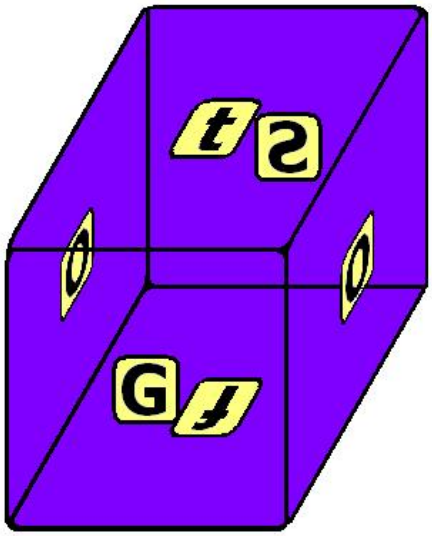
CENG 490
Senior Project

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

Developer's Manual

by



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

Table of Contents

1. Introduction

This document briefly outlines the steps that need to be taken for enhancing the functionalities of the Ma3e and therefore should be treated as introductory material. Some of the modifications presented in this document can easily be achieved by developers familiar with Network Programming and XML. However, for advanced upgrades, familiarity with OGRE-3D Software Development Environment is essential.

The Ma3e is developed and executed on the Windows XP (SP2) operating system. The protocol specification and the format of the exchanged messages can be found in the Final Design Report [1]. The graphical functionalities have been developed on top of the OGRE-3D Graphics Engine. For further information, the developer is recommended to consult its Application Programming Interface specifications [2].

The document proceeds as follows. First a brief overview of the architecture is given. It is followed by a section that explains how modifications on the communication (client-server) protocol can be made. The subsequent sections elaborate on graphical modifications (scene and menu preparation). The "Puzzle Deployment" section describes how new puzzles can be integrated into the game.

2. Architectural Overview

The main architecture of the game lies on the two distinct parts: Client Side and Server Side. Client Side is responsible for the playing of the game on a single computer. Server Side is responsible for the communication and synchronization of the players.

Architecture of the Client Side depends on the 3 main classes, ClientSideGameEngine, GraphicsEngine and MyListener. ClientSideGameEngine is responsible for the interaction of the game with server side. GraphicsEngine is responsible for the rendering of the game. It uses some classes such as Scene which is responsible for the rendering of a room. It also manages the menus in the game and the handles of these menus. MyListener is responsible for the real-time I/O interaction of the game. It takes the I/O messages and handles them.

The Server Side consists mainly of classes that are responsible for the underlying Client-Server interaction (Message, MessageFactory, MessageResolver, Channel, Connection); classes that represent the global game state (Cube, Room, Player, Object) and finally the core (Server, Skeleton).

[1] GOSOft, "The Ma3e – Final Design Report." Online. Available: http://senior.ceng.metu.edu.tr/2007/gosoft/documents/final_design_report.pdf
[2] "OGRE: Documentation." 09 March 2007. Online. Available: http://www.ogre3d.org/index.php?option=com_content&task=view&id=407&Itemid=106

3. Enhancement on Client-Server Communication

3.1. Message Preparation

Messages make up the core of the client-server communication. "Message.h" and "Message.cpp" contain the necessary implementation.

IMPORTANT: *Both the client and the server should either access the same code segments ("Message.h" / "Message.cpp") or their exact copies.*

The Message is a container class for all attributes (e.g. Identifier, Timestamp, ChatMessage, GeometricVector, Position, Direction, etc.) that need to be conveyed during client-server communication. If the developer decides to use any of these building blocks to compose different messages, then he[3] is encouraged to study the implementation on his own.

While implementing new message types, the developer should modify the following functions as to integrate the desired functionalities:

```
bool Message::fromBitStream (string bitstream);
string Message::toBitStream();
```

"fromBitStream" parses the incoming bitstream and constructs the corresponding Message entity. If the bitstream has been parsed successfully then it should be reported as "true" in its return value. "toBitStream" accomplishes just the opposite task.

IMPORTANT: *At all times, the developer is encouraged to consult the Final Design Report [[4]] for more information on the design and the protocol specification.*

3.2. Integration into Client/Server Code

On the server side, the following classes are responsible for interaction with the communication backbone:

- MessageResolver.h
- MessageFactory.h

As the name implies, "MessageResolver.h" processes and categorizes an incoming message and invokes the relevant methods that make the necessary changes in the global game state. It is important to realize that some messages such as MOVEMENT and OBJECT_STATE need also be distributed to other players in the room. This functionality is implemented within the MessageResolver.

[3] All references of the pronoun refer to third persons without bias towards a specific gender.
[4] GOSOft, "The Ma3e – Final Design Report." Online. Available: http://senior.ceng.metu.edu.tr/2007/gosoft/documents/final_design_report.pdf

"MessageFactory.h" contains the static methods that generate different types of messages from various user arguments. The return types of each function within the MessageFactory class are Message entities.

Similarly on the client side, the following classes/functions accomplish the similar tasks:

- MessageTransmitter.h
- MessageTransmitter.cpp
- ClientGameEngine.cpp (DWORD WINAPI getMessages(LPVOID n))

The MessageTransmitter class enables the client machine to seamlessly send messages to the server. The server then takes on the job of distribution of the messages to other client machines.

The "getMessages" function within the ClientGameEngine acts as a separate thread for listening to all messages coming from the server. These messages imply changes in the local game state and therefore the relevant functions of the Graphics Engine are called.

IMPORTANT: *Due to its multi-threaded nature, care has to be taken when the getMessages function modifies a local game state parameter while the Graphics Engine is constantly seeking for changes within the local game state. Appropriately assigning critical sections and enabling state modification only through function calls was the method taken during development.*

The developer should be cautious as to make the necessary modifications in these integration classes/functions whenever a new message type is to be implemented.

4. Room Scene Preparation

4.1. What is dotScene?

DotScene is the XML representation of the 3d scene to be loaded by Ogre3D Engine. The complete syntax can be found in 'dotscene.dtd' in the folder 'documents'. The specific structure to be used by The Maze is as follows:

```
<scene id=....>
        <environment>
                <colourAmbient r="?" g="?" b="?" a="?" />
                <colourBackground r="0" g="0" b="0" a="1" />
        </environment>
        <nodes>
                <node name="Players" id="999">
                        ** CANNOT HAVE ANY CHILDREN**
                </node>
                <node name="NonCollisionObjects" id="333">
                        ** light Nodes**
                        ** plane Nodes (walls)**
                </node>
                <node name='CollisionObjects' id="666">
```

```
                    ** object Nodes**
                </node>
        </nodes>
</scene>
```

4.2. Room Structure

Rooms should have specific boundaries. In terms of world coordinates:

- upVector of the camera : +y axis
- ground : xz plane at y=0 where x in [-1500, +1500], z in [-1500, +1500].
- ceiling : xz plane at y=1500 where x in [-1500, +1500], z in [-1500, +1500].

Initial placement of the camera is static, and looks at the direction of –z.

4.3. Types of Nodes

There are mainly three types of nodes, light, plane and object nodes.

Light Nodes

These represent light in the scene. The schema and an example instance are as follows:

```
<!ELEMENT light (position?, normal?, colourDiffuse?, colourSpecular?, lightRange?,
lightAttenuation?, userDataReference?)>

<!ATTLIST light
        name            CDATA           #IMPLIED
        id              ID              #IMPLIED
        type            (point | directional | spot | radPoint) "point"
        visible         (true | false) "true"
        castShadows     (true | false) "true"
>

<node name="LightNode" id="0">
        <light name="Light1" id="1" type="point" visible="true">
                <position x="0" y="500" z="0" />
                <lightAttenuation range="8000" constant="1" linear="0" quadratic="0"
                />
                <colourDiffuse r="1.0" g="1.0" b="1.0" a="1" />
                <colourSpecular r="0" g="0" b="1" a="1" />
        </light>
</node>
```

Plane Nodes

These represent static planes(walls) with textures. The schema and an example instance are as follows:

```
<!ELEMENT plane (normal, upVector?, vertexBuffer?, indexBuffer?)>
<!ATTLIST plane
        name                CDATA       #REQUIRED
        id                  ID          #IMPLIED
        distance            CDATA       #REQUIRED
        width               CDATA       #REQUIRED
        height              CDATA       #REQUIRED
        xSegments           CDATA       #DEFAULT "1"
        ySegments           CDATA       #DEFAULT "1"
        numTexCoordSets     CDATA       #DEFAULT "1"
        uTile               CDATA       #DEFAULT "1"
        vTile               CDATA       #DEFAULT "1"
        material            CDATA       #IMPLIED
        normals             (true | false)  "true"
>
<node name="groundNode" id="20">
        <plane name="ground" id = "21" distance="0" width="3000" height="3000"
        xSegments="20" ySegments="20" numTexCoordSets="1" uTile="5" vTile="5"
        material="PrairieWind" normals="true">

                <normal x="0" y="1" z="0" />
                <upVector x="0" y="0" z="1" />
        </plane>
</node>
```

The coordinates are probably not subject to any change. The important part is related with textures:

- uTile: how many times the texture is multiplied in u dimension.
- vTile: how many times the texture is multiplied in v dimension.
- material: the name of the material applied to the wall

Material scripting is explained later in the section 4.5.

Object Nodes

These represent static/dynamic object entities displayed my model meshes. The schema and an example instance are as follows:

```
<!ELEMENT entity (vertexBuffer?, indexBuffer?, userDataReference?)>
<!ATTLIST entity
        name            CDATA       #IMPLIED
        id              ID          #IMPLIED
        meshFile        CDATA       #REQUIRED
        materialFile    CDATA       #IMPLIED
        static          (true | false)  "false"
        castShadows     (true | false)  "true"
>
```

```
<node name="dragonNode" id="39">
	<position x="250" y="400" z="-500" />
	<rotation qx="0" qy="1" qz="0" qw="0.000796274" />
	<scale x="3" y="3" z="3" />
	<entity name="dragonEntity" id="77" meshFile="dragon.mesh" static="false"
/>
</node>
```

The elements are explained as follows:

- position: x-y-z coordinate
- rotation: Quaternion representing the orientation (see table at section 4.7, Quaternion Table)
- scale: multiplier in x-y-z dimensions
- meshFile: name of the model mesh file

4.4. Media Archieve Structure

The media archieve is found under 'media' folder, and classifies the files such that:

- media
  - models: mesh files
  - materials
    - textures: image files used by material scripts
    - scripts: material scripts used by our nodes

4.5. Material Scripting

When you want to use a texture image, you have to write the corresponding script file for it. It is suggested to add the script in the file 'NewMaterials.material' under /media/materials/scripts. A sample material file is as follows:

```
material PrairieWind
{
	technique
	{
		pass
		{
			texture_unit
			{
				texture PrairieWind
				scale 0.5 0.5
			}
		}
	}
}
```

4.6. Viewing Model Meshes

To view the meshes in the media archieve, you can use ShowMesh program under the folder:

\ma3e\490.2007\gosoft\tools\ShowMesh

4.7. Quaternion Table

Rotation around +y axis:

Rotation amount	**qx**	**qy**	**qz**	**qw**
PI/4	0	0.3825	0	0.923956
PI/2	0	0.706825	0	0.707388
3PI/4	0	0.923651	0	0.383235
PI	0	1	0	0.000796274
5PI/4	0	0.92426	0	-0.381764
3PI/2	0	0.707951	0	-0.706262
7PI/4	0	0.383971	0	-0.923345

Copy-paste these values to achieve that much of a rotation.

5. Menu Preparation

Menus in the game are prepared through an XML file which is loaded to the game in the beginning. For the time being there are 3 menus in the game: Main Menu, Game Menu and Object Menu. Main menu is displayed in the beginning and inside the game when the game is paused. Game Menu is displayed through the game. Object Menu is displayed when a selectable object is selected. It shows the actions of the object. Main Menu and Game Menus are stable but new Object Menus can be added to the game. The menus in the game are handled by making the active menu visible and other menus invisible.

You can find a layout of our menu file(XML file) below:

```
<?xml version="1.0" ?>
- <GUILayout>
- <Window Type="DefaultGUISheet" Name="MainMenu">
- <Window Type="DefaultGUISheet" Name="debug_overlay">
<Property Name="AlwaysOnTop" Value="True" />
<Property Name="Disabled" Value="True" />
</Window>
- <Window Type="DefaultGUISheet" Name="OpenMenu">
<Property Name="RelativeMaxSize" Value="w:1 h:2" />
<Property Name="Size" Value="w:1 h:2" />
<Property Name="Visible" Value="True" />
-  <Window  Type="TaharezLook/FrameWindow"
Name="OpenMenu/MainWin">
```

```
<Property Name="Position" Value="x:0.4 y:0.1" />
<Property Name="RelativeMaxSize" Value="w:0.23 h:0.43" />
<Property Name="RelativeMinSize" Value="w:0.23 h:0.43" />
<Property Name="Size" Value="w:0.23 h:0.43" />
<Property Name="Text" Value="Main Menu" />
<Property Name="Alpha" Value="0.75" />
<Property Name="CloseButtonEnabled" Value="False" />
- <Window Type="TaharezLook/Button" Name="ResumeGame">
<Property Name="Position" Value="x:0.25 y:0.2" />
<Property Name="RelativeMaxSize" Value="w:0.12 h:0.04" />
<Property Name="RelativeMinSize" Value="w:0.12 h:0.04" />
<Property Name="Size" Value="w:0.33 h:0.04" />
<Property Name="Text" Value="Play Game" />
<Property Name="InheritsAlpha" Value="False" />
  </Window>
  </Window>
  </Window>
  </Window>
  </GUILayout>
```

The example above shows the layout of a menu called Open Menu with one button: Resume Game. You can adjust some properties of the menu like size, position or text via this XML file format.

6. Puzzle Deployment

The interaction of players with objects in the room and the rules that determine how the states of objects interrelate constitute the puzzles in the Ma3e. It is assumed that every object has a current state and that the state of objects can be manipulated either by player interaction or by the consequence of rule application.

Within this respect, we may categorize the implementation into classes that are responsible for:

- describing what to do when an object is in a certain state,
- communication-wise, enabling the player-object interaction to take place,
- checking the current states of objects in the rooms and applying the predefined rules.

6.1. Describing Objects

A new object can be registered into the framework by implementing the header and body classes that extend the Object class. It is necessary and sufficient to override the following methods of inherited from the Object class:

- *constructor* (string id, int roomID);
- virtual void performAction (string actionName);
- virtual void drawState ();

Every object has an id and every object must be placed in a room. The constructor enables an instance of this object to be generated depending on these two parameters.

When the client application receives a message indicating a player-object interaction, the performAction method is called with the *actionName* parameter. For instance, the TURN-ON action on a switch that has a current state of OFF causes it to change its state to ON. The function should be implemented such that this functionality is achieved.

Finally, the drawState method interacts with the Graphics Engine such that the current state of the object is drawn.

IMPORTANT: *The developer is strongly encouraged to study the code in Object.h and Object.cpp before implementing a new object.*

IMPORTANT: *Objects are registered into the Graphics Engine inside the bool GraphicsEngine::setup (void) function. The developer is encouraged to study the current implementation for a better understanding.*

6.2. Enabling Player-Object Interaction

The Player-Object interaction takes place whenever a player picks an object and selects an appropriate action to be performed from the pop-up menu.

The Message that enables this information to be communicated among the hosts is the OBJECT_STATE message. It carries the:

- object ID,
- current object state,
- room ID (in which the object is located),
- an additional parameter to be used on future extensions,
- and a timestamp.

It is very likely that developers will not need to modify this message structure, therefore further details are avoided.

6.3. Rule Processing

Whenever a client sends to the server an OBJECT_STATE message, the server records the latest state of the object. Please note that in the server side, only "Object.h" and "Object.cpp" are present. Inherited classes are not included. The reason is that the server does not need to know anything about the implementation details specific to a given object. The server is only interested in the state of the objects.

IMPORTANT: "Object.h" and "Object.cpp" on the server-side are different from their complements on the client-side. The developer is strongly encouraged to study the difference on his own.

The Room::resolveObjectState (string objectID, string state) method is responsible for processing and incoming OBJECT_STATE message. The state of the object is

recorded and then the Rule Engine is invoked to see if any rules are now applicable. The Rule Engine is implemented by the static functions within "RuleResolver.h".

***IMPORTANT**: For integrating other rules, it is necessary and sufficient that the developer modifies the static void applyRules (Room * room) function of RuleResolver.*

IMPORTANT: Objects are registered into the global game state of the server inside void Cube::generateAllObjects (). The developer is encouraged to study the current implementation for a better understanding.

7. Conclusion and Future Work

When we first start the project we have many goals and challenges for the project. We accomplished some of them and we failed some of them due to some reasons, especially the insufficient time is the main reason for the points we couldn't finish. Let's go over the some points of the project.

The main server and client properties are accomplished. The server can take a load of 100 players. The game can be played with many players synchronously. The actions of any player can be viewed by the other players simultaneously.

The main 3D rendering and visualization is accomplished. We have lots of 3D models. And the rendering quality of the scene is very high. FPS in a scene is between 50 and 70 which are very desirable.

The database feature is not implemented due to time reasons, so we don't take the records of the players and we don't do any authentication. Before the players get into the game, they adjust the properties of the player by the Ma3eClient.config, then get into the game by Ma3eClient.exe.

Puzzles are implemented partly in the game. Due to time reasons we couldn't prepare as many puzzles as we wanted. Also lack of good 3d models is a reason. We aimed to implement the puzzles via XML file, but for the time being the puzzles in the game are implemented partly by XML file, partly by hard-coded. But we have the enough infrastructures to implement all the puzzles in XML format.

Menus are also implemented successfully. All the menus are implemented. But new object menus can be added when new puzzles are added.

Room preparation is accomplished successfully. All the rooms in the game are implemented via XML files.

As you see most of the goals in the project are achieved. Lack of many puzzles and database interaction are the main flaw of our game, but new puzzles, menus and rooms can be added to the game without changing the architecture.