CENG 490 Senior Project

"A 3D – Massively Multiplayer Online Game"

Initial Design Report



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

Table of Contents

1. Introduction	.4
1.1. Motivation	. 4
1.2. Project Definition	. 4
1.3. Overview of Project Scope	.4
1.4. Goals and Challenges	
1.5. Comments on the Initial Design Procedure	.6
1.6. Current Status in Prototype Implementation	. 7
2. Game Play and Story	
2.1. General Story of the Game	.9
2.2. Puzzle1	12
2.3. Puzzle2	13
2.4. Puzzle3	14
2.5. Puzzle4	15
2.6. Game Play 1	16
3. Interface Design1	17
3.1. Game Menu Design	17
3.2. In Game Screen Design1	
3.3. Inventory Menu Design	19
4. Game Architecture	21
4.1. Overall Architecture	21
4.2. Message Types	22
5. Detailed Design	23
5.1. Classes Used	23
5.1.1. Server-Side Network Backbone	23
5.1.1.1. Server	
5.1.1.2. Skeleton	24
5.1.1.3. Connection	25
5.1.1.4. Channel	25
5.1.1.5. Message Decoder	26
5.1.1.6 Event Orderer	26
5.1.2. Server-Side Game Engine	26
5.1.2.1. Game State	27
5.1.3. Client-Side Network Backbone	27
5.1.3.1. Client	28
5.1.3.2. Stub	29
5.1.4. Client-Side Game Engine	30
5.1.4.1. Client Game State	
5.1.4.2. Core Objects	
5.1.4.3. Graphics Engine	
5.1.4.4. AI Engine	
5.2. Sequence Diagrams	
5.2.1. Server-Side Network Backbone Sequence Diagram	
5.2.2. Server-Side Game Engine Sequence Diagram	
5.2.3. Client-Side Network Backbone Sequence Diagram	

5.2.4. Client-Side Game Engine Sequence Diagram	45
5.2.5. Chat Engine Sequence Diagram	
5.2.6. Inventory Sequence Diagram	
5.2.7. AI Engine Sequence Diagram	
5.3. Activity Diagrams	
5.3.1. Player Movement Activity Diagram	48
5.3.2. Object Interaction Diagram	
6. Project Schedule	

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.

1.3. Overview of 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 Cube¹" 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 Initial Design Procedure

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.

1.6. 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:

An overly simplified version of the network backbone that can be seen in the initial design was implemented. Several clients get connected to the server and each connection is dealt with in a separate thread. The clients try modifying a global state variable on the server side. Upon modification a message is sent to each of the client that is connected to the server. The client uses separate threads for sending and receiving messages.

On the other hand, for computer graphics we have extended some sample applications that were developed on the OGRE3D library. We were able to generate a room, with a texture on each of its walls and several objects positioned on various locations inside the room. For the time being, our character is a ninja but that is planned to be changed as soon as possible. Of course, we need to get more acquainted with the OGRE3D library. We can change the camera position and orientation and

view the room from different angles. Furthermore lighting and shading is also accomplished. Below is a screen-shot from the implementation.



A scene from prototype implementation

2. Game Play and Story

In this section of our initial design report I will try to tell 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 scenerio, 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 position. Game flow is generally incremental in these games. But in our game these concepts of usual FPS games don't 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 won't 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.



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. 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, since each door can only be unlocked for 10 seconds. Others are free to get out of the room whenever the door is unlocked.



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 gameplay 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.

Untitled Page - Microsoft Internet Explorer Dosya Düzen Görünüm Sik Kullanılanlar Araçlar Yard	im	1111					
🕞 Geri 👻 🕑 - 💌 🛃 🏠 🔎 Ara 🌟 S	ik Kullanılanlar 🕢 🍰 -	🎍 🗹 🔹	, 🛍				
dres 🚳 http://localhost:4045/Main_Menu/						>	Git Bağlantılar
	Enter The Cube View Profile Hall Of Frame Options Credits Help Leave The Cube	Game Title Introductory For Game	Part				
	Menu Item(*)		~				
				-			
Bitti						Sere Yere	l intranet
🛃 Başlat 👘 🤌 🥙 🕼 🔌 🗿 YouT	🗁 dass 🧐 metu	🗁 Main	Main	🦞 adsız	inven	Ø Untitl	TR 🔇 🔽 17:32

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 gameplay of the game, controls etc.), Leave The Cube(quit the game).

🗿 Untitled Page - Microsoft Internet Explorer									
Dosya Düzen Görünüm Sık Kullanılarlar Araçlar Yardım									.
😋 Geri 🔹 🐑 🔹 😰 🏠 🔎 Ara 🌟 Sik Kullanilanila	ar 🧭 💈	3 · 🎍 🛛	- 🗌	Ø.					
Adres 🗃 http://localhost:4084/Paused_Game_Menu/Default.aspx							~	🔁 Git	Bağlantılar »
		Game Tit	le						
		Resume Ga	me						
		View Profi							
		Options Help							
		Leave The C	ube						
🙆 Bitti							S	/erel intrane	et 🗸
🛃 Başlat 🖉 🥙 🟟 🎽 🌁 You 🎓 das	🧐 met	🗁 Mai	Pau	🛛 🦉 ads	🚺 Yen	🖹 inv	🖉 Unti		 17:31

Paused Menu Interface

3.2. In Game Screen Design



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

🕘 Untit	tled Page - Microsoft I	nternet Explorer							- 7 X
Dosya	Düzen Görünüm Sık K	ullanılanlar Araçl <mark>ar Yardım</mark>							A.
Ge	ri - 🕥 - 💌 💈] 🏠 🔎 Ara 🤸 Sik Ku	ilanianlar 🧭 🍃	🎍 🗷 •					
Adres 🧕	http://localhost:4000/Inv	entory_Menu/						× [Git Bağlantılar 🌺
			Invento	ry Menu]				<u>^</u>
					Selecte	d Item			
lter	m	Item	Item						
lter	m	Item	Item						
					Name±	xplanatory Text			
lter	m	ltem	ltem						
lter	m	ltem	ltem						
							_		
Ec	quip Item Unequip	Item Leave Item	Resume Game						
] []							(•
🕘 Bitti			The second	1.00				×	el intranet
💾 Ba	aşlat 🔰 🧭 🕫 🕯	🥱 🤲 🍪 YouT 📁	class 🧐 metu	Main	Inve	🦉 adsız	inven	🖉 Untitl	TR 🔇 🚰 17:33

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. 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.



Overall Game Architecture

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 and allowing it to support large number of clients. Client game engine generates and receives events and holds a reflection of the game state in 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.

4.2. Message Types

The format of the messages that are to be sent between the clients and the server are provided below. It does not cover all of the cases however a majority of the type of messages are outlined below:

synchronization	player_id	state	position	direction
movement	player_id	movementType	toPosition	{currentPosition}
changeDirection	player_id	toDirection	currentDirec	tion
interaction	player_id	object_id	action_id	parameter_list

Synchronization Messages:

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.

Movement Messages:

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 is for synchronization purposes.

Direction Change Messages:

The player indicated by *player_id* changes his/her direction from *currentDirection* to *toDirection*.

Interaction:

Interaction messages are more generic than the others, that can be used for various purposes. However, the intended purpose of this type of message 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.

5. Detailed Design

5.1. Classes Used

5.1.1. Server-Side Network Backbone



5.1.1.1. Server

Server
-serverProperties : Properties -globalGameState : GameState * -rooms : Vector <roomdispatcher *=""> * -players : Vector<playerdispatcher *=""> * -connnections : Hashtable * -channels : Vector<channeb -skeleton : static Skeleton</channeb </playerdispatcher></roomdispatcher>
+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 hostname 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</message>
+Connection(in game_connection : GameState *, in RoomDispatcher : RoomDispatcher *, in player_connection : PlayerDispatcher) : void -receiveSingleMessage() : string throw ConnectionException +processIncomingMessages() : void +sendBack(in msg : Message) : 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 *=""></connection>
-waitingEventsBuffer : Vector <events></events>
+subcribe(in sub_connection : Connection *) : void
+unsubscribe(in unsub_connection : Connection *) : void throw ConnectionNotFoundException
+getAllSubscribers() : Vector <connection *=""></connection>
+cleanAll():void
+pushEvents(in events : Vector <event>) : void</event>
+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 () funcation call.

5.1.1.5. Message Decoder

MessageDecoder
-instance : MessageDecoder
-MessageDecoder() +decode(in serializedMessage : string) : Message throw ParseException

The "Message Decoder" is designed based on the singleton design-pattern. In theory, the server should contain only one instance. If this restriction is found unnecessary, it might be dropped before the final design.

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

5.1.1.6 Event Orderer

EventOrderer
-buffer : BinarySearchTree <event></event>
-decoderInstance : MessageOrderer
-generateEvent(msg:Message) : Event
+flushBuffer : Vector <event> throw OrderingException</event>
+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

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. 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 consequnces 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.

5.1.3. Client-Side Network Backbone



5.1.3.1. Client

Client
-clientID : string
-clientProperties : Properties
-hostProperties : Properties
-clientGameState : ClientGameState *
-rooms: Vector <roomdispatcher *=""> *</roomdispatcher>
-players : Vector <playerdispatcher *=""> *</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

Messegaencoder
-instance : MessegaEncoder
-MessageEncoder
+encode(in event : Event throw MessegaEncodeException) : Message

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 *</stubstate>
-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



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

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.

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</couple>

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

Object class contains meshfile, id, state and Rules of that object. Meshfile 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
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 elapsedTime (beginTime is used to calculate the elapsedTime). 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

PlayerDispatcher

#position:Position #player:Player * #dispatcher:RoomDispatcher *

Inventory
-equippedItem : Object * -itemList[] : Object *
<pre>+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 *[]</pre>

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.

	RoomDispatcher
-r	oom : Room *
-p	osition : Position *
-p	

Renderable	
+getPosition(): Position	
+getMesh():string	
+getAction() : int	

Dispatcher

+generateEvent() : Event *

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. Graphics Engine



GraphicsEngine is our core class of rendering the scenes of the game. According to the mode we either render a menu or a scene(renderScene(), displayMenu()). Our scenes are composed of the rendering of Objects, Players and the Rooms by using the EntitiyRenderer and RoomRenderer classes which renders by adding nodes to the scene nodes of the main scene. Graphics Engine is abstracted by the Keyboard, Text and Mouse Listener classes which are our IO handlers. The rendering will change according to that IO. The camera is also instantiated here. But mainly we are bound to OGRE for our graphics rendering. It contains all the functions and attributes for a clean rendering.

GraphicsEngine
<pre>#roomRen : RoomRenderer* #entRen[] : EntityRenderer * #cam : Camera *</pre>
#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
+RoomRenderet(in ren : Renderable *)

+render() : void

Renderer	
+render():void	

TextListener
+getText() : void

KeyboardListener
+keyPressed() : void

MouseListener
+mouseMode() : void



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 rulememory for the AI player by resolving events and adding rules to that list. We do that by using resolveEvents() and addRelation() functions.

AlEngine
#AIPlayerDispatcherVector : Vector <aiplayerdispatcher *=""></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*					

AlPlayer
+AlPlayer()

5.2. Sequence Diagrams



5.2.1. Server-Side Network Backbone 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. Whenever a message 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 Sequence Diagram

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 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 Sequence Diagram

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 gameplay. 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 Sequence Diagram



Chat Engine

ChatEngine has the only responsibility of sending and receiving ChatMessage objects, which are simply text messages for the interaction of the players. ChatEngine is accessed by the client and server engines by means of send/getChatMessage() methods.

5.2.6. Inventory Sequence Diagram



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. Al Engine Sequence Diagram





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. Activity Diagrams

5.3.1. Player Movement Activity Diagram



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 Diagram



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.

6. Project Schedule

Ø	Took likano	Stor	Finish	Duration	
1	Managing Project	02.10.2005	25.05.2007	34w	·
2	Project Management	02.10.2005	25.05,2007	34w	
3	Propasal	08.10.2005	09.10.2006	tw	
4	Analysis Roport	20.10.2006	07.11.2005	2w 3d	
6	Initial Desing Report	15.11,2005	04.12.2005	2w4d	
6	Final Design Report	15.12,2005	15.01.2007	4w 2d	
7	Prototype Domo	22.01.2007	23.01.2007	51	
8	Developing Game Concept	(0.10,2005	07.05.2007	31w	
9	Game Subject Decision	03.10.2005	09,10,2005	5w	
10	Developing Sceneric and Character	03.10.2005	04.12.2005	9w	
11	Designing Core Game Play	01.12.2005	18.01.2007	7w	
12	Designing Nenu	01,12,2005	18.01,2007	7₩	
1	Developing Game Engine	03.10.2006	07.05.2007	31w	
4	Identifying Engine Requirements	08.10.2005	28.11.2005	Øwitd	
15	Designing Engine Detailed	05.12,2005	29.12.2005	3w 4d	
倍	Designing engine Architexture	11.12.2005	25.12.2005	2w2d	*
17	Implementing Engine Prototype	05.01,2007	10.01.2007	40	
18	aD Graphics Module	15.02.2007	07.05.2007	11w-3d	8
19	Basic Physics Module	15.02,2007	07.05.2007	11w3d	**************************************
20	Califsian Dataction Module	15.02.2007	07.05.2007	t1w.3d	**************************************
21	I/O Modure	15.02.2007	07.05.2007	tiw 3d	*
22	Sound Module	15.02,2007	07.05.2007	11# 3d	() () () () () () () () () () () () () (
23	Lighting Module	15.02.2007	07.052007	11w3d	ALC: 100 100 100 100 100 100 100 100 100 10
24	N Modulo	15,02,2007	07.05.2007	ttw 3d	ALCON.
25	Network Modulo	15.02.2007	07.05.2007	11w3d	*
26	Darabase Module	15.02.2007	07.05.2007	11w-3d	ALL
27	GUI Module	15.02.2007	07.05.2007	11w3d	*
28	Final Release	07.05.2007	28.05.2007	3w1d	()
29	Alpha Testing	07.05.2007	10.05.2007	40	*
30	Bota Tosling	10.05.2007	15.05.2007	40	
31	Installation Manual	15.05.2007	25.05/2007	twi4d	****
32	User Manual	15,05,2007	25.05.2007	1w 4d	****
33	Final Demo	28.05.2007	28.05.2007	10	