University of Lodz

Faculty of Physics and Applied Computer Science

Piotr Chojnacki

Justyna Duczmańska

The design and implementation of multiplayer, web-based, 2D game made in client-server architecture

Engineer's thesis on APPLIED COMPUTER SCIENCE specialization

Under the supervision of

PhD Grzegorz Szewczyk

Centria University of Applied Sciences

May, 2014

Keywords

computer game, multiplayer, client-server

Table of contents

1. Abstract

Even though Ludwig Wittgenstein – Austrian-British philosopher – was the first man to address a definition of word “game”[1] in his *Philosophical Investigations* book in 1953, games were known to human since the ancient times[2]. Because people are social beings and they long to interact with each other, a purpose of majority of games was – and still is – to spend time with others in a friendly atmosphere. It is also in human nature to face challenges and compete with other people and it is reflected in social games as well. Of course the topic is much more complex, but the fact is that games are one of many important experiences in human’s life – they teach us creativity, solving problems, interacting with other people and reducing stress of everyday life [3]. One could say that today’s popularity of games is caused by nothing else but marketing. The truth is that marketing only makes people decide to choose one title over another, but the need of playing itself lies in human nature. Psychologists still cannot fully answer the question of why do people like to play games so much, but most of them agree that it is stress in school, in work, but also boredom that motivates people to do it.

Interactive entertainment industry is currently one of the most influential branches of the industry. Its beginnings in the mid-1970s were humble – only small groups of people decided to develop games and they lacked players interested in their products. Most games were created as a hobby projects without purpose of earning money. Soon the amount of players was increasing rapidly along with the games popularity. Tiny market of video games turned into a giant industry after few decades and now its income is estimated to be 24.75 billion US$ in 2011 only in the USA[4].

A few of most recognizable video games developers are: Electronics Arts, Inc., Ubisoft Entertainment S.A., Activision Blizzard, Nintendo.

For us, video games were always one of the greatest hobbies. Since we remember, we were interested in playing, but also in thinking about how do those games are developed – how people can create such interactive, virtual – but realistic - worlds.

During studies we already had an opportunity to work on one video game project, which was created by us from scratch without using any external engine. It was a 3D game which allowed the player to walk in three-dimensional abandoned mine in order to find as much gold as possible and leave before the mine collapses. It helped us a lot in understanding problems considering game development and thanks to that we also noticed how working on a complex project like this can boost one's programming and designing skills. Though we were excited about results we achieved, we saw mistakes we made during development and we were more motivated in creating something even more extended and complicated. This is how the idea for this project was born.

2. Glossary of Terms

3D	Three-dimensional
2D	Two-dimensional
RPG	Role-playing game – lets user to create his character and develop its skills and experience.
MMORPG	Massive multiplayer online RPG – online role-playing game, which allows hundreds or thousands of people to play together at the same time.
OS	Operating system
IDE	Integrated development environment
NPC	Non-player character
WYSIWYG	"What you see is what you got". It is a type of editors that shows map exactly in the way it will be later displayed in game.

3. Introduction

3.1. Motivation

In our minds there was always an idea of creating our own computer roleplaying game, because we really liked this genre, but we found that modern games are insufficient for some groups of players. Modern games are short and too easy, because they're targeted to the large amounts of players from around the world. It is clear that video game developers want to sell as much copies of their product as possible but with our hobby project we are planning to satisfy the needs of players like us, who cannot really find a game playing which they would spend many hours, because they would find it to be a real challenge for them.

We noticed that majority of game projects created at universities are educational games. When we asked one of our friends who was working on educational game project at his university why he chose this game type, he replied that because it was more likely to be accepted. We decided not to follow this pattern, as so-called educational games are not in fact that educational as people think they are[5] – mainly because they are not so exciting, whereas other games let the player dive into its world and therefore make him more susceptible to its content. This means that it does not matter if the game focuses on shooting aliens from other planet, or gathering new equipment through making quests for non-player characters - if only the game is exciting and additionally forces the player to think creatively of how to solve problems, teaches to make ethical choices. It is much more valuable than educational games for kids.

3.2. Responsibilities and contribution

- Piotr Chojnacki
  - General gameplay and game mechanics concept
  - Managing and supporting project repository
  - Game client project
  - Game client implementation
  - Data exchange model between client and server (communication)
  - Writing documentation

  Thesis project contribution: 50%

- Justyna Duczmanska
  - General gameplay and game mechanics concept
  - Game server project
  - Implementation of game server application
  - Data model used by client and server applications
  - Sample graphics
  - PostgreSQL database design and implementation
  - Writing documentation

  Thesis project contribution: 50%

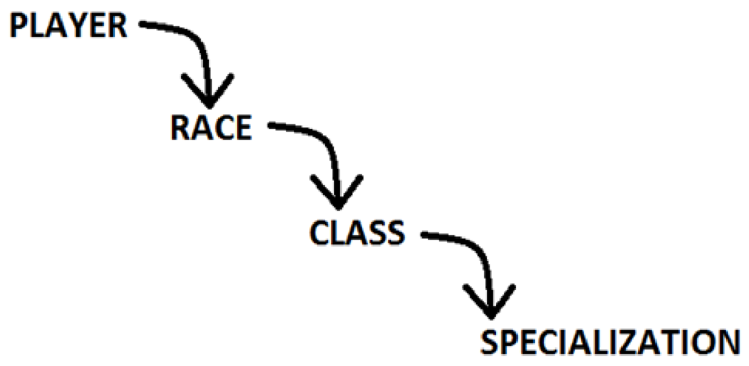
4. Technical design and requirements

4.1. Brief description of the project

The Trinea project should meet the following requirements:

- The game is 2D MMORPG.
- The game's action takes place in a mystic realm of Trinea.
- The land will be dwelled by mythical creatures and four playable humanoid races:
    - **Skalds –** Tall, powerful people who live in winter, mountainous terrains. They are excellent warriors.
    - **Cerians –** Mysterious people living in woods. They are not well known by other races. They are great rangers.
    - **Him'Jar –** People living in deserts. As well as they are prepared for living in very unfavorable conditions, they are perfect mages.
    - **Wardens –** Most similar to normal people.

  Each race has its own culture, traditions, skills, architecture and original look.
- One of advantages of RPG games is a player's ability to choose his role. In Trinea, player will choose his race, class and then specialization in which he will gain experience.



- During game, player will have the ability to:
    - Increase the strength of his character by gaining experience

- Complete quests initiated by non-player characters
- Fighting with other players
- Competing with other players by gaining achievements for heroic actions
- Socialize with other players by chatting, trading, completing quests together

4.2. Examples of MMORPG games

Since the beginning of video games, developers had an idea of creating a game which would let the player to play the same game with the large amount of other people and – what is most important – to interact with each other. Unfortunately due to weak internet connections and small number of people who had internet connection at all, it was not possible to allow hundreds or thousands of people to play together in the same virtual world. Today, there are a lot of MMO game titles – most of which are unfortunately not well designed and therefore not very popular.

Ultima Online – This is probably the first MMO game which gained popularity. It was Richard Garriott - the creator of Ultima Online – who coined MMORPG term[6]. Ultima Online was released in 1997 and it still has a large amount of fans all around the world who still play it. Only six months after release, it reached 100.000 of paid subscriptions, which in those times was an astonishing number.

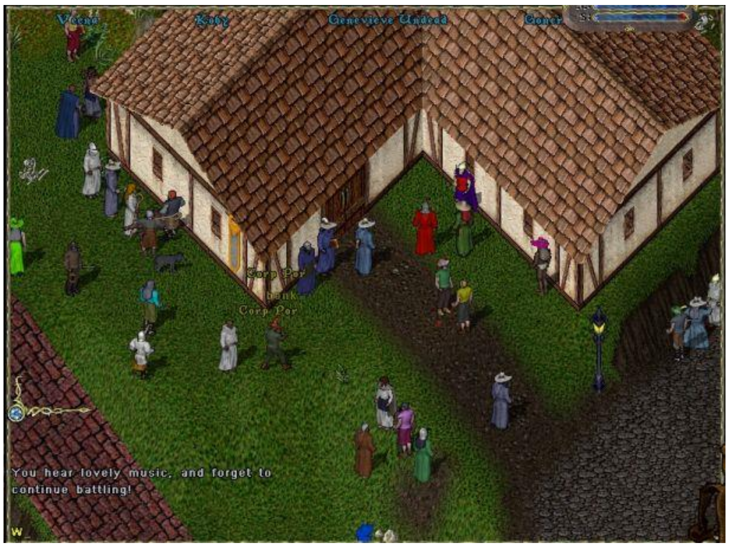


Fig. 1. Ultima Online game. (http://pc.gamespy.com/pc/ultima-online/1169826p1.html)

Lineage II – MMORPG game created by NCSoft company, launched in 2003 in South Korea. It reached over 1.000.000 paid accounts in 2007. The game was a prequel to Lineage, which was also a MMORPG game, but has never become as popular as Lineage 2. It has very standard gameplay if we compare it to other MMORPGs – player creates a character and by killing monsters he receives experience and better equipment. Lineage 2 was targeted to hardcore players, because it took a lot of time for the player to reach the highest level possible.

Fig. 2. Lineage 2 game. (http://lineage2.com)

World of Warcraft – Released in 2004, by Blizzard Entertainment game which is to this day the most popular MMORPG in the world. With over 10.000.000 paid subscriptions[7] it became the challenging competitor between other MMORPGs developers for a full decade.

It is considered as the game with the most extended gameplay due to unique battlegrounds, raids and dungeons systems, character professions and classes, enormous amount of quests, big world, and a plot which is the continuation of the one presented in Warcraft III game.

Fig. 3. World of Warcraft game. (http://us.battle.net/wow/en/)

4.3. Detailed description of the project

Our project consists of four stand-alone applications which we divided into two groups.

- Game group – consists of client and server applications.
- External tools – consists of map editor and game object editor. [8]

It is possible to say that the game group is isolated from the external tools group, because it does not require tools to work alone. The tools are just an additional feature providing developers easy method of extending gameplay by creating or modifying the map and game objects such as creatures, NPCs and items.

Project applications are designed this way so that they can cooperate with each other and exchange data. Figure 4 shows the graph of communication directions.

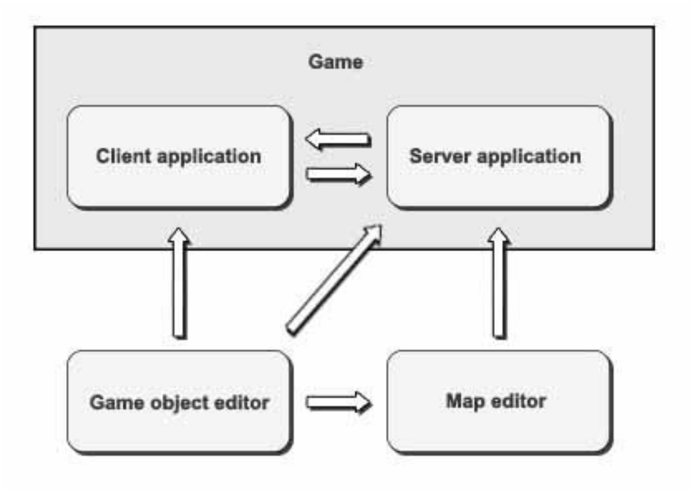


Fig. 4. Graph of communication directions between project applications

4.3.1 Game client application

Along with game server, game client is the most important application in a project. This is an application that the players will install on their computers and launch to play a game. It is written in C++ language and uses OpenGL graphics library to display a current game state. After the launch, the application displays the game main menu (Fig. 5) where the user can log in to the game with his login name and password. The game client application will not let the player to enter the game if it is not able to authenticate the player with the given credentials.

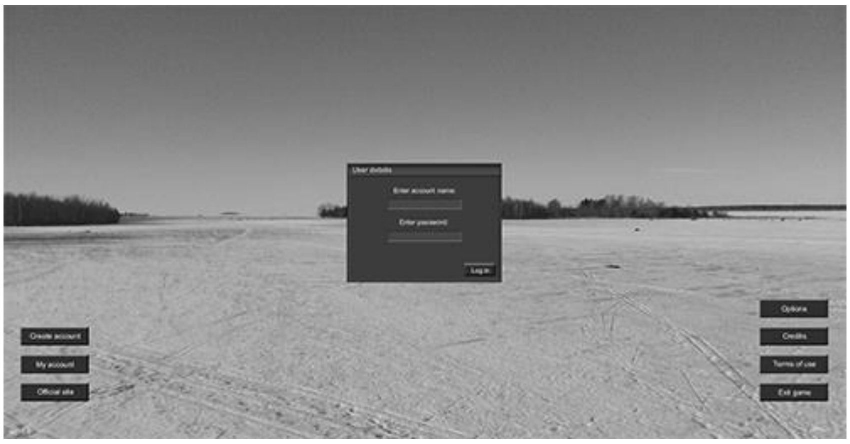


Fig. 5. Main menu in game client application.

4.3.2 Game server application

The game server is one of the most important applications of the project. Only the developers have access to it. It manages all game states and is the only trusted application. The server will only receive information about an action the client wants to perform, but it will send only the result of this action according to the game state.

4.3.3 Map editor [8]

The map editor is a tool for developers which allows them to easily create the map and save it as the XML file, which then will be used by the server application. It is a WYSIWYG editor, so while creating map the developer sees it as it will be represented later in the game.

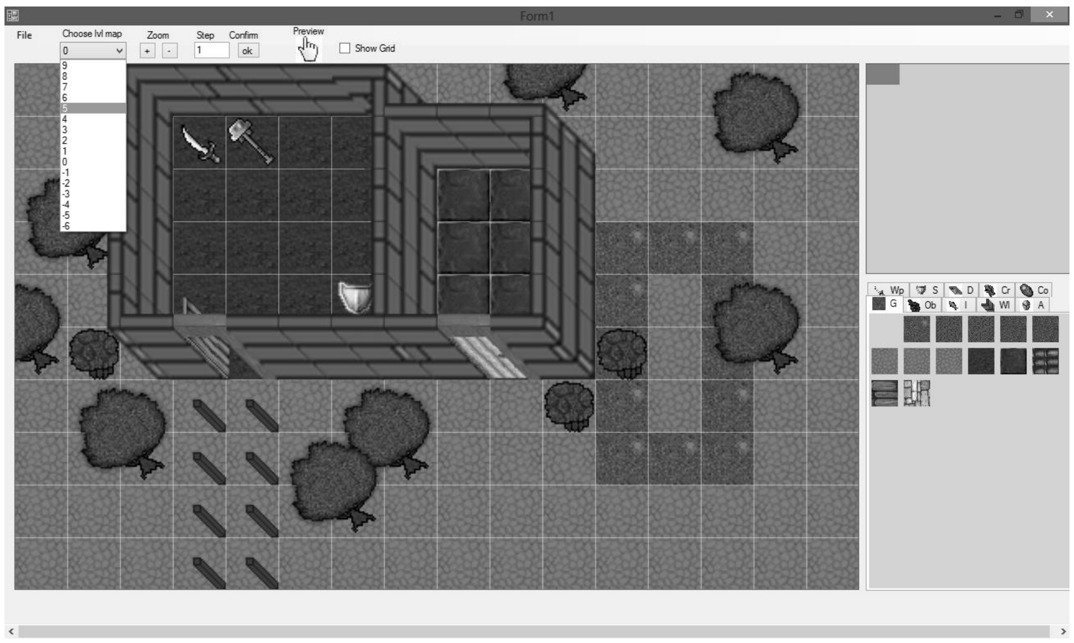
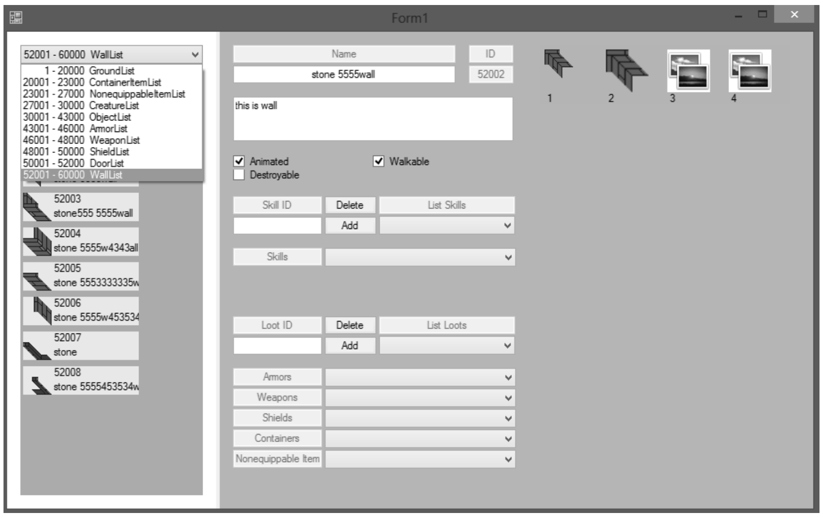


Fig. 6. Screenshot from Map Editor application.

4.3.4 Game objects editor [8]

The game objects editor is a tool which allows developers to create game objects such as creatures, NPCs and items by setting their graphics, descriptions, names and properties. It produces separate files for the game server application and the client application, because the client needs less information about objects than the server.



52001 - 60000 WallList

1 - 20000 GroundList
20001 - 23000 ContainerItemList
23001 - 27000 NonequippableItemList
27001 - 30000 CreatureList
30001 - 43000 ObjectList
43001 - 46000 ArmorList
46001 - 48000 WeaponList
48001 - 50000 ShieldList
50001 - 52000 DoorList
52001 - 60000 WallList

52003 stone555 5555wall
52004 stone 5555w4343all
52005 stone 5553333335w
52006 stone 5555w453534
52007 stone
52008 stone 5555453534w

Name: stone 5555wall
ID: 52002

this is wall

[✓] Animated
[✓] Walkable
[ ] Destroyable

Fig. 7. Screenshot from game objects editor application.

4.4. Scope and context of the project

The project should meet the following technical requirements:

- Application will be supported by PC, notebook and netbook computers with Windows XP/Vista/7/8 OS.
- At the beginning the game will not be commercial, but it will be designed to be easy to commercialize it in the future if there will be such an opportunity (premium accounts).
- Application will not be supported by multiple servers. If the number of players will be too high in the future, an extra server will have to be set up, but players from different servers will not be able to interact with each other.

- There will be four applications in a project:
  - Server
  - Client
  - Map editor [8]
  - Game objects editor [8]

- Server, client, map editor and database creator applications will be created and compiled in Visual Studio 2012 IDE.
- Client application will have advanced graphical user interface.
- Server application will be a console application.
- Information about the users and their progress will be stored in MySQL database.
- Quests and scripts will be created in Lua language.
- Game's website will present the information about the game.

4.5. Proposed solution description

Server application:

a. Application launch

b. Administrator runs „start" command through console interface

c. Server is now up and clients are able to connect to it

Client application:

a. Application launch

b. Application checks if system requirements are met

c. If yes, main menu pops up. If not, user receives an error message with error description.

d. In the main menu, the user writes his login and password

e. Application establishes connection between it and the server

f. If connection is established, user receives a list of characters stored on his account. If not, user receives error message with error description

g. User chooses one of characters and clicks "enter" button

h. User is now able to play with a chosen character

4.6. Functional requirements

The project should meet the following functional requirements:

- Server application should be a console application in order to decrease unnecessary usage of server computer's resources. It should be able to store advanced logs so that the administrator should be able to look into a state of the game every time he wishes to.
- When the server application is launched, clients should be able to connect to it right away after the administrator runs "start" command through the console interface.
- The target user should only have access to the client application which he should be able to download for free from the game's website.
- The client application needs to have a simple installation wizard.
- After installation, the user can launch a game. The game checks if the system requirements are met – if yes, the user can write his login and password and click the "enter" button. Now the game tries to establish a connection between the server and the client applications. If it succeeds, the user receives a list of characters that are created on his account from which he can choose a character he will be playing. After choosing a character, the user is already in a game and can play.
- Game's website should be full of information about the game for beginners and advanced users. It should also display information about all players' achievements. Each character should have his own subpage on which the other players should be able to check information about this character unless its owner will hide them.
- The client application should have a graphical user interface. It - along with the server application – should be written in C++ language. The map editor and the database creator should be written in C# language. For all applications, Visual Studio 2012 platform should be used.

4.7. Non-functional requirements

Feature	Restriction description
Performance	Time to launch a client application not longer than 5 seconds. Average number of frames per second not less than 40 FPS during gameplay. Average latency not higher than 100 ms during gameplay.
Resources usage	Required 50 MB of hard drive disk space.
Simplicity of usage	User should not spend more than 1 hour to know the basics of a gameplay.
Security	Server application needs to make a backup of all data every day. It needs to be simple to load data from backup. Users' passwords need to be encrypted.
Supportability	Client application should be portable between operating systems from Windows family – Windows XP/Vista/7/8.

5. Phase of analysis and designing

5.1. Project implementation model

Project is being developed according to Classical Waterfall model.

After determining the application requirements a stage of implementation and testing follows. During whole process of development, the documentation is written. The last step is to create user documentation – help file and articles on game's website – and supporting the target user by releasing patches and extensions which will improve gameplay quality (project maintenance).

5.2. UML Diagram

5.2.1. GUI Engine

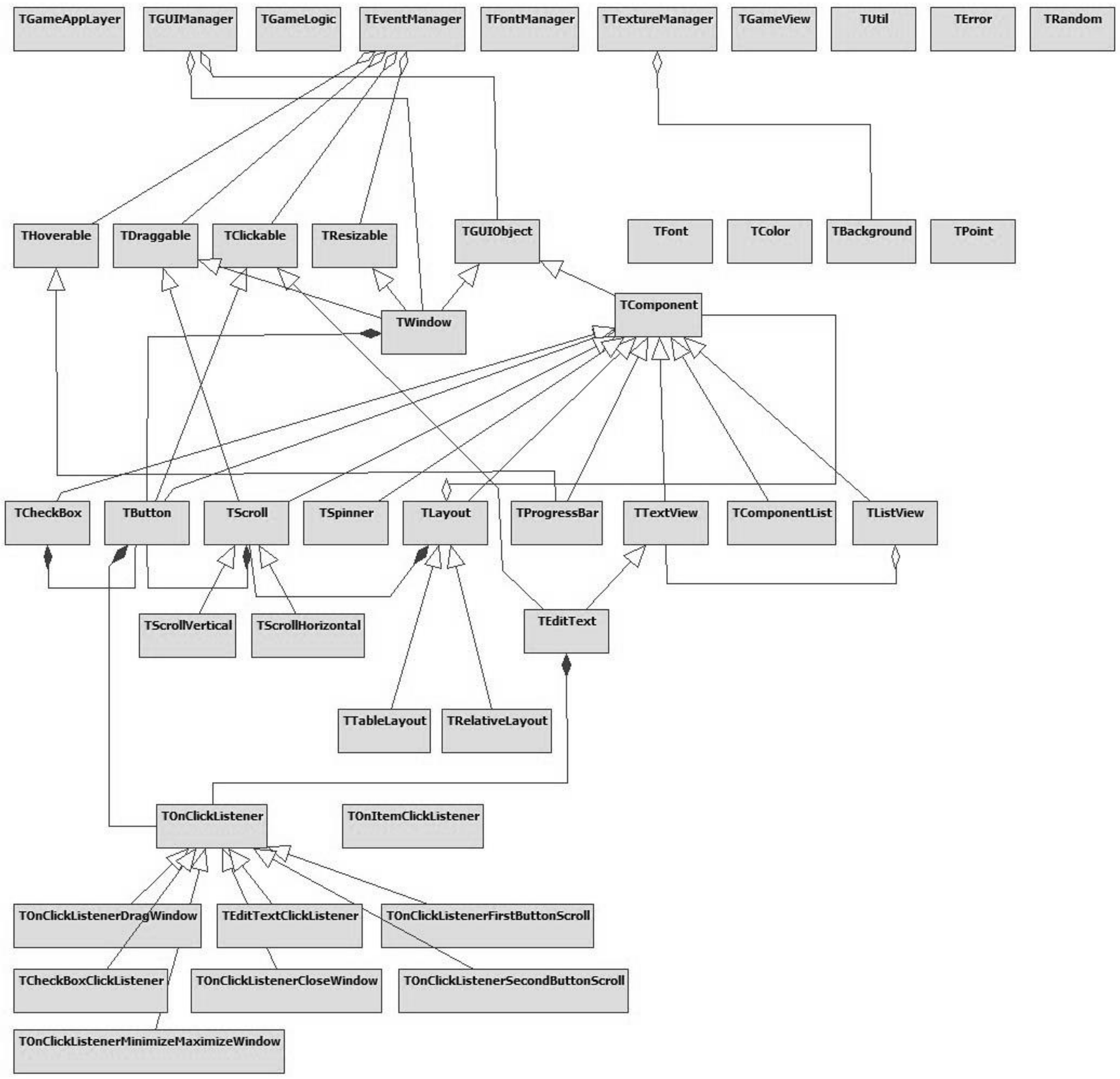


Fig. 8. Relations between classes in Game application

5.3. Client-server communication model

Both client and server applications can send messages to each other, but only the server is trusted application. The client's messages are called requests, because the client asks the server only for information, whereas the server's messages are called responses, because the server does not have to ask the client anything – it just responds to the client's requests.

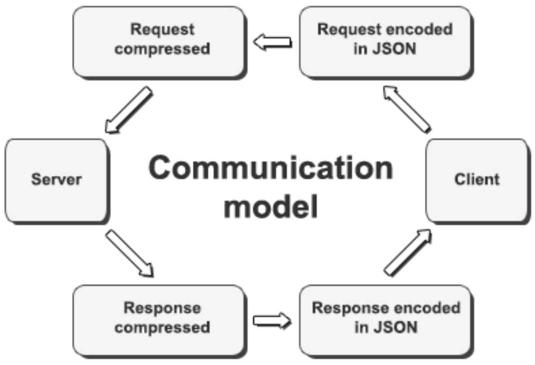


Fig. 9. Client-server communication model graph

We use JSON format to encode messages in order to send data between the client and the server. We chose it over the other solutions because for example XML is more human-readable, but it is more space consuming which – along with security - is most important issue when designing client-server communication for the online game. In these kinds of applications, the data transfer speed is crucial, because the server may support a large amount of players (clients) who cannot encounter any delays in order to keep them satisfied with the game experience. As stated earlier, security is also a main issue, as we have to keep personal information such as emails, passwords and account names safe. This can be achieved with proper encrypting with help of RakNet library.

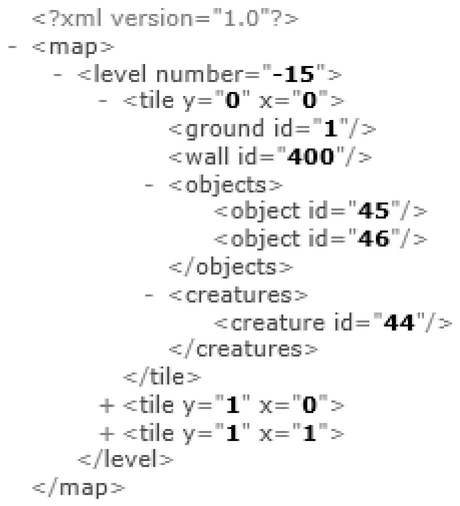
The message life cycle begins on either Client or Server side. Assuming the message is sent from Client to Server - which is constantly listening for requests – it is first encoded to JSON format. Along with actual data, a unique UID is added to the message. UID is a value which uniquely identifies specific computer and game client instances which assures that the message is sent from the game client and not from other sources. It is to prevent possible man-in-the-middle attacks. Then, the String containing JSON message is yet compressed by RakNet library in order to save as much space as possible to create very small packages. RakNet uses Huffman encoding to compress Strings. After that, the compressed message (request) is sent directly to the server. The server receives a request from the client and checks if the message is sent by authenticated game client. If not, the message is discarded immediately. If it's authenticated, the message is decoded and handled properly by the server's message processor.

The communication model is presented in fig. 9.

5.4. Data model used by Map editor [8] and Server application

The map editor and the server application use the same data model. It is relevant, because the map editor is meant to create map files ready to be read by the server immediately without unnecessary conversions. We decided to use the XML format, because it is human-readable and easy to edit manually. It is also very popular and the Windows Forms framework has built-in XML parser so we do not have to use external libraries for the map editor.

The map is 4-dimensional. It consists of several levels. Each level is represented as matrix of tiles. Each tile can stack specific amount of objects. The example of the map file is presented in fig. 10.



```
  <?xml version="1.0"?>
- <map>
  - <level number="-15">
    - <tile y="0" x="0">
        <ground id="1"/>
        <wall id="400"/>
      - <objects>
          <object id="45"/>
          <object id="46"/>
        </objects>
      - <creatures>
          <creature id="44"/>
        </creatures>
      </tile>
    + <tile y="1" x="0">
    + <tile y="1" x="1">
    </level>
  </map>
```

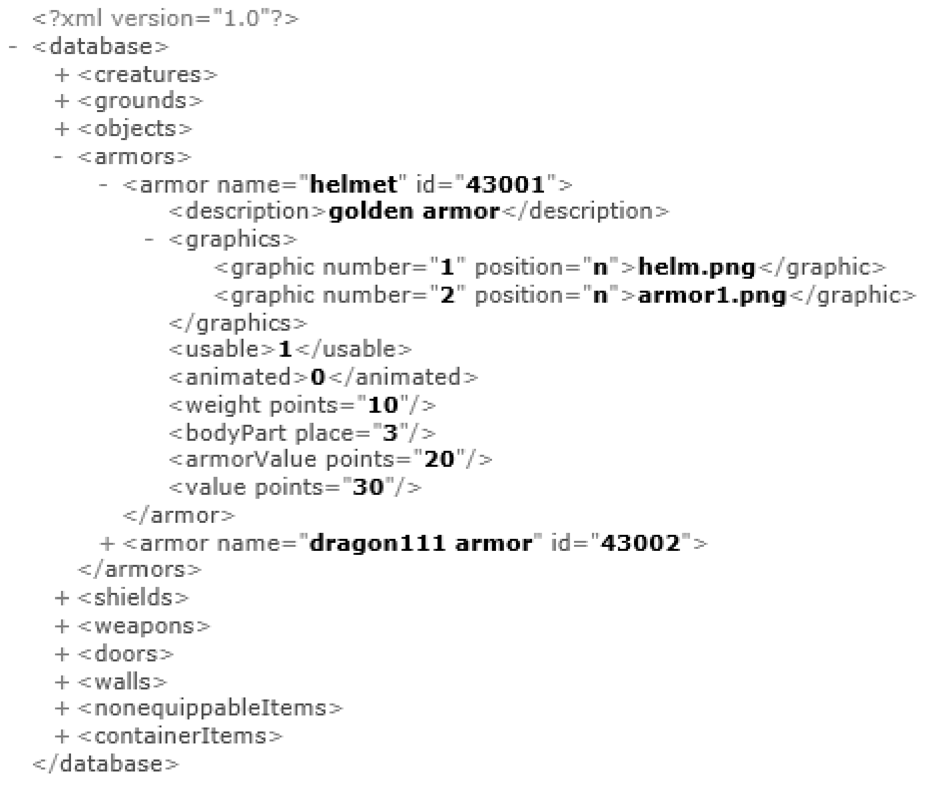
Fig. 10. XML map file example

In the example above, the map has only one level – its number is -15. It consists of three tiles which store specific objects, walls, grounds and creatures.

5.5. Data model used by Game objects editor [8] and Server application

In game objects editor's case, it is also crucial to create objects database file in a format which can be easily read by the server application. This tool also uses the XML format for the same reasons for which we decided to use it in the map editor – it is human-readable and easy to edit.

The model consists of objects which are categorized to main types such as creatures, grounds, objects and weapons. Each type has its own properties. Some of them are common for all, like "id", "name" and "description" and others are unique for specific types, like for example "walkable" for grounds or objects.



```
  <?xml version="1.0"?>
- <database>
    + <creatures>
    + <grounds>
    + <objects>
    - <armors>
        - <armor name="helmet" id="43001">
              <description>golden armor</description>
            - <graphics>
                  <graphic number="1" position="n">helm.png</graphic>
                  <graphic number="2" position="n">armor1.png</graphic>
              </graphics>
              <usable>1</usable>
              <animated>0</animated>
              <weight points="10"/>
              <bodyPart place="3"/>
              <armorValue points="20"/>
              <value points="30"/>
          </armor>
        + <armor name="dragon111 armor" id="43002">
      </armors>
    + <shields>
    + <weapons>
    + <doors>
    + <walls>
    + <nonequippableItems>
    + <containerItems>
  </database>
```

Fig. 11. XML game objects database file example

5.6. Use cases

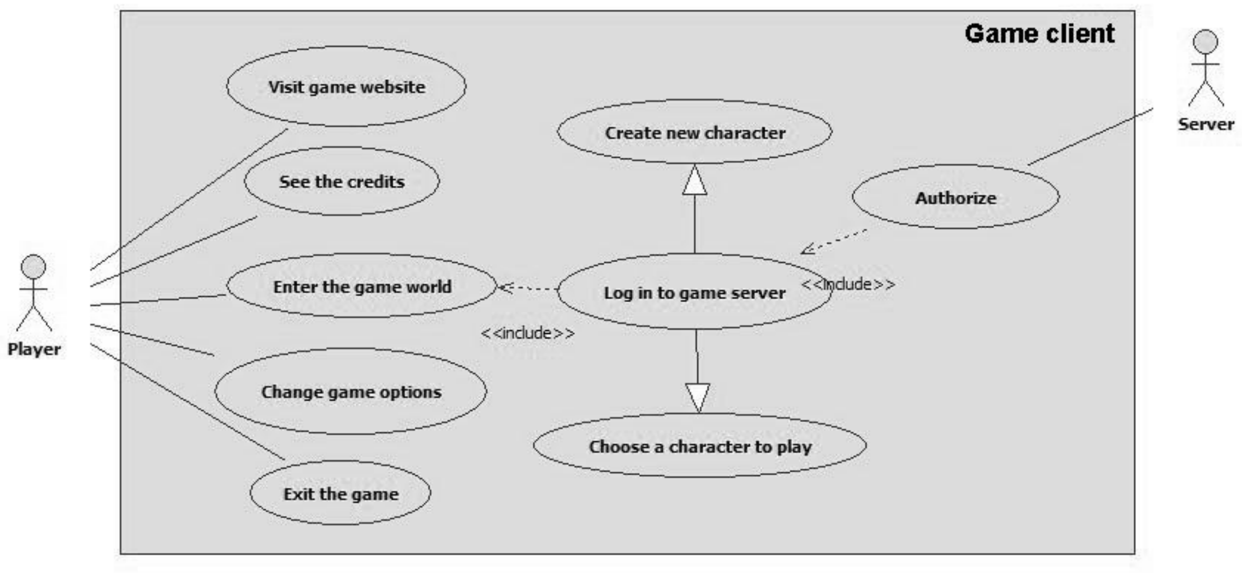


Fig. 12. Game client application use cases

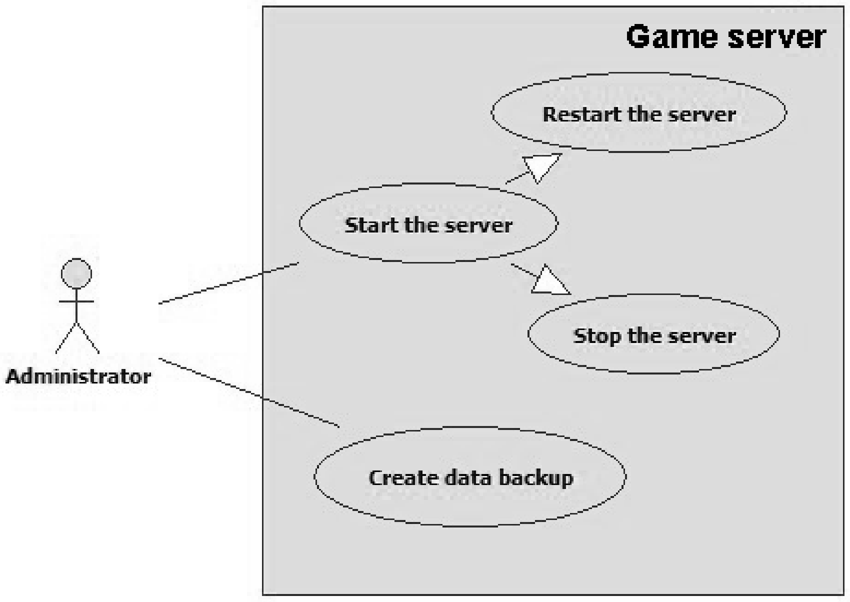


Fig. 13. Game server application use cases

6. Used libraries and technologies

6.1. Used libraries preview

In order to work efficiently on complex projects, it is crucial to decrease the amount of time one has to spend on working and focus only on parts that are unique for the project itself. We also have to remember that adding new functionalities to the program is not just writing code, but also testing it which is very long process. That is why it is very important to use already existing code if it serves functionality we require. This code is usually well packed and separated from everything else in our project. It is also already tested and we can be sure about its quality. It is called a library. Here is a list of libraries that we decided to use and a short description of each of them.

- **OpenGL** – One of two most popular graphics libraries. The second one is Microsoft's DirectX, which is currently leading graphics library on Windows platform – mainly because there were many problems in OpenGL's history which we will not cover in this thesis. Right now OpenGL retrieves its former popularity as its new versions are released by the Khronos Group. Main OpenGL's advantage over DirectX is that it is cross-platform. It means that if we write the project for Windows platform, we do not have to worry that if one day we decide to make a version for Linux we would have to rewrite the whole rendering system.
- **pugixml** – Light-weight library providing XML reading and writing classes for C++. The library is very easy to use and it is portable.
- **GLFW** – Portable framework for OpenGL library. Allows us to easily create OpenGL's rendering context without taking care of OS-specific requirements.
- **GLEW** – Next library providing additional functionalities for OpenGL. It is required because Windows OS for a long time have not updated its system OpenGL headers/libraries and we could not use the most modern version of OpenGL. GLEW provides features of newer versions.

- **GLM** – OpenGL Mathematics library for C++. It is very light-weight, because it consists of one header only. It provides such additional capabilities as matrix transformations, quaternions and procedural noise functions.
- **JsonCpp** – JSON data format parsing library for C++. It is very fast, light-weight and easy to use. It provides also a class which allows user to see JSON data presented in formatted form, therefore it is easier to verify it with eyes.
- **LodePNG –** Very easy to use, light-weight PNG format image encoder and decoder library for C++.
- **libpqxx** – Library for C++ integration with PostgreSQL database.
- **RakNet** – Powerful library used for networking. Unfortunately there is redundant functionalities, but ease of use and the security features compensate this. It provides TCP and reliable UDP transport, and supports multiple platforms, including Microsoft Windows and Linux, which is most important for us.
- **FTGL** – Light-weight font rendering library based on OpenGL which requires only one header to be included. It can draw three popular types of fonts – raster, vector and textured. User can easily manipulate the look of the rendered text, by changing the font color, texture, size, spacing.
- **glog –** Very light logging library from Google which allows to easily create logs with different levels of severity. It can also distinguish whether application runs in debug or release mode and log different messages depending on this. The library was chosen because it has very clear documentation and is much simpler and more efficient than the other free logging libraries.

6.2. Used technologies preview

Before starting any serious project, developers have to decide which technologies to use in order to provide as good solution as possible. It is worth to note that most of the time there is not something like the best solution – each technology has its advantages and disadvantages but every language or tool is specialized to help with specific tasks, for example Java language is very simple and it's easy to write proper code efficiently due to its Garbage Collector which takes care of automatic memory deallocation. Unfortunately this useful feature comes with a price of lower efficiency, that is why most demanding video games are written in C++ language instead of Java.

- **C++** – We chose C++ language to write client and server applications. Its efficiency along with portability were the reasons for us to choose it. It was very important to use C++, especially for the server application because we needed it to be very fast to be able to handle a large number of players at once.
- **XML** – Language which is used to encode data in a format that is easily readable by both human and machine. XML is used by us to store data about the map - created by the map editor and loaded by the server application – and the game objects database. Its advantage is that it is very easy to verify its correctness and to modify it even without a help of a machine.
- **JSON** – As well as XML it is used to encode data in a format readable by human and machine. The reason for choosing it over XML was that JSON is faster. We are using it as object serializer in the client-server data exchange.
- **Microsoft Visual Studio 2012** – IDE for Windows platform in which we write C++ and C# code. It also provides the XML editor. It is professional studio used by large companies and it is considered as the only right choice for C++ language on Windows platform.
- **Subversion** – Software versioning and revision control system. It is very useful tool which helps preserving all the data changes done on project files. We use it to maintain versions of source files and documentation.

- **PostgreSQL** – Very popular database system similar to MySQL. It is free for commercial use and because it was created some time after MySQL, most of known bugs were already fixed in the process of designing which makes it more stable.

7. Implementation phase – sample subsystems overview

7.1. Game client application

7.1.1. Graphical User Interface

GUI engine is very important part of the game client application as it is responsible for creating all the controls that the user can interact with, for example all kind of buttons and lists.

The main goal in designing and implementing the GUI engine was to create the TGUIManager class which was responsible – as the name indicates – for managing all GUI controls. It has an internal list of all GUI controls, therefore whenever it is needed, it can remove existing (as well as deallocate memory associated with it) component or add another one.

In order to provide as good functionality as possible, all GUI controls derive from one class called TGUIObject:

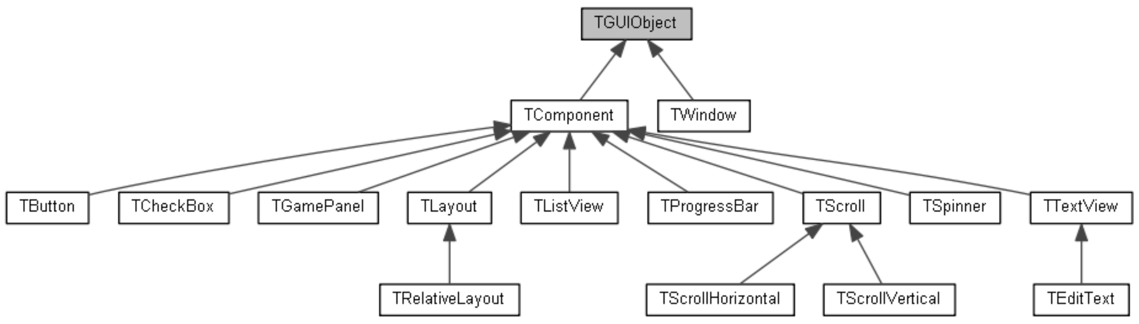


Fig. 14. Classes deriving from TGUIObject

TComponent and TWindow derive directly from TGUIObject. The difference between those two classes is very slight, but it was necessary to separate them.

TLayout is an object which allows the user to store other objects in it. It is mostly used in class instances of TWindow.

It is very easy to draw each control – it can be done by calling draw() method. Developers do not have to remember to draw each control, because it is automatically done by the TGUIManager, which draws all the objects every game cycle:

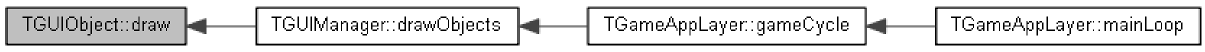


Fig. 15. Caller graph for TGUIObject::draw() method

Implementation of TGUIManager::drawObjects() method is represented in code snippet below:

```
void TGUIManager::drawObjects() {
    // Draw all objects except windows
    unsigned int guiObjectListSize = m_guiObjectList.size();
    for(unsigned int i = 0; i < guiObjectListSize; i++) {
        TGUIObject* obj = m_guiObjectList[i];
        if(obj != nullptr) {
            if((obj->getParentId() < 0) && (dynamic_cast<TWindow*>(obj) == 
nullptr)) {
                // GUIManager should only draw objects which don't have 
parents specified
                // And those that aren't instances of TWindow class
                // Rest objects will be drawn by their parents
                // But only if that parent is able to draw children (i.e. 
TWindow or TLayout)
                obj->draw();
            }
        }
    }

    // Now draw windows
    unsigned int windowListSize = m_windowList.size();
    for(unsigned int i = 0; i < windowListSize; i++) {
        TWindow* w = m_windowList[i];
        w->draw();
    }
}
```

7.1.2. Events system

Systems such as GUI engine also require implementing kind of event system in order to make user to be able to interact with it. This is the main purpose of the TEventDispatcher class – to make some responsive system to all the events that the and the itself is sending.

We were thinking about types of actions the user can perform on GUI controls and we have chosen clicking, resizing, dragging and mouse hovering.

In order to achieve flexibility, we created separate classes with pure virtual methods for each action and whenever any TGUIObject needs to react on this specific action, it just has to derive from this class and implement the required methods. It is achieved thanks to multi-inheritance.

The TEventManager is responsible for calling methods associated with specific action for those TGUIObjects that should react on them.

As an example we can describe the TButton class. It is usual that a button can be clicked, so it just has to derive from the TClickable class and implement wasClicked(const TPoint& clickPos) method. We also need to specify the TOnClickListener object for it.

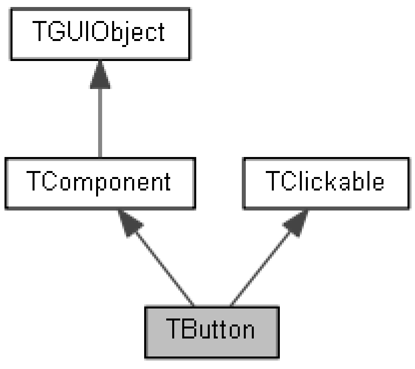


Fig. 16. TButton inheritance graph

Example of implemented wasClicked() method in the TButton class:

```
bool TButton::wasClicked(const TPoint& clickPos) {
    // First check if click position is inside parent
    TGUIObject* parent = getParent();
    if(parent != nullptr) {
        if(!parent->pointInObject(clickPos)) {
            return false;
        }
    }

    return pointInObject(clickPos);
}
```

It is worth to note that the reason behind adding TOnClickListener class instead of pure virtual method was the fact that if it was a virtual method, we could have only one task for all TButton instances and we need separate tasks for each instance. That is why we added a short TOnClickListener class. Its body is as follows:

```
/*!
 *  @brief     Class to add reaction on mouse click event.
 *  @details   Each reaction should have separate class that inherits from
<em>TOnClickListener</em>.
 *  @author    Justyna Duczmanska
 *  @author    Piotr Chojnacki
 *  @version   1.1
 *  @date      2012-10-26
 */
class TOnClickListener {
public:
    /*!
        Everything that should be performed when component is clicked by a
mouse.
    */
    virtual void onClick(const TPoint& clickPos) = 0;

    TOnClickListener();
    virtual ~TOnClickListener();
};
```

The above code snippet is very basic body of this class. Whenever we create another class which derives from it, we have to implement onClick() method. For example when we needed action to close TWindow (close button is in almost all windows), we could easily create it with the following code:

```
/**************** Close window button ********************/
class TOnClickListenerCloseWindow : public TOnClickListener {
private:
    TWindow* m_window;
public:
    void onClick(const TPoint& clickPos);
    TOnClickListenerCloseWindow(TWindow* window) : m_window(window) {}
};

void TOnClickListenerCloseWindow::onClick(const TPoint&) {
    m_window->close();
}
```

All the other actions work in the similar way as TClickable. Here are the call graphs for specific actions in the TEventManager class:

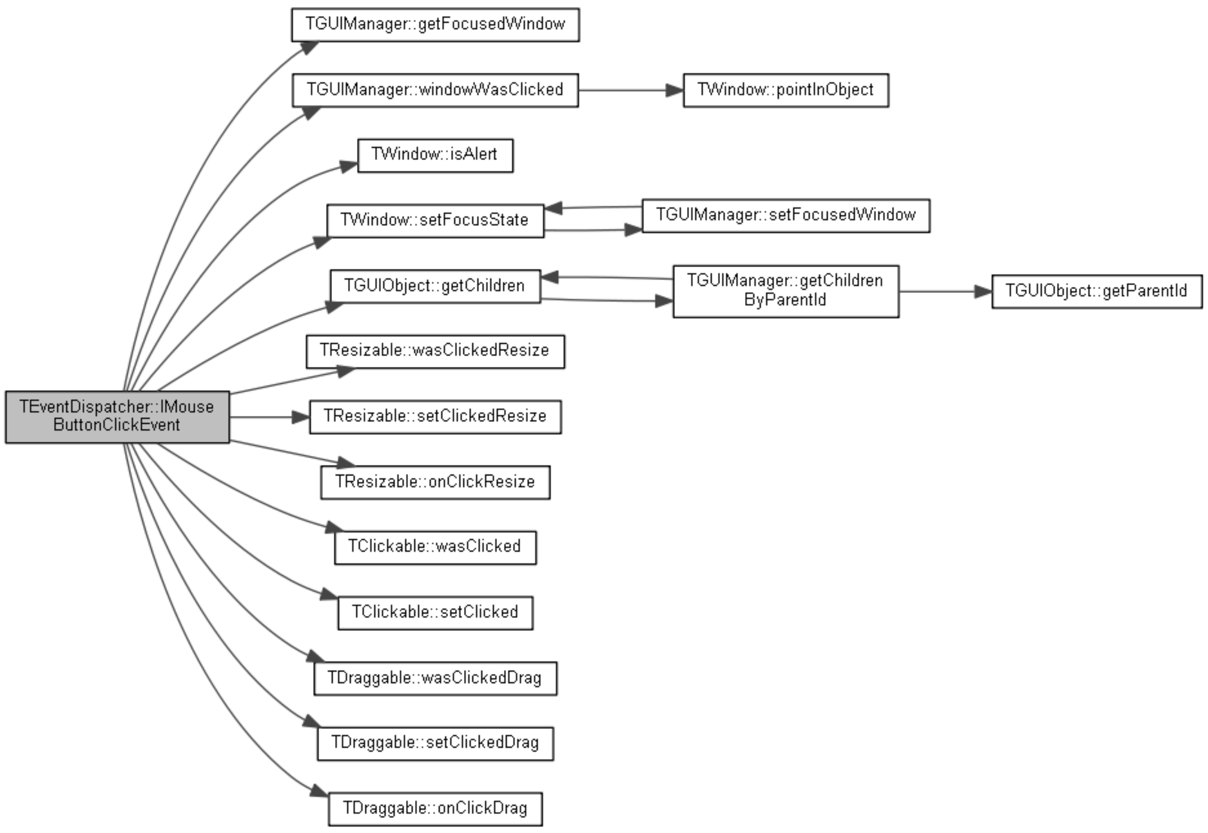

TEventDispatcher::lMouse ButtonClickEvent
TGUIManager::getFocusedWindow
TGUIManager::windowWasClicked
TWindow::pointInObject
TWindow::isAlert
TWindow::setFocusState
TGUIManager::setFocusedWindow
TGUIObject::getChildren
TGUIManager::getChildren ByParentId
TGUIObject::getParentId
TResizable::wasClickedResize
TResizable::setClickedResize
TResizable::onClickResize
TClickable::wasClicked
TClickable::setClicked
TDraggable::wasClickedDrag
TDraggable::setClickedDrag
TDraggable::onClickDrag

Fig. 17. Left mouse button clicked event call graph

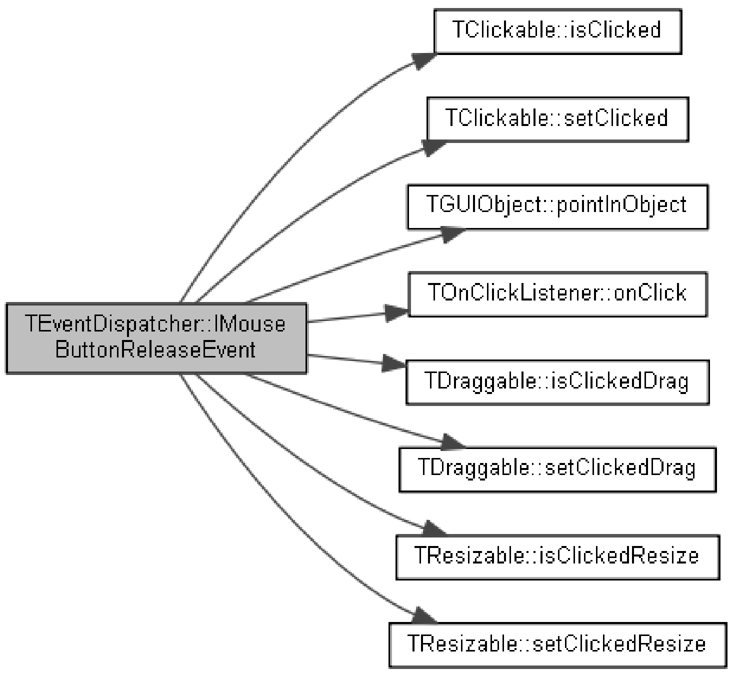

TEventDispatcher::lMouse ButtonReleaseEvent
TClickable::isClicked
TClickable::setClicked
TGUIObject::pointInObject
TOnClickListener::onClick
TDraggable::isClickedDrag
TDraggable::setClickedDrag
TResizable::isClickedResize
TResizable::setClickedResize

Fig. 18. Left mouse button released event call graph

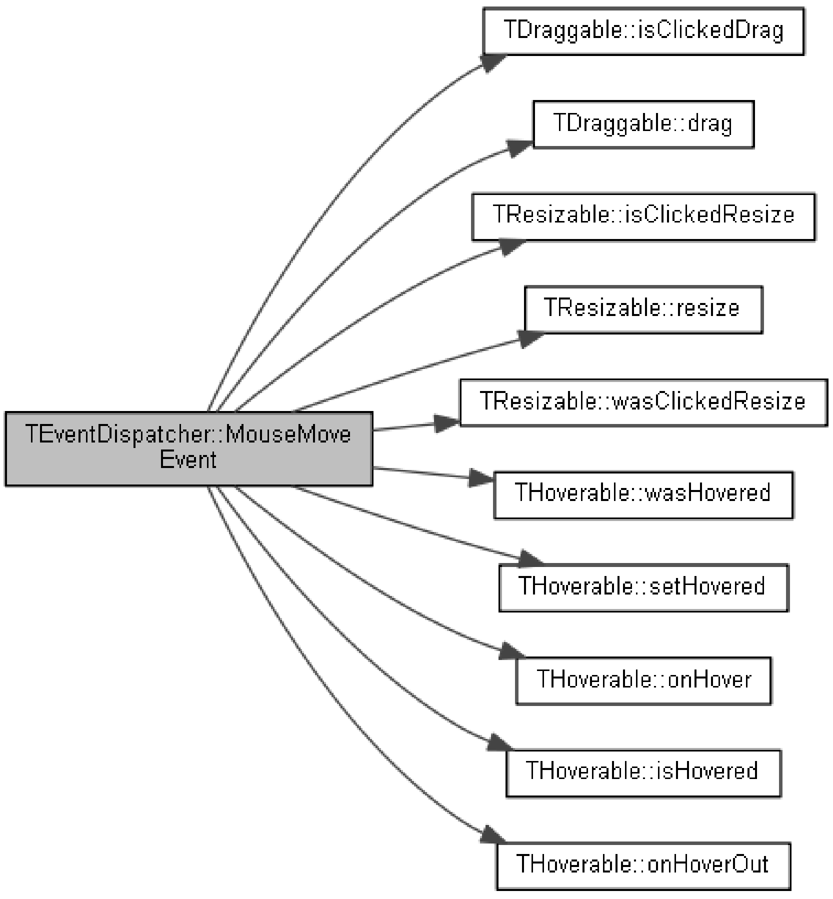


Fig. 19. Mouse move event call graph

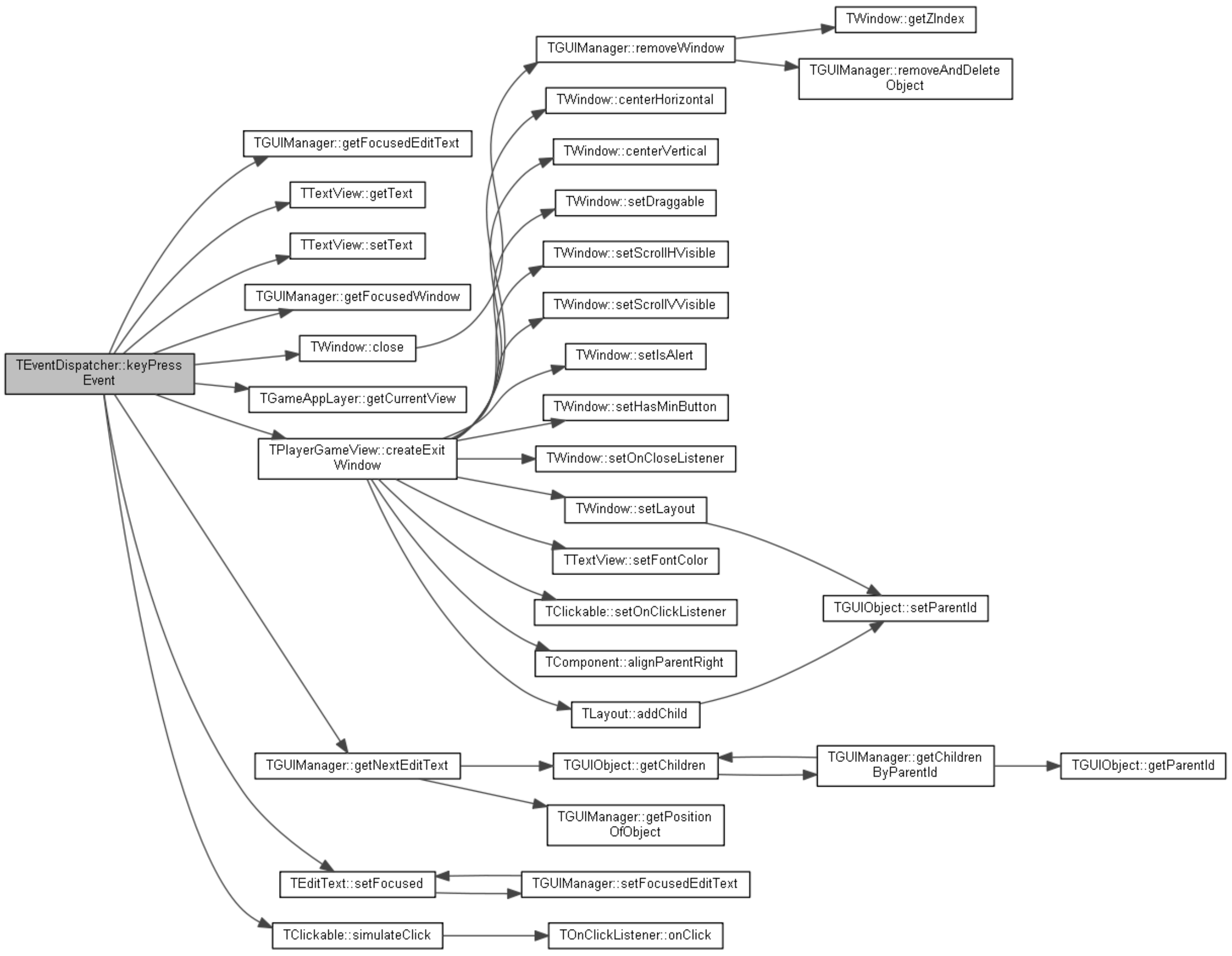

TEventDispatcher::keyPress Event
TGUIManager::getFocusedEditText
TTextView::getText
TTextView::setText
TGUIManager::getFocusedWindow
TWindow::close
TGameAppLayer::getCurrentView
TPlayerGameView::createExit Window
TGUIManager::removeWindow
TWindow::getZIndex
TGUIManager::removeAndDelete Object
TWindow::centerHorizontal
TWindow::centerVertical
TWindow::setDraggable
TWindow::setScrollHVisible
TWindow::setScrollVVisible
TWindow::setIsAlert
TWindow::setHasMinButton
TWindow::setOnCloseListener
TWindow::setLayout
TTextView::setFontColor
TClickable::setOnClickListener
TComponent::alignParentRight
TLayout::addChild
TGUIObject::setParentId
TGUIManager::getNextEditText
TGUIObject::getChildren
TGUIManager::getChildren ByParentId
TGUIObject::getParentId
TGUIManager::getPosition OfObject
TEditText::setFocused
TGUIManager::setFocusedEditText
TClickable::simulateClick
TOnClickListener::onClick

Fig. 20. Key on keyboard pressed event call graph

7.1.3. Game events processor

In order to provide an easy way of responding and delegating events, we required a system that would process the game messages such as “player logged in”, or “player entered the world”. The class that is responsible for this task is called TGameEventProcessor.

From every place in the code we can call TGameEventProcessor to process an event with specific identifier:

```
void TGameEventProcessor::processEvents(GAME_EVENTS eventType,
game_message_response_ptr msg) {
    switch(eventType) {
    case GAME_EVENTS::LOGGED_IN:

eventLoggedIn(std::dynamic_pointer_cast<TGameMessageLogIn_Response>(msg));
        break;
    case GAME_EVENTS::FAILED_LOG_IN:
        eventFailedLogIn();
        break;
    case GAME_EVENTS::ENTERED_WORLD:

eventEnteredWorld(std::dynamic_pointer_cast<TGameMessageEnterWorld_Response
>(msg));
    break;
    case GAME_EVENTS::DISCONNECTED_FROM_SERVER:
        eventDisconnectedFromServer();
        break;
    case GAME_EVENTS::SERVER_FULL:
        eventServerFull();
        break;
    case GAME_EVENTS::SERVER_CONNECTION_ATTEMPT_FAILED:
        eventServerConnectionAttemptFailed();
        break;
    }
}
```

If we come up with an idea for another game event, we have to follow only three steps. First of all, we have to add a new identifier to the GAME_EVENTS enum:

```
enum class GAME_EVENTS {
    LOGGED_IN,
    FAILED_LOG_IN,
    ENTERED_WORLD,
    DISCONNECTED_FROM_SERVER,
    SERVER_FULL,
    SERVER_CONNECTION_ATTEMPT_FAILED,
    PLAYER_MOVE
};
```

Then we have to create a method specific for this event and then only call it inside the processEvents() method as shown previously.

Let’s for example analyze the call graph of the TGameEventProcessor::eventEnteredWorld method:

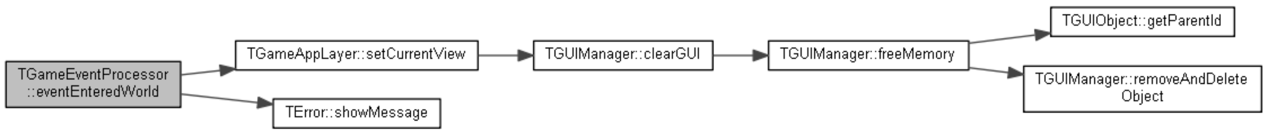

TGameEventProcessor ::eventEnteredWorld
TGameAppLayer::setCurrentView
TError::showMessage
TGUIManager::clearGUI
TGUIManager::freeMemory
TGUIObject::getParentId
TGUIManager::removeAndDelete Object

Fig. 21. Caller graph for TGameEventProcessor::eventEnteredWorld() method

As shown in fig. 24, the current view is set as the world view, and GUI manager clears all the GUI objects from the previous view (main menu), which means that it frees up memory allocated by them and performs all required operations associated with this.

7.1.4. Game Application Layer

Game application layer is responsible for initialization, main loop, networking and processes that are strongly associated with the operating system. Therefore in our project it consists of frames per second counter, timer, message processor and server connection classes and the main game loop. We can call it the core of the whole application. In our code it is represented as TGameAppLayer.

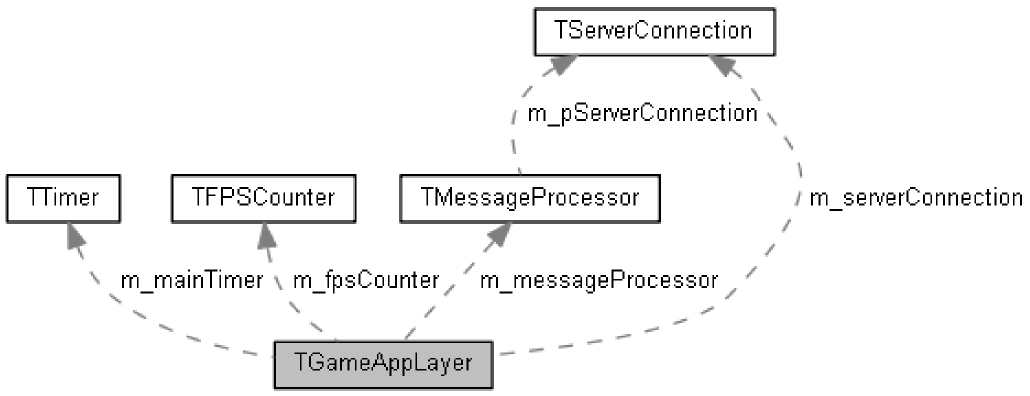


Fig. 22. Collaboration diagram for TGameAppLayer

The main loop is very important part of the TGameAppLayer as well as of the whole application, because it is the place where all calculations and drawings are processed. There is a loop, which calls the gameCycle() method continuously. The gameCycle() is a method which performs all actions that need to be called an every frame.

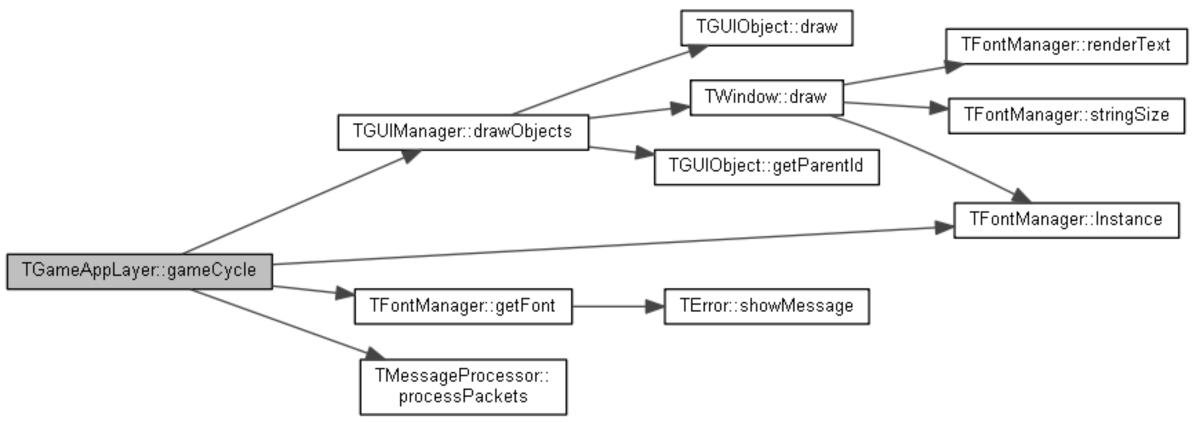


Fig. 23. Call graph for TGameAppLayer::gameCycle() method

7.2. Game server application

7.2.1. Server core

As every larger system, the server application needed a class that could be called a server core which is responsible for managing all subsystems. Its name in our code is very intuitive – it's just TServerCore. It owns such subsystems as account manager, database connection, game state data, game event processor and message processor.

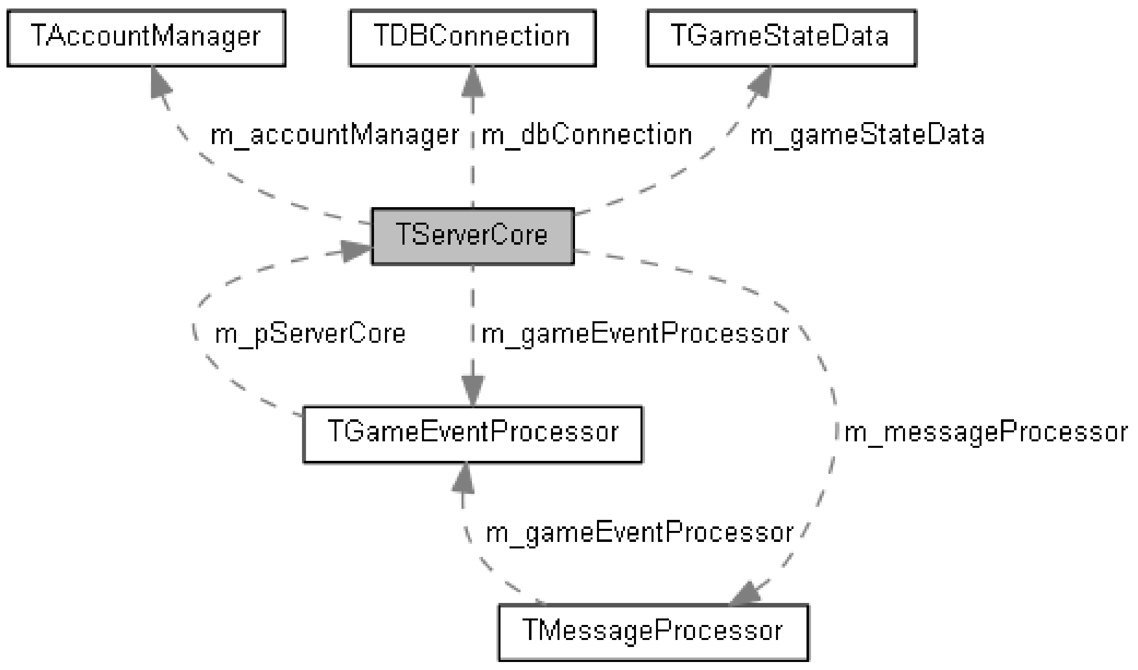


Fig. 24. Collaboration graph for TServerCore

When server is launched it has to initialize all subsystems. If any of those systems' initializing fails, server cannot be started.

```
bool TServerCore::initialize() {

    m_server = RakPeerInterface::GetInstance();

    SocketDescriptor sd(CONNECTION_PORT,0);
    RakNet::StartupResult result = m_server->Startup(MAX_CLIENTS, &sd, 1);
    if(result != RakNet::StartupResult::RAKNET_STARTED) {
        return false;
    }
    m_server->SetMaximumIncomingConnections(MAX_CLIENTS);
    m_messageProcessor.initialize(m_server, &m_gameEventProcessor);

    /**
    * Parsing XML with game objects database and storing data
    * in TGameObjectManager
    */

TGameObjectManager::Instance().setPrototypes(TXMLParser::parseXML("serverGa
meObjects.xml"));
    std::cout << std::endl << "Loaded XML with game objects database.
Result: " << std::endl;
    TGameObjectManager::Instance().printPrototypesState();

    m_dbConnection.connect();

    try {
        m_dbConnection.loadAccounts(m_accountManager);
    } catch(std::exception& e) {
        return false;
    }

    if(!m_gameStateData.initialize())
        return false;

    return true;
}
```

7.2.2. Message processor

We wanted to make a structure of both the client and the server applications to be as similar as possible in order to avoid writing the same solutions more than once. That is why some of the subsystems have very similar structure. For example the TMessageProcessor. It is used to process messages sent by the client and passes those messages to the TGameEventProcessor which properly responds to those messages.

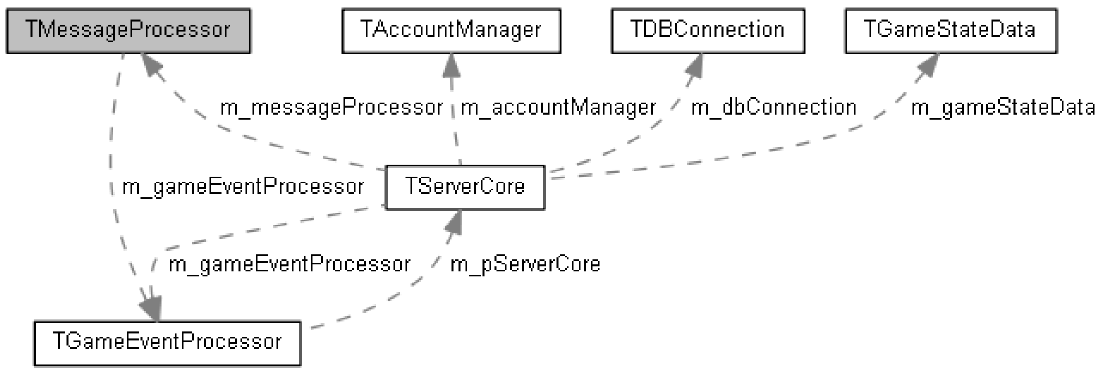


Fig. 25. Collaboration graph for TMessageProcessor

7.2.3. Game event processor

As previously explained, the TGameEventProcessor is responsible for responding to messages sent by the client. The example scenario could be:

a. Client sends message encoded in JSON that the user wants to log in with specific credentials

b. The server's TMessageProcessor receives message sent by the client, decodes it and passes it to the TGameEventProcessor

c. The TGameEventProcessor receives a message passed by the TMessageProcessor and checks if credentials are correct and sends encoded in JSON message to the client with the result.

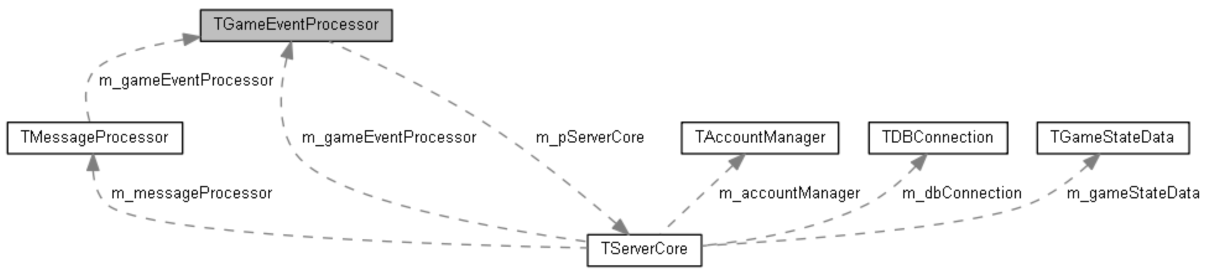


Fig. 26. Collaboration graph for TGameEventProcessor

8. Testing phase

8.1. Testing types

8.1.1. Data and database testing

We should ensure that data is properly saved and loaded from the database and sent between the server and the client applications. Data should be properly encoded and decoded from JSON format in order to provide successful communication flow between applications.

All fatal errors affecting data will be found as exceptions while using applications. Other, minor errors which are not in fact errors, but rather bugs (e.g. wrong values in character statistics or wall which allows a character to pass through it) will have to be caught during intensive applications usage, most likely in beta version release.

8.1.2. Function testing

This testing type is related to testing specific use cases which were determined earlier. It is easier to divide the testing process into those functions (use cases), in order to be able to focus on a small part of the project at once.

We should start from testing each case separately using valid and invalid data. After positive verification, we should perform testing on functions sequences until we finally test the whole application running.

8.1.3. User interface testing

Another testing type considers the user interface. The user interface is one of the most important parts in our applications (especially in the game client) as it is directly exposed to the user.

GUI should provide intuitive, simple menu, with a short learning curve. All components should be functional, uniform and should work according to their destination.

Most important parts of the GUI (functionality and the errors occurrence) will be tested by us and after we eliminate most bugs, it will be presented to other testers for further testing.

8.1.4. Performance profiling

High performance is crucial for a MMO game as it has to support multiple players at once and provide good user experience (e.g. no latency problems due to server traffic). The game client application should run efficiently with most – even older – hardware.

In order to examine a performance of our applications, we will use profiler tool provided by Microsoft Visual Studio and we will create multiple fake instances of game clients.

8.2. Testing tools

Purpose	Tool
Test management	Microsoft Word Microsoft Excel
Test design	Microsoft Excel
Detecting memory leaks	Visual Leak Detector
Defect tracking	Trac
Performance testing	Microsoft Visual Studio Profiler

9. Summary

Computer games are very specific projects. Even a small game which seems to be very simple and primitive from the user's point of view, in fact is a very complex project which requires knowledge from different areas – physics and computer graphics but it also needs creativity and analytical skills from developers.

Before starting this project, we did not have much experience with computer games development and client-server architecture. Even though we decided to face this challenge which in the end appeared to give us much – not only priceless knowledge, which surely will be used in our future work even if it is not related to game development itself - but it also gave us joy and feeling that if we really want something, we can achieve this.

At the beginning it was very hard to start, because we did not really have experience in working in a team either. We had to design and think over numerous solutions where everyone had his own ideas which we considered brilliant, yet we had to choose only one. It was difficult to separate tasks between team members and to cooperate, as each of us had to prepare his own piece of puzzle and make it fit the project as a whole. Over time we learned to think as a team, not as individuals. We started meeting at one place at the university regularly. Working outside home was very helpful for us because thanks to that, we were not distracted all the time with other things. Sometimes we could not achieve our daily goal because of problems we met or courses we had during this time. But it only mobilized us to work harder the next day.

So far the project is not a finished game, but a big part of game's engine is finished, including the GUI engine, which is very difficult to create from scratch. There are a few minor bugs which we encountered, but it is usable and probably can be distinguished from the project itself if needed in the future. It means that it was well designed and it is flexible – it allows a developer to easily add new components and to manage them. The game server and the client can connect to each other and send basic messages. The user can already log in with his own credentials stored in the database and enter the world with a chosen character. The world data is properly sent to the client by a server.

10. References

[1] *http://en.wikipedia.org/wiki/Game*

[2] *Radoff, Jon (May 2010), "History of Social Games," http://radoff.com/blog/2010/05/24/history-social-games/*

[3] *http://blog.usoapps.com/2012/03/why-do-people-play-games/*

[4] *http://www.theesa.com/facts/index.asp*

[5] *http://www.teachthought.com/featured/why-people-play-video-games/*

[6] *https://en.wikipedia.org/wiki/History_of_massively_multiplayer_online_games*

[7] *http://www.bbc.co.uk/news/technology-15672416*

[8] *Grzelaczyk, Jacek (2015), "The design and implementation of developers' tools for multiplayer, web-based, 2D game made in client-server architecture" (in print). University of Lodz, Lodz, Poland*