Contents

1 Introduction

1.1 The subject and issues raised.

The subject of the thesis is to create a server and client prototype for the MMORPG game (*Massively Multiplayer Online Role-Playing Game*). Games of this type are distinguished primarily by the fact that they are played through the server, by a very large number of players simultaneously, reaching tens or hundreds of thousands. Creating a server that can handle this number of players is a big challenge, even if we have very efficient equipment. The purpose of the work is to create such a server, as well as a client that can connect to it, which is in fact a prototype of a full-fledged game. However, the assumption of this work is the implementation of an MMO game in which the number of players does not exceed several hundred. The 4 and 5 chapters describe how to implement server and client prototypes, the 6 chapter provides details about how the project works.

1.2 Description of the game.

The created game will be a two-dimensional platformer with an open world. The player will be able to explore locations, fight monsters, collect equipment and experience points from them that will increase the player's level of experience. These are elements that can be found in almost every RPG.

The player will have the opportunity to acquire and set up items that will increase his statistics. In addition, souls can be won from each opponent (not in the usual sense of the word, so an opponent may have more than one "soul"). Unlike standard RPG, where new skills are gained with experience levels, in my game skills can only be acquired from opponents. They will be dropped randomly, similar to items.

The world will not have artificial barriers, which means that to get somewhere you have to do a task or have the right level. Instead, the locations available to the player will be limited by certain obstacles that must be overcome with the appropriate item or special ability. Games with this structure are called *metroidvania*. The combination of these two genres (i.e. MMORPG and *metroidvania*) is quite innovative, because there is virtually no game in which it is done. The sensibility and practicality of such a combination is one of the things that this master's thesis is to check.

As for the interaction between players, which is even required in online games, players will have the opportunity to jointly explore the world and fight its dangers. As a standard, a text chat will be available, through which each player will be able to easily communicate with other people in the same room, or on the server in general.

2 Overview of similar games.

2.1 MMORPG

The MMORPG genre we know today has been around since the 1990s. The first titles were games such as Meridian 59 (The 3DO Company, 1996), The Realm Online (Sierra Entertainment, 1996), and the popular Ultima Online (Electronic Arts, 1997), which works until today. However, the first MMORPG with a graphical interface, actually older than the above, was Neverwinter Nights (Strategic Simulations, 1991).

Today's best-known MMORPG is World of Warcraft (Blizzard Entertainment, 2004). It has been operating since 2004 and until then it has had 7 additions and has attracted over 10 million players. He is considered by many to be an exemplary representative of the [1] genre. There is some truth in this, because it has all the elements characteristic of MMORPG - rich interactions with other players, complex character development, a huge world full of various tasks and demanding challenges designed for the best players that need to be taken in groups. Each player creates his character (which can have several), sometimes called an avatar, which he develops, discovering new lands and creating task lines (so-called *quest chain*) associated with them. The player can also belong to a guild, which is jointly developed by many people. Interactions between characters of different players are possible, such as trade, fight or creating so-called *party*, i.e. teams of several people playing together. One of the more characteristic things are the so-called instances - separated areas of the world (dungeons), which can have many copies. Normally, players are all on the same map, while instances are intended for smaller groups, and each group passes its copy of the instance. The instances are designed so that they can be passed only in a group. Among them are the so-called rallies that require groups of several dozen people.

Of the other MMORPG representatives worth mentioning is Maple Story (Nexon, 2003) [2]. While the vast majority of MMORPGs are three-dimensional [3], Maple Story is a two-dimensional side-scroller - platformer. The game has been running continuously for 16 years and is very popular.

2.2 Metroidvanias

The name *metroidvania* comes from two series of games that this genre initiated: Metroid and Castlevania [4]. The distinguishing feature of the genre is that we have a huge, open world to explore, but obstacles stand in our way that we can overcome by acquiring new skills. As a result, openness is in some ways an illusion, because usually the world is planned so that we get the needed improvements in a certain order. Some exceptions are situations when we have many paths to choose from at the same time (which is quite common and is an indicator of well-designed *metroidvanii*) and so-called *sequence breaking*, i.e. passing the game in a different sequence than the developers intended, using various tricks and loopholes in mechanics.

When it comes to specific games, first of all it is worth mentioning "Castlevania: Aria of Sorrow" (Konami, 2003) [5], on which the prototype I create is modeled directly.

3 Selected tools.

3.1 Godot

Both the client and server are created using the Godot engine. It is a free, open source game development engine, licensed from MIT [6], developed since 2007 (the code was opened in 2014). Is one of the fastest growing open projects on GitHub [7], and has been gaining popularity very quickly over the past two years. It provides a wide set of tools for creating 2D and 3D games, so you don't have to waste time writing all systems, such as rendering or audio playback. It also provides a high-level API for creating network games. However, in my work I do not use this API, but only implement my own data transfer via TCP. The reason for choosing this engine is that it is (subjectively) the easiest and most intuitive to use, and at the same time so advanced that you can create any game in it. Godot's undoubted advantage over other popular engines (Unity3D, Unreal Engine) is also that it has a separate 2D graphics renderer.

However, Godot's disadvantage is that it is relatively new and no very large projects have yet been created that would make it "proven". Therefore, using it for larger games is somewhat risky.

3.2 C#

C# is an object oriented programming language for the Microsoft .NET Framework [8] runtime. It has features typical of modern programming languages, such as automatic garbage collection, parametric polymorphism or lambda expressions. .NET is normally only intended for Windows, but cross-platform is provided by Mono - a set of tools implementing .NET libraries for non-Windows operating systems, including Linux and Mac OS. In syntax and mode of operation, C# is most similar to Java.

Godot has good support for C#, so that C# is one of the three main programming languages in this engine (the other two are GDScript and C ++). In my design I use it as the main language of the server code.

3.3 MongoDB

MongoDB [9] is a NoSQL database management system. Databases in MongoDB do not have any strictly defined structure, but are only documents in JSON format. The system is characterized by high scalability, and in addition it is horizontal scalability - extending it involves adding more equipment (instead of improving it, as in the case of vertical scalability, which is e.g. in SQL) [10]. On my server, communication with MongoDB is via a C# driver.

4 Prototype specification - server

This chapter describes the operation of the game from the server side, and also introduces more important topics related to the Godot engine.

4.1 General description.

The server is written in C# and runs on the Godot engine, but for logic not closely associated with the game uses the pure C# API. The same code as in the client is used to simulate the game, only with some modifications. The server has a main stage (see the next section) and accepts incoming connections from clients and communicates with the database, and the game is divided into independent scenes operating simultaneously. At the same time, he is authoritarian towards the client. All information about items acquired, damage dealt, collisions with the environment are verified on the server and only there they matter. The client only sends input from the player, i.e. pressed keys, chat messages, interface interactions, etc.

Communication between the client and server takes place via packets sent over the TCP protocol. The server and client have appropriate methods for creating such packages. Each packet contains in turn: its size (one byte), header (one byte, enumerated type) and data (mixed types). When receiving a packet, the server or client reads the packet header and reads the data one by one based on the packet.

4.2 Godot engine architecture.

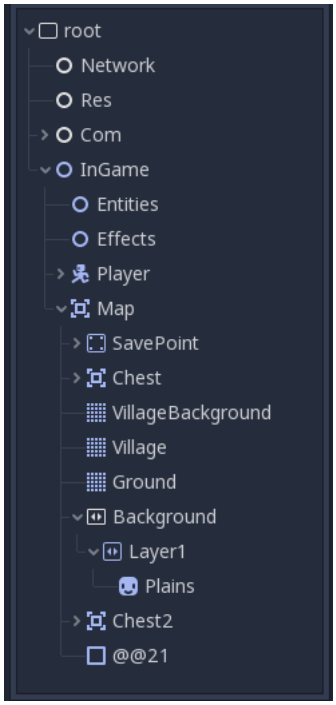

Figure 1: Example scene tree during gameplay

Scenes are the basis of every Godot game. The `SceneTree` class, which has the root of the `Viewport` class, is responsible for the main program loop. Nodes coming out of the root form a tree structure (fig **??**), where the direct descendant of `Viewport` is the so-called current scene. Equivalent to the current stage can be nodes called singletons that exist globally throughout the game. Child nodes of the current scene create what we see in the game - it can be the game menu or the game itself. Each element of the game is a node in the stage tree and at the same time is a stage itself. Saved scenes can be instanced on the main stage, which allows logic to be enclosed in smaller elements and modularity. Nodes in the tree can be additionally connected by so-called signals. The signals operate on the principle of subscribing to events - when the signal is emitted, all connected nodes get information about it and call the method that was connected to the signal. You can send various data to the method with the signal, so you can transfer information asynchronously. This is a very important tool that is used in many places in my work.

An important feature of Godot, which, however, does not matter much, is the architecture based on servers [11]. Logic, physics and rendering operate independently of each other and exchange data only through specific channels,

which allows for their very efficient parallelism. Users can refer to servers only through the classes and their methods. Sometimes this can improve the performance of heavy code fragments, but in practice it is rarely used.

It is also worth mentioning the stability provided by Godot. The engine is designed so that it does not stop if the game is in an incorrect state. It is practically impossible to cause an exception or violation of memory, and thus, unexpected interruption of the game. This only happens when you encounter an engine error, which is not very common, and detected errors of this type are usually quickly patched by the developers or contributors.

4.3 Architecture of the prototype.

My game server is a normal scene with a script attached that is responsible for all logic (each node of the game tree may have a script). Its primary task is to listen to incoming calls. For this purpose, the class `TcpListener` is used, which is in the standard C# language library. When a new client connects to the server, a new thread (class `Thread`) is created, which in the loop expects packets from the client and responds to them.

When a player logs in to the game, he is placed on the appropriate map. The game world is divided into smaller rooms. Each such room has its representation on the server, if it is active. When the player enters the room, an instance of the room scene is created, which has the class `Viewport` as the base type. This ensures independence, because `Viewport` in Godota do not interact with each other, unless they have the same so-called. world; otherwise they even have separate physics. On this stage, the game world is simulated in the same way as at the client's, with some differences for some actions. For example, if an opponent behaves randomly, the draw takes place only on the server, and the client only gets information about how the opponent's state has changed. More about rooms in the subsection 4.6.

The server's task, in addition to exchanging packages, is to create and manage rooms. All these activities, including the simulation of individual rooms, take place in parallel, so the server is adapted to a large number of players. However, the simulation is not free, so if there are a large number of rooms at the same time, stability problems may occur.

4.4 Packets.

4.4.1 Creating packets.

Packages are created directly in the Packet class. The class constructor adopts the header, and the data is attached to the packet by sequentially calling methods that add the appropriate data type. Each of the methods returns the same packet, so you can call them in a string.

Example lines of code creaating packets:

```
player.SendPacket(new Packet(Packet.TYPE.INVENTORY)
.AddU16Array(player.GetCharacter().GetInventory()));

var packet = new Packet(command).AddU8(mode)
.AddString(player.GetCharacter().GetName()).AddString(message);
```

Base methods used to create packets are:

- `AddString(string)` - adds a sctring to packet (ASCII);
- `AddStringUnicode(string)` - adds a string to packet (UTF-8);
- `AddU8(byte)` - adds an unsigned byte to the packet;
- `AddU16(ushort)` - adds an unsigned short number to the packet (short int);
- `AddU32(uint)` - adds a non-negative number to the packet (unsigned int).

Example implementation of one of these methods looks like this:

```
public Packet AddU16(ushort i) {
    data.Add(new byte[] {(byte)(i / 256), (byte)(i % 256)});
    length += 2;

    return this;
}
```

Each of these methods converts data to bytes and automatically increases the variable that stores the packet size. These bytes are packed into small tables that are added to the package's data list. At the very end, all these arrays are combined into one:

```
var bytes = new byte[length];
bytes[0] = (byte)length;

int offset = 1;
foreach (var unit in data) {
    Buffer.BlockCopy(unit, 0, bytes, offset, unit.Length);
    offset += unit.Length;
}
```

Derivative methods that use the above are:

- `AddU16Array(ushort[])` - adds an array size (U8) oraz and sequence of numbers (U16) from the array;
- `AddBoolArray(bool[])` - writes a sequence of boolean values (true/false) contained binarly in a byte (U8).

There are also more advanced methods that are intended for specific data from the game:

- `AddStateVector(Array, Array, bool[])` - adds a state vector to the packet (more about the state vector in the section 4.6);
- `AddStats(Player, string[])` - adds the given player's statistics to the vector. The first two bytes of statistics are a bit mask specifying which statistics are sent. The statistics list is in a fixed table and each bit corresponds to the statistics from the list. One bit means that the statistics are sent, zero bit that is not in the packet. The mask is followed by a sequence of U16 numbers containing the values of individual statistics;
- `AddEquipment(ushort[])` - adds equipment to the vector, empty slots omitted. It works like statistics - one byte is a mask that determines which slots are sent, followed by the identifiers (U16) of items (or souls) on individual slots.

The `Packet` class also includes the `Send()` method, which accepts the stream of the specific player's network socket to which this packet should be sent. Most often it is called inside the class `Player`. Each packet can be sent to many players.

4.4.2 Reading packets.

The package is read by the class `Unpacker`. It contains methods that are technically the opposite of the package creation methods: `GetString ()`, `GetU8()` and `GetU16()`, returning the appropriate values from the data in the package. The object of the `Unpacker` class gets raw bytes and with each call to the data mining method, it automatically shifts the index in the byte array, which makes it easy to read what it needs. The data in the packet is completely arbitrary, you know what's in it only based on the second byte, which indicates the type of packet.

An example implementation of the method, which reads the number from the packet (to be compared with the reverse method, `AddU16()`, in the subsection 4.4.1) looks like this:

```
public ushort GetU16() {
    offset += 2;
    return (ushort)(data[offset-2] * 256 + data[offset-1]);
}
```

The `HandlePacket()` method uses the command contained in the package to perform the appropriate operations and extracts the necessary data. It is quick and convenient to use, but it has a disadvantage - it is prone to mistakes when creating the package. If it contains invalid data, reading it will fail. That is why it is important to have documented package types and what should be included in them. The fact that types are referred to as an enumeration type is very helpful.

Sample code from the `HandlePacket()` method, which supports the package responsible for receiving a key press by a player and sending it to other players:

```
case Packet.TYPE.KEY_PRESS:
var id = player.GetCharacter().GetPlayerId();
var key = GetU8();

Server.GetControls().Call("press_key", id, key);
player.GetCharacter().BroadcastPacket(new Packet(command)
.AddU16(id).AddU8(key));
```

4.5 List of packet headers.

The types of packets sent between clients and the server are described below. Each item contains: PACKET NAME - the name of the item in the enum type structure that identifies packages and [PACKET DATA] - a list of data that is sent in the package. Individual data types are described in the previous chapter.

4.5.1 Packets sent from sesrver to client.

- `HELLO` - a welcome packet sent to the player after connecting to the server, containing the game version. If it turns out that the client is out of date, information will appear to update it and further action will be impossible until then.
- `LOGIN[U8]` - result of the attempt to login the player. 0 means success, above zero is the error code (e.g. incorrect password).
- `REGISTER[U8]` - result of an attempt to register a player. 0 means success, above zero is the error code (e.g. an account already exists).
- `ENTER_ROOM[U16, U16, U8, U8]` - information about entering the room. The first number is the map number, the second is `ID` of the player's object, the other two describe his starting location. The customer after receiving this package loads the map given to him and sets the character in the right place and assigns the received `ID` to it.
- `KEY_PRESS[U16, U8]` - information for other players that the player with the given `ID` pressed the given key.
- `KEY_RELEASE[U16, U8]` - information for other players that the player with the given `ID` released the given key.
- `ADD_ENTITY[U16, U16]` - information that the object of the given type was created and what `ID` got. After receiving this package, the client creates an object of this type.
- `REMOVE_ENTITY[U16]` - information that the object with the given `ID` has been deleted.

- `TICK[U8, [U16, ...] ...]` - a synchronization packet sent cyclically to all players in a given room. It contains information about important properties of all objects, e.g. position, current status, etc. Each object defines a set of this information itself. See subsection 5.3.
- `SPECIAL_DATA[String, data]` - similar to a synchronization package, but used by objects that have different state with other players (e.g. chests). See subsection 5.3.1.
- `INITIALIZER[U16, ...]` - a synchronization package containing data for initializing one object. It is used for objects about which the client must have initial information, but their further status is irrelevant from the server side (e.g. an effect that is not created on the client side)
- `CHAT[U8, String, StringUnicode]` - chat message of the specified type. The package also contains the player's nickname and content.
- `DAMAGE[U16, U16]` - information about damage inflicted on an object with a certain `ID`. It can be a player or an opponent.
- `STATS[stats]` - a list of player statistics. Sent at the beginning and whenever statistics change. The first 2 bytes specify which statistics are included in the packet (information is saved in bits), followed by the values of individual statistics.
- `INVENTORY[U16Array]` - information about the content of the player's backpack. The first byte is the number of items, followed by `ID` of subsequent items from the backpack.
- `EQUIPMENT [equipment]` - information about the player's inventory.
- `SOULS[souls]` - information about the souls owned by the player, sent on the same basis as `INVENTORY`.
- `SOUL_EQUIPMENT [equipment]` - information about the player's spiritual equipment.
- `ABILITIES[U8]` - a byte that specifies which player's abilities are enabled (one bit corresponds to each ability).
- `MAP[U16Array]` - information about the rooms discovered by the player. Each pair of numbers are coordinates on the map that the player discovered.
- `ITEM_GET[U16]` - information about obtaining an item of this type.
- `SOUL_GET[U16]` - information about getting a soul of a given type.
- `SAVE` - packet sent to the client as confirmation of saving the game (see subsection 6.4.3).

- `GAME_OVER[U16]` - information about the death of the character. Sent during the game when `HP` drops to 0 or when trying to login when the character is still dead. Contains information about the remaining seconds to spawn.

4.5.2 Packets sent from client to server.

- LOGIN[String, String] - login information (username and password) sent to the server.
- REGISTER[String, String, U16] - username, password and character color, sent to register a new account.
- KEY_PRESS[U8] - information about pressing a given key.
- KEY_RELEASE[U8] - information on releasing the given key.
- CHAT[U8, StringUnicode, [String]] - sends a chat message to the server. The first byte is the message type, followed by the content, and finally the optional player name if the message is whispered.
- CONSUME[U8] - the action of using the item from the backpack at the given position.
- EQUIP[U8, U8] - the action of placing an item on a given slot from the given position in the backpack.
- EQUIP_SOUL[U8, U8] - as above, but with souls.

4.6 Rooms.

The game world on the server is organized into rooms (class objects `Room`). Each such room is a separate stage in Godota, with a root of type `Viewport`. This means that they are independent of each other - although they exist in parallel, there are no interactions between the objects within them. Each room corresponds to a map in the game, which is a fragment of the game world. It can be, for example, the entrance to a cave, some underground corridors or a path between the villages. The maps are rectangular rooms consisting of tiles and objects such as enemies or treasure chests. After logging in, the players go to one of the rooms and are free to move between them. The transition to another room is done by going beyond the screen. Of course, this is not possible everywhere, because you can hit, for example, a dead end or obstacle.

From the logical side, each object on the stage is mapped on the server and assigned `ID`. Each `ID` is unique and new objects are simply given a sequential number. The range of these numbers is large enough to always have enough numbers for each new player or object that appears on the map. `ID` is used to identify the node in all operations such as change of state or deletion, the information about which must be sent to clients. Sending identifiers as numbers is much more efficient than using names (strings) because less data is sent.

The class `Room` has several methods to improve room management. The most important of these is `BroadcastPacket()`, which sends the packet to every player in the room. It is used for sending chat messages, information about new objects and synchronizing the game state (*tick*). Synchronization involves sending a state vector for each object that contains the values of certain variables that have changed since the previous synchronization. Most often, they are the coordinates of the object's location, but the client also needs information about the state of the object (e.g. in the case of opponents it may be the direction in which they are facing or the action they are taking). Values that have not changed since the previous *tick* are not transferred to optimize data transfer. If the entire object vector has not changed, information about it is not included in the packet at all. Information about whether a given value is in the vector is stored in the first byte of the vector. Bit 0 means that the value has not changed, bit 1 means that it has changed, and a new value can be downloaded. The synchronization packet is sent every 0.24 seconds (15 frames in the game). More about synchronization in the subsection 5.3.

The server room performs a full game simulation. It executes the same code as the client, thanks to which it can easily verify events such as hitting opponents or collecting items. This makes cheating very difficult. The downside is that simulation on the server is very expensive, especially since there can be several hundred such instances at the same time if there are a lot of players. For some optimization, unused rooms are removed. If no packet is sent for 100 synchronization periods (i.e. there is no player in the room), the room is automatically removed from memory and stops being simulated until the next player enters. Then the room is created again - it has `ID` counted from 0, all objects return to their starting positions, and defeated opponents are reborn.

5 Prototype specification - client

This chapter describes the operation of the client application from the network side and describes its interaction with the server.

5.1 General description of the operation

Unlike the server, the client is entirely written in the GDScript programming language. It is a scripting language, syntactically very similar to Python. Is the main language used to write scripts in the Godot engine, created especially for this purpose. It is much slower than compiled languages (C ++ type), but efficient enough to create logic in games (including simple AI and interaction between objects, for example).

The customer uses the entire Godot API for network communication. The most important element in this process is the class `StreamPeerTCP`, which is responsible for connecting and exchanging data. Most network operations are performed in a singleton called `Network`. This is a global scene that deals with connecting the client to the server and receiving and interpreting incoming packets. The same classes as on the server are responsible for packet operations: `Packet` and `Unpacker`, but implemented in GDScript. The reason for this split is that the data in C# and GDScript are represented differently and continuous conversion can affect performance. In addition, while using C# is quite fast, mixing two languages calls various intermediary methods, which also take time. The choice of GDScript instead of C# is due to the simplicity of writing the code in that language; its performance is sufficient.

Singleton `Network` defines many signals that are emitted when most packages are received. These signals contain various data received from the server. Many systems take advantage of the fact that `Network` is available everywhere in the code and attaches various signals to each other's needs. For example, if the customer receives the `EQUIPMENT` package, the data is extracted from it and a signal is emitted containing the information received.

Below is the code snippet responsible for this:

```
match unpacker.command:
        Packet.TYPE.EQUIPMENT:
                var equipment = []

                var equipped = unpacker.get_u8()
                for i in 8:
                        if (equipped & Data.binary[i]):
                                equipment.append(unpacker.get_u16())
                        else:
                                equipment.append(0)

                emit_signal("equipment", equipment)
```

Connecting to the `equipment` signal looks like this:

```
Network.connect("equipment", self, "on_eq")
```

The client uses this signal in two places: in the character menu to update the inventory information and in the player's node, where he uses it to display the appropriate weapon during an attack. Other signals work in a similar way and are mainly related to data such as equipment, skills or the area of the map.

5.2 Wymiana pakietów

As mentioned in the previous section, the client uses similar classes `Packet` and `Unpacker`, implemented in GDScript.

Below, for comparison with 4.4.1 and 4.4.2, is the implementation of the `AddU16` and `GetU16` methods in the client code:

```
func add_u16(i : int) -> Packet:
        data.append(i / 256)
        data.append(i % 256)
        data[0] += 2
        return self

func get_u16() -> int:
        offset += 2
        return data[offset-2] * 256 + data[offset-1]
```

Apart from the syntax, there are many similarities. GDScript is dynamic, but has optional typing (which, for now, has no effect on code performance). The slight difference here is the array type `data`. In C# it is of type `byte[]`, fixed size, and in GDScript it is dynamic `PoolByteArray`. The client implements only these simple methods. The advanced ones, such as `AddU16Array()` or `AddEquipment()`, are not needed there.

5.3 Synchronization

As described in the subsection 4.6, each object in the server room is assigned a unique `ID`, i.e. a number that identifies it. The same numbers are mapped to the customer. After entering the room, the player receives the `ADD_ENTITY` package for each object that is in that room. This packet contains information about the object type and `ID`. Based on this information, the appropriate node is created on the client side. Then the customer receives the so-called complete package `TICK`. This is a typical synchronization packet, but unlike those sent periodically to each player, it sends all information about all objects.

The `Data` class is responsible for handling synchronization packages, with the static method `apply_state_vector()`. It takes the object of type `Unpacker`, the node on which the synchronization operation is to be performed and the change vector. Each object with data that needs to be synchronized has 3 methods: `state_vector_types()`, `get_state_vector()` and `apply_state_vector()`. The first returns data types that are in the vector so that you know how to pack them in the package (in the case of the server) or how to extract them from the package (in the case of the client). `get_state_vector()` returns an array with this data, and `apply_state_vector` accepts the data array and applies it to the object (at the same time it gets a state vector, because sometimes this information is needed). The data in `apply_state_vector()` depend on the state vector. If it is written in the vector that some information has not changed (bit 0 and is not included in the packet), the old value is thrown into the final data table, obtained from `get_state_vector()` (i.e. this method is called is on both the server and the client).

Below are sample implementations of these methods for one of the opponents in the game, the skeleton:

```
func state_vector_types():
    return [
        Data.TYPE.U16,
        Data.TYPE.U16,
        Data.TYPE.U8,
        Data.TYPE.U8,
        Data.TYPE.U8
    ]

func get_state_vector():
    return [
        round(position.x),
        round(position.y),
        state,
        direction+1,
        walk+1
    ]

func apply_state_vector(timestamp, diff_vector, vector):
    var old_position = position
    position.x = vector[0]
    position.y = vector[1]
    if has_meta("initialized"):
        sprite.position = (old_position -
                position) + sprite.position

    state = vector[2]
    direction = vector[3]-1
    walk = vector[4]-1
```

As you can see, the data must be prepared, as per the instructions in the package. To remove, my code cannot be passed numerically negative or fractional, so you must be one of them, which are converted into an invalid integer and are restored when received. For fractional numbers (`float`)) You lose the sporadic part of precision, but this is irrelevant because it is information about fractions of pixels in the positions of objects.

Each object in Godota has so-called metadata. It is a set of key-value pairs that can contain any values. For the selected object assigned by the server `ID` object appear in the metadata. In the code below, the value `initialized` contains information about whether the object is initialized.

```
if has_meta("initialized"):
    sprite.position = (old_position - position) + sprite.position
```

It is set to `true` when the object is first synchronized from the `TICK` package.

Until initialization, every object in the game (only for the client) is invisible and does not change its state.

The reason that this information is checked in the example with the skeleton is that the change of its position is immediate only for the first packet (i.e. when it is still uninitialized). For the next packages, its actual position changes, however, its child node `Sprite`, which is responsible for displaying the graphics, does not change. This node then moves towards the real position, but it happens gradually. Thanks to this, instead of visibly jumping position changes in the event of a large change, we see a smooth, interpolated transition from the old position.

5.3.1 Special objects

The so-called special objects. Their status is not synchronized, but is different for each client. An example of a special object is a chest. There are several of them in the game, each has a different item and they work so that each player can open each chest only once, regardless of other players. Information about whether the chest has already been opened is sent to the customer immediately after joining the room (in a separate package from each special object). Information about opening the chest is stored in a database in a special table for each character. This table contains unique identifiers for each chest.

Similarly, the second special object, which is the equivalent of chests for souls. But this mechanics can be adapted to other things, such as hidden passages or mechanisms for levers/buttons.

5.4 Lag compensation

As mentioned in the subsection 4.1, the server is authoritarian towards the client, which means that all decisions about moving and interacting with objects are made on its side. However, the player sending information to the server and displaying what the server sends him is not enough. There are delays of several dozen to several hundred milliseconds between the client and the server, which with standard game smoothness (about 16ms per frame) is simply unacceptable [12]. Therefore, even though the server simulates everything that happens in the rooms, the same happens to some of the clients. Synchronization packages are used only for error correction (the situation where the opponent is in a different place on the server than at the client's is not so rare; hence sometimes big jumps in position) and in some cases, for making decisions about random actions performed on the server . The game code is divided into parts that take place only on the server (e.g. AI decision making), only at the client (e.g. animation change) and in both places. Because what the client sees does not always agree with the state on the server, corrections are made based on the information received about the current state of the world.

5.5 Players and controls

The `Controls` class is responsible for control. The most important what it contains is an enumerated type with all available actions in the game. These are: `ATTACK`, `JUMP`, `UP`, `RIGHT`, `DOWN`, `LEFT`, `SOUL`, `ACCEPT` , `CANCEL`, `MAP`, `MENU`, `CHAT`, `COMMAND`, `CLOSE_CHAT`. Each action has a key assigned to it. In addition, `Controls` may be in different states, which differ in the keys that are being processed and the way they are done. These states are `ACTION`, `CHAT`, `MENU`, `MAP`, `GAME_OVER` and `QUIT`. Each game interface can accept input from the player only in certain states, i.e., being in the menu, we cannot move the character. The keys can be operated in two modes: normal and local, and each has its own method - `process_key()` and `process_key_local()`. They differ in that `process_key_local()` does not send the server information about the key being pressed.

The class `Controls` has a key-press signal that is connected in various places (mainly interfaces). The character itself is also controlled through this class. Information about pressing and releasing the key received from the server is also processed here. Thanks to this, the control is standardized in every place and can be controlled from the outside. This is how other players' characters move - when the customer gets input information, it is passed to the `Controls` class, from where the signal received by each character on the board is sent. In addition to the key identifier, the signal also sends the current control mode and `ID` of the player who pressed the key. If a character receives this signal and `ID` matches their assigned `ID`, then it will perform the appropriate action.

Each character in the game is an instance of the scene named `Player`. Avatar control is completely unified, but one of them is highlighted - it's called main player, i.e. the character currently controlled by the player during the game. It differs from the rest in that it is not in a sense assigned to the map; when changing rooms, it is not destroyed, but only transferred to another stage. In addition, as child nodes it has a camera (which moves the screen during the game) and a user interface, i.e. character menu, map and HUD (Head-Up Display). The main player character also receives position information differently - it is partially deferred when moving. Thanks to this, the character's movement is smoother and not interrupted by step changes in position. When the character stops for a moment or is too far from the correct position, the change of position is forced.

6 Prototype specification and user documentation

This chapter describes the most important aspects of the game, elements related to the game and the organization of the application project.

6.1 Project organization

The project is organized in such a way that it is easy to separate the client from the server. All server-related scripts and scenes are located in a folder named `Server`. In addition, the project folder contains subfolders:

- `Audio` - containing all sounds;
- `Graphics` - containing all graphics;
- `Maps` - in which there are room scenes;
- `Nodes` - a folder with scenes of objects and opponents;
- `Resources` - a folder with various other resources, including fonts and JSON files with data;
- `Scenes` - main game scenes;
- `Scripts` - all scripts used by the client.

In addition to game data, the project folder has several helper scripts and programs that simplify project management. The scripts are:

- `build_game.bat` - script for exporting a game (creating an executable file) from the command line;
- `run_database.bat` - script to run the database;
- `run_game.bat` - a script to run an instance of the game (the Godot editor gives the opportunity to run only one instance);
- `run_server.bat` - script to run the game server, i.e. directly to the `Server.tscn` scene.

There are also 2 programs written in Ruby:

- `MapEditor.rb` - a program used to edit information contained in maps, and more precisely, to determine the location of walls (see subsection 6.5.3);
- `ResourceList.rb` - a program that displays a list of all resources with their numbers (see the next section).

6.1.1 Resource files

The previously mentioned program `ResourceList.rb` prints two types of resources. The first of them for maps, available in the files `.tscn`. This is the scene information format used in the Godot engine. It's on text, so it's easy to extract information with it, even manually. Each map (room) contains information about its position, part and position of the walls, and above all, the content ID, which is available in the program with all maps printed, is useful to use if a given number is occupied `ID` is something worth taking care of.) The second type of resources are `.json` files, which contain information about items, souls and opponents. This is information such as name, description or statistics. The JSON format is very good for collecting such data. They are grouped into folders by form (`Items`, `Souls` and `Enemy`), but each file contains a lot of information. The data in each file is available in the table, and the files are sorted alphabetically and combined in one list. Information about `ID`, e.g. the item is not saved and is determined only on the basis of indexes, so a program such as `ResourceList.rb` is useful if information about `ID` is sometimes needed.

6.2 Running the game

For the game (client) to work, a server connection is needed. The prototype does not have an official server, typical for MMORPGs, but setting up your own server is not difficult, and the client connects to the local host by default.

Starting the server requires a working database. You must first install MongoDB on your computer, preferably in such a way that the software is available via the command textttmongodb on the command line. Then you can use the batch file `run_database.bat`, which will take care of starting the database server correctly.

The next step is to start the server itself. For this you need a Godot engine version 3.1 MONO. It can be downloaded from the official site: `https://godotengine.org/`. To use the script `run_server.bat`, the downloaded executable file must be available globally in the system terminal and have the name `godot_mono.exe`. Alternatively, you can run the command `godot -s Server / Nodes / Server.tscn` in the project folder, where `"godot"` is the path to Godot.

The game can be called by starting Godot (version 3.1) in the project folder or by opening the project in the Godot editor and pressing F5.

6.3 General game descripion

Figure 2: Title screen

The game I created is called *The Soulhunter*, which refers to the skills of our hero, enabling the acquisition of strong opponents, here called souls. Shortly afterwards, the application connects to the server and waits for the HELLO package (if it is not available for 5 seconds, an error message will appear). If you see, the user will see the title screen of the game (Fig **??**). There, he has the option of selecting login, account usage and exit. The menu is fully accessible via the keyboard. If the user wants to register, he asked for the login, password and character color (there is no character wizard as in a typical MMO, but you can change the avatar color so that everyone doesn't look the same). Due to the fact that the game is only a prototype so far, no email address is provided, which also means that there is no change as to whether you have recovered your password.

After consideration, the customer has an available package with information to accommodate should be loaded. Later, all objects are initialized and the character is placed in the starting position. From this moment the player gains control of his avatar and can move around the world. As in a typical platformer, you can walk and jump. You can also attack. At the beginning of the character there is no equipment, so you attack with your fists, but the first weapon is available in the crates next to the starting point, and then you can use those available in other places, you need to download from opponents.

An important issue is also the distinction between the service account and the service form. The account consists of a login and password, and its name and various statistics. Player after registration and account creation, and the character is created only after connecting. Normally you can have many different characters in MMORPGs, but due to the fact that it's just a prototype, and

creating a character management system is not easy, each user has only one avatar. It is automatically created at login and has the same function as user login.

6.3.1 Basic game rules

As in any MMORPG, my prototype has no specific purpose. The player is placed in an open world and has the freedom to take action. The most common motivation of players in games of this genre is to develop their own character, by gaining further levels of experience and improving equipment. Both of these things are possible in my game. Gaining experience is done by defeating opponents. Of course, the player has the choice whether he wants to kill or not, but if he does not, he does not gain experience points, i.e. his character does not advance to the next levels and does not become stronger. Equipment upgrades are achieved by acquiring better items. New equipment can be obtained from chests hidden in various places or by killing enemies who may drop rare items.

In a normal MMORPG, the stronger character gets access to more difficult challenges. Can fight stronger monsters and get better and better items. However, in the prototype, despite the lack of limit of experience levels, acquiring them quickly loses its meaning due to the small game world.

6.3.2 Game's world

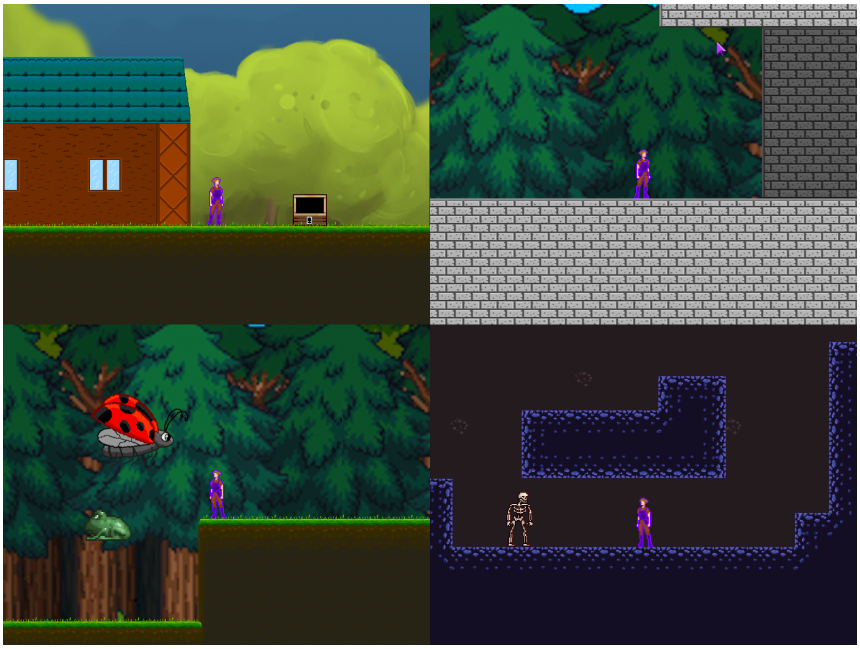

Figure 3: Various locations in the game's world

There are 5 locations in the prototype: village, plains, forest, cave and fortress. They differ in the appearance of the environment and opponents that can be found (there are 7 types). Figure fig: locations shows examples of them. There are 15 rooms in the game assigned to different locations (from 1 to 4 for each). Each room is rectangular, with dimensions that are multiples of Full HD (1920x1080) resolution. The village serves as the starting location. It contains only one room and also has one of three save points in the game. There are no opponents there. There are plains to the east and west of the village. The enemies that can be found there are overgrown ladybugs and frogs. The first ones fly left and right, and the frogs only jump. Behind the eastern plains is a forest in which the same opponents are. At the end of the forest is a fortress with skeletons and undead knights who throw knives. There is also another save point and you can get the double jump ability. This skill is needed west of the village in the cave. Having a double jump you can get to the room where the strongest opponent in the game is located and which also drops the strongest weapon. Although the world is so small, it was possible to put in it the most important aspect of metroidvanii, which is an area that is inaccessible at first because of an insurmountable obstacle without the previously acquired skill.

Technically, each room is a separate stage, prepared in the Godot engine editor. The root of each of these scenes has a script called `MapSettings`, which

contains variables that store map information. This is primarily `mapid`, which is the map identifier, which is e.g. sent to the client when the server tells him which map to load. Other variables are `map_x` and `map_y`, which specify the location of the map, and `width` and `height`, which determine its dimensions. `map_x` and `map_y` are primarily used to determine in which room to place the player. When the player leaves the screen, the server looks for the adjacent room that is located there and loads its `mapid`. Dimensions determine the screen restrictions at the client (the camera cannot go outside the current room), and on the server they are used to check that the player is not outside the map and that the next one should not be loaded.

In addition to the script, each map has one or more `TileMap` nodes, which are the basic element building the environment. The game world is built of 64x64 tiles. The tiles have different shapes (flat or sloping) and graphically represent various substrates and walls. These graphics are drawn so that they combine seamlessly, which gives the impression of a whole. All opponents and objects are also placed on the stages. Each such element is an instance of a certain scene, which has a script responsible for the correct representation of these elements on the server and at the client. All are properly registered, assigned `ID` and this is done automatically.

6.4 Mechanics

This section describes the mechanics found in the prototype.

6.4.1 Statystyki postaci

As in a typical RPG, each character has several statistics that affect how he performs in combat. There are 7 in my prototype: `HP` (*Hit Points*), `MP` (*Mana Points*), `attack`, `defense` , `magic attack`, `magic defense` and `luck`. The first two additionally have their maximum values as separate statistics. `HP` are character hit points. If they fall to 0, the character dies and the player must wait for it to be reborn. The `MP` points are intended to use some souls (casting spells). If the character doesn't have enough, they may not be able to use the skill. `attack` determines how much damage the opponent takes with a regular weapon attack. `defense` means resistance to enemy attacks; the value of this statistic is simply subtracted from the damage dealt, but they cannot fall below 1. `magic attack` is responsible for the power of magic attacks. The `magic defense` and `luck` statistics have no effect on the prototype. The first should be responsible for resistance to magical attacks, and the second should increase the chance of obtaining rare items and souls.

At the beginning, each statistic has a value of 1, except `HP` and `MP`, which are 120 and 80 respectively. Statistics can be increased by gaining experience levels. Each character starts at the first level and can gain more thanks to experience for defeating opponents. After gaining a certain amount of experience points, the character level and all basic statistics increase by 1 (10 and 5 for `HP` and `MP`), and the amount of experience to reach the next level increases.

In addition to experience levels, the assumed equipment and souls influence the statistics. The equipment that can be worn is a helmet, armor, shoes and a cover (e.g. a cape). There are 10 items in the prototype that can be put on different inventory slots and affect different statistics. It is a weapon that increases attack statistics and armor (armor and helmets) that increases character defense.

6.4.2 Souls

A very important element of the game are souls (from which the title The Soulhunter took shape). They represent the power of defeated opponents. Each monster has several (2+) souls that the player can acquire and which are somehow related to that monster. There are 9 types:

- `trigger` (red), similar to spells. They are used by holding the key up and attacking. Example: throw a magic knife.
- `active` (blue), also similar to spells, but they are not cast immediately, and last for some time. They are used with the Shift key and can work while holding this key, or switch the effect by pressing it. Example: short levitation while holding the key.
- `augment` (yellow) directly affect the character's statistics or other aspects such as jump height.
- `enchant` (pink), affect the character's inventory. They can, for example, increase the statistics of a certain type of item, or the weapon's attack speed.
- `extension` (translucent gray), increase the potential of other souls. They can, for example, affect the speed of magic bullets or enhance the effect of increasing sttistics.
- `catalyst` (turquoise), used for spiritual craftsmanship (combining souls, extracting from items, etc.). They do not exist in the prototype.
- `ability` (green), give characters special skills. This skill can be a new action or give some lasting effect. They can be activated and deactivated without restrictions. Example: double jump (possibility of jumping a second time in the air).
- `mastery` (black), similar to green, but have a greater impact on the gameplay and cannot be switched. They do not exist in the prototype, except for one entry called *Soul Hunt*, which allows you to win other souls.
- `identity` (white), specify the character class. They affect the type of weapon a character can put on and increase its statistics at subsequent levels of experience. In addition, they can increase or decrease the chances of getting some souls, depending on their compatibility with the associated character class. They do not exist in the prototype, except for one entry called *Mage*, which has no effect on the gameplay.

6.4.3 Game state

Unlike a typical MMORPG, you can lose progress in my game. Normally, every action performed by the player is immediately saved in the database. Any changes to statistics, inventory, items are mapped in the database, so if a player quits the game at any time, he won't lose anything. In my game I used a more traditional approach. You have to save the game, otherwise you lose progress since the last save if you leave the game or the character dies. You can save the game at special save points. At the time of saving, the current state of the character is recorded in the database. Normally it is stored in the server's memory. Of course, in the event of a breakdown, this data is lost, so in an actual game (not a prototype), it would have to be solved differently (e.g. by a mirror database with "temporary" data). Fig **??** shows the appearance of the save point.

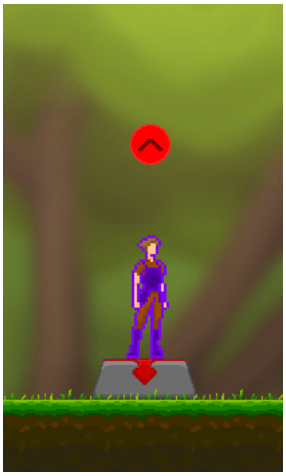

Figure 4: Save point appearance

If the character dies, there is a loss of progress, and you also have to wait for some time. The waiting time in the prototype is 30 seconds. After the defeat screen appears, the player can wait until the counter reaches zero and continue playing or log out. However, even after logging out and logging back in again, the character cannot be used until it is reborn (the player does not have to be in the game for the time to count down).

6.5 Interfaces

This section describes the interfaces that the player has to deal with in the prototype game.

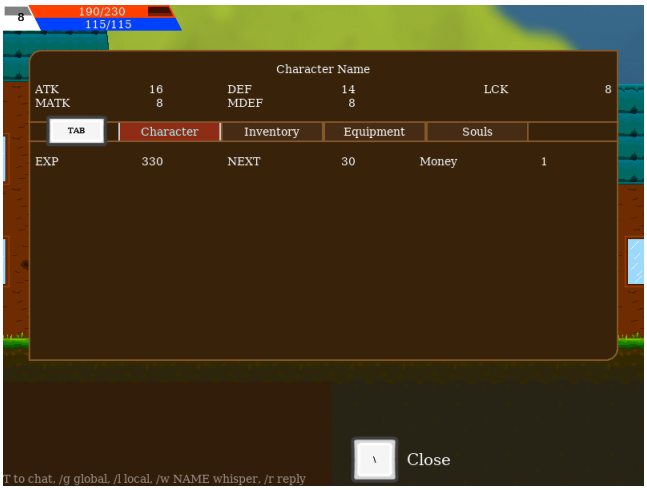

Figure 5: Character menu

6.5.1 Character menu

Fig **??** presents a character menu that can be opened during the game and contains all important information about our character. It is divided into 4 cards: character (*Character*), equipment (*Inventory*), equipment (*Equipment*) and souls (*Souls*). The character card contains all statistics that are not displayed in the menu header, i.e. experience points, experience required to the next level, and money (unused).

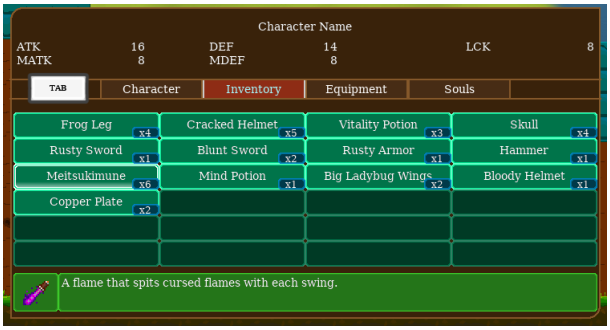

Figure 6: Inventory tab

The equipment card (fig **??**) shows all items that the player possesses that are not in the inventory. Users can view their items here. The currently selected item is highlighted and its icon and short description are shown. Here you can also use some items: food and drinks that restore health or mana (magic energy). On the technical side, the server sends a raw list of items, and the client after

receiving groups it by type (the quantity in a given group appears as a small label next to the name).

6.5.2 Equipment and souls

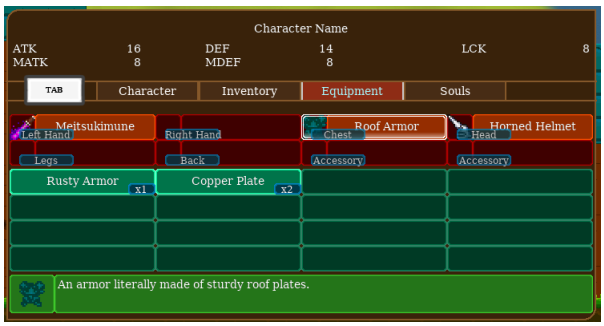

Figure 7: Equipment tab

Fig. **??** shows the inventory submenu, where the player can put on and remove items from the character. It is divided into two parts: the slot part, in which we choose what type of equipment to put on (they are signed with small labels under the name of the item) and the equipment part, which is activated after choosing the equipment slot (fig **??**). The selected slot then changes the color of the highlight and a cursor appears at the bottom of the menu, which also, like the equipment card, contains all the items you have that can be put on the slot. You may also notice that when the item is highlighted, some statistics change color. This is to view the strength of the items. Blue backlight means that putting on the item will increase the statistics, red will decrease it. Unfortunately, in the prototype this preview works entirely on the client side, which means that there is no access to e.g. information about soul effects (which are implemented only on the server) that affect objects.

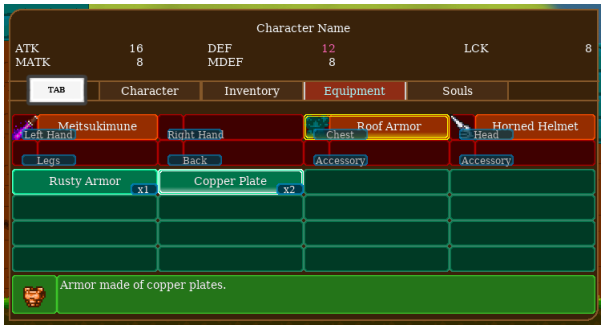

Figure 8: Equipping items

Technically, the item is created by sending the `EQUIP` package to the server, with the item position that the player wants to create. The list of items is updated with the signal `inventory`, which contains a raw table of owned items. They are then grouped (small labels next to the item name are the number of repetitions of a given item in the table), and each group is assigned the position of the first item on the list of a given type. The server gets this information and is able to determine which item should be put on and removed from the equipment.

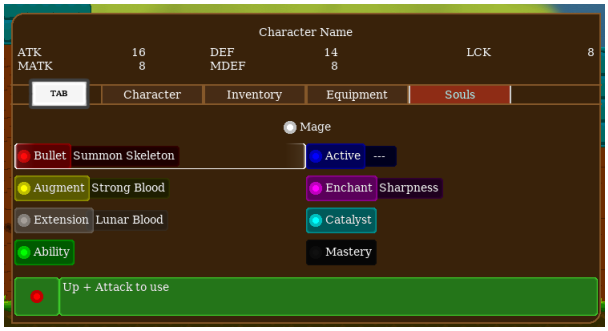

Figure 9: Souls tab

Fig. **??** presents a soul card that works in a similar way to the inventory card, but the soul list is not on the same screen as the slots list, but a separate one. We have a list of types of souls with their colors, and each item, except the name of the type, shows what is currently founded (if applicable), and when choosing a slot, a description is shown what souls of a given type do, and how to use them. After selecting the slot, the acquired souls and their number for each type are shown (Fig **??**). At the bottom is the description of the currently highlighted soul and how many mana points it consumes (textit MP).

Figure 10: Equipping souls

In the case of green souls (fig **??**), we do not have to do with donning but switching, so each of them has a small icon next to the name, informing about the state of the corresponding skill. This information is updated by the signal `abilities` from the node `Network`, which also reaches the player's node to tell him what skills he can use.

Figure 11: Ability souls

6.5.3 Mapa

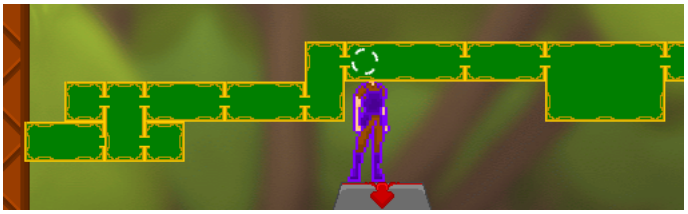

Figure 12: Map preview

During the game, the player has access to the map (Fig **??**). It is represented as a grid of rooms, illustrating the layout of the rooms. Each room is appropriately highlighted and has passageways marked. One square on the map corresponds to an area of 1920x1080 in the game. The map clearly shows that the rooms are rectangular. The player's position is also shown. In a full-fledged game, various POIs (*Points Of Interest*) are usually marked on such maps, e.g. save points, unopened chests, tasks, etc.

On the technical side, the server stores in the database the coordinates of each map square visited by the player. They are saved as strings in an array, because because of the documentary nature of the database, they occupy the same amount of space. However, they are already sent to the client in the form of pairs of numbers (in the case of a larger map, partitioning would probably be useful, because sending all data at once may be inefficient). Discovering the map to make it more efficient occurs when crossing the "squares" of the map. The server then checks if this new item is already discovered and if not, it is added to the discovered rooms.

The way of drawing rooms is determined on the basis of data saved in the maps themselves. Each map has a table of walls in which each square from the room rectangle is assigned 4 values, one for each of the directions. A given wall may not exist at all, it can show a border (straight line) or a transition (a line with a cut in the center). These data are properly interpreted and drawn in the game. However, this system has a downside. The client has all the information about the arrangement of the rooms and theoretically draws only those about which he got information from the server that they are discovered. However, customer modification that allows you to draw all map rooms is unfortunately

not particularly difficult. This is a compromise that must be agreed if we want the server not to have to send a lot of map information itself.

6.5.4 Chat

The chat is located at the bottom left of the screen (see: Fig **??**). The player can write at almost any time of the game (also when he is in the character menu or viewing the map). Chat can work in 3 modes: global, local and whispered. The first sends a message to every player who is currently in the game (in a normal MMORPG, the servers are clustered and global chat works only for people from the same cluster). Local mode sends messages only to players in the same room. Whisper mode only sends a message to one selected player. Chat mode changes with special commands starting with /. These are `/ g` for global chat, `/ l` for local and `/w [playe_name]` for whispering, where `[player_name]` specifies to whom we want to send this message. Messages of different types are properly signed and have different colors (Fig **??**).

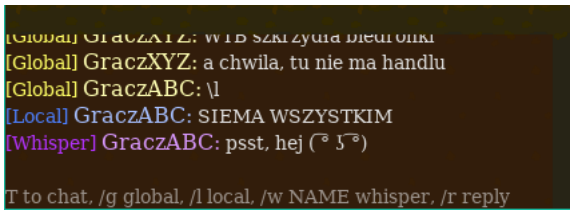

Figure 13: Example chat conversation

7 Application testing results

After completing the prototype, performance tests (so-called *stress test*) were performed, checking how the software copes with a large number of players. Unfortunately, it was not possible to collect a large number of people (after advertisements on social media), but the obtained data can be used to estimate the server's capabilities.

During 109 minutes and 50 seconds of server operation, 18 players joined, with 5 playing simultaneously at the peak moment. During this time 11010 packets were received (55068 bytes or 275268 counting TCP headers), and sent 41533 (448321 bytes, 830660 with headers TCP). Assuming an average of 4 players all the time, it gives us a data transmission of about 19 bytes per second per player. With 100,000 players active at the same time, this gives a data transfer of 1.9 MB / s, so when it comes to networking capabilities, the prototype works well in this respect. Of course, this is an underestimation (not including the TCP headers mentioned earlier), but if part of the network worked as it should based on the UDP protocol, the excess protocol would not increase the order of the amount of transmission amount.

What tests have not tested is CPU performance. It is difficult to say how stable the server would behave with more players. The more load on the server, the slower the simulation, i.e. the greater desynchronization of clients (and in critical cases - failure and shutdown of the server process). Theoretically, such tests can be done automatically, but this would require the creation of appropriate tools and simple AI to guide artificial players. Although incomplete tests have not checked the real limitations of the server, their results are promising.

8 Summary

A working prototype of a two-dimensional MMORPG game was created in the work. Players have the option of creating an account and logging into the game, and in the game itself, you can explore the world and meet other people. The whole is written in such a way that it is impossible to cheat (apart from the creation of so-called bots, i.e. programs that automatically control the player's character, which would be quite difficult, however). All important operations are performed on the server, so the customer has no way of cheating, e.g., saying that he has acquired an item that he should not have or that he got somewhere that he should not be able to.

Data exchange between the client and server is done in a fairly efficient and easy to extend way. For this purpose, the packets are byte in form, although some data is sent redundantly to simplify some operations. The way you implement package creation and unpacking allows you to easily add and manipulate new packages. Similarly, adding new maps, items, souls and their effects is not difficult. Automation of state vectors also helps add opponents without writing large amounts of code responsible for communication with the server.

From the very beginning, the goal was to create a prototype, but because it was an MMORPG game, there were plans for a little more functionality that was not implemented due to lack of time.

The prototype lacked primarily a plot. Currently, after starting the game, you simply walk around aimlessly, discover the few rooms that exist and defeat your opponents. Of course, exploration-oriented MMORPG is not something bad, but some remnants of the plot, or at least the history of the rearranged world, would be useful. Setting specific tasks for the player would make the game more immersive and interesting.

Trade was also lacking. Developed economics is one of the most basic things of every MMORPG. Even with such a small amount of content as in my prototype, there would be, for example, the possibility to set the chances of getting items so that players had something to exchange. What would most encourage players to trade would be crafting. Exchanging rare items for materials that are rather time consuming to obtain is not uncommon. In my game, ultimately, there was to be the possibility of forging souls, i.e. combining some of their types, extracting from objects (there would be special objects with enchanted souls), or some broadly understood transmutation. Exchange of souls was also to be possible.

There is also another mechanics typical of MMORPG, i.e. the system of teams (*party*). In the prototype, players can only chat and no other interactions. Everyone fights alone, and even if several people beat the opponent, only the one who dealt the last blow gains experience. Monster loot is player-independent; it just falls out and whoever gets it first will have it. This is virtually unheard of online games. The team system also makes little sense if you don't need to play in a group. Therefore, there were plans for one very strong opponent (so-called *boss*), which would require a few people to fight.

From the technical side, there was mainly no security. There is virtually

no encryption, the bytes are packed and unpacked in raw form. In addition, passwords are not hashed. While it is difficult to cheat, hacking, account theft or server destabilization are not difficult for a person who knows how to hack. There is no package verification, so any incorrect data may cause an exception and shut down the server. Of course, apart from deliberate operation, this is impossible because TCP provides it. However, if it were to be a serious MMORPG, and not a prototype, then these things would necessarily have to appear in it.

An important element that should be in the game is handling broken connections. Currently, when the player disconnects, he is automatically logged out, which can lead to loss of progress. Connection interruption can occur for various reasons and the server is able to distinguish between deliberate disconnection and e.g. cutting off the internet, so the player should be able to log in for 5 minutes to keep progress when such a crisis occurs.

In addition to the elements mentioned above for implementations which did not have time, there are many other possibilities for project development. Apart from the mechanics typical of MMORPG, you can use the fact that the game is *metroidvania*. A popular phenomenon associated with this species is the so-called *speedrun*, which means getting through the game as quickly as possible. *Metroidvanias* give a lot of space here because of their non-linearity. If you can do *sequence breaking*, optimize the so-called *buildu*, i.e. all the equipment and skills of the character, or simply training their proficiency in moving the character, players will definitely use it. That's why you can create the right conditions for them and incorporate *speedruny* as game mechanics. It can exist e.g. as an arena with challenges or a special system based on instances. This is definitely a mechanic that you can consider introducing in the context of the further development of my project.

On the commercial side, there are also some options for monetization. In addition to typical strategies in games such as selling character skins or items that facilitate certain aspects of the game, you can take advantage of the possibility of losing progress. Many players would probably pay for the opportunity to save a rare item that they lost because they didn't manage to save the game before their character died. Apart from ethics (such practices can be considered as exploitation of players), the mechanics protecting against loss of progress do not give an unfair advantage over other (non-paying) players, of course, as long as the difficulty of the game is fair.

It is also worth considering optimization. The prototype is built on the TCP protocol, which is practically not used at all in this type of games, because it is known for generating considerable delays. Packages are sometimes unnecessarily retransmitted, and there is data redundancy associated with confirmations, etc. Switching to the UDP protocol is a must if you want to significantly develop the project. The server architecture itself is ready for this. All operations are abstract enough that replacing the protocol will not break compatibility. Instead, you need to introduce some mechanics related to the reliability of packet transmission. In the case of UDP, they may come in parts, in the wrong order or not at all. The client and server must be able to handle this. In addition,

the transmission of some packets can be avoided. For example, when creating inventory, the server sends the entire contents of the equipment again. This is unnecessary; the customer already knows what items he has, so he can act accordingly on this data. In the same way, after logging in the player gets all information about items, equipment and statistics. Instead, he could only receive it when he needed it. The downside of such a solution is a small delay, e.g. when opening the character menu. However, to reduce the amount of data transfer it is worth doing something like this. In the case of MMORPGs, every byte saved in the packet counts because thousands of such packets are sent. You can also save on the compression of transmitted data, which was also missing in my prototype.

Creating a *metroidvania* game combined with MMORPG is a difficult process. A game based on exploration must have a sufficiently large world. This requires, for MMORPG, quite frequent updates with new locations. They must be designed so that they harmonize well and are not repeatable. To avoid repetition, completely new mechanics will be needed from time to time. And while adding ordinary content is not particularly problematic (from the programming side), new mechanics require more work, due to the network nature of the game. So while the game *metroidvania* / MMORPG is right to exist in itself (assuming the ideal situation in which it has a rich and often expanded world), from the production side it would be very difficult to maintain.

The creation of my prototype showed that MMORPG is not as unattainable for small teams or even lonely game developers, as many people think. Probably it can be done more simply, there is ready software and solutions for creating mass network games. Ultimately, my protoyp tested one of many possibilities, and looking at the final effects it can be presumed that there is some potential to create a good online game for several hundred players.

List of Figures

References

[1] *pc genre awards best role-playing (rpg or mmorpg) - world of warcraft", (dostęp 14 sierpnia 2019). [online]. dostępny w internecie: Https:// web.archive.org/web/20080511162839/ 2004/pc/index12.html.*

[2] *maplestory pc | gryonline.pl", (dostęp 25 sierpnia 2019). [online]. dostępny w internecie: Https://www.gry-online.pl/gry/maplestory/z13ad9.*

[3] *list of massively multiplayer online role-playing games", (dostęp 21 sierpnia 2019). [online]. dostępny w internecie: Https://en.wikipedia.org/wiki/ list_ of_ massively_ multiplayer_ online_ role-playing_ games.*

[4] *metroidvania (concept)", (dostęp 21 sierpnia 2019). [online]. dostępny w internecie: Https://www.giantbomb.com/metroidvania/3015-2440/.*

[5] *castlevania: Aria of sorrow | gryonline.pl", (dostęp 25 sierpnia 2019). [online]. dostępny w internecie: Https://www.gry-online.pl/gry/castlevania-aria-of-sorrow/zf1734.*

[6] A. Manzur and G. Marques, *Sams teach yourself Godot engine game development in 24 hours*. Pearson, 2018.

[7] *top open source projects 2018: Vscode, react-native, tensorflow", (dostęp 14 sierpnia 2019). [online]. dostępny w internecie: Https://www.zdnet.com/ article/top-open-source-projects-2018-vscode-react-native-tensorflow/.*

[8] M. Mysior, *C# w praktyce*. Wydawnictwo MIKOM, 2005.

[9] K. Banker, *MongoDB in Action, Second Edition*. Manning Publications, 2016.

[10] *will nosql databases live up to their promise?", (dostęp 14 sierpnia 2019). [online]. dostępny w internecie: Http://www.leavcom.com/pdf/ nosql.pdf.*

[11] *why does godot use servers and rids?", (dostęp 14 sierpnia 2019). [online]. dostępny w internecie: Https://godotengine.org/article/why-does-godot-use-servers-and-rids.*

[12] *fast-paced multiplayer", (dostęp 14 sierpnia 2019). [online]. dostępny w internecie: Https://www.gabrielgambetta.com/client-server-game-architecture.html.*