Calla Reimplementation

The calla chat client is created using the Godot Engine v3.4.4. The project was coded using gdscript with external libraries such as PJSIP integrated using gdnative. The following is an exhaustive list of scenes (.tscn) currently present in the client.

- Avatar1
- Avatar2
- AudioStreamPlayer
- ControlledPlayer
- GUI
- Map1
- PeerPlayer
- PJSIP
- TitleScreen
- Tree

Avatar1 and Avatar2

These scenes contain AnimatedSprite nodes. These are used depending on what avatar number the user chooses as the AnimatedSprite node used in either ControlledPlayer or PeerPlayer.

AudioStreamPlayer

This scene is auto loaded into the project. It contains an AudioStreamPlayer which is used as audio output for the voice call.

ControlledPlayer

This scene is used as the base prototype for the character that the player using the client locally would control. It contains assets needed for emoting as well as a collision shape used as the character's hitbox.

GUI

This scene is loaded in when the client is connected and in the map. It contains the user interfaces that are used by the client. It has the emote panel, settings panel, etc. It is where all control nodes used outside of the initial registration screen are placed.

Map1

This scene contains the map used by the client when connecting to the game. The map is created through a mixture of tile maps and objects placed in the scene manually such as trees. These objects also have hitboxes which players can collide with. The map also automatically generates a navigation mesh for players to pathfind properly to where they clicked to go. The scene also contains a YSort node which character sprites are supposed to be a child of in order for them to render in the correct order. The map also contains with it the main camera.

PeerPlayer

This scene is used to instantiate characters into the game which are not controlled by the local player. It is identical to ControlledPlayer in most ways and only different in that it is attached to a different script since it has different logic from a ControlledPlayer.

PJSIP

This scene contains the PJSIP node which is used in order for the client to properly interface with SIP and start calls with the server. It is auto loaded into the project.

TitleScreen

This scene is the main scene, meaning it is the entrypoint of the client. It contains control nodes where one should configure details in order to properly connect to their desired server.

Tree

This is a scene which is used to instantiate trees in the map

The following is an exhaustive list of scripts (.gd) currently present in the client.

- Character
- ControlledPlayer
- Emote Panel
- Map1Navigation
- Multiplayer
- OptionsPanel
- PeerPlayer
- PlayerCamera
- Resources
- Settings
- SpeechBubble
- TitleScreen

Character

This script is a base class used by both ControlledPlayer and PeerPlayer. It describes common functions used by both ControlledPlayer and PeerPlayer. This contains things such as setting the sprite’s avatar, pathfinding, animation and emoting.

ControlledPlayer

This script inherits from Character, it adds logic for letting the local player control the specified character.

Emote Panel

This script contains logic for showing and hiding the emote panel

Map1Navigation

This script contains logic for baking the navigation mesh. It handles the navigation mesh of trees as well as tiles with hitboxes. It does this by getting hitboxes in the map, combining them and then cutting them out of the navmesh. Note that hitboxes that create donuts (polygons with holes inside them) when combined may break the current implementation.

Multiplayer

This script is autoloaded into the project. It contains all the functions needed for the networking capabilities of the game. It contains rpc calls as well as functions which are expected to be rpc called.

OptionsPanel

This script is the logic behind the options menu. It handles visibility of the panel as well as actually changing the values that are present in the options menu in their respective nodes.

PeerPlayer

This script inherits from Character, it adds logic for updating the position of the sprite from the server and also displaying any emotes that other players should want to show.

PlayerCamera

This script controls the position of the camera. It smoothly follows the player. It also listens for scroll inputs and zooms in or out respectively.

Resources

This script is autoloaded into the project. It contains a bunch of preloaded resources that are used in the client.

Settings

This script describes the logic of the settings panel as well as being responsible for its visibility.

SpeechBubble

This script is responsible for properly showing emotes on characters and then hiding them after a timer is hit.

TitleScreen

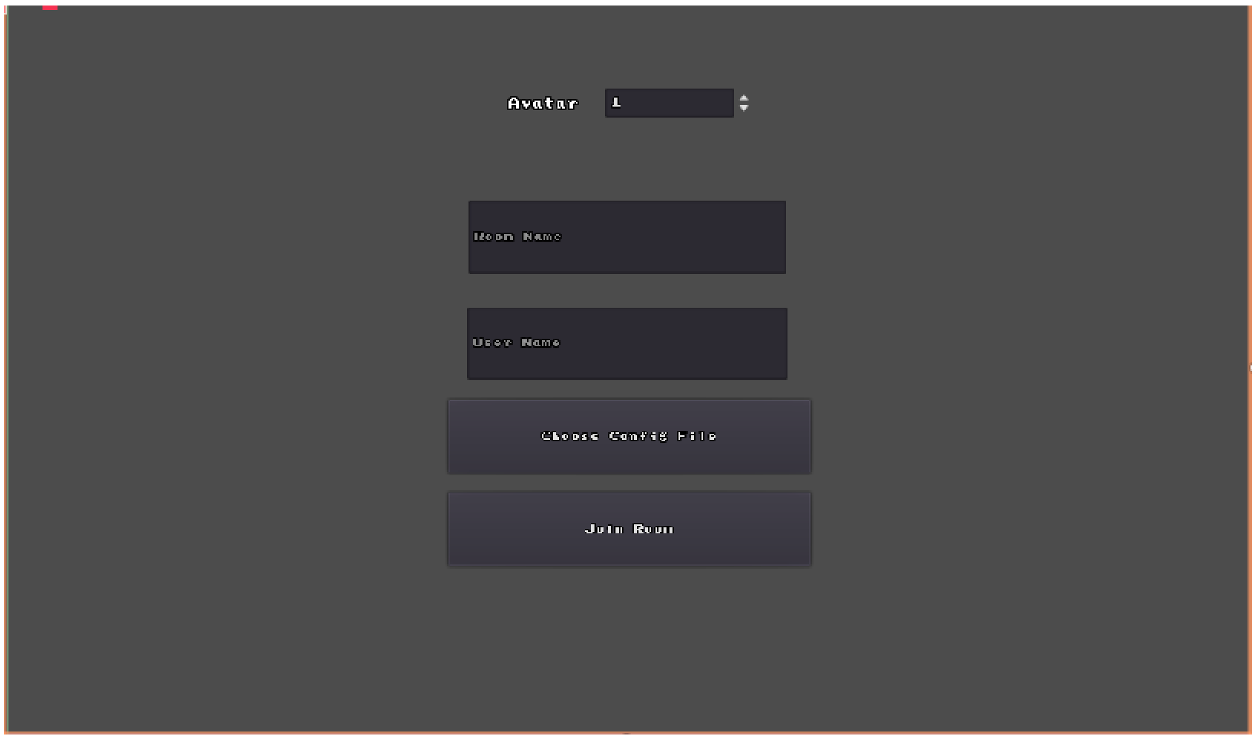
This script contains all the logic of the initial configuration screen and makes checks in order to make sure that the configuration is correct.

General Architecture

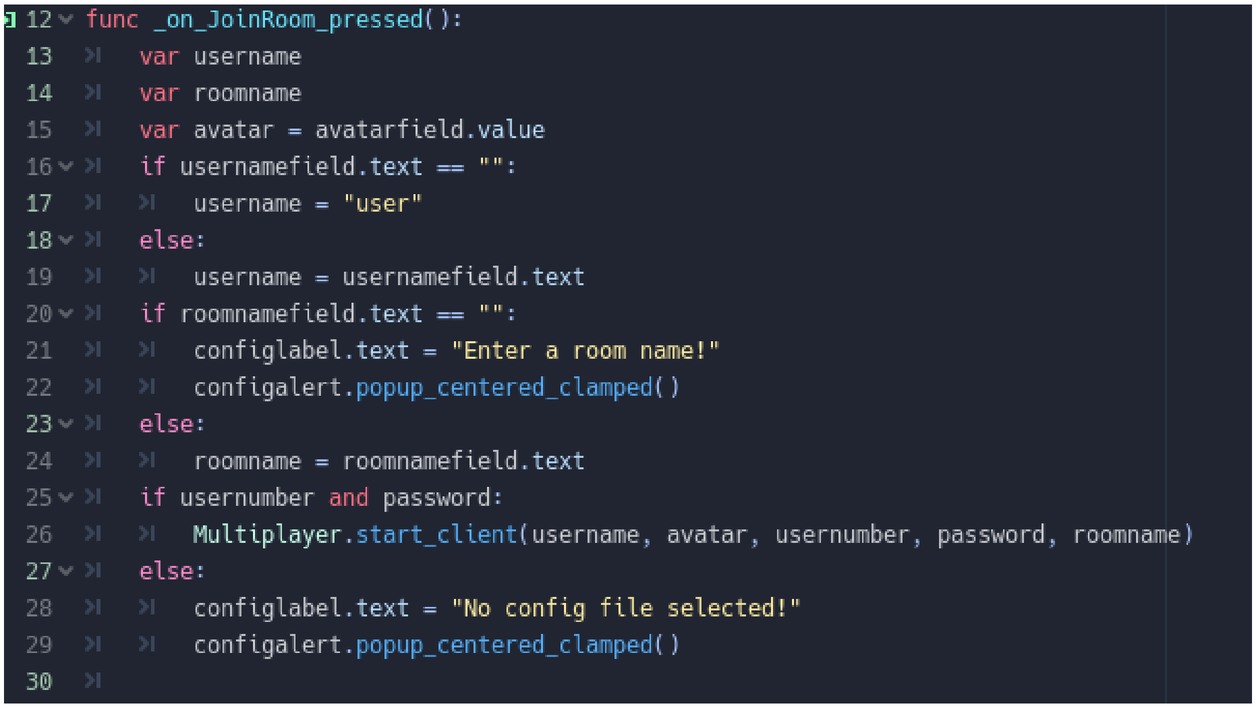
The code is generally structured so that all network aspects of the client are handled in Multiplayer.gd. The client also has two main scenes, the title screen where all configurations are done (TitleScreen.tscn) and the “game” scene where the actual call takes place (Map1.tscn).

Call Initialization

The client first starts with the TitleScreen.tscn scene. This is where all initial configurations to connect to the server must be done:

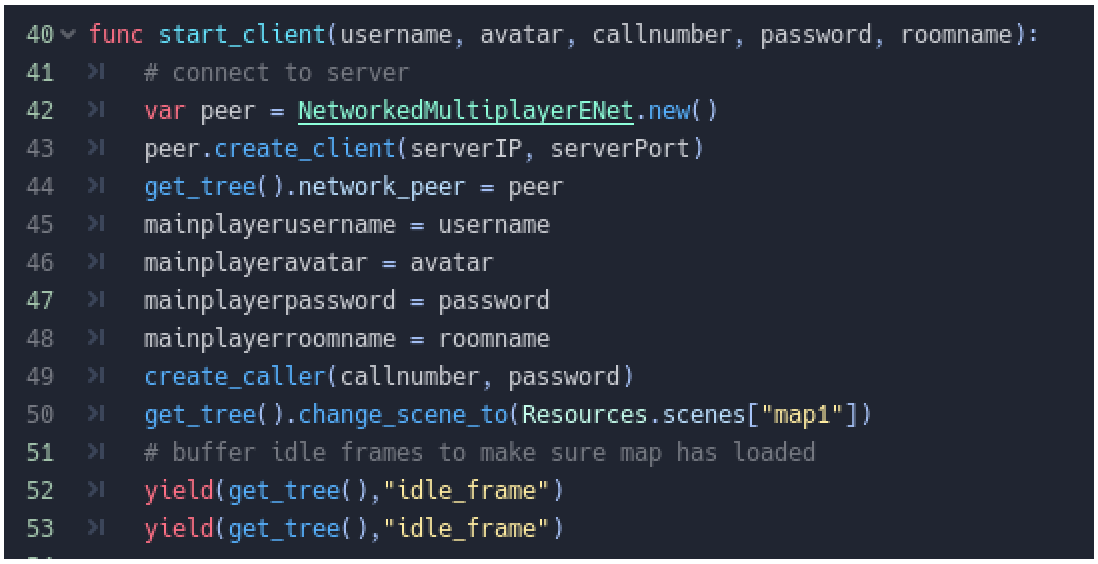


After all is said and done, pressing the join room kicks off this function:



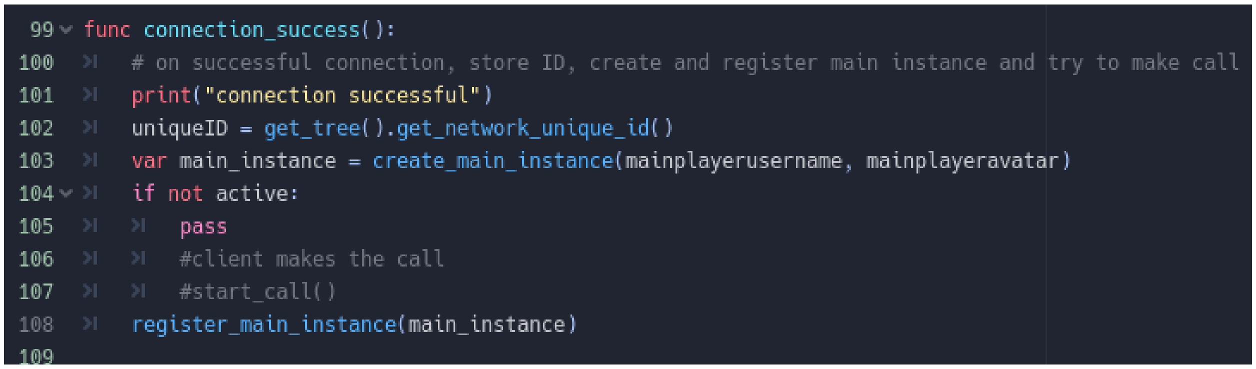
```
func _on_JoinRoom_pressed():
	var username
	var roomname
	var avatar = avatarfield.value
	if usernamefield.text == "":
		username = "user"
	else:
		username = usernamefield.text
	if roomnamefield.text == "":
		configlabel.text = "Enter a room name!"
		configalert.popup_centered_clamped()
	else:
		roomname = roomnamefield.text
	if usernumber and password:
		Multiplayer.start_client(username, avatar, usernumber, password, roomname)
	else:
		configlabel.text = "No config file selected!"
		configalert.popup_centered_clamped()
	
```

All the fields are checked and validated. After validation, it then passes off relevant data to the start_client function in the Multiplayer.gd node.



```
func start_client(username, avatar, callnumber, password, roomname):
	# connect to server
	var peer = NetworkedMultiplayerENet.new()
	peer.create_client(serverIP, serverPort)
	get_tree().network_peer = peer
	mainplayerusername = username
	mainplayeravatar = avatar
	mainplayerpassword = password
	mainplayerroomname = roomname
	create_caller(callnumber, password)
	get_tree().change_scene_to(Resources.scenes["map1"])
	# buffer idle frames to make sure map has loaded
	yield(get_tree(),"idle_frame")
	yield(get_tree(),"idle_frame")
```

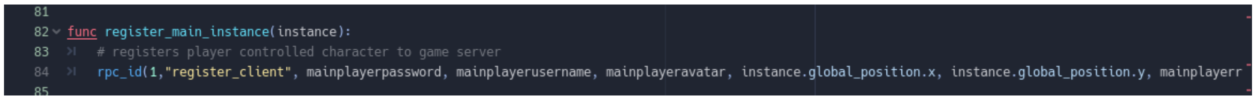
In the start client node, the client tries to establish a connection to the server. It then stores all the passed data into local variables for later use. It then registers itself as a SIP endpoint with the create_caller function. After that it loads the map. After this we expect a successful connection which triggers a signal which calls the ff function:



```
func connection_success():
	# on successful connection, store ID, create and register main instance and try to make call
	print("connection successful")
	uniqueID = get_tree().get_network_unique_id()
	var main_instance = create_main_instance(mainplayerusername, mainplayeravatar)
	if not active:
		pass
		#client makes the call
		#start_call()
	register_main_instance(main_instance)

```

Upon successful connection, we store our network ID. We then create the character instance that the player will control through the create_main_instance function, after creating the main instance we then register it to the server using the register_main_instance.

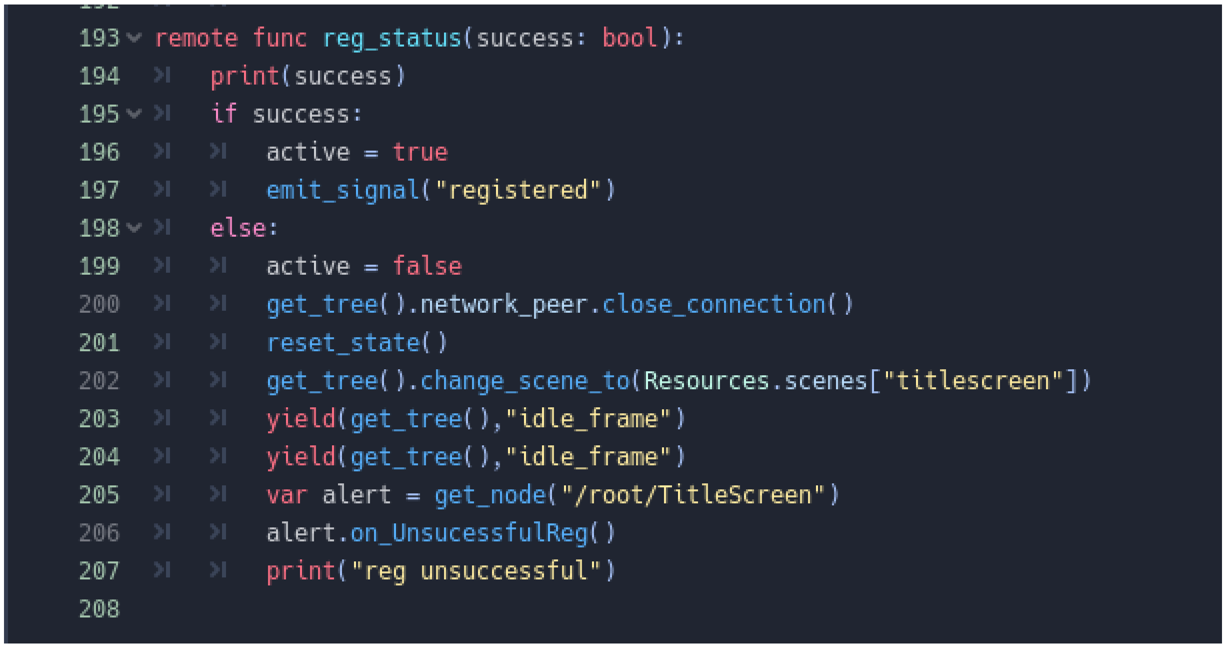


```

func register_main_instance(instance):
	# registers player controlled character to game server
	rpc_id(1,"register_client", mainplayerpassword, mainplayerusername, mainplayeravatar, instance.global_position.x, instance.global_position.y, mainplayerr

```

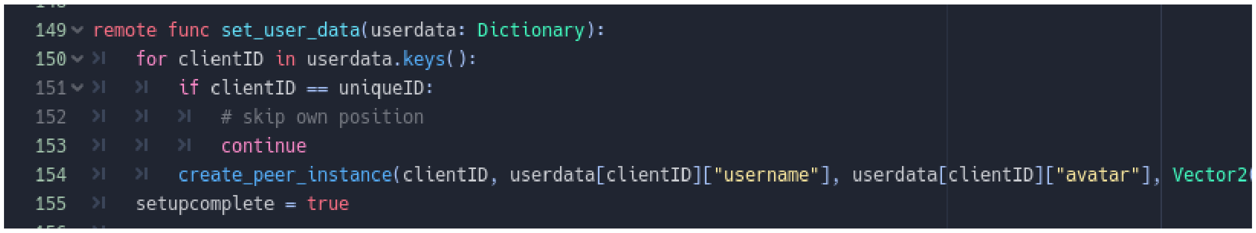
Register main instance simply calls an rpc function to the server to relay all of our character's details to the server. We then wait for a reply from the server in the form of an RPC call to the ff function:



```
remote func reg_status(success: bool):
	print(success)
	if success:
		active = true
		emit_signal("registered")
	else:
		active = false
		get_tree().network_peer.close_connection()
		reset_state()
		get_tree().change_scene_to(Resources.scenes["titlescreen"])
		yield(get_tree(),"idle_frame")
		yield(get_tree(),"idle_frame")
		var alert = get_node("/root/TitleScreen")
		alert.on_UnsucessfulReg()
		print("reg unsuccessful")

```

If registration was a success, we now expect the following function to be called by the server to our client:

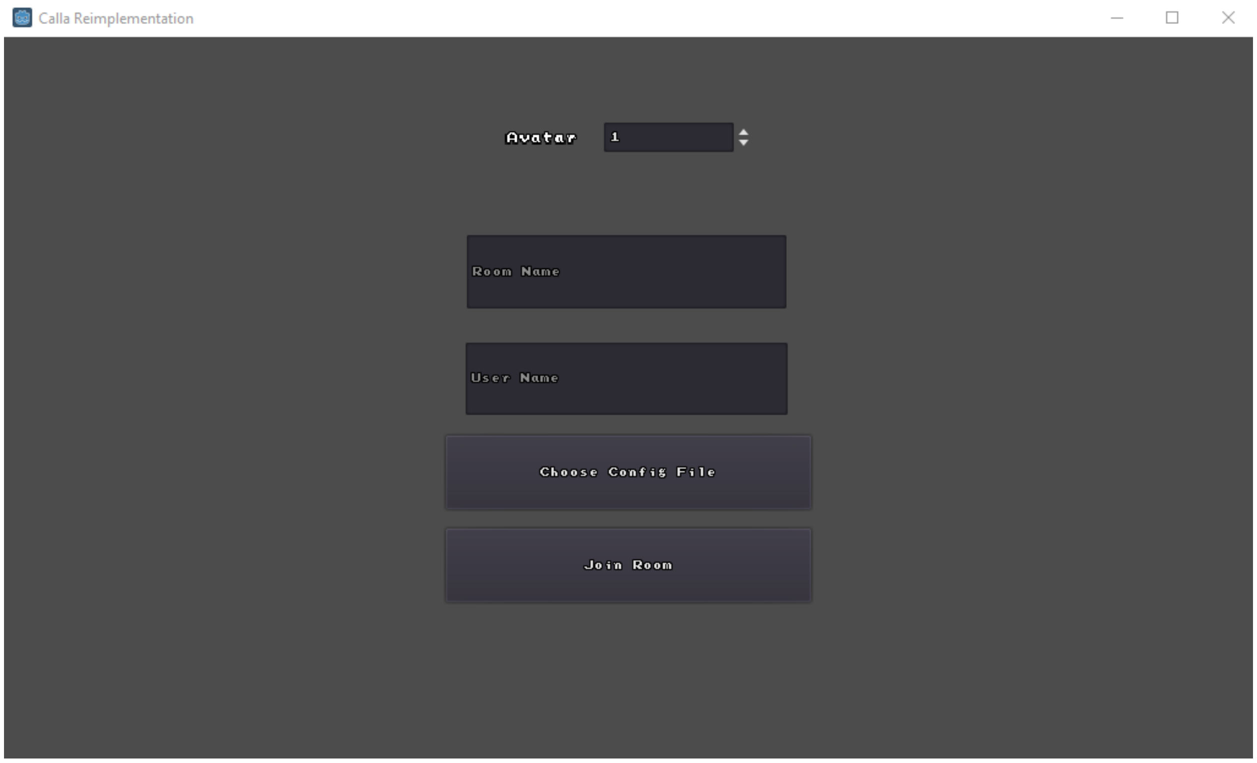


```
remote func set_user_data(userdata: Dictionary):
	for clientID in userdata.keys():
		if clientID == uniqueID:
			# skip own position
			continue
		create_peer_instance(clientID, userdata[clientID]["username"], userdata[clientID]["avatar"], Vector2(
	setupcomplete = true
```

This function is called by the server in order to share all current clients already connected to the server and their usernames, avatars and positions. It is given through a dictionary. It then calls create_peer_instance which creates a character in the scene for each client already connected. After this step, setup is complete and a variable is set in order to show that.

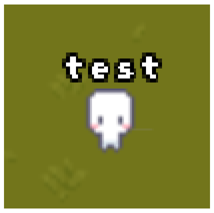
Title Screen Configuration

There are 4 things to configure in the title screen:

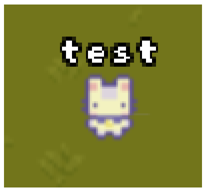


The avatar can be changed in order to change the sprite shown when joining in as a character:

When avatar is set to 1:



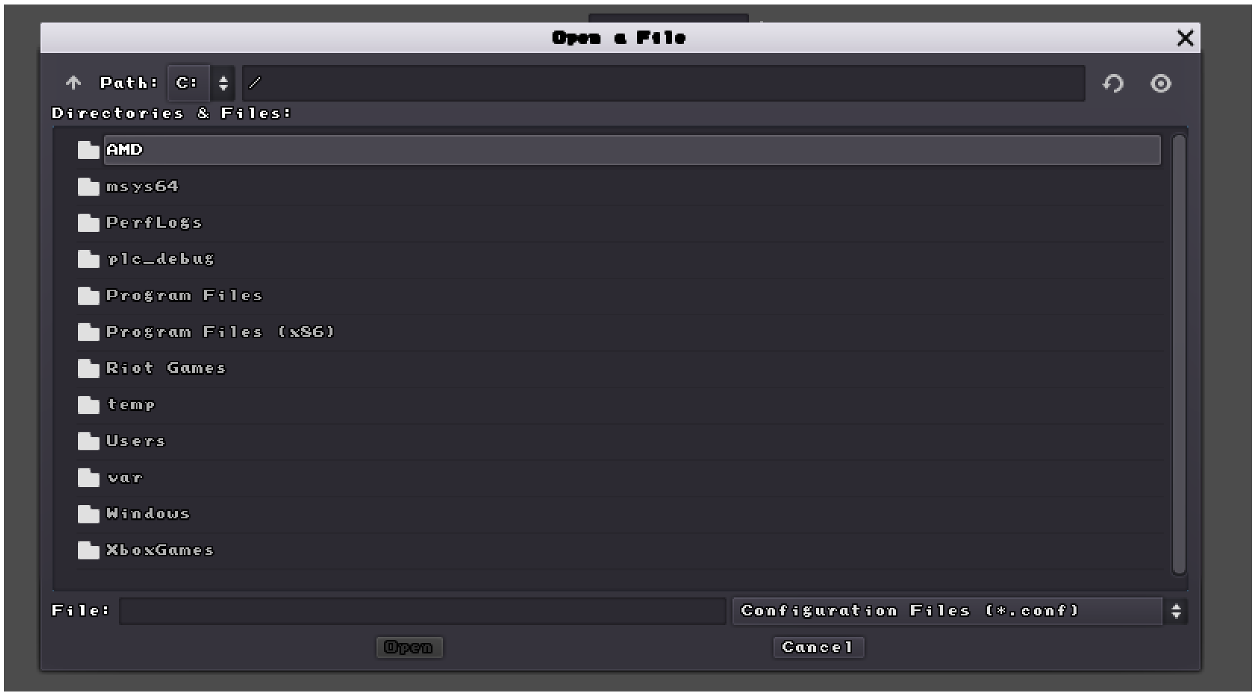
When set to 2:

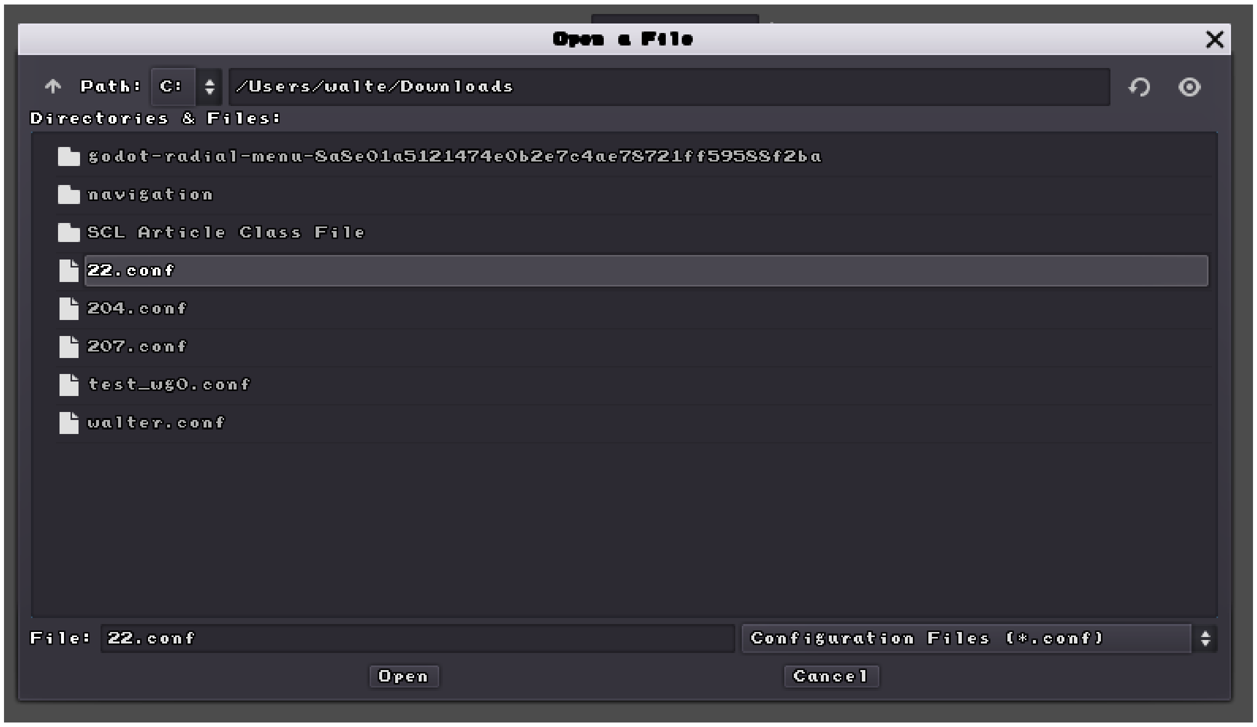


The room name, which does not do anything currently.

The username which determines the name floating above the sprite's head.

And the choose config file button. This button popups a file dialog window with which you are expected to choose the Wireguard configuration file given in order to connect to the conference.

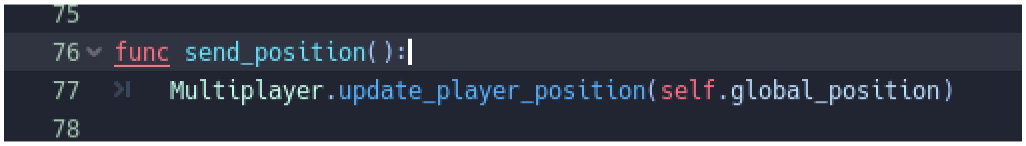




```
Path: C: /Users/walte/Downloads
Directories & Files:
  godot-radial-menu-8a8e01a5121474e0b2e7c4ae78721ff59588f2ba
  navigation
  SCL Article Class File
  22.conf
  204.conf
  207.conf
  test_wg0.conf
  walter.conf
File: 22.conf        Configuration Files (*.conf)
```

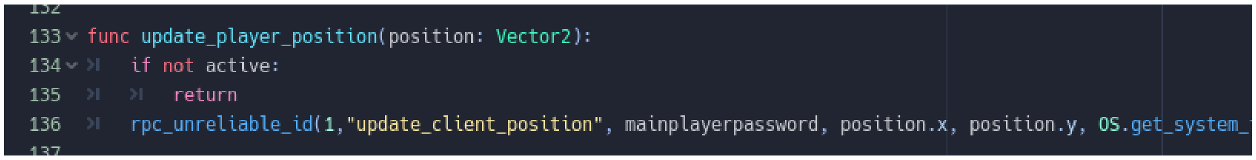
Client Updates

The client updates the server through calling the following function in Character.gd



```
func send_position():
	Multiplayer.update_player_position(self.global_position)
```

This function is called by the character controlled player in the scene. This then calls the update_player_position function in Multiplayer.gd

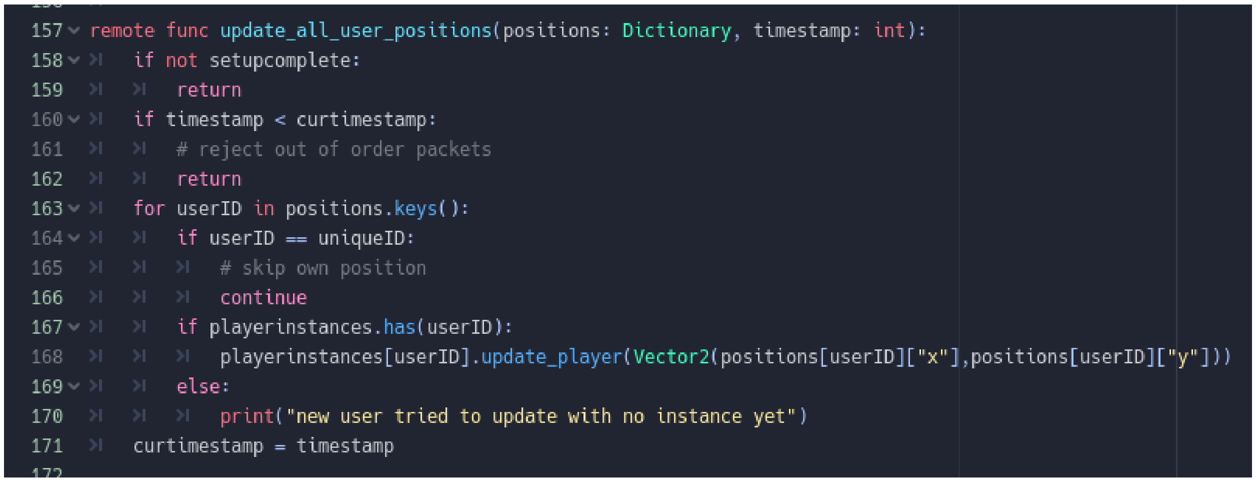


```
func update_player_position(position: Vector2):
	if not active:
		return
	rpc_unreliable_id(1,"update_client_position", mainplayerpassword, position.x, position.y, OS.get_system_
```

It then calls an unreliable rpc to the server in order to update our current position to the server.

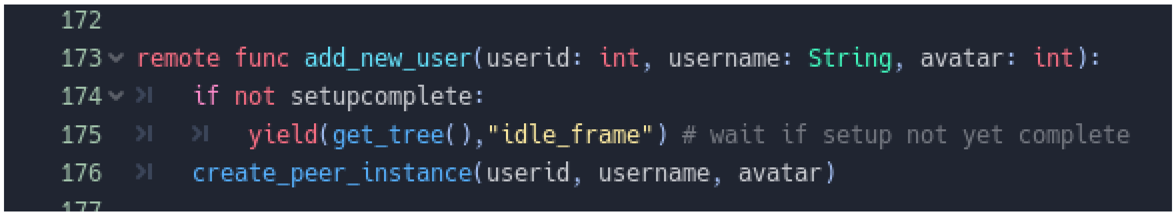
Server Updates

In order to update us about the current state of the meeting, the server can call four different functions through rpc. They are tagged with remote in order to signify that they can be called by the server. First is the update_all_user_positions function which updates all peer instances in the client and their positions.



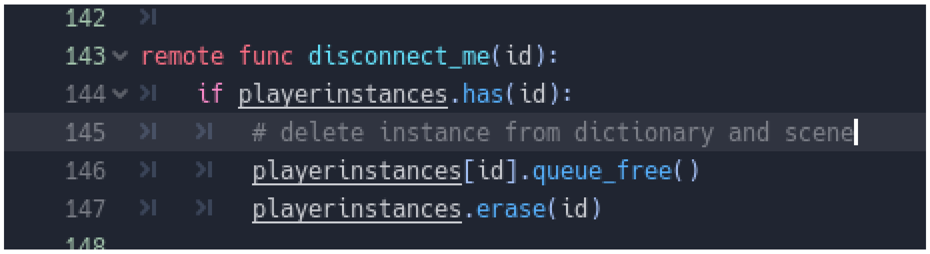
```
remote func update_all_user_positions(positions: Dictionary, timestamp: int):
	if not setupcomplete:
		return
	if timestamp < curtimestamp:
		# reject out of order packets
		return
	for userID in positions.keys():
		if userID == uniqueID:
			# skip own position
			continue
		if playerinstances.has(userID):
			playerinstances[userID].update_player(Vector2(positions[userID]["x"],positions[userID]["y"]))
		else:
			print("new user tried to update with no instance yet")
	curtimestamp = timestamp
```

Next is the add_new_user function which is called every time a new user joins the call.



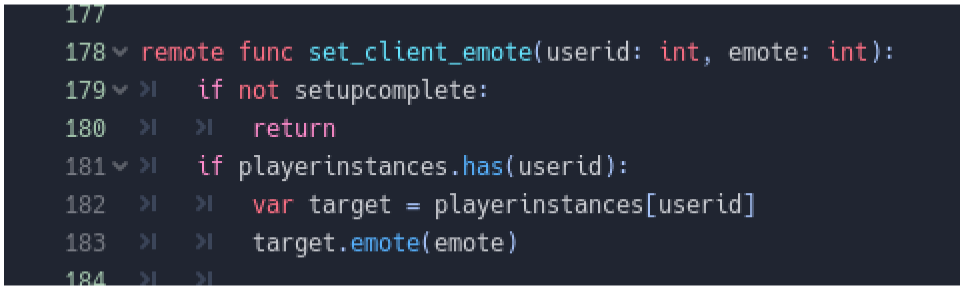
```
remote func add_new_user(userid: int, username: String, avatar: int):
	if not setupcomplete:
		yield(get_tree(),"idle_frame") # wait if setup not yet complete
	create_peer_instance(userid, username, avatar)
```

Next is disconnect_me which is called every time someone disconnects from the call.



```
remote func disconnect_me(id):
	if playerinstances.has(id):
		# delete instance from dictionary and scene
		playerinstances[id].queue_free()
		playerinstances.erase(id)
```

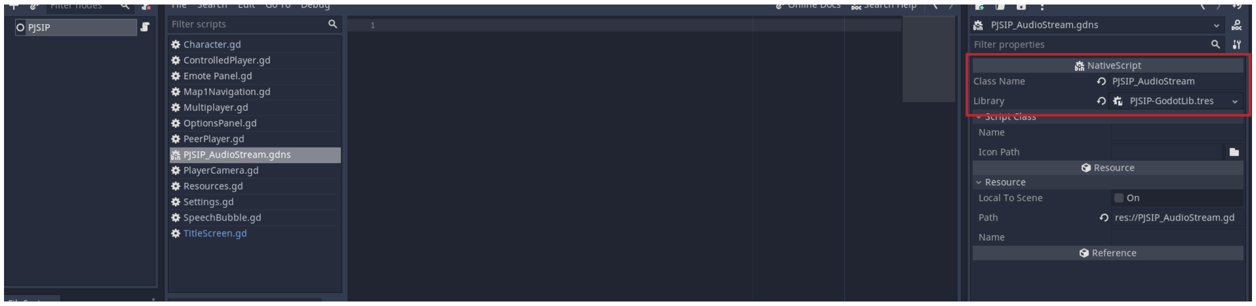
Lastly, set_client_emote is called every time someone emotes



```
remote func set_client_emote(userid: int, emote: int):
	if not setupcomplete:
		return
	if playerinstances.has(userid):
		var target = playerinstances[userid]
		target.emote(emote)
```

PJSIP Node

The PJSIP node is important as it handles all SIP related (meaning the call part of the client) data. It is currently auto loaded into the scene. Make sure it is setup correctly by looking into PJSIP.tscn:



PJSIP

Filter scripts
- Character.gd
- ControlledPlayer.gd
- Emote Panel.gd
- Map1Navigation.gd
- Multiplayer.gd
- OptionsPanel.gd
- PeerPlayer.gd
- PJSIP_AudioStream.gdns
- PlayerCamera.gd
- Resources.gd
- Settings.gd
- SpeechBubble.gd
- TitleScreen.gd

PJSIP_AudioStream.gdns

NativeScript

Class Name	PJSIP_AudioStream
Library	PJSIP-GodotLib.tres

Script Class

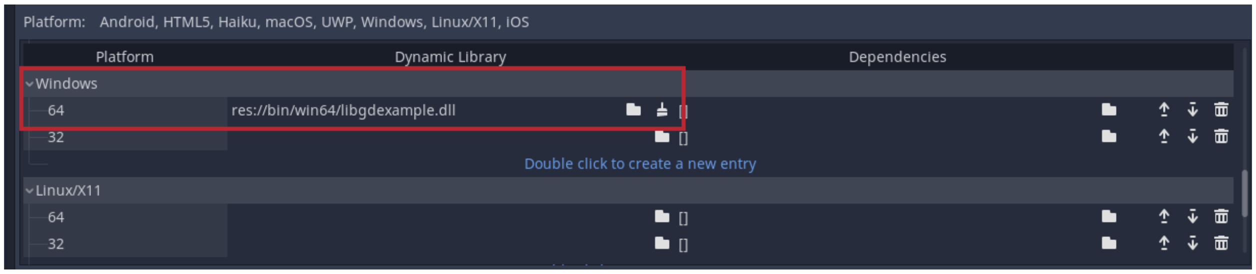
Name	
Icon Path	

Resource

Local To Scene	On
Path	res://PJSIP_AudioStream.gd
Name	

Reference

In the red box see that the following is set, furthermore click on library and make sure that the .dll is properly selected:



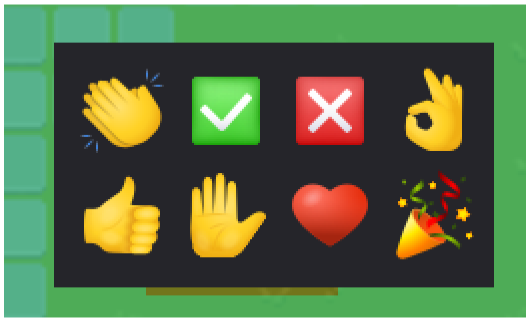
Platform: Android, HTML5, Haiku, macOS, UWP, Windows, Linux/X11, iOS

Platform	Dynamic Library	Dependencies
Windows		
64	res://bin/win64/libgdexample.dll	
32		
	Double click to create a new entry	
Linux/X11		
64		
32		

See:
https://docs.godotengine.org/en/stable/tutorials/scripting/gdnative/gdnative_cpp_example.html#doc-gdnative-cpp-example
For further explanation to GDNative and how the library is integrated into Godot.

Emotes

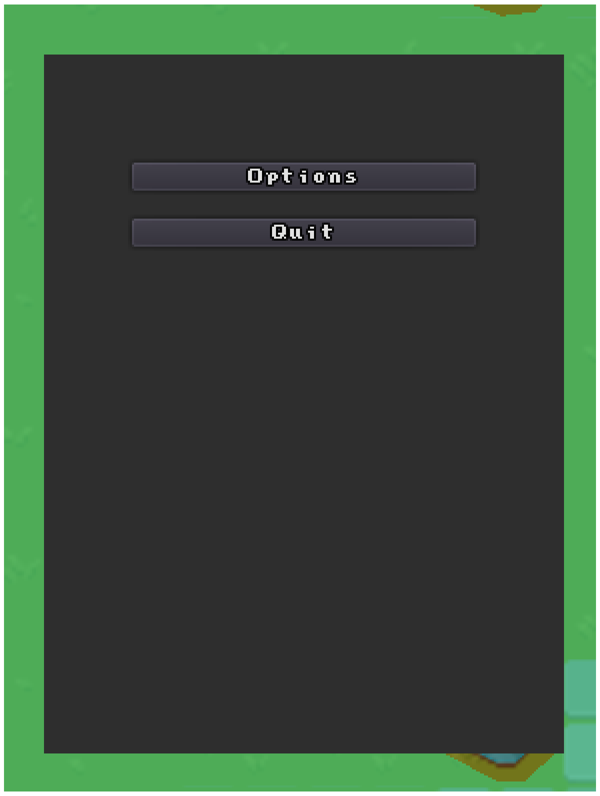
Emoting can be done through pressing the ‘E’ button in order to show the panel:



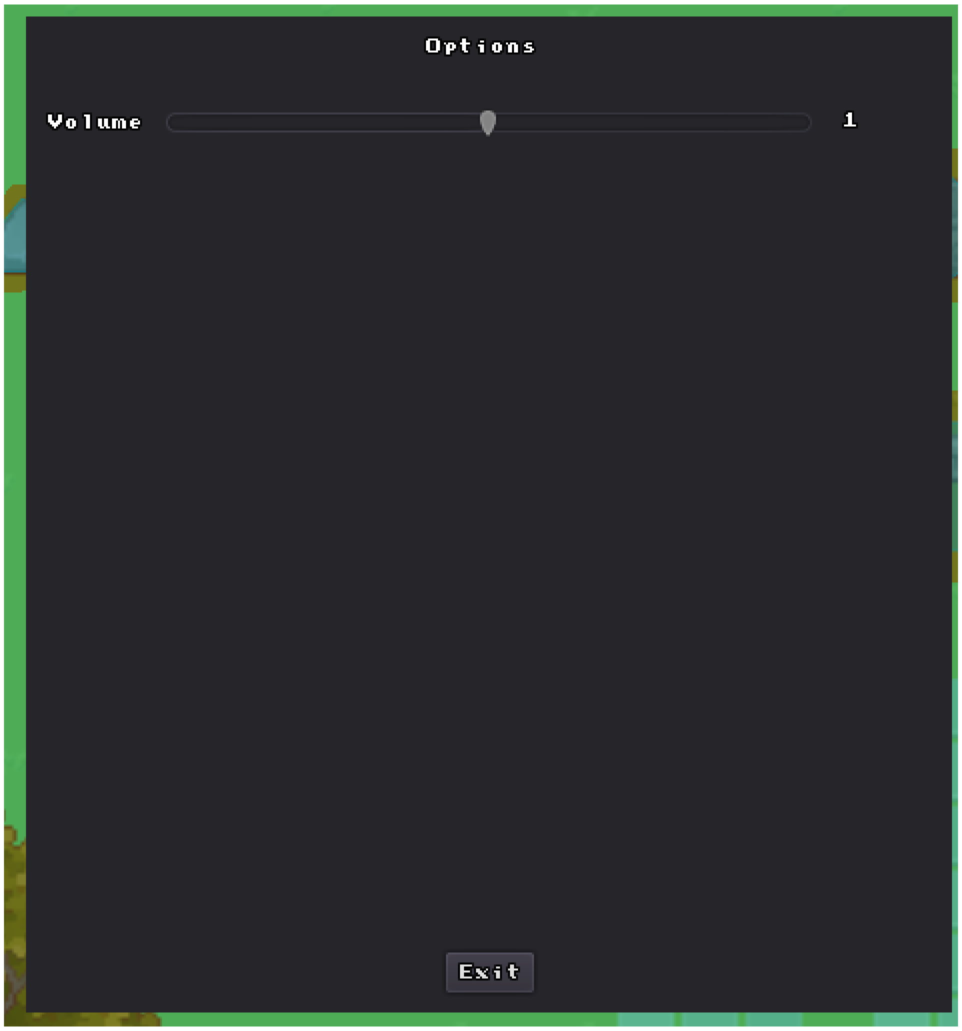
Pressing any of the emotes will result in your character emoting

Options

You can bring up the options menu by pressing ESC:



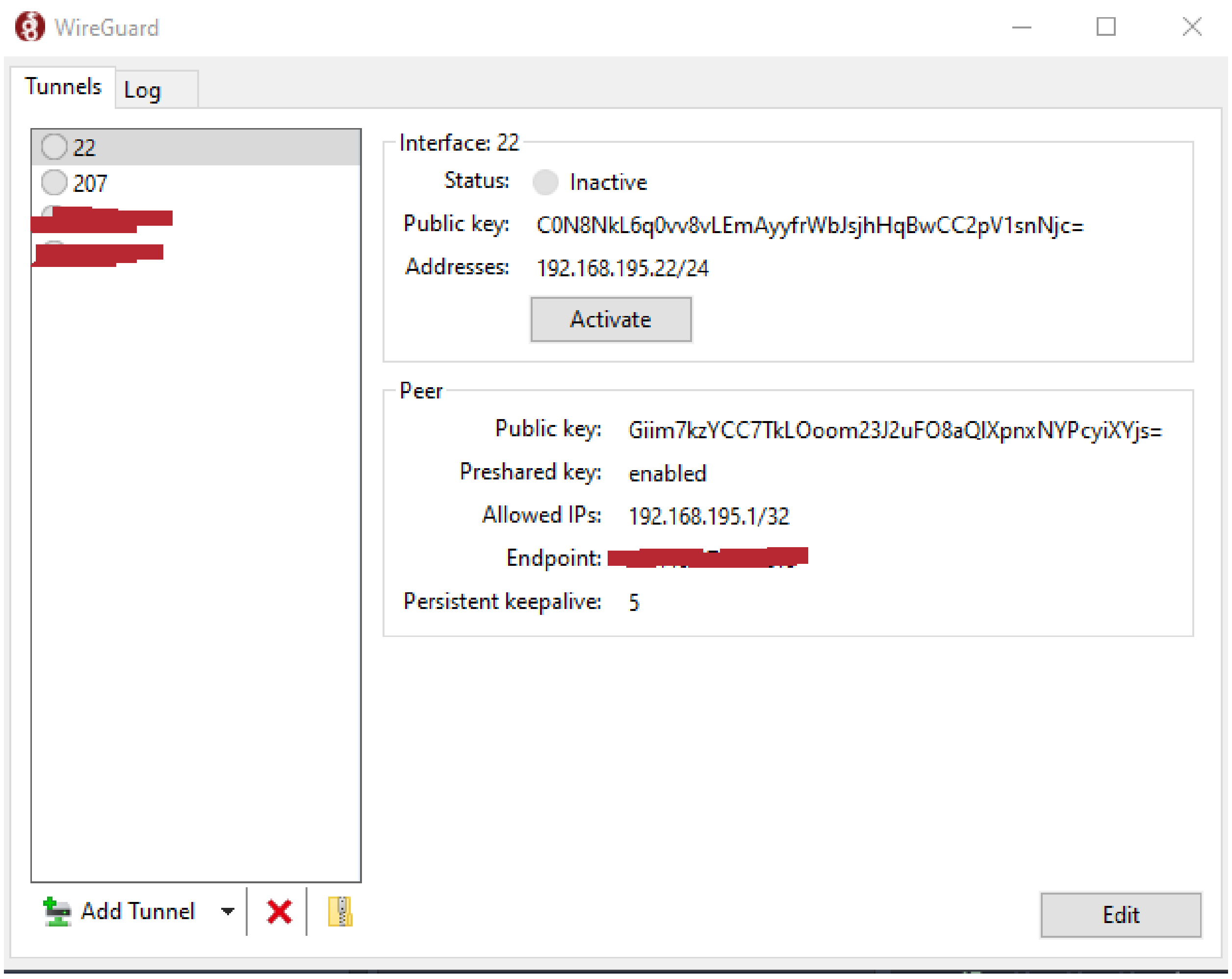
Pressing options will lead you to a menu where you can change the volume of the call:



Pressing quit will bring you back to the title screen.

Wireguard

Currently, [Wireguard]() is needed in order to tunnel through IP addresses and properly connect to other peers. A Wireguard configuration given by the conference holder should be then put into a wireguard client like so:


WireGuard

Tunnels | Log

22
207
[redacted]
[redacted]

Interface: 22

Status:	Inactive
Public key:	C0N8NkL6q0vv8vLEmAyyfrWbJsjhHqBwCC2pV1snNjc=
Addresses:	192.168.195.22/24

Activate

Peer

Public key:	Giim7kzYCC7TkLOoom23J2uFO8aQIXpnxNYPcyiXYjs=
Preshared key:	enabled
Allowed IPs:	192.168.195.1/32
Endpoint:	[redacted]
Persistent keepalive:	5

Add Tunnel

Edit

After activating the tunnel, one should then use the same configuration file put into the Wireguard client into the Calla client to properly connect to the call.

Function Signatures

Function signatures with relevant follow up documentation

Character.gd

func _ready()
func set_avatar(inputavatar: int)
func control(delta)
func generate_path(start: Vector2, end: Vector2)
func traverse_path(delta)
func set_navigation(nav: Navigation2D)
func _physics_process(delta)
func send_position()
func animate() https://docs.godotengine.org/en/stable/tutorials/2d/2d_sprite_animation.html

func emote(emote: int)
func set_username(name: String)

ControlledPlayer.gd

func _ready()
func player_arrow_controlled()
func _unhandled_input(event)
func control(delta)

EmotePanel.gd

func _ready()
func _input(event)

Map1Navigation.gd

a lot of this script was modeled after this video, this script is used to create the navmesh of the scene: Godot Navigation | Subtracting Collision Shapes from Navigation Pol…

https://docs.godotengine.org/en/stable/classes/class_navigation.html?highlight=navigation
https://docs.godotengine.org/en/stable/classes/class_geometry.html

func _ready()
func polyreduce(polygonarray)
func addpolystonavmesh(polygonarray)
func refreshnavmesh()
func trees()
func tileuse(Tilemap, ID)
func tileuseall(Tilemap)

Multiplayer.gd

main networking script, see rpc documentation:
https://docs.godotengine.org/en/stable/tutorials/networking/high_level_multiplayer.html
Also see signals documentation used to trigger functions on connect and disconnect:
https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html

func _ready()
func create_caller(callnumber:String, password:String)

```
func start_call()
func start_client(username, avatar, callnumber, password, roomname)
func create_main_instance(username = "user", avatar = 1, position = Vector2(20, 0))
func register_main_instance(instance)
func create_peer_instance(ID, username = "user", avatar = 1, position = Vector2(0, 0))
func connection_success()
func connection_failure()
func initiate_disconnect()
func reset_state()
func disconnected()
func update_player_position(position: Vector2)
func send_emote(emoteint: int)
remote func disconnect_me(id)
remote func set_user_data(userdata: Dictionary)
remote func update_all_user_positions(positions: Dictionary, timestamp: int)
remote func add_new_user(userid: int, username: String, avatar: int)
remote func set_client_emote(userid: int, emote: int)
remote func reg_status(success: bool)
```

OptionsPanel.gd

See signals documentation used to trigger _on_Button_pressed() and _on_HSlider_value_changed(value):
https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html

```
func _ready()
func _on_Button_pressed()
func _on_HSlider_value_changed(value)
```

PeerPlayer.gd

See tween documentation used to smooth player position updates:
https://docs.godotengine.org/en/stable/classes/class_tween.html

```
func control(delta)
func animate()
func update_player(position: Vector2)
```

PlayerCamera.gd

See Camera2D documentation:
https://docs.godotengine.org/en/stable/classes/class_camera2d.html

func _ready()
func set_player(instance)
func _process(delta)
func _input(event)

Resources.gd

See autoload documentation:
https://docs.godotengine.org/en/stable/tutorials/scripting/singletons_autoload.html

func _ready()

Settings.gd

func _ready()
func _input(event)
func _on_Quit_pressed()
func _on_Options_pressed()

SpeechBubble.gd

func _ready()
func startEmote(emotenumber)

TitleScreen.gd

See Control documentation as it is the base class for all UI related nodes:
https://docs.godotengine.org/en/stable/classes/class_control.html
See LineEdit documentation for text fields:
https://docs.godotengine.org/en/stable/classes/class_lineedit.html
See FileDialog documentation:
https://docs.godotengine.org/en/stable/classes/class_filedialog.html
Also WindowDialog:
https://docs.godotengine.org/en/stable/classes/class_windowdialog.html

```
func _on_JoinRoom_pressed()
func _on_ChooseConfig_pressed()
func on_UnsucessfulReg()
func on_disconnect()
func _on_FileDialog_file_selected(path:String)
```

That’s all, good luck!