COLLABORATION SPACE

TEAMPROJECT, WS22/23

Khaled Aboualnaga, Assylbek Bugybay, Yusuf Kütük, Fatma Sude Kütük, Amai Rivas, Daniel Dedoukh, Asian Hossain

Client: Philipp Graf | SUPERVISOR: PHILIPP GRAF

Table of Contents

1. Introduction

1.1 Project introduction

Prolonged quarantine restrictions around the world have led to an unprecedented demand for virtual event platforms, especially immersive ones that offer a complete “immersion” in computer worlds. Over the past few years, collaboration with others has increasingly been reduced to interacting online. Although virtual collaboration is an effective replacement for face-to-face meetings and conferences, after few years of using online meeting platforms, the need for “real” meetings is not felt so crucial.

Effective communication between teammates plays a significant role in the work of a company or organization. They contribute to the effective achievement of goals, more coordinated work within the company, and the creation of a friendly working environment.

So, the growing demand for platforms that provide a more complete immersion and involvement in virtual spaces is not surprising.

The tools that are used in this case are divided into two categories: 2.5D and 3D.

In our project we will use 2.5D concept. 2.5D platforms provide interfaces that mimic the physical environment (venue) and avatars (computer incarnation of the participant). They are a step ahead of meeting platforms used in companies today, but it's also more like an optimized and captivating wrapper for 2D content.

1.2 Mission

To create an application that the users can use to join a 2D space where they can move, chat, see other users, and interact with each other; in a completely online manner.

1.3 Vision

To bring the online world a step closer to the real world. Enabling our users to have the same experience they would have if they were physically in that space, while making it open source.

2. Requirements

2.1 Initial Requirements

Initially we had this set of initial requirements stated by the client:

- Users can join a joint two-dimensional world they can navigate by walking around.
- Administrators can design a world layout using pre-existing building blocks which in the beginning at least need to support floors and impassable walls/objects.
- Users which are in proximity to other users can interact through chat. (Not video or audio streaming.)
- Users can select an avatar.
- Pinboards available, where users can leave messages, or attach images or files.

From this set of requirements most of them where successfully integrated into the project.

The third project requirement, which says users can chat if they are in the proximity to each other, was implemented with a minor difference. In our project we have separated different chats according to their rooms. So, if you are in the same room, you can see the chat history and also send new messages. This is done due to code simplicity.

The second last user requirement about administers being able to design a workspace and also the availability of the pinboards was not implemented and left for future releases.

2.1 User Stories

In this section you will see the most important user stories which we did in our sprints:

1. A user needs to be able to enter a space
2. A user needs to be able to move while in a space
3. Camera should follow the user
4. User names should be displayed above avatars
5. A user can login
6. A user can register
7. Users can customize an avatar
8. A user joins a library default space
9. A user can open and close doors
10. Customize the color of the avatar's body part
11. Create UI for avatar customization
12. Create animated sprite sheets for avatar
13. Configure the high-level multiplayer API implementation and make it run on Google Could in VM
14. A user can configure his avatar and select username after registration
15. A user can chat with users over comfortable chat UI
16. A user needs to be able to join a space with the avatar configuration it had chosen previously
17. A user can experience joyful taste of the coffee from coffee machine
18. A user can exit a space
19. A user can see who is online when in a space
20. A user can experience a vending machine
21. Setup server for chat backend
22. Setup mongodb database for storing chat history
23. Configure chatUI client-side for communicating messages over WebSocket
24. Setup python WebSockets server for chat messaging system
25. As a user I should be able to communicate with different groups using chat system depending on meeting room
26. A user can change their password
27. A user can join a 2nd space
28. A user can join a 3rd space
29. A user needs to have access to the program as executable file
30. A user is able to pick which space to join with a preview of the space/ a small description (UI only)
31. Research about how to use emojis on the chat
32. A user needs to be able to join a space anytime
33. A user can't see the participants button when ESC menu is accessed
34. A user enjoys the UI of the project

35. A user can see who sent each message in the chat
36. A user can see updated list of participants
37. A user should not see server as participant
38. A user avatar should only be applied to him and see other users with their avatar
39. A user can select a female avatar
40. A user can see how many people are online when deciding which space to join UI
41. As a user can see the location of the participant on the participants list
42. A user needs to be able to see in which room they are in the popped out UI title
43. As a user I should be presented with chat history based on rooms
44. A user can change it's state to AFK
45. A user needs to be able to react with emojis (in speech balloons) without entering the chat
46. Fix that when pressed on exit meeting, it restart the game. Just use similar thing as unregister_participant to inform server about client leaving
47. Create an architecture diagram

Of course, this is not the full list of user stories. On appendix A you can find the backlog with all the user stories and tasks.

3. Most important features

3.1 Multiuser Support

The most important feature of the Collaboration Space application is multiuser support. The application allows multiple users to enter a space online.

3.2 Chat

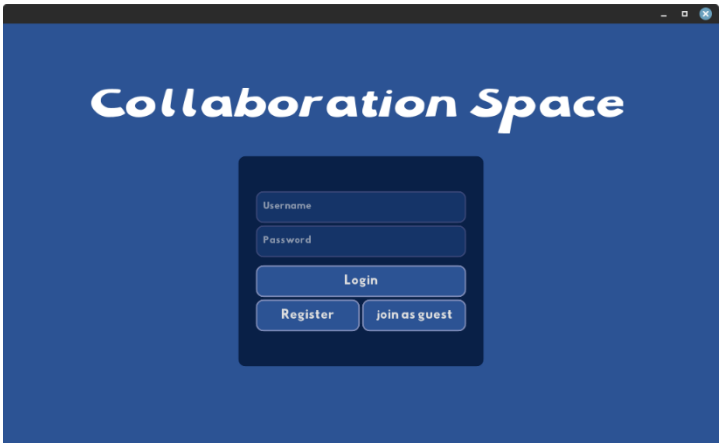
As it is mentioned in Vision, the target of the application is providing spaces for multiple users to contact via chat. Collaboration space provides users a chat functionality. Users can only interact with the users which are in the same place with them. Every space includes several rooms and areas. Each room and area have their own chat. For instance, if a user enters a room this user can only talk (chat) people who are inside this room. This provides users to have private chat areas.

3.3 Join any Time

Another main feature of our application is that a user can join a space any time. There are 3 spaces exist and users can join them any time without waiting. We achieve this feature by running the instance of the spaces on our Linux server (Ubuntu) all the time.

3.4 Registered User & Guest User

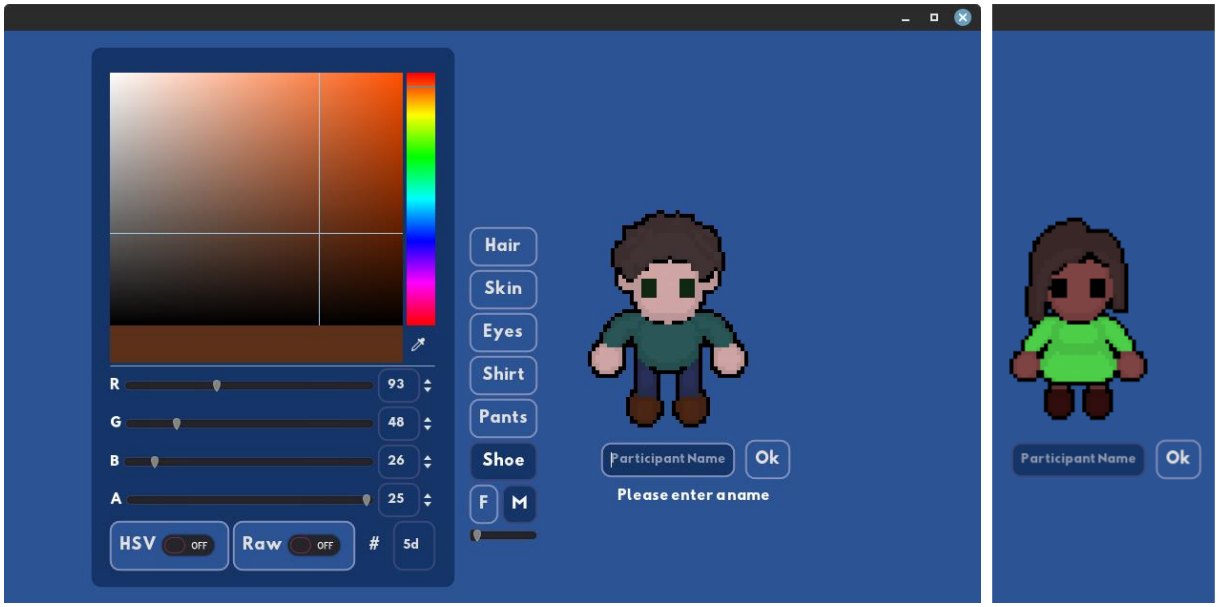
Our application provides two options to users to join a meeting. First option is registration. Registration requires an email address and a password. Registered users can modify their password as well. Registered users can customize their avatar and it is automatically saved. Therefore, when they log in, their custom avatar and username are automatically loaded.



Second option is joining as a guest user. It only requires a username. Guest users can also customize their avatar. However, their customization data is not stored in our database (Firestore) after they left the application.

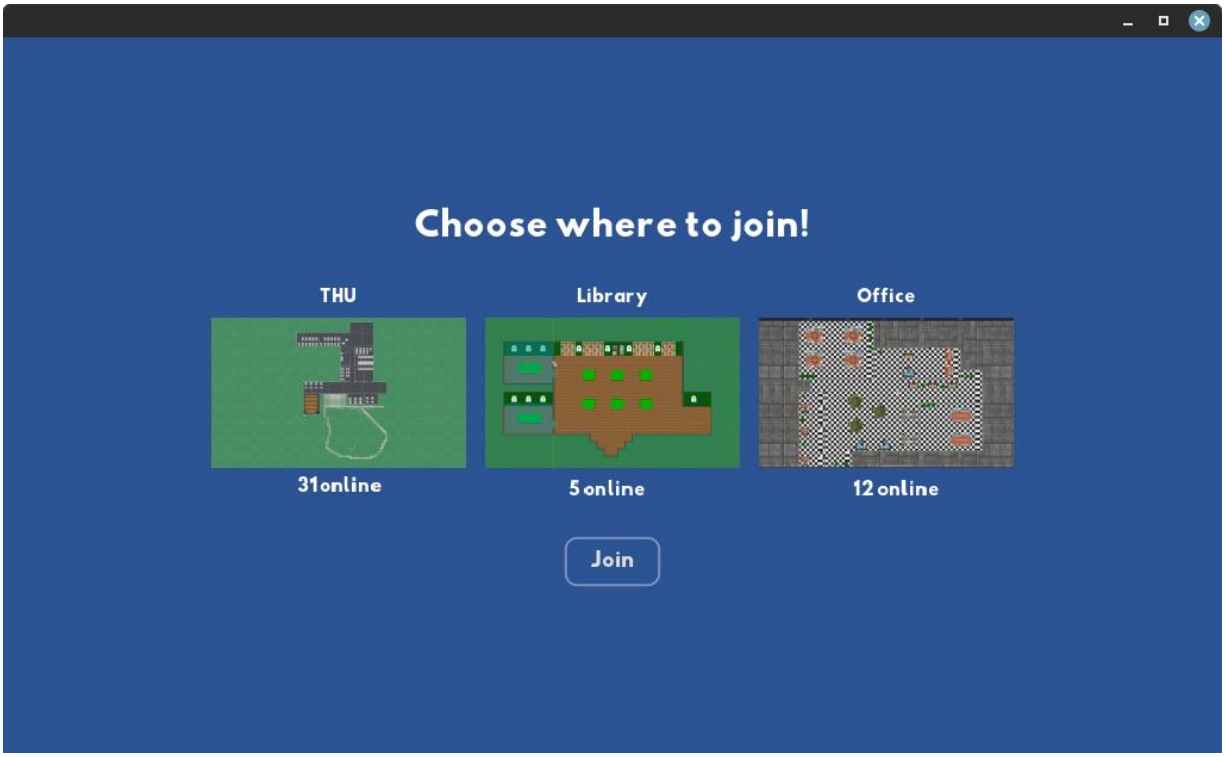
3.5 Avatar Customization

Every user, either registered or guest, can customize its avatar and user name. The customization feature provides female and male avatar options. Users can modify the color of avatar parts (hair, skin, eyes, shirts, pants and shoes) as their wish.



3.6 Multiple Meeting spaces

Our application provides users three different spaces which are a university (THU), a library and an office. Users can join these three spaces any time. In space selection screen, users can see how many participants are in every space.

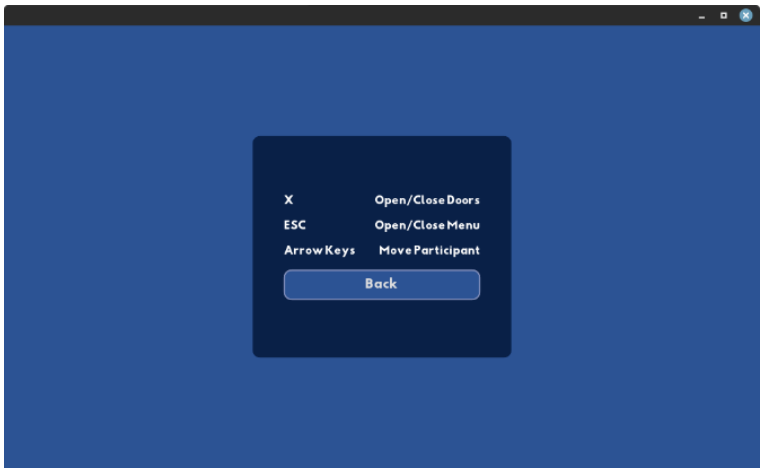


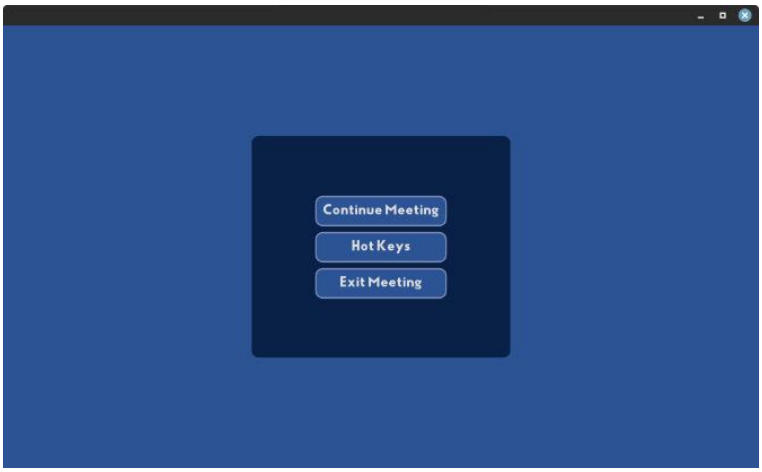
3.7 See List of Participants and their Locations

Since some of the space maps are quite big, we decided to provide a feature which shows the active participants list and the location of each participant in a space.

3.8 User Interface

Godot allows developers to create a custom global theme via its theme editor. We replaced the standard Godot UI theme with our custom theme. Our frontend theme created a color palette and selected a suitable font which makes Collaboration Space application stylish, consistent and user friendly. Moreover, to have better user experience, we keep the UI simple and clean.

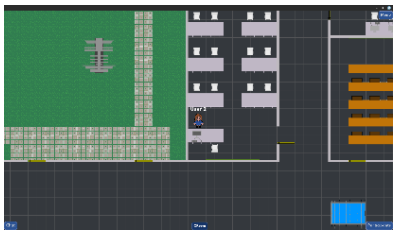




3.9 Graphics

Pixel art is selected as the graphics style of user avatars. The avatar has 2 animation states which are idle and walk. Also, these animations can be seen from 8 directions.

In addition, our application has 3 spaces with comprehensive design which means many objects including different tables, book shelves and sofas are designed to imitate the real world.



3.10 Gameplay

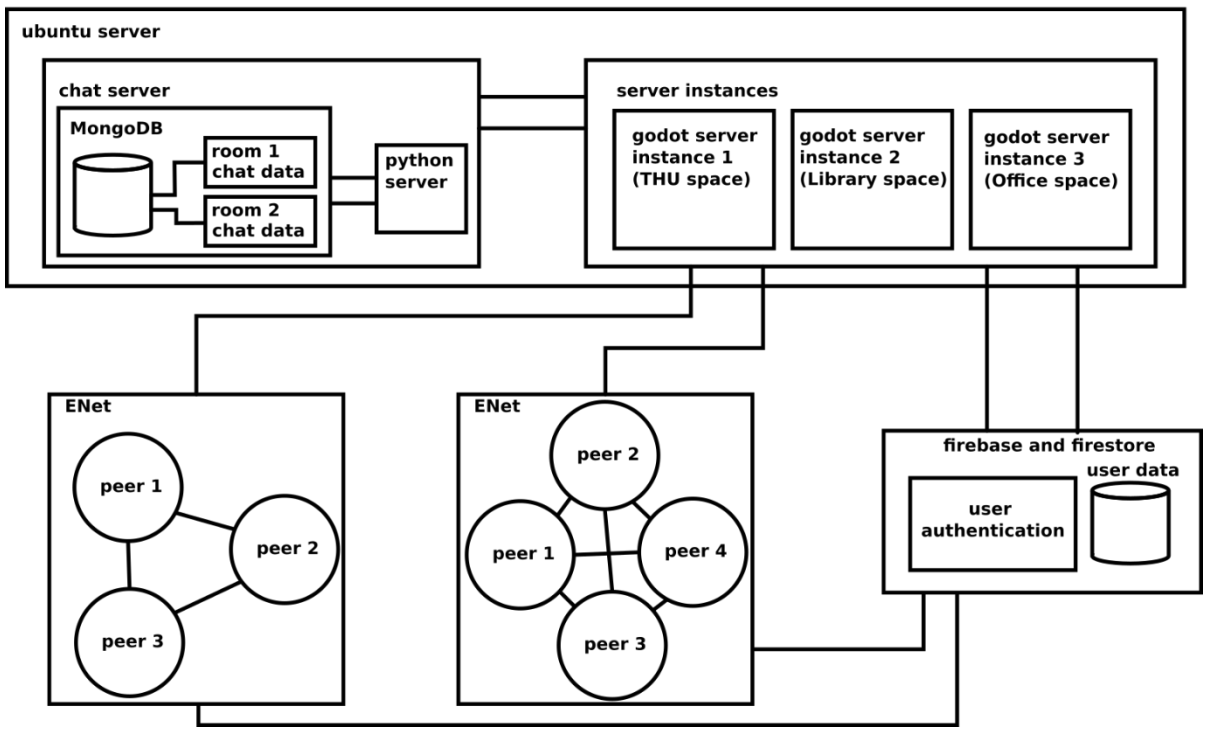
Godot provides stable collision system. Therefore, users can collide walls and other objects such as tables. Moreover, camera follows the user across the space.

3.11 Support for Multiple Platforms

Godot provides different export option for several platforms. Collaboration Space application can be run on Windows, MacOS and Linux operating systems.

4. Technical concepts

4.1 Architecture diagram



4.2 User management

4.2.1 Introduction

For proper user management Firebase platform was chosen due to its performance and simplicity. Firebase is Google-backed application which acts as a set of hosting services for any application type, offering cloud-hosted NoSQL database named “Firestore” and user management name “ Firebase Auth”. There are many more useful Firebase tools, but they are not mentioned, as they are not in the scope of this project.

4.2.2 Godot HTTPRequest

Before moving to how Firebase Auth and Firestore DB work, we should understand Godot’s component “HTTPRequest”. This component is used for sending and receiving HTTP requests.

We must instantiate the HTTPRequest instance and connect the signal “request_completed” to the method in our script. This method should have *result*, *response_code* , *headers* and *body* as input parameters. When HTTP response is received, signal “request_completed” is triggered and our method attached to the signal is called with provided input parameters from HTTPRequest instance. To read the content of the *body*, we should parse JSON object as Dictionary.

4.2.3 Firebase Authentication

A user can query the Firebase Auth backend through a REST API, which uses HTTP POST Requests in JSON format. For each type of request there is a specific endpoint URL with WEB API KEY inside it. The requests are sent and received using HTTPRequest Godot component.

In our project Firebase Auth is used to perform the following operations:

- **Sign up with email / password**

  The request is sent containing email and password. As a response we get UID, session token and refresh token, which are stored in global variable of type Dictionary.

- **Sign in with email / password or anonymously**

  Login is similar to register request, but another endpoint URL is used. With login anonymously option we do not provide any credentials

- **Refresh of session token**

  Since session tokens expires after 1 hour, they have to be refreshed for the users and server instances. Request is sent containing user refresh token. As response request we get new session and refresh tokens which are updated in the global variable.

- **Delete account**

  If user uses authentication option “ join as a guest “, then his account has to be deleted after he logs out from the game in order to not overload our DB with unused guest accounts. In order to delete the account, we send a request containing user session token.

- **Change password**

  The request is sent containing session token and new password. On successful execution, the password is changed in Firebase without logging out the user.

4.2.4 Firestore Database

A user can query Firestore Database through a REST API after he was authenticated. Same as with Firestore Auth, the requests are sent and received via HTTPRequest Godot component. For each request the header is needed, which contains Content-Type as JSON and user session token to check if the user is authorized to perform requests, path to the document in Firestore and different endpoints URLs depending on the action.

In our project Firestore Database is used to perform the following operations. User data contains username with the avatar configurations. Server data contains current online of each space.

- **Save data**

  After user's registration his username and avatar configurations are sent to Firestore DB as JSON document.

- **Update data**

  This request is very similar to saving the data, only different URL endpoint is used. It is used for saving user data and server data.

- **Get data**

  In order to get the document we just need to provide the endpoint URL alongside default entries.

4.2.5 Delete data

When anonymous user exits the game, not only his Firebase account has to be deleted, but also Firestore Database entry. The request is very simply to get data, but endpoint URL is different.

4.3 Multiplayer

4.3.1 Introduction

Godot High-Level Multiplayer API was chosen as a solution for multiplayer implementation for the project. Godot always supported standard low-level networking via UDP, TCP and some higher level protocols such as SSL and HTTP. These protocols are flexible and can be used for almost anything. However, using them to synchronize application state manually can be a large amount of work. Therefore, Godot's high-level networking API sacrifices some of the fine-grained control of low-level networking. In general, TCP and UDP protocols are re-build in a way that important parts as kept, such as optional reliability and packet order, but unwanted parts are removed, such as congestion/traffic control features, Nagle's algorithm and etc. [4]

4.3.2 Mid-level abstraction

Before going into how an application across the network is synchronized, we should understand the base of network API.

- **NetworkedMultiplayerENet** [5]

  ENet's purpose is to provide a relatively thin, simple and robust network communication layer on top of UDP. This object extends from PacketPeer, so it inherits all the useful methods related to data manipulation. On top of that, it adds methods to set a peer, transfer mode, etc. It also includes signals that will let you know when peers connect or disconnect.

- **PacketPeer** [6]

  PacketPeer is an abstraction and base class for packet-based protocols (such as UDP). It provides an API for sending and receiving packets both as raw data or variables. This

makes it easy to transfer data over a protocol, without having to encode data as low-level bytes or having to worry about network ordering.

4.3.3 Network initialization and communication

SceneTree is the object which controls networking and everything tree-related in Godot. To initialize high-level networking, the SceneTree must be provided with NetworkedMultiplayerENet object. [7]

- **Initialization of server example**
  var peer = NetworkedMultiplayerENet.new()
  peer.create_server(DEFAULT_PORT, MAX_PARTICIPANT)
  get_tree().set_network_peer(peer)

- **Initialization of client example**
  var peer = NetworkedMultiplayerENet.new()
  peer.create_client(ip, DEFAULT_PORT)
  get_tree().set_network_peer(peer)

On every peer connected to the server when a new peer connects or disconnects the below mentioned signals are called. Methods can be attached to those signals, so that specific actions are performed when they are received.

- **Server and Clients**

network_peer_connected(int id) and *network_peer_disconnected(int id)* signals inform in-app users and server that we have to register/deregister new user, both on server and client side.

- **Clients only**

Signals *connected_to_server* and *connected_failed* inform only the current user who is trying to join if he was connected or not. Another important signal is coming from the server in case it shuts down is *server_disconnected*

In order to communicate between peers, we use RPC (remote procedure calls). For each RPC call the method name is provided, which should exist on client side with keyword *remote*. Additionally, the sender can provide arguments. There are 3 methods that we should know about.

- rpc("function_name", <optional_args>). This function sends RPC call to all connected peers besides the sender.
- rpc_id(<peer_id>,"function_name", <optional_args>). This functions send RPC call to a specific peer.
- SceneTree.get_rpc_sender_id(). This function is used to check who is the RPC sender.

4.3.4 Application initialization

For each player the application has to be initialized, which involves initialization of the space and participants. Initialization of the space is simple, as it is loaded on the client side. It happens with a few commands.

var meeting_area = load(<scene path>).instance()
get_tree().get_root().add_child(meeting_area)

Participants initialization, on the other side, is more complicated, as the new user should know which users are already in the application and configure their user data. Dictionary with all participants and their configurations is stored on the server side, which sends RPC call to new user

providing the dictionary. Upon receiving, a new user should attach the participant ID to the network peer in his SceneTree and populate the avatar with relevant information. Here is the code sample with comments for the new user adding other participants. Initialization of the user himself is happening similar to this method, but *participant.set_participant_camera(true).*

```
remote func new_user_adds_ingame_participants(participants_list_from_server : Array) -> void:
    # Load the participant scene
    var participant_scene = load("res://Participant.tscn")
    # Iterate through all the participants
    for p in participants_list_from_server:
        var spawn_point = p["spawn_point"]
        # Instansiate the participant scene
        var participant = participant_scene.instance()
        # Attach the network peer ID to the participants
        participant.set_name(str(p["NetworkID"]))
        # Set spawn locations for the participants
        participant.position = Vector2(spawn_point)
        # Each participant has control over his node.
        participant.set_network_master(p["NetworkID"])
        # Camera is set to false, as we are not the user
        participant.set_participant_camera(false)
        # set data (name and colors) to participant
        participant.set_data(p)
        # Add participants to the list which is needed to see who's online
        participants[p["NetworkID"]] = p
        # Adds participant to participant list in Default scene
    get_tree().get_root().get_node("Default").get_node("Participants").add_child(participant)
```

4.3.5 Application Synchronization

In our application we need to keep participants and doors in sync. In order to do the synchronization Godot provides easy to use functions: [8]

- rset("variable", value)
- rset_id(<peer_id>, "variable", value)
- rset_unreliable("variable", value)
- rset_unreliable_id(<peer_id>, "variable", value)

rset “Remotely changes a property's value on other peers (and locally).”

Lets now move on to how specific components are being synchronized.

- **Participant**

  Each participant in our application has the following main properties: *velocity, current_animation, current_location.*

  Because every participant has to be in the correct position, have correct animation and see the correct current location, these three properties have to be synchronized.

  - **Participant Position Synchronization:**

```
if is_network_master(): # if the local system is the master of the node
    # jump to get_input() which updates the velocity variable according to the key
presses
```

```
            get_input()
            # set puppet properties of the puppets of this participant on other clients
            rset("puppet_motion", velocity)
            rset("puppet_pos", position)
    else: # if the instance is not the master of the node, it is a puppet
            # apply the puppet properties with the values, which are
            # coming from rset call from the actual master of the puppet
            velocity = puppet_motion
            position = puppet_pos
            # simulate the physics
            velocity = move_and_slide(velocity)
    …
```

The idea of the synchronization of current_animation and current_location is similar as position synchronization.

- **Participant Animation Synchronization:**

```
func decide_animation() -> void:
        if is_network_master():
                if direction.x == 0 and direction.y > 0:
                ...
                current_animation = "idle_s"
        elif direction.x > 0 and direction.y > 0:
                ...
                rset("puppet_current_animation", current_animation)
        else:
                current_animation = puppet_current_animation
```

- **Door:**

Each door has a property called open which indicates whether the door is open or not. And according to this property, correct rotation value is set.

Doors also have a property called interaction_active which is set to true if a participant's interaction area collides with the door's interaction area.

The door script has a remotesync function called interact() for synchronization of the doors. During the _process() if the user presses 'X' key this function is called for all peers using rpc().

- **Emojis:**

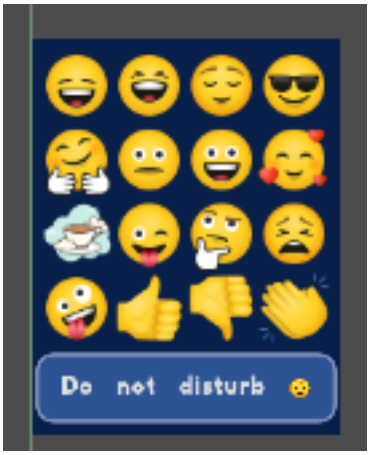
Emojis UI was created separately from Participants scene, we have added Emojis scene and a childe node. Each button is made from TextureButton so they can contain pictures instead of text. When the buttons are pressed the signal is sent to Participants.gd. It is then received in get_emoji method

```
func _get_emoji(which) -> void:
    if which== "":
            which = "DefaultEmoji"
    if which != "Seventeen" and which != "":
            $Emojis.get_node(which).show() # Shows emoji for network master Avatar
            $Timer.start()
    ....
    if is_network_master():
            if which != "Seventeen":
```

current_emoji = which

….

In this method depending on if it is AFK emoji or not several “filtration” steps are done. “DefaultEmoji” is a placeholder for the invisible emoji. If this step was done, then the Timer if started for normal emojis(except AFK emoji). In order to synchronize the appearance at each frame we call _display_emoji() method on _process method. _display_emoji() method shows or hides the emojis for both network master and non network masters.



4.4 Chat

4.4.1 Introduction

Our application was intended to enable users to interact with each other through Gamification. One core feature, of course, is chatting with other users and being able to exchange messages.

4.4.2 Basic approach

One of the metrics for smooth communication is performance and non-latency. This is achieved by one of the famous protocols around, which is WebSocket. We depend on WebSocket to ensure two-way communication between the user and the server and to eliminate the continuous polling that wastes resources. This is achieved by pushing notifications from the server in case of any new messages to any of the interested users. [13][14]

4.4.3 User libraries

In our application, we depend on python as the main programming language. One of the most performant and famous libraries used for WebSocket protocol is the websockets library, which works on the asyncio library. The websockets library supports asynchronous operations out of the box. This way, the server is optimized to process requests in the most efficient way. [15][16]

As will follow, the Motor library for handling the database from the python server is used. Motor is also an asynchronous library by default, built on top of asyncio, so no CPU time or resources are wasted. [17][18]

4.4.4 Chat database

The main goal from the beginning is to have a database enabling users to have a chat history and not to lose any messages exchanged between users. NoSQL, along with MongoDB, is selected as the main database for storing and retrieving messages for the chat history. NoSQL or MongoDB is used for its easiness and compatibility in integrating into most projects. [19][20]

4.4.5 Communication between client and server

As already known, the WebSocket protocol barely provides messaging mechanism and does not impose restrictions on message format. So, JSON format is used as a WebSocket subprotocol to exchange messages between client and server. There are four different kinds of messages a client and server can communicate; They are shown below.

```
    # Register user information when they first log in

{
  "type":"register",
  "user_id":"id",
  "name":"user_name"
}
```

```
    # Assign users to which room they are in currently

{
  "type":"assign_room",
  "user_id":"id",
  "room":"room",
  "state":"state"
}
```

```
    # Send messages from chat(client) to server

{
  "type":"message",
  "msg":"msg"
}
```

```
    # Retrieve message when the user joins a room

{
  "type":"retrieve_message",
  "result":"array_of_results"
}
```

4.4.6 Saving messages in the database

Messages of the chat are saved in the same database, with every Collection corresponding to one room and every message corresponding to a document. This way, clear differentiation between messages can be easily made while maintaining a clear architecture. For example, the following message was taken from the database as an example:

```
{
  _id: ObjectId("636ffa8bbb6a45111b6d148e"),
  sender_id: 'xiPxeauF5Mb2juXmEnsrBv0PFOp1',
  sender_name: 'User1',
  recipients_id: [ 'V7BL5G7R7rS2MxuSrNCXo635T7w2' ],
  message: 'Hello',
  date: ISODate("2022-11-12T19:56:59.885Z")
}
```

"recipients_id" holds the array of recepients that were in the room at the time the message was sent, allowing to easily retrieve the messages that were only communicated with the user. Date allows tracking and retrieving messages according to a specific time frame.

4.4.7 Chat history

As mentioned above, when the user joins a new room, the client communicates the event with the server, and hereby the server can retrieve the message based on the user ID and the room he

has joined. When the client receives the message, it is responsible for parsing the retrieved array of messages and updating the user's chat. [21]

4.4.8 Identifying the room of the user

We depend on the Godot signaling system to identify which room the user belongs to. This mechanism allows sending a signal to all subscribers to notify them of an event. Along with that, Area2D AND CollisionShape2D are used to detect when the user enters or exits a certain room. [22][23][24]

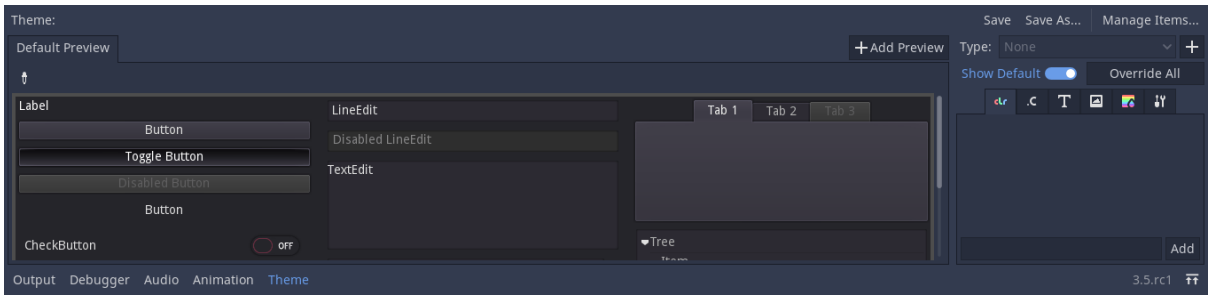
When the user joins or departs a room, the client automatically sends a message to notify the server, along with useful information. The server then assigns the user to a suitable room. The server has for every room a Python Dictionary where the user ID is assigned to a room.

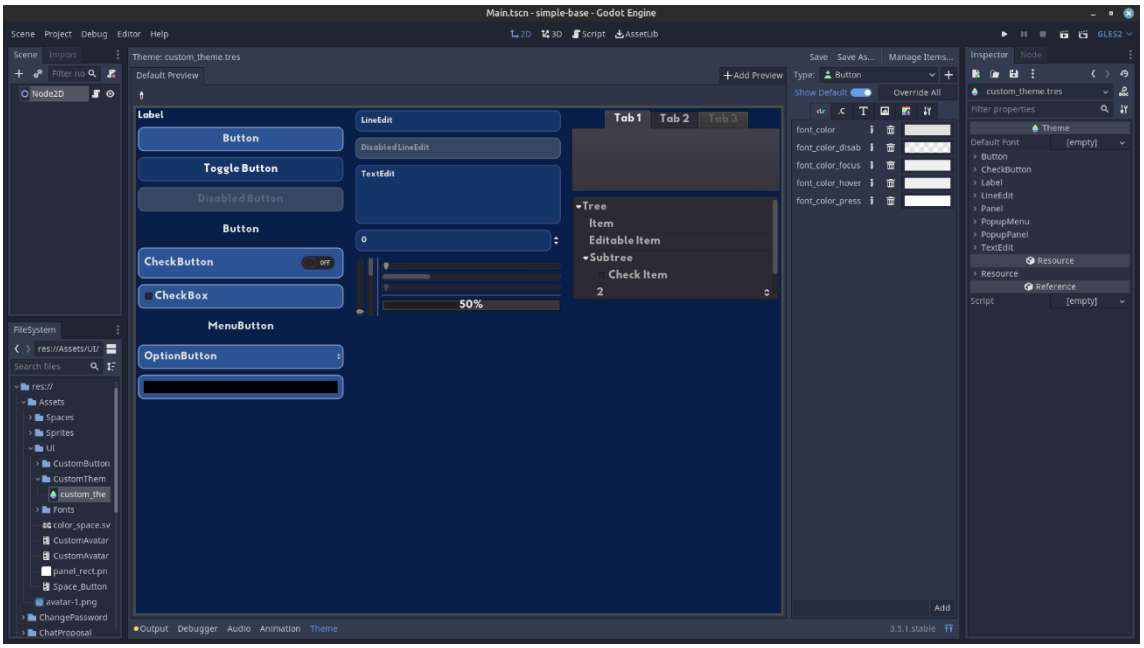
This is useful, for example, so that the server knows the state of users against rooms and allows communication only to occur between users in the same room.

4.5 User interface

4.5.1 Custom theme

Godot has a theme editor which provides developers to create their own custom global theme. In order to create a stylish, consistent and user-friendly UI, Godot's global theme functionality is used for this project.



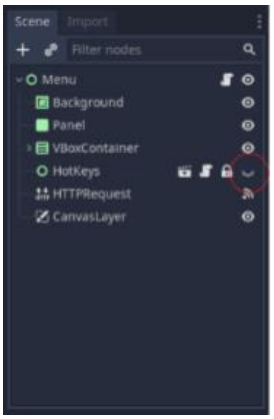


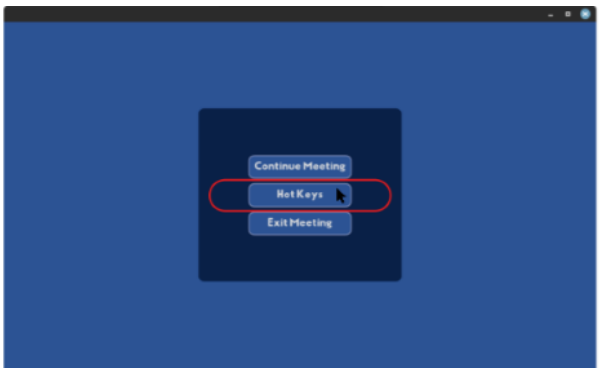
To create a global theme, firstly a color palette is created, and a font is selected. Sztylet Font (bold style) is used as UI font.[25][26]

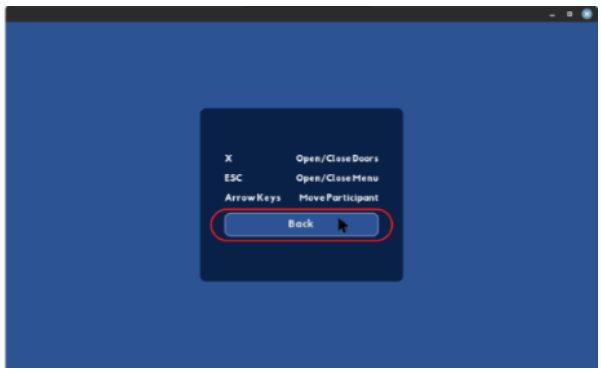
In Godot, all user interface nodes inherit from Control which has anchors and margins to adapt its position and size relative to its parent.[27]

4.5.2 Scene Transition

In the application, the transition between different scenes is designed by two ways. First way is changing from one scene to another scene. This is done by Godot's SceneTree.change_scene() function.[28] In some cases, the application does not require to a scene change. Therefore, instead of changing the scene, Godot's hide() and show() functions are used to show or hide the scenes which are already existed inside a scene.







Opening and closing the menu is one of the examples of hiding and showing the scenes. The figure above shows the scene tree for Menu scene. The Menu scene already contains the HotKeys scene which is hidden by default. Therefore, when the HotKeys button is clicked, the HotKeys scene is shown and when the "Back" button in HotKeys scene is clicked, the Hotkeys scene is hidden again.

4.6 Space design

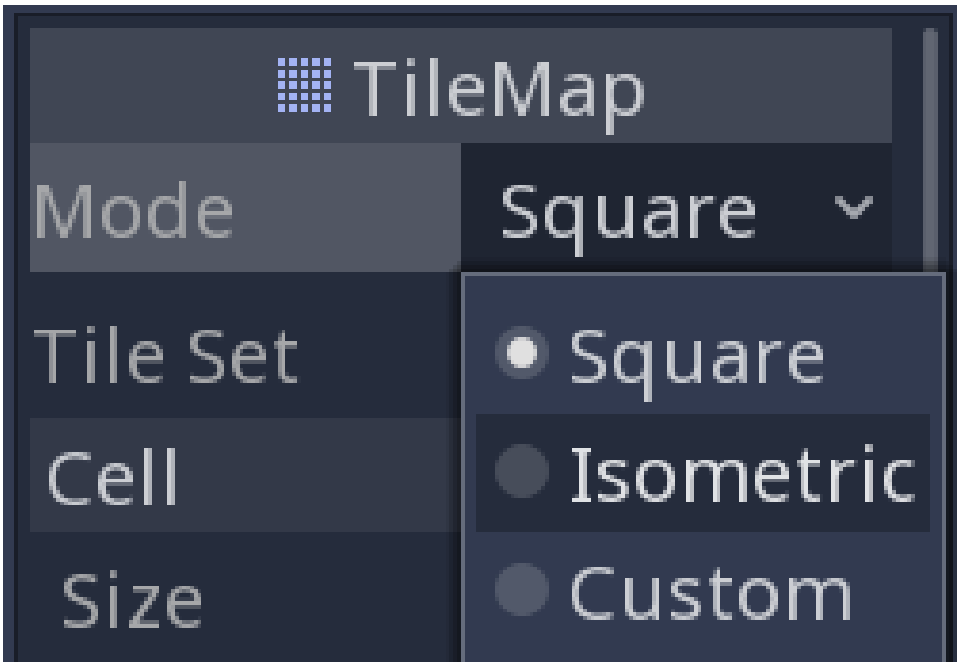
4.6.1 Introduction – using TileMaps

A tilemap is a grid of tiles used to create a game's layout. There are several benefits to using TileMap nodes to design your levels. First, they make it possible to draw the layout by "painting" the tiles onto a grid, which is much faster than placing individual Sprite nodes one by one. Second, they allow for much larger levels because they are optimized for drawing large numbers of tiles. Finally, you can add collision, occlusion, and navigation shapes to tiles, adding additional functionality to the TileMap.

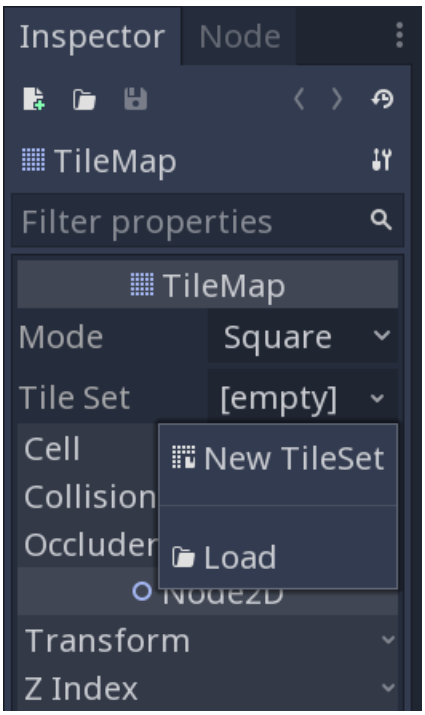
4.6.2 TileMap Node

Add a new TileMap node to the scene. By default, a TileMap uses a square grid of tiles. You can also use a perspective-based "Isometric" mode or define your own custom tile shape.

Under the "Cell" section in the Inspector are many properties you can adjust to customize your tilemap's behavior.



4.6.3 Creating a TileSet

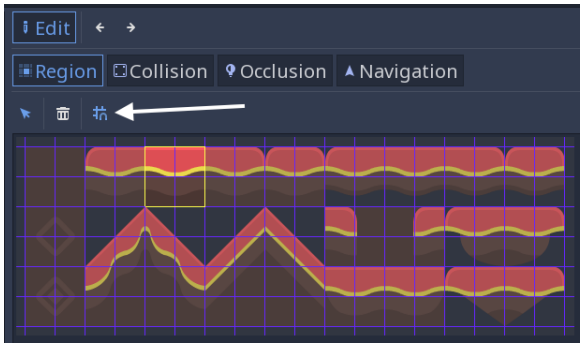


Once you've configured your tilemap, it's time to add a TileSet. A TileSet is a Resource that contains the data about your tiles - their textures, collision shapes, and other properties. When the game runs, the TileMap combines the individual tiles into a single object. that contains the data about your tiles - their textures, collision shapes, and other properties. When the game runs, the TileMap combines the individual tiles into a single object.

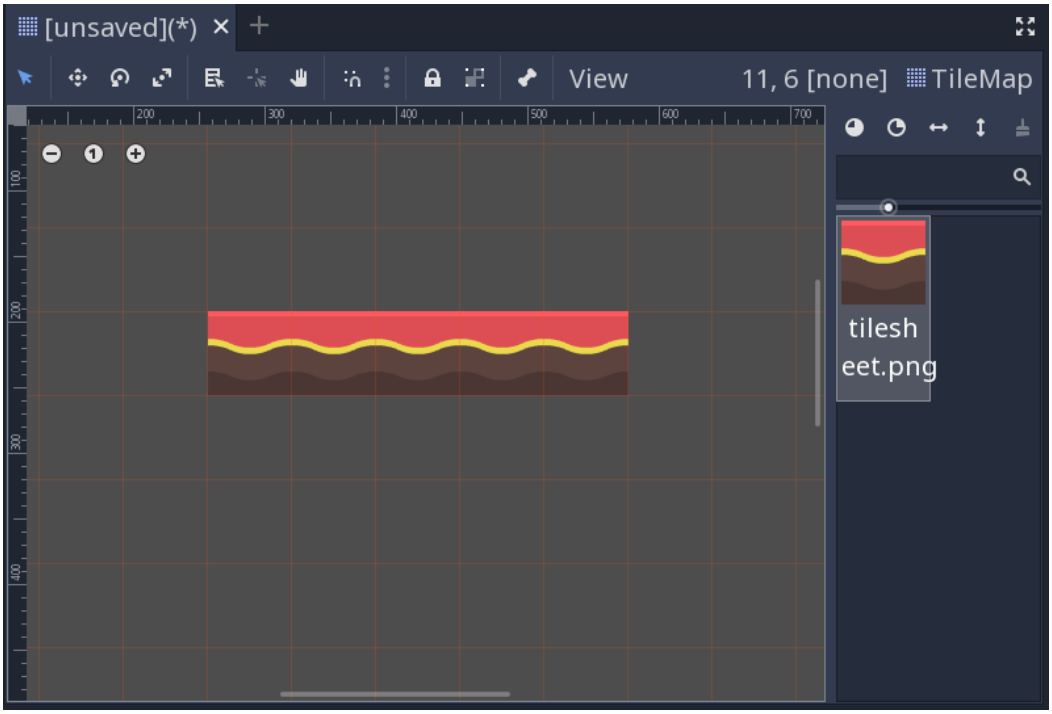
To add a new TileSet, click on the "Tile Set" property and select "New TileSet". First, you need to add the texture(s) that you'll use for the tiles. Click the "Add Texture(s) to TileSet" button and select the tilesheet.png image. Next, click "New Single Tile" and drag in the image to select the tile you want. Click the "Enable Snap" button to make it easier to select the entire tile. A yellow rectangle appears around the selected tile.

If you're making a map that needs collisions - walls, floor, or other obstacles, for example - then you'll need to add collision shapes to any tiles that you want to be considered "solid".

Click "TileSet" at the bottom of the editor window to return to the tileset tool. Click the tile you previously defined (outlined in yellow). Select the "Collision" tab and click the "Create a new rectangle" button. Make sure you still have grid snap enabled, then click and drag in the tile.



Click on the TileMap in the scene tree, and you'll see that the newly created tile now appears on the right side. Click in the viewport and you can place tiles. Right-click to remove them.



4.6.4 Autotiles

Autotiles allow you to define a group of tiles, then add rules to control which tile gets used for drawing based on the content of adjacent cells.

Click "New Autotile" and drag to select the tiles you wish to use. You can add collisions, occlusion, navigation shapes, tile priorties, and select an icon tile in the same manner as for atlas tiles.

Tile selection is controlled by bitmasks. Bitmasks can be added by clicking "Bitmask", then clicking parts of the tiles to add or remove bits in the mask. Left-clicking an area of the tile adds a bit, right-click removes "off", and shift-left-click sets an "ignore" bit.

Whenever Godot updates a cell using an autotile, it first creates a pattern based on which adjacent cells are already set. Then, it searches the autotile for a single tile with a bitmask matching the created pattern. If no matching bitmask is found, the "icon" tile will be used instead. If more than one matching bitmask is found, one of them will be selected randomly, using the tile priorities.

5. Project organization

5.1 Project organization - general

To achieve this ambitious project, we decided to organize our team as follows:

- Process model: Scrum (fully online)
  - One-week Sprints
  - From Thursday to Thursday
  - Daily meetings every day at 12:00
  - Client meetings on Thursdays at 12:30
  - Sprint retrospective meeting on Thursdays at 14:00
  - Sprint planning on Thursdays at 14:15
  - Coaches with Prof. Balser variable

- The main roles we as a team were:
  - Scrum master/Procut owner:
    - Amai Rivas
  - Front end:
    - Fatma Sude Kütük
    - Enver Yusuf Kütük
  - Back end:
    - Asian Hossain
    - Assylbek Bugybay
    - Daniel Dedoukh
    - Khaled Aboualnaga

Although this might seem very static, most people worked simultaneously on both front end and back end. For a more detailed view on each person's contributions see section 6 of the documentation called "Individual contribution".

- The tools used for the project organization were the following:
  - For the scrum organization we used Jira
  - Client meetings and coaches from Prof. Balser were done on Webex
  - Team meetings (Daily, Retrospective, Development, etc.) were done on Discord
  - Main team written communication was done through WhatsApp
  - For version control we used GitHub
  - For documentation we used OneDrive

See the Appendix B: Project organization, where right at the beginning we tried to write down how the project would be organized, and at the end it didn't see very different.

5.2 Milestones

In the Appendix B: project organization, we also tried to summarize the project milestones per week. Since it was very early in the development of the project, we hadn't even started programming anything yet. That means we took the first milestones document as a sketch or a template, more than our hard planning.

That meant that even at the first week we started developing or researching things planned for the next few weeks. But looking back we think it was a good idea, because sometimes we would have a misinterpretation of the complexity of some tasks, which ment that starting sooner would give us an advantage.

Later, around the last 3 to 4 weeks of the project we created a second milestones document (see appendix C: Milestones 2.0), where we tried to plan what should be completed each week, until the last week. This second time we did a much better planning, having the experience of managing the previous sprints, and knowing what the team was capable of for each sprint.

We think we did a good job on planning the last few weeks of the project, and we followed the document closely, with minor exceptions.

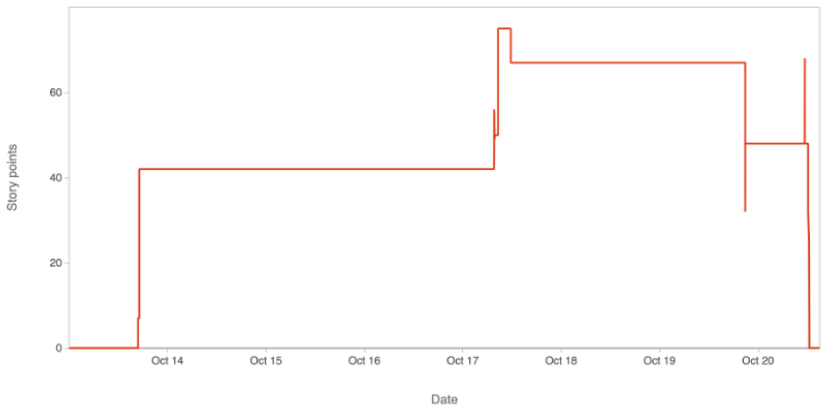
5.3 Jira reports

Jira has many advantages, and one of them is the tab “Reports” it provides us, to analyze our performance thorough the project. Thank you to the help of Prof. Balser, after learning how to properly use the scrum model, he also taught us how to use Jira, and finally, how to analyze some of the reports.

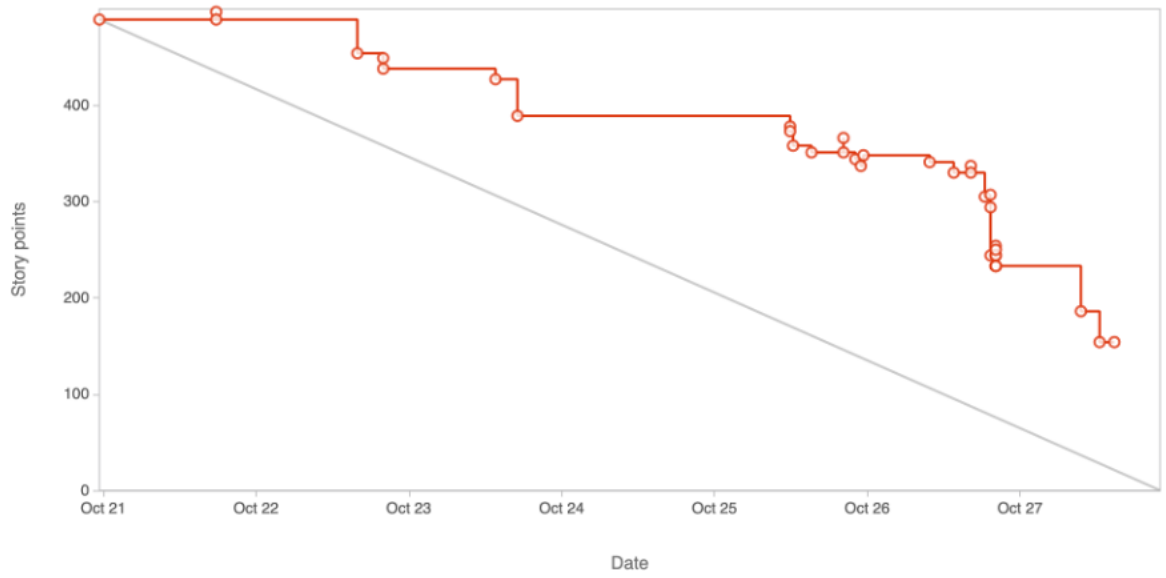
We would like to analyze those reports now, at the end of the project, since both have helped us in the middle of the endeavor to improve our scrum abilities and see where we were failing to follow the process model.

The first report we want to analyze is two burndown charts. The first one is from the beginning of the project, when we still were not using Jira the way we were supposed to, but also not following scrum model how we should. You can see that it’s called “burndown chart” because it should start high and end low, which would mean you start with a lot of open story points that you need to finish, and over the course of the sprint they are reduced as you complete the tasks.

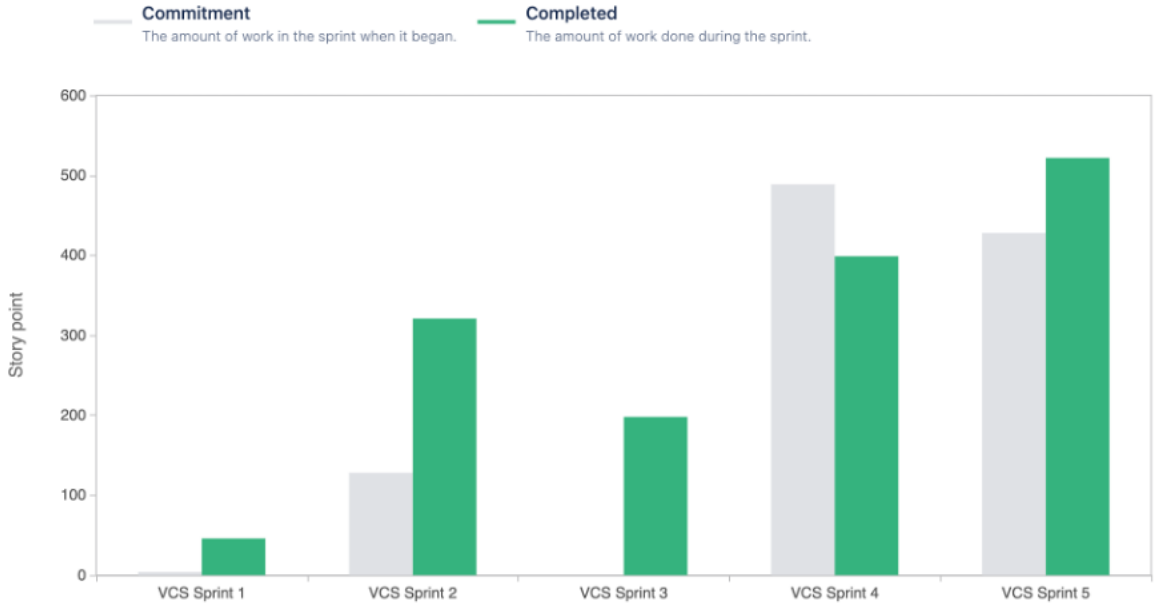
But as can be seen in the image, the chart looks very messy, with no story points at the beginning, and some getting added in the middle and even at the end, with them also getting finished right after.



Now, after some coaching from Prof. Balser, and knowing what we were supposed to do; so we can see on the second image, that we have a nicely looking burndown chart with a lot of story points at the beginning. Story points that need to be done, with people marking them as complete over the course of the sprint, represented by the down trending graph. With some story points left open, which we didn’t manage to complete on the sprint. This indicates a good sprint planning, and a good use of the tool Jira and the scrum process model.



Secondly, we would like to analyze the velocity report of our project, which shows you mainly two things: how many story points were committed at the beginning of the sprint, and how many were completed. If we add story points in the middle of the sprint, they won't count as "committed", since it will be after the start of the sprint, but if we manage to complete them, then they will be counted as "completed".



As you can see in the image, the first sprint we didn't really know what we were doing, since almost no story points were committed at the beginning of the sprint, but at least we marked some tasks/user stories as completed at the end.

On sprint two we were starting to learn how to use Jira and adding user stories/tasks, and their respective story points, so we can see that at the beginning some story points were committed but towards the middle a lot more were added and completed.

In sprint 3 we had user stories and tasks assigned and planned for the whole sprint, but we didn't add any story point on the same day, and that's why it shows that no story points were committed on that sprint, although a lot were completed.

On sprint four, thanks to coaches from Prof. Balser again, we learned what the velocity chart (this chart in particular) works, and how we are supposed to directly add story points to the user stories/tasks on the sprint planning, and it shows that we did, since we can see almost 500 story points committed at the beginning of the project. We can also appreciate that on this sprint we didn't manage to finish all our tasks, but that was not a problem, since we simply moved them to the next sprint.

Finally in the fifth sprint we can see again a velocity chart looking right with the same amount of committed story points vs the completed story points. We have a bit more story points because we had to add tasks/user stories in the middle of the sprint, when we would find some new feature missing, or some bug corrections.

Overall, regarding project organization, we can very easily see the improvement of our understanding not only about how to use the tool Jira, but also about how we are supposed to use it, following the scrum process model.

6. Individual contributions

6.1 Daniel Dedoukh

- **User management**

This part was fully implemented by Daniel. Detailed description of the implementation can be found in user management documentation. Here are the main tasks he has performed.

  - Learning about Firebase, Firestore and HTTPRequest component in Godot
  - Authentication with e-mail and password
  - Anonymous authentication
  - Change of password
  - Saving/fetching user data
  - Safe logout from the game / delete account
  - Change of user data
  - Online of spaces when user is selecting which space to join
  - Refresh of session token for users and server instances

- **Multiplayer**

Daniel and Yusuf have implemented multiplayer for our project using Godot high-level multiplayer API. Detailed description of the implementation can be seen in multiplayer documentation. Here are the main tasks Daniel has performed.

  - Detailed analysis of high-level multiplayer API documentation
  - Creation of multiple demos to test behaviour of different multiplayer components
  - Moving local host implementation to server
  - Reconstruction of host implementation into joining the game any time
  - Safe disconnection from network peer when user is leaving

- **Documentation and presentation**
  - Multiplayer documentation (expect for synchronization part)
  - Multiplayer presentation
  - User management documentation

- **Another contributions**
  - Google Cloud VM as server configuration and maintenance
  - Support of another participants with merging, testing and implementation
  - Support with project organization

6.2 Fatma Sude Kütük

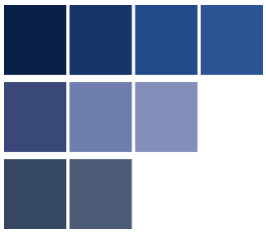
User Interface

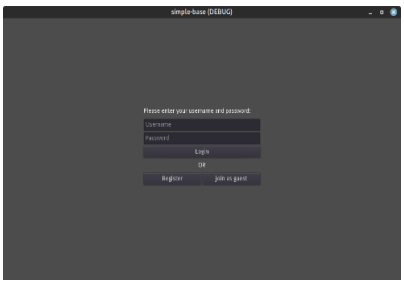
The UI is implemented by Fatma Sude Kütük. She designed most of the scenes which user interacts, including the entry, customization, space selection.

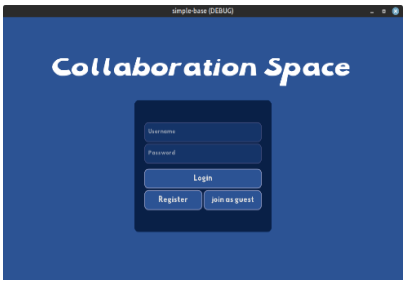
Custom Theme Creation

Godot provides standard themed UI elements such as button, input field. However, it is not stylish. Therefore, she used Godot's theme editor to create a custom theme which provides more stylish and user-friendly UI elements.

She created a color palette and selected a font to create a custom theme. **Sztylet Font** (bold style) is the selected font for UI elements.[9][10] After creating the custom theme, she replaced the existing default theme with the newly created one so that by default Godot uses the custom theme instead of the standard one.







Avatar Customization

She is fully implemented the avatar customization page. For the customization logic, she worked with Yusuf Kütük.

Chat UI

She created the Chat UI. A user can enter its message and view the messages which he/she received.

Menu UI & Hotkeys

She implemented the Menu UI which includes following options: continue meeting, hotkeys and exit meeting. Hotkeys scene shows the game controls.

Space Selection

Users can select a space to join. After joining, if they want to go back to the space selection scene, they can go back by clicking "back" button.

Documentation & Presentation

Documentation: Most Important features, UI. Presentation: Demo, Reached Goals

Other contributions

- Disable participant's movement when the chat is focused
- Preventing users from joining the meeting without name
- **Location Label UI and logic:** it shows every user's location on the bottom of the screen
- **Name Label UI:** it shows every user's name on top of its avatar
- **Scene connections:** Connecting the scenes via scene change or showing/hiding

6.3 Yusuf Kütük

Yusuf worked in both front-end and back-end team during the project and the following tasks were performed by him.

Front-end contributions

- Creation of customizable male and female avatar graphics
- Creation of Idle and walk sprite sheets for avatars
- Creation of 16 animation in Godot using the sprite sheets
- Implementation of customization logic using UI created by Fatma
- Creation of participant scene and implementation of participant logic which includes movement, collision and animation
- Application icon design

Back-end contributions

Yusuf and Daniel have implemented multiplayer for our project using Godot high-level multiplayer API. The main tasks of Yusuf:

- Research on high-level multiplayer API documentation
- Implementation of first minimal multiplayer demo where multiple clients and one host can join. Each of them has an image with collision body and they can move while their positions are synchronized.
- Showing the correct customized avatar for every participant in a meeting
- Testing example network projects [11], and creating minimal example
- Synchronization of participants and doors
- Synchronization of animations
- Adding command line argument support to our application in order to have control over aspects like IP address, port number and starting mode (server or client)
- Implementation of space selection logic

Documentation & presentation

- Synchronization part of the multiplayer documentation
- Architectural diagram for documentation
- Used technologies presentation

Other contributions

- Support some of the team members with Godot, git, merging, testing and implementation
- Some help on package installation on ubuntu server
- Initial implementation of room scene and script
- Removal of server's participant
- Script for running multiple servers and chat server with a single command

6.4 Khaled Aboualnaga

- **Back-end:**
  1. Chat:
     - The chat of our application was fully implemented by Khaled both on server- and client-side, which included these tasks:
     - Built the Python server for the chat application using websockets library which uses asyncio (asynchronous Python library).
     - Researching which solution and technologies would work the best for the chat system, in addition to research on which database and schema we would use.
     - Setting up the MongoDB database to store messages and to support chat history.
     - Implementation of chat based on rooms, and chat history both on server- and client-side.
     - Using Python Motor asynchronous library to handle MongoDB database to store and retrieve messages
     - Working on a subprotocol mechanism to communicate between client and server.
  2. Multiplayer:
     - Research on how the Multiplayer can be built and the technologies we can use.
     - Contribution to the first implementation of multiplayer like positioning.
     - Setting up the server for both the Multiplayer and chat.
- **Front-end:**
  3. UI:
     - Designing of Register and Login screens and the first version of Chat-UI, which was used at the beginning, but we later switched to Fatma's Chat UI.
  4. Client-Side:
     - Worked on refining the Chat UI logic and to provide implementation to update Chat UI according to incoming messages.
     - Parsing and processing messages received from server-side on different parts of the application.
     - Worked on the client-side Rooms implementation and a mechanism on how to notify server when user leaves or enter room areas.
     - Retrieval of messages and processing on client-side when user enters/leaves the room.
     - Implementation of WebSocket protocol on client-side (Godot-engine).
- **Documentation and Presentation:**
     - Chat Documentation.
     - Chat presentation slides.

6.5 Amai Rivas

Project organization

Amai was both the product owner and the scrum master on the project. So, he was mostly responsible to organise the team, to set goals and to interact with the customer. Of course, we had a very democratized environment, where big decisions were voted if there was any sort of disagreement, and he didn't do all on his own. But, as the main project organiser, he mainly accomplished the next tasks:

- Create the project organisation document with the vision, mission, milestones, and team organisation
- He was in charge of Jira, creating most of the User Stories, prioritizing them and creating Tasks, Meetings, etc. (again with the help of the team)
- He was managing most of the meetings (Daily, Sprint Planning, Client meetings, Coaches, Retrospectives…). As well as taking notes, taking feedback from our client or our coach; and letting the team know about the next meeting, or what was important, etc.
- He was responsible to create the Sprint objectives, and make sure they were realised at the end of the Sprint, with the prioritisation of the User Stories and Tasks on the Sprint Planning.
- He also created the Milestones 2.0 document
- He created the layout of the presentation, document, and a poster sketch, and organised the team's tasks to populate them
- He created the project management, and the result part of the documentation
- He put the documentation together and applied an adequate style
- He created the intro, retrospective and outlook part of the presentation

Spaces

Even though organizing the project has taken a lot of time, Amai also took the main role as a Space Designer, first creating the library, and then the University. Amai tried to find assets online to create the Spaces, but he didn't like any, so he drew by hand all required assets (the only one taken from the internet is the grass on the outside). With the respective tasks, summarized:

- Draw all the assets that will be used on the Space: floor, walls, libraries, tables, chairs, sofas, sculptures, piano, sits, door, outside path, barrier, caffe machines, dispenser machines, printer, sink, computers, kicker, ping-pong, microwave, blackboards, etc.
- Create the layout of each space with the floor tile map
- Create the wall layout with the required collision
- Create and program the door to open and close with a key (with help from his teammates for the synchronization)
- Add assets to the space using tile maps, and add the pertinent collision to each, so that they act like a normal object
- Assist his colleague Ash on the creation of the Office

6.6 Assylbek Bugybay

Assylbek worked on both backend and frontend and performed the next tasks:

Front-end contributions:

- Minor Chat UI improvements
- Implementing hide and show options in Participant.tscn scene, synchronization of menu, participants, chat and emojis UI
- Implementation of the Participants UI
- Improvements in the scene Tree of Participants and Default scene
- Implementation of the Emojis UI
    - Search and download of the 48x48 Emojis
    - Implementation of the panel to store the emoji
- Additions of the new sprites and changes of avatars collision scene, changes of its collision shape, changes to toggle options and z-value
- Key listeners to Avatar and Chat UI scenes

Back-end contributions:

- Research on localhost multiplayer and Firebase solutions
- Initial research on Firestore and Firebase Realtime databases
- Node JS and Python server Socket connection prototypes
- UI prototype for the chat, with button RichText and the Text field- was not used further
- Installing the second Godot server on the Google Cloud, so multiple tests can be run simultaneously
- Usage of the emojis in the chat
- Participants list implementation-connection of the Participants UI buttons and panels to store the list with the methods in Meeting.gd script
- Participants locations implementation:
    - Extension of the predefined Godot methods to get Realtime updates of the participants locations- solves the issue when new user is entered the scene or when someone left it
    - Implementation of the get method to store the signal from different scene
- Emojis UI implementation:
    - Synchronization of the puppets
    - Signals interface from Emojies.gd
    - Network master and non-network master current_emojis property synchronization
- Implementation of the Do Not Disturb emoji
    - Synchronization of other emojis with Do not disturb emoji

Documentation & presentation

- Presentation slides on User management
- Introduction to this documentation
- Section 2 of this documents, on initial requirements and user stories
- 7.2 Outlook of this ducment

Other contributions:

- Creating the Discord server
- Bugs fixes on server failures
- Creation of the executable of the program
- Constant Merge and bugs fixes

6.7 Asian Hossain

Research & Backend

- Asian helped with the research of Realtime Database implementation in Godot. Tried to implement it at first but was unsuccessful due to a bug in the api, later team decided to use Firestore database.
- Helped Khaled a little bit with debugging error in the chat system

Frontend

- Space Design (Office)
  - Create a background for the space
  - Create a layout of the space with TileMaps
  - Used TileSet to create the furniture (sofa, office chairs, meeting tables reception computers and chairs), walls.
  - Draw collision shapes for all the above so that avatar is not able to walk over them
  - Create the room areas for the chat

Documentation & Presentation

- Documentation: Technical Concept – Space design
- Presentation: Space design
- Poster: Created the poster with Adobe InDesign

Other contributions

- Helped with the setup of Google Cloud Server

7. Conclusion

7.1 Final retrospective

Finishing this documentation, we will have a look at the final retrospective. We had a meeting where instead of doing the regular per-spring retrospective, we did the final project retrospective, trying to look back at the whole project and write down the three bullet points of a retrospective meeting: what went well, what went not so well, and what we can improve. Here are our findings:

- What went well?
  - The team had a very similar vision about how the project should be
  - Teamwork kept improving each time we spent on meetings o working together
  - We didn't have big fundamental disagreements on what technologies to use
  - Organization in terms of meetings, documents, and tasks went very well
  - We were mostly on time to meetings
  - Daily meetings were very useful, with feedback from teammates when needed
  - We were helpful to people who don't understand some concept or part of the project
  - Code was well commented
  - We were able to organize ourselves to work on tasks between different disciplines (frontend vs backend)

- What went not so well?
  - Sometimes we would do double work because of bad communication
  - Some of the tasks were more complex than we anticipated
  - We didn't divide tasks to the simplest subtasks
  - We sometimes didn't communicate decisions to the team
  - Not communicating when we had issues with our tasks
  - We had a slow start because Jira was a new environment
  - At the beginning client meetings were about showing your work, instead of the team's work
  - Not a "team sentiment", but an "individual sentiment"

- What we can improve:
  - Try to think as a team, instead of just about ourselves
  - Have one person show the Sprint Results to the client
  - Arrange meetings with urgency, to avoid losing time
  - Realize that you are not the only one working hard on the project

In general, we think that we had a bit of a slow start because of some miscommunications, and bad organization, but we also think that we have exponentially improved our team working and communication skills.

Having a slow start was expected, since we were just getting started and getting to know each other as a team. But towards the end of the project our teamwork was challenged, since there were a lot of code merging, and multidisciplinary development, and yet we succeeded without any major problem.

7.2 Outlook

Here we would like to specify the next features could be implemented in the future releases:

- Pinboards in the classrooms to provide users useful information
- Implement more motion Sprites to enable the user to drink coffee near coffee machine
- Whiteboards to write, enabling teachers or professors to teach
- More spaces
- Possibility to create their own spaces with editor
- Add video chat/voice chat
- Add pin boards to allow some sort of asynchronous interaction between users
- Private spaces like a Webex Meeting Rooms
- Extend the app to mobile
- More avatar designs
- Introduce online chemistry classes, like in Minecraft Education Edition
- Enable switch from 2.5D to 3D

8. References

[1] Firestore REST API docs: https://firebase.google.com/docs/firestore/use-rest-api (accessed Nov. 14, 2022).

[2] Firebase REST API Auth docs: https://firebase.google.com/docs/reference/rest/auth (accessed Nov. 14, 2022).

[3] Godot HTTPRequest docs: https://docs.godotengine.org/en/stable/classes/class_httprequest.html (accessed Nov. 14, 2022).

[4] Godot High-Level Multiplayer: https://docs.godotengine.org/en/stable/tutorials/networking/high_level_multiplayer.html (accesse d Nov. 14, 2022).

[5] Godot NetworkedMultiplayerENet: https://docs.godotengine.org/en/stable/classes/class_networkedmultiplayerenet.html#class-networkedmultiplayerenet (accessed Nov. 14, 2022).

[6] PacketPeer: https://docs.godotengine.org/en/stable/classes/class_packetpeer.html#class-packetpeer (accessed Nov. 14, 2022).

[7] SceneTree: https://docs.godotengine.org/en/stable/classes/class_scenetree.html#class-scenetree (accessed Nov. 14, 2022).

[8] rset: https://docs.godotengine.org/en/stable/classes/class_node.html?highlight=rset#class-node-method-rset (accessed Nov. 14, 2022).

[9] Designed by PlusOne Fonts, https://www.fontspace.com/sztylet-font-f22264 (accessed Nov. 14, 2022).

[10] Licensed as SIL Open Font License (OFL), https://scripts.sil.org/cms/scripts/page.php (accessed Nov. 14, 2022).

[11] Network example projects https://github.com/godotengine/godot-demo-projects/tree/master/networking (accessed Nov. 14, 2022).

[12] Using TileMaps https://docs.godotengine.org/en/stable/tutorials/2d/using_tilemaps.html#creating-a-tileset (accessed Nov. 14, 2022).

https://github.com/godotengine/godot-demo-projects/tree/master/networking

[13] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, https://www.rfc-editor.org/info/rfc6455.

[14] An overview of HTTP, Mozilla.org. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (accessed Nov. 14, 2022).
[15] Websockets 10.4 documentation, readthedocs.io. https://websockets.readthedocs.io/en/stable/index.html (accessed Nov. 14, 2022).
[16] Asynchronous I/O, python.org. https://docs.python.org/3/library/asyncio.html (accessed Nov. 14, 2022).
[17] Motor: Asynchronous Python driver for MongoDB, readthedocs.io. https://motor.readthedocs.io/en/stable/ (accessed Nov. 14, 2022).
[18] PyMongo 4.3.2 Documentation, readthedocs.io. https://pymongo.readthedocs.io/en/stable/ (accessed Nov. 14, 2022).
[19] MongoDB Python Connection, mongodb.com. https://www.mongodb.com/languages/python (accessed Nov. 14, 2022).
[20] MongoDB Manual, mongodb.com. https://www.mongodb.com/docs/manual/ (accessed Nov. 14, 2022).
[21] Signals – Godot Engine Documentation in English, godotengine.org. https://docs.godotengine.org/en/3.3/getting_started/step_by_step/signals.html (accessed Nov. 14, 2022).
[22] Observer – Game programming patterns, gameprogrammingpatterns.com. https://gameprogrammingpatterns.com/observer.html (accessed Nov. 14, 2022).
[23] Area2D - Godot Engine Documentation in English, godotengine.org. https://docs.godotengine.org/en/stable/classes/class_area2d.html (accessed Nov. 14, 2022).
[24] CollisionShape2D - Godot Engine Documentation in English, godotengine.org. https://docs.godotengine.org/en/stable/classes/class_collisionshape2d.html (accessed Nov. 14, 2022).
[25] Designed by PlusOne Fonts, https://www.fontspace.com/sztylet-font-f22264 (accessed Nov. 14, 2022).
[26] Licensed as SIL Open Font License (OFL), https://scripts.sil.org/cms/scripts/page.php (accessed Nov. 14, 2022).
[27] https://docs.godotengine.org/en/stable/classes/class_control.html#class-control (accessed Nov. 14, 2022).
[28] https://docs.godotengine.org/en/stable/tutorials/scripting/scene_tree.html?highlight=change_scene#changing-current-sceneScene Transition (accessed Nov. 14, 2022).

9. Appendix

A. Backlog

It was too long to add to the documentation, so it can be accessed on jira: https://collaborationspace.atlassian.net/issues/?jql=created%20%3E%3D%20-30w%20ORDER%20BY%20created%20ASC

B. Project organization

Creating a 2D Virtual Collaboration Space – Team E

Project mission

To create an application that the users can use to join a 2D space where they can move, chat, see other users, and interact with each other; in a completely online manner.

Project vision

To bring the online world a step closer to the real world. Enabling our users to have the same experience they would have if they were physically in that space, while making it open source.

Milestones

2 weeks
1st Creation of a 2D space with an avatar that can move and collide with some walls.
2nd Enable the space online.
3rd Add the ability for other users to join the space online.

4 weeks
4th Give the users the ability to chat and interact with each other.
5th Creation of a UI to allow for some configuration, exiting, joining, seeing who is online…
6th Creation of different spaces (university, library, caffe, park…)

2 weeks
7th Create the avatar selection screen, enabling to customize their avatars (different avatars, hair color, hair length, skin color…).
8th Extras, ideas: with the remaining time, we can work on giving users the ability to create/join private rooms, to create their own spaces (level editing); and what we can think of, that could improve our users' experience.

Project organization

To realize this ambitious project, we have decided to organize ourselves as follows:

- Process model: Scrum (with 1-week Sprints, finishing on Thursdays)
- Roles:

    - Scrum Master: Daniel Dedoukh
    - Product owner: Amai Rivas
    - Developers:
        - Front end:
            - Fatma Sude Kütük
            - Enver Yusuf Kütük
            - Amai Rivas
        - Back end:
            - Khaled Aboualnaga
            - Asian Hossain
            - Assylbek Bugybay
            - Daniel Dedoukh
- Version control: **GitHub**
- Online meeting tool: **discord** (for team meetings), **Webex** (for client meetings)
- Weekly client meetings: **Thursdays at 12:30**, at: https://thu.webex.com/meet/amai
- Working model: **online**
- Game engine: **Godot**
- Programming language: **gdscript**
- Graphics style: **Pixel Art**
- Backend: **to be determined** (research needs to be done still)

C. Milestones 1.0

This week (20.10 – 27.10):

- Have a working chat on multiplayer (show usernames, per-room listening of messages)
- Have a working multiplayer (being able to join any time, name correctly shown, avatar customization correctly shown, exiting the meeting)
- Have 2 extra Spaces a user may choose to join
- Have a uniform/improved UI theme

Week 6 (27.10 – 3.11):

- Being able to join the 2 extra Spaces in multiplayer
- Last week to add features/improvements (Pin boards? Emojis? Reactions? Arrow to show where a user is?)

Week 7 (3.11 – 10.11):

- Last week to add small features/improvements
- Clean the code
- Find bugs/repair them
- Have a working deliverable at the end of the week

Week 8 (10.11 – 17.11):

- Prepare presentation day (poster, documentation, power point)

- Finalize project
- Create final project deliverable