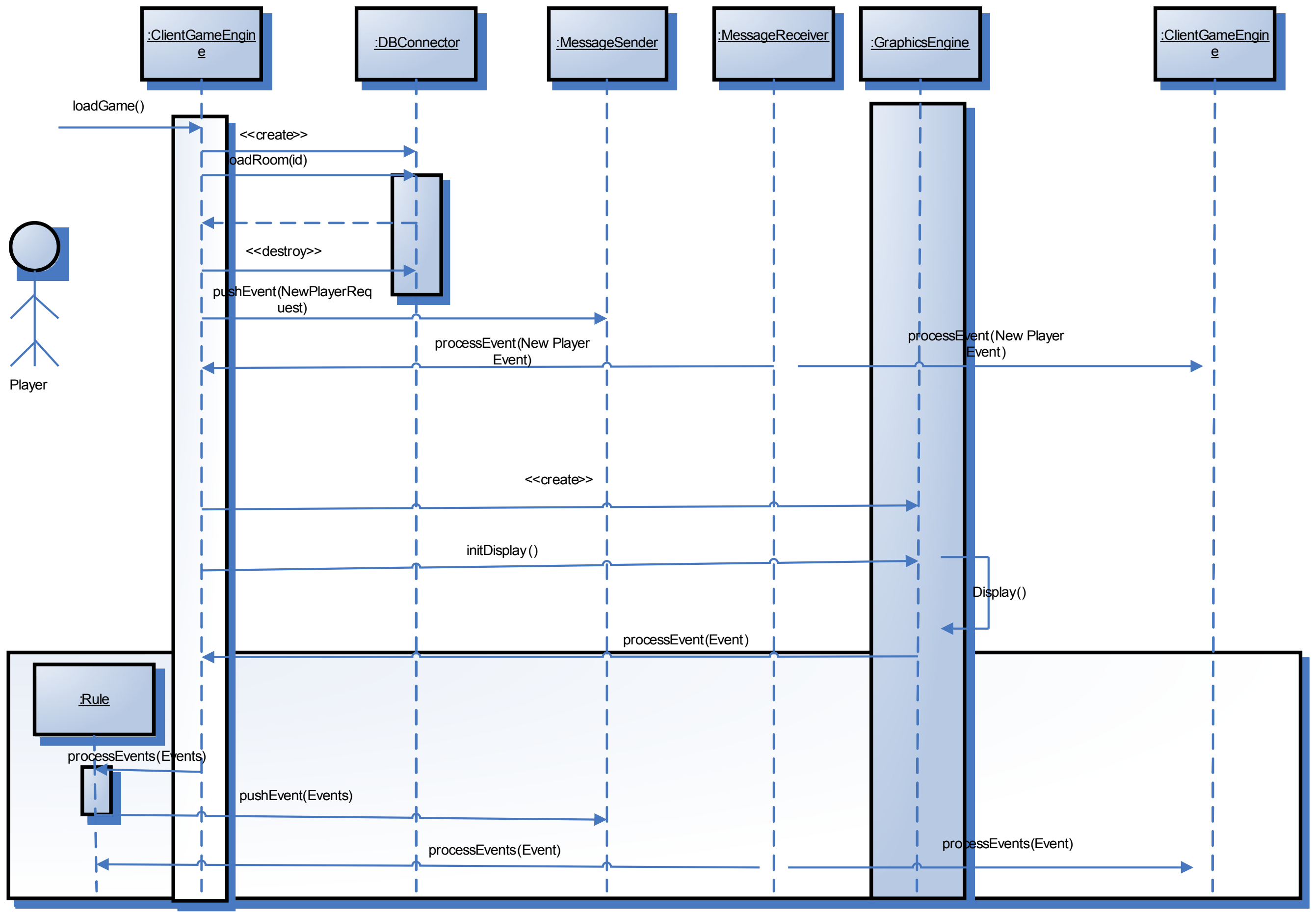
Client Side Sequence Diagram

Although not all of the class-to-class interactions are provided here, the diagram provides the reader a thorough understanding of how the client behaves upon initiation, until closing. The states through which the client goes are outlined as follows:

- An external entity – in our scenario the Player – after configuring and setting appropriate parameters of the "Client" for connection, directs it to obtain a connection with the host machine (or the server). [openConnection()]
- Upon this request, the "Client" generates a new (or obtains the existing) instance of the "Stub". [create()]
- After a connection is successfully established, the "Client" tries obtaining the static and dynamic representations of the world [loadStaticData(), loadDynamicData()]. Loading static representations are essential so that the Graphics and the Game Engine on the client side can generate the view and game functionalities. The dynamic data is necessary so that a player can start up in a room where there are other players that have already made some changes. Otherwise, concurrency cannot be maintained. Although physically two players are in the same room, they would not be playing in the same conditions.
- The data structures representing static and dynamic game information should be updated based on the previously loaded instances. [setGameData()]

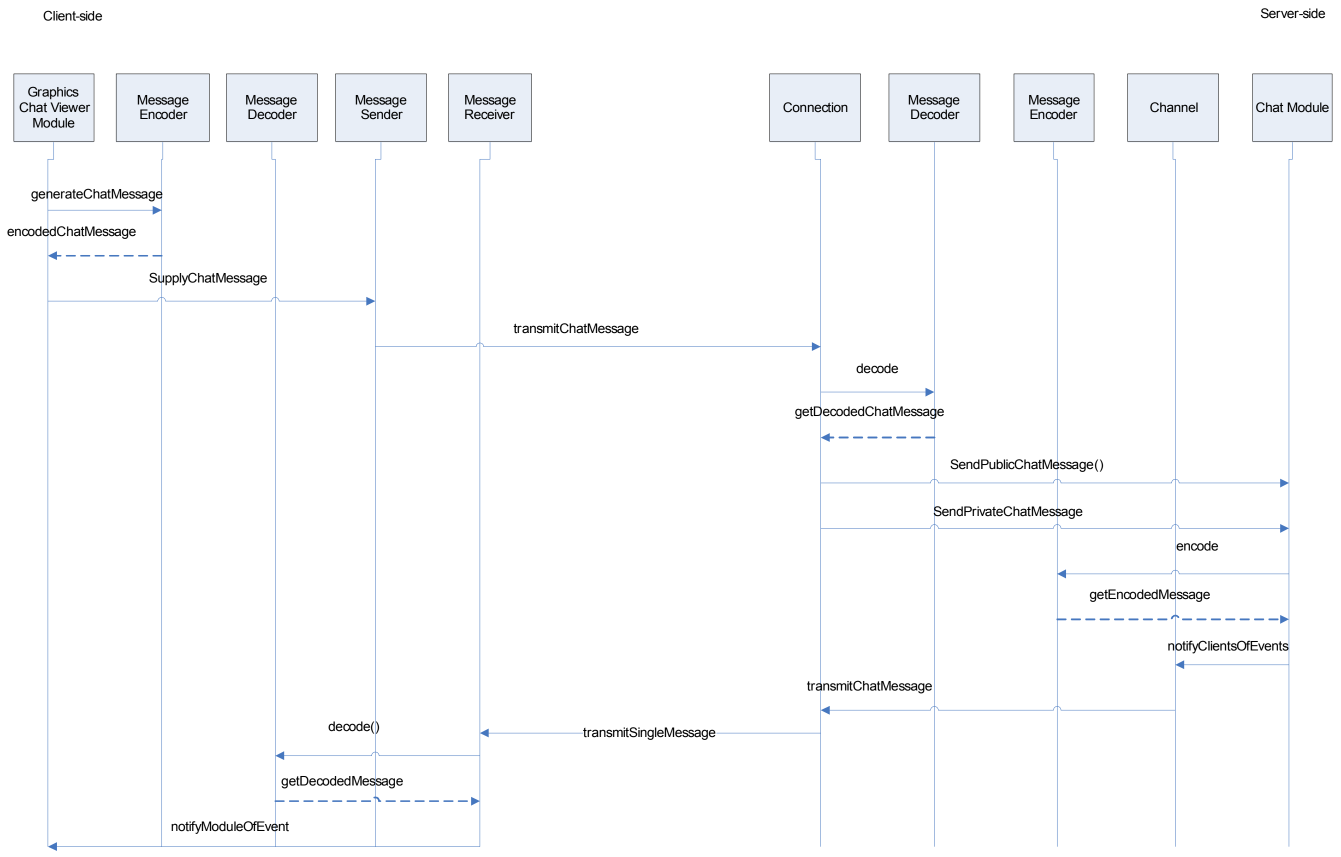
- When everything is established, the “Client” is ready to exchange messages with the server. For this purpose, it creates an instance of both the “Message Sender” and the “Message Receiver” class. [create()]
- The “Message Receiver” sits idle unless there is a new message from the server [receiveSingleMessage()] . Upon such an event, processes such as decoding, event ordering are executed [decode(), addToBuffer(), isReady(), flush()]. For more information please review the paragraphs on the Server-Side Sequence Diagram. Finally the list of ordered events are supplied to the Game Engine for further processing.
- The “Message Sender” on the other hand acts as the mediator between the player and the network. Whenever the “Game State” changes due to the current player's interaction with his/her machine an event is generated and sent to the “Message Sender” for processing. [pushEvent()]
- The “Message Sender” encodes (serializes) [encode()] the message and sends it to the “Server” [sendSingleMessage()].

5.2.4. Client-Side Game Engine: Load-game & Event Processing Scenario



The initialization of the game is when the loadGame() method is called by the user. This triggers the creation of the database connector class, which is used to load the static data from directly database, which is to be performed only once per game play. After the room data being loaded, the ClientGameState creates a NewPlayerRequestEvent, which is sent to the server and results in the addition of the new player to the room specified. This change in the game state is represented by a NewPlayerEvent and sent to all the players in the corresponding room. Only then the main loop of the game can be started. A GraphicsEngine object is created and initialized, and then it automatically starts displaying the scene. Any event spawned by the IO Handler, in our case implemented by the GraphicsEngine itself, is sent back to the game engine. The event is evaluated against a set of rules (relations actually) in the Rule object existing in the ClientGameEngine, and the resulting event(s) are transmitted to MessageSender, to be distributed to all related players. The client is now ready to receive messages and process them, to modify the game state. The main loop executes as long as the player is in game.

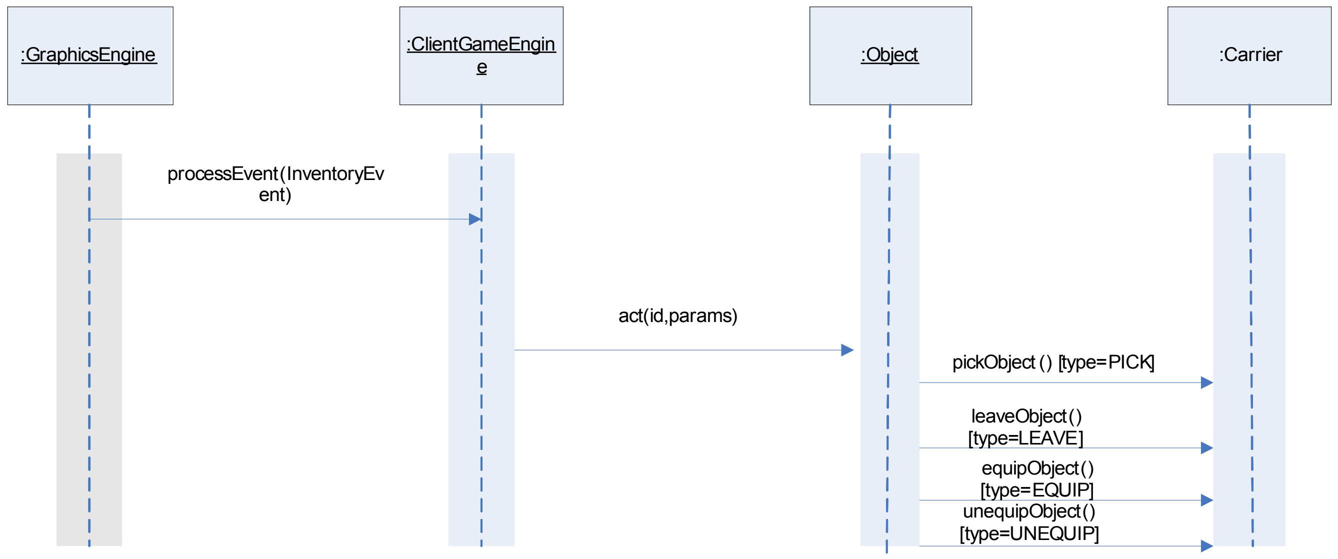
5.2.5. Chat Engine: Client-Server Single Message Transmission Scenario



Chat functionality among the players is handled separately with its own message type and graphics rendering module. When a player wants to send some text to other players, that text content is captured by the GraphicsChatViewerModule and sent to MessageEncoder to be encoded. The resulting ChatMessage is sent to the Connection object via MessageSender. Received and decoded, the actual ChatMessage is transmitted finally to ServerChatModule by means of its sendPublicChatMessage() or sendPrivateChatMessage() methods. This implies a request from the server to distribute the ChatMessage that it received, and so it does. ServerChatModule encodes the ChatMessage and sends it to Channel object through its notifyClientsOfEvents() method. Channel is responsible for distributing the ChatMessage to the relevant targets. MessageReceiver module of the clients gets the message, decode it and send it to the GraphicsChatViewer to be displayed.

Note that this is not implemented through the Event mechanism since it semantically forms a different module. Those ChatMessages neither need ordering facility or no validation to check its integrity. They simply sent and received without any concern, and displayed on the screen. This way, the network interaction between players is compartmentalized into independent modules handled separately.

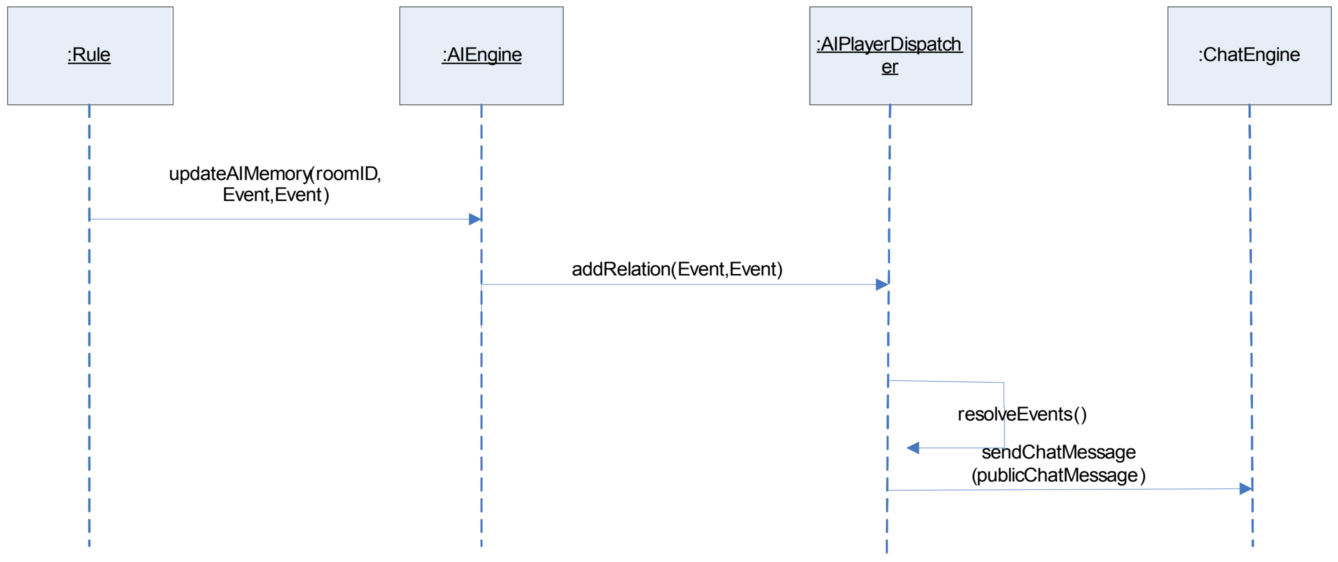
5.2.6. Client-Side Game Engine: Inventory Functions Scenario


:GraphicsEngine
:ClientGameEngine
:Object
:Carrier
processEvent(InventoryEvent)
act(id,params)
pickObject () [type=PICK]
leaveObject() [type=LEAVE]
equipObject() [type=EQUIP]
unequipObject() [type=UNEQUIP]

Inventory

Any change in the inventory is first triggered by an event generated by the GraphicsEngine (implementing IOHandler) and sent to ClientGameEngine. The processing of the event by the engine results in the call of act() method of the related object. Next, according to the type of the inventory action, a corresponding message (pickObject(), leaveObject(), equipObject() or unequipObject()) is sent to the Carrier object and the necessary adjustment of the items in Carrier's inventory is achieved.

5.2.7. AI Engine: Learning, Resolution and AI-Human Interaction Scenario

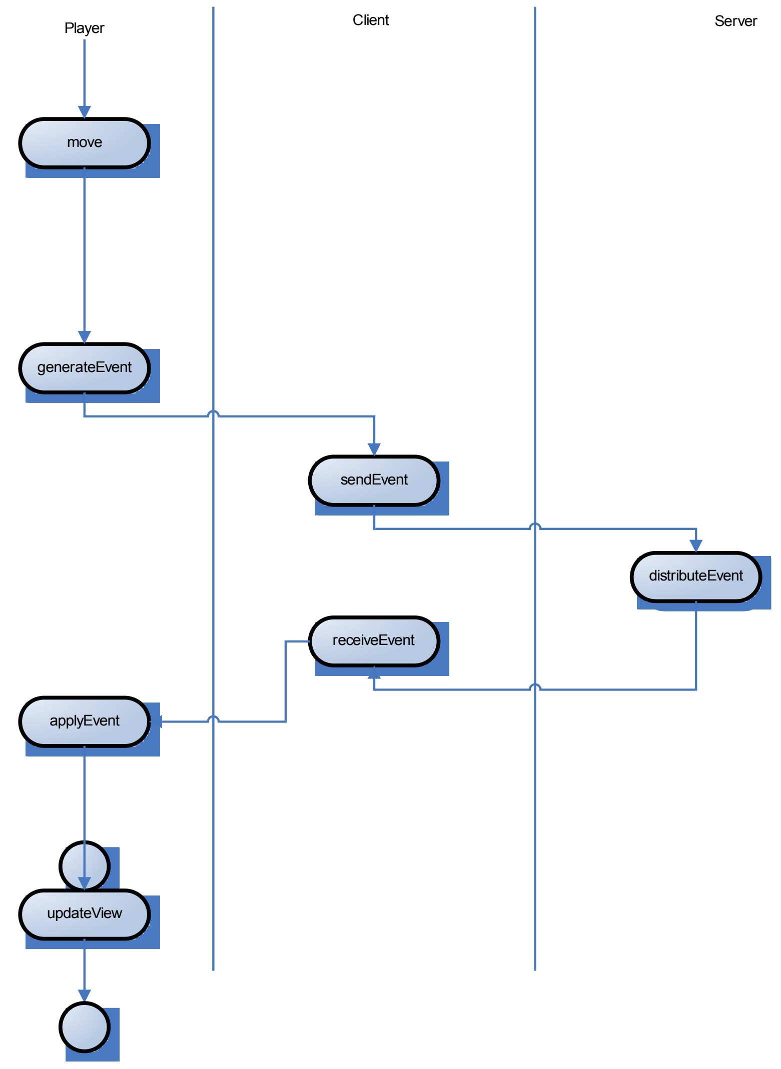


AI Engine

The AIPlayerDispatcher object, representing a living AI player, holds the data in the AI memory. Any processing in the rule object resulting of the matching of the resultant Event given a causing Event triggers the call of updateAIMemory method in the AI engine. Next AI Engine passes the Event pair to all the responsible AI players (in our design all the players in the same room with the occurring event), which is stored in their memory. After that, resolveEvents() method is called, generating resultant Events given the pool of cause-consequence pairs in their memory. With this being completed, AI informs the players by sending public messages to the ChatEngine, containing information about the event they have deduced.

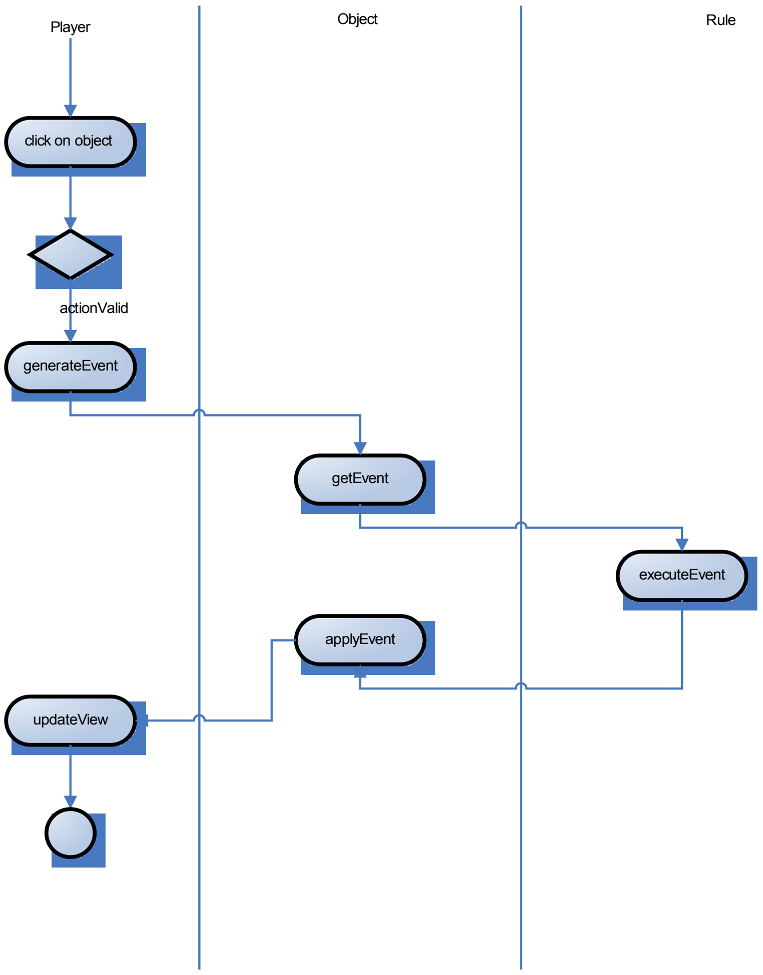
5.3. Behavioral Design – Process Modeling

5.3.1. Player Movement Scenario



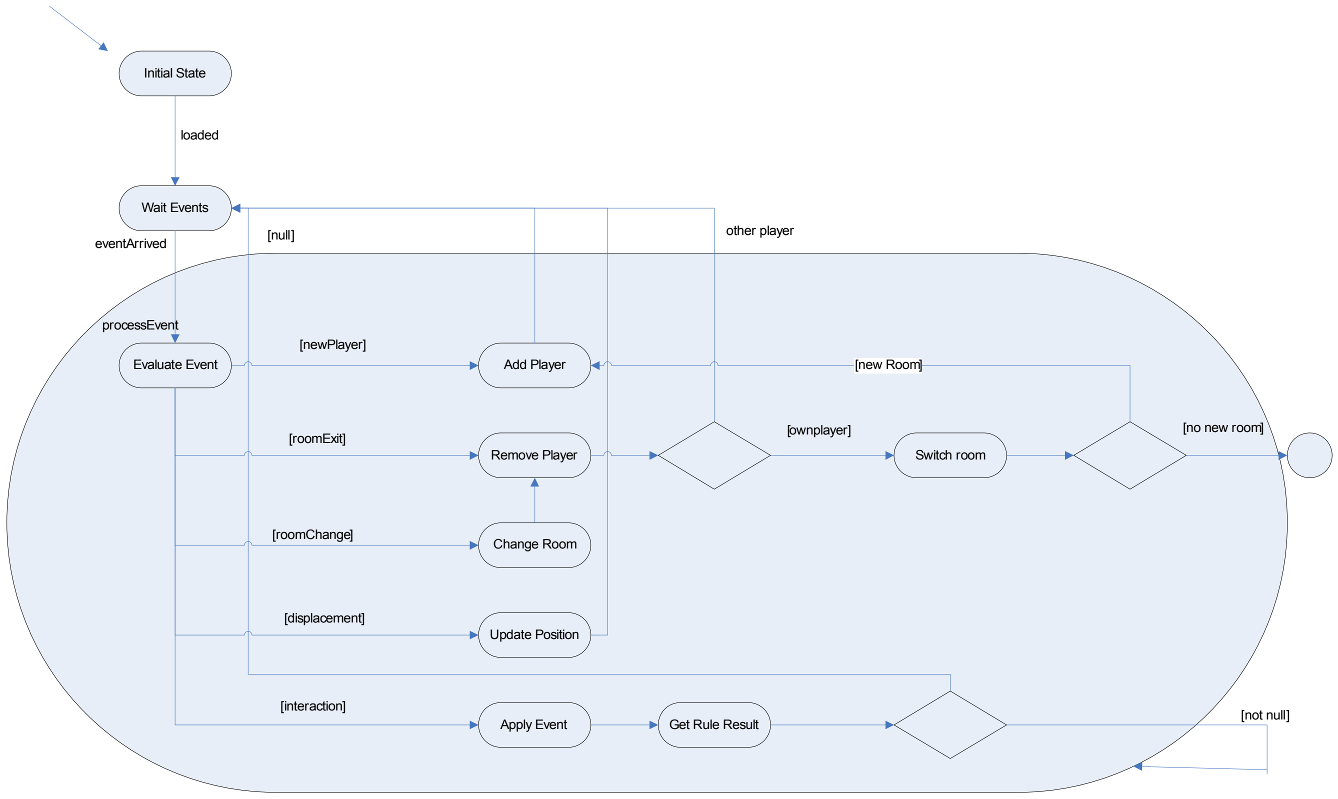
Any movement resulting in a change in position or orientation of the player generates a corresponding event and passes to the client. Client sends this to the server, where the event is distributed and sent back to a number of clients. Having received the event, player applies this event and updates its view of the world accordingly.

5.3.2. Object Interaction Scenario



Player can click on an object in the room. This triggers the request of performing an action and checked against validation. If it passes, a corresponding event is generated and sent to the object interacted. The event is executed according to the rules defined in the object (determining its behavior) and the resulting events are returned to the object. Next the object applies the event and makes the relevant changes in the world. The user updates its view accordingly.

5.3.3. Client-Side Event Processing Scenario



We have defined our Event to be abstract, having many implementing classes, all of which are handled through one channel in Client Game Engine: processEvents() method. In this scenario, how different types of events will be handled is clarified. Once any event arrives from the server to the client, this method is called. It resolves the type of that event and processes accordingly. For example, one of the more complex events, a room change, is resolved and the state machine goes to Change Room state. Remove Player state removes the exiting player from the room. If changing player is the not the owner player (of the executing client), another player has exited the room, so it returns back to wait more events. Otherwise, it leads to the state of Switch Room. Current room is switched to the new room here, and if the new room is null (meaning system exit rather than only room exit), client simply exits. Otherwise, it updates the new room information and gets back to wait for events.

6. Interaction Plan

The game is highly interactive and responses in real time. We identified all possible interaction methods. We have two main entities in our interaction model: Player and Object. The real players and AI players are considered in the Player category and all the other things in the game are considered as objects, such as walls, doors, lasers, keys etc. The methods are then assigned to relevant couples as described below.

Player Player

Players can only speak to each other via chat engine. There are two types of chat message in the game: public chat and private chat. Real players can talk to each other as well as with AI players. AI players can also talk in the public chat. Other than chat players do not interact with each other. Of course, we handle collision detection by defining bounding boxes around players.

Player Object

Our objects in the game can be divided in two groups: the ones that can be used by hand after picking it up into the inventory and the others that can not be picked up into the inventory. Both of the objects can be used in our game. For all objects in the game an action list is defined which contains the actions that can be done with that object. We are thinking of two methods in selecting these actions. First method is using objects' and player's bounding boxes. When an object and player come closer below a predefined distance the action menu of that object prompts to the scene and you can select the desired action from that action list. Second method is using picking strategy. You pick an object in the scene and its action list is prompted to the scene. Of course the desired object must be within a predefined distance. Picking up an object is an also action and you pick up an object by selecting 'pick up' command from the action list. The picked up objects can be used by hand, like keys, and these objects also has defined actions for usage in hand, such as a key lock or unlock locks when it is used by hand. Items in the inventory can also be dropped . The object will free fall to ground from the position of the player's hand. We handle all these interactions with our generic messages and our 'event' strategy.

Object Player

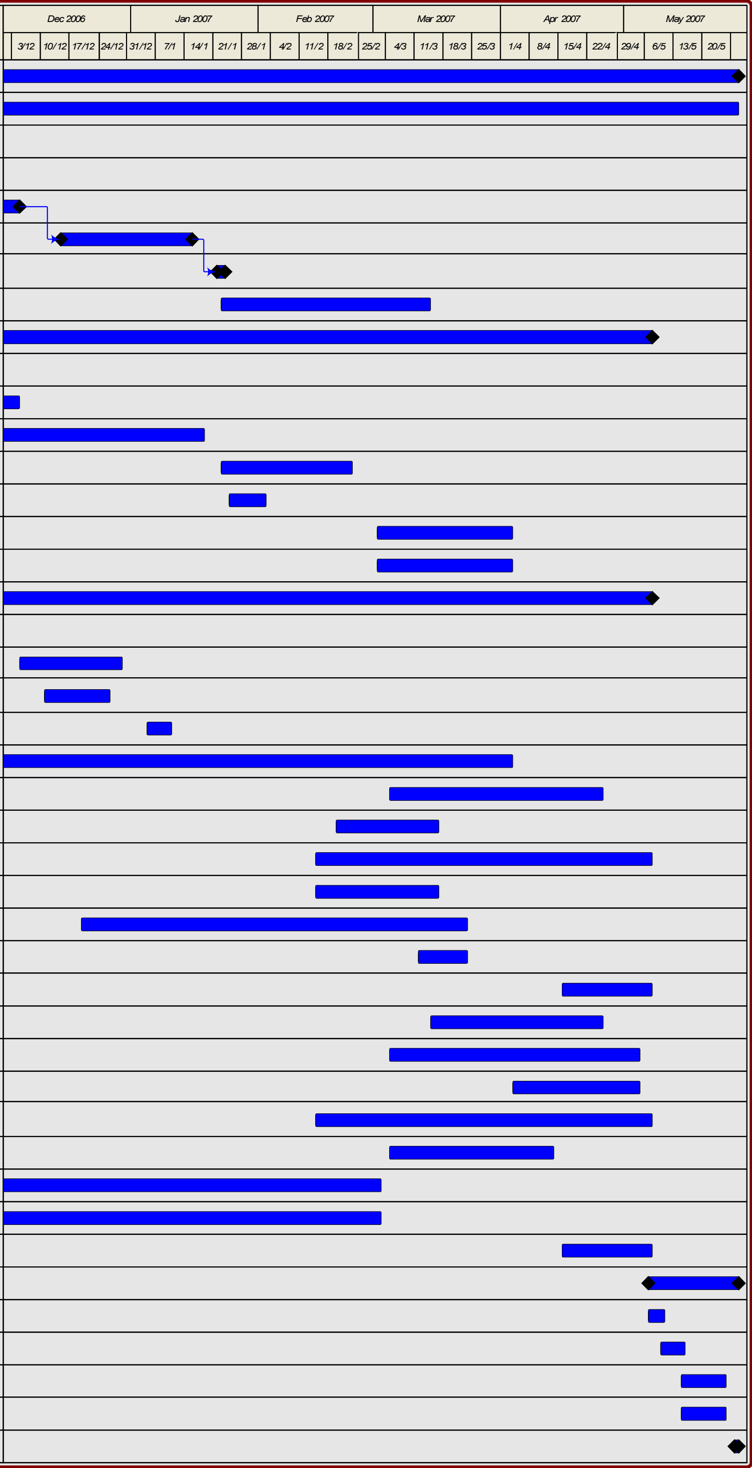
An object can do some effects according to the actions in its action list. They can hit or even harm the players, such as lasers wound or kill people who touch them.

Object Object

An object can also interact with other objects in the game(in fact most of the rules in the game are defined according to these interactions.). Of course these interaction are defined in the action lists of that objects. For example, water and electric wire in our game are two objects. If they interact each other an electric shock is triggered in the environment. Actually, their interactions are mostly defined in the rules.

7. Project Schedule

ID	Task Name	Start	Finish	Duration
1	**Managing Project**	03/10/2006	28/05/2007	34w
2	Project Management	03/10/2006	28/05/2007	34w
3	Propasal	03/10/2006	09/10/2006	1w
4	Analysis Report	20/10/2006	07/11/2006	2w 3d
5	Initial Desing Report	15/11/2006	04/12/2006	2w 4d
6	Final Design Report	15/12/2006	15/01/2007	4w 2d
7	Prototype Demo	22/01/2007	23/01/2007	2d
8	Developing Website	23/01/2007	14/03/2007	7w 2d
9	**Developing Game Concept**	03/10/2006	07/05/2007	31w
10	Game Subject Decision	03/10/2006	09/10/2006	1w
11	Developing Scenerio and Character	03/10/2006	04/12/2006	9w
12	Designing Core Game Play	01/12/2006	18/01/2007	7w
13	Developing Puzzles	23/01/2007	23/02/2007	4w 4d
14	Designing in Game Menu	25/01/2007	02/02/2007	1w 2d
15	Designing Start Menu	02/03/2007	03/04/2007	4w 3d
16	Designing Inventory	02/03/2007	03/04/2007	4w 3d
17	**Developing Game Engine**	03.10.2006	07/05/2007	31w
18	Identifying Engine Requirements	03.10.2006	28/11/2006	8w 1d
19	Designing Engine Detailed	05/12/2006	29/12/2006	3w 4d
20	Designing engine Architexture	11/12/2006	26/12/2006	2w 2d
21	Implementing Engine Prototype	05/01/2007	10/01/2007	4d
22	Main 3D Graphics for a Room	10/10/2006	03/04/2007	25w 1d
23	Implementing Models	05/03/2007	25/04/2007	7w 3d
24	Lighting and Shadows	20/02/2007	16/03/2007	3w 4d
25	Basic Physics Module	15/02/2007	07/05/2007	11w 3d
26	Collision Detection Module	15/02/2007	16/03/2007	4w 2d
27	I/O Module	20/12/2006	23/03/2007	13w 3d
28	Chat Module	12/03/2007	23/03/2007	2w
29	Sound Module	16/04/2007	07/05/2007	3w 1d
30	General AI Module	15/03/2007	25/04/2007	6w
31	Implementing Puzzles to the Game	05/03/2007	04/05/2007	9w
32	Puzzle Learning of AI Players	04/04/2007	04/05/2007	4w 3d
33	Database Module	15/02/2007	07/05/2007	11w 3d
34	GUI Module	05/03/2007	13/04/2007	6w
35	Designing Client Side Network Detailed	01/12/2006	02/03/2007	13w 1d
36	Designing Server Side Network Detailed	01/12/2006	02/03/2007	13w 1d
37	Integrating Modules	16/04/2007	07/05/2007	3w 1d
38	**Final Release**	07/05/2007	28/05/2007	3w 1d
39	Alpha Testing	07/05/2007	10/05/2007	4d
40	Beta Testing	10/05/2007	15/05/2007	4d
41	Installation Manual	15/05/2007	25/05/2007	1w 4d
42	User Manual	15/05/2007	25/05/2007	1w 4d
43	Final Demo	28/05/2007	28/05/2007	1d



We have divided our timeline according to different types of work. We have 4 main categories:

- Project Management
- Game Concept Development
- Game Engine Development
- Final Release

Our project management will go through all the second semester. We are planning to develop website for the second semester during the winter break. This site will contain information about our project team, people can follow our project progress from there when it is ready. We will try to update our website whenever something is added to the game.

We have developed most of our game concept so far. We are certain of our scenerio. Our game's success relies on the quality of the puzzles. The puzzles in our game forces players to work collobaretevely to get out of the rooms and survive. We have created a few puzzles to show this point. But we need to crate more puzzles. This winter break will be good opportunity to develop fantastic and innovative puzzles. Besides puzzles we are planning to finish our in-game menu till the prototype demo. But the start menu and inventory menu can only be developed at the end of March.

Developing the Game Engine is our main responsibility. Everything in the game is contained under this concept. But we have also divided this part into modules. We will try to finish our main modules till the end of the March which are Main 3D graphics for the rooms, design of the client and server side nerwork. Additionally we will add modules that we have designed till the end of April, and lighting and shadows will be handled even earlier. Our game contains limited game physics but it doesn't need very wide physics engine because players will be always in a room. So we can think we can handle this by Ogre's physics engine ODE which is embedded in Ogre. We are planning to finish basic physics module design till the end of April with its main functionality collision detection. I/O Module and Chat Module are very related with eachother and they will be both finished by the end of March. Sound Module is still an optional module for us. At the end of our progress if we would find time we will try to add this module. AI Module implementation is also one of the though tasks that we are facing, we have divided this module into 2 main parts: General AI Module, and Puzzle Learning of AI Players, Both of them are related with Implementing the Puzzles to the Game. All of these parts have to be finished by the end April. Besides these modules Database have to be finished by the end of April as well. At the end we have to integrate all of these modules.

After integration there comes the Final Release phase. For final release we have to finish alpha and beta testing till the middle of the May. We know that we haven't got enough time for testing but we wont have much time left after integration. Of course we will prepare our installation and user's manual before the Final Demo. We hope we will come up with an enjoyable and addictive game at the end of second semester.