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

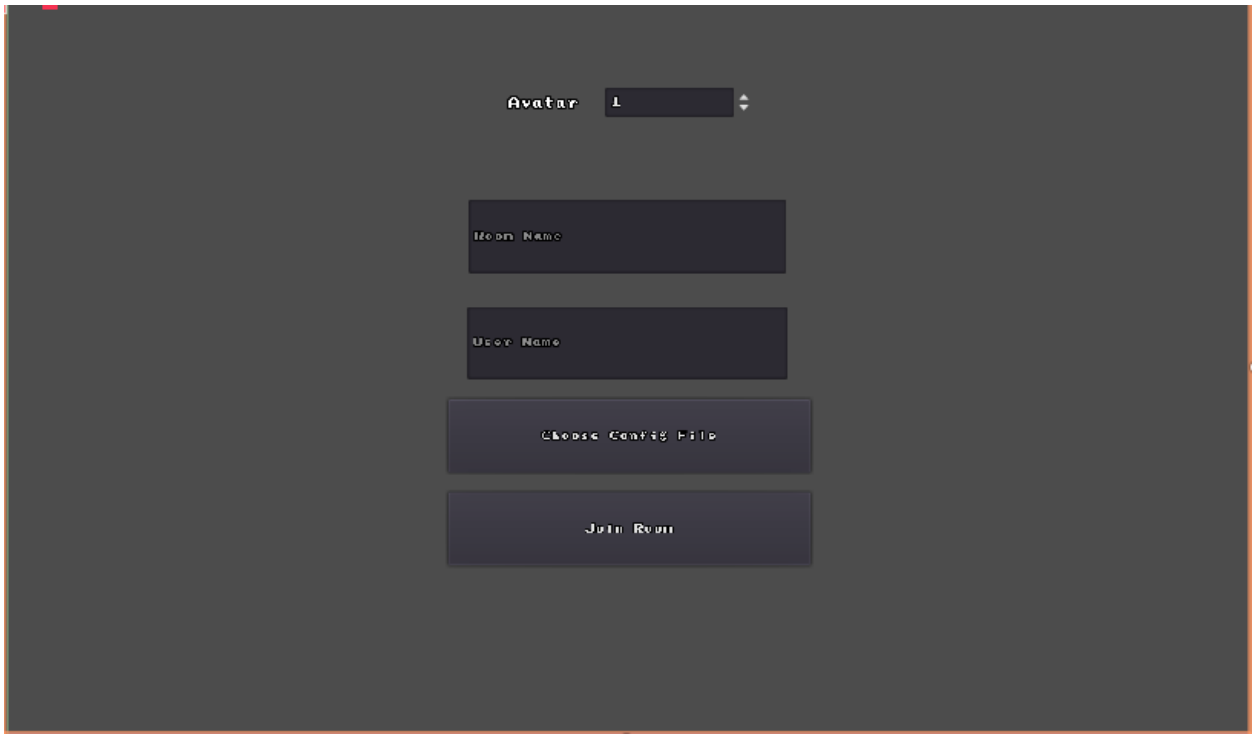
This script is responsible for properly showing emotes on characters and then hiding them after a timer is hit.

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.

Call Initialization

The client first starts with the titlescreen 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:

```gdscript
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 node.

```gdscript
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:

```
99  func connection_success():
100     # on successful connection, store ID, create and register main instance and try to make call
101     print("connection successful")
102     uniqueID = get_tree().get_network_unique_id()
103     var main_instance = create_main_instance(mainplayerusername, mainplayeravatar)
104     if not active:
105         pass
106         #client makes the call
107         #start_call()
108     register_main_instance(main_instance)
109
```

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.

```
81
82  func register_main_instance(instance):
83      # registers player controlled character to game server
84      rpc_id(1,"register_client", mainplayerpassword, mainplayerusername, mainplayeravatar, instance.global_position.x, instance.global_position.y, mainplayerr
85
```

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:

```
193 remote func reg_status(success: bool):
194     print(success)
195     if success:
196         active = true
197         emit_signal("registered")
198     else:
199         active = false
200         get_tree().network_peer.close_connection()
201         reset_state()
202         get_tree().change_scene_to(Resources.scenes["titlescreen"])
203         yield(get_tree(),"idle_frame")
204         yield(get_tree(),"idle_frame")
205         var alert = get_node("/root/TitleScreen")
206         alert.on_UnsucessfulReg()
207         print("reg unsuccessful")
208
```

If registration was a success, we have successfully initialized the call and can now go into the main loop of the protocol.