Documentation Vovracar

Kevin Mündel
Mobile Computing

Moritz Süß
Mobile Computing

Patrick Alves
Mobile Computing

***Abstract* — This is a documentation of our project Vovracar. This contains the birth of the idea, the materials and the whole process.**

I. Introduction

At the beginning we got presented with some interesting ideas for our project. „Robopets“ was one of them, but we had nothing to work with and with the limited time we have for our project, it meant we have to scale down a lot. We want to set up some basic applications, so other groups could build upon our ideas. The basic things a Robopet has to do is to react to an users voice and follow its orders. To follow its rules it has to move and the easiest thing to move is a model car. But this wasn't enough for us. We wanted to do something exciting. We had some prior experiences with VR applications, so it didn't take long for us to finalize our thoughts. We want to make a model car being voice controlled and using cameras on the car to stream the video to a VR headset. That was the birth of Vovracar.

II. The Beginning

A. Idea

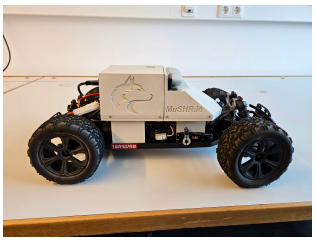
First of all we had to make some clear goals and functions we want to deliver. We want to build an application which can control a racing car through voice commands. The focus will be on simple commands such as „forward“ and „left/right“ and more. The cars speed should be controllable with voice inputs.

In addition we want to use the existing camera on the race-car to stream video to a VR headset. This enables the user to make better and more reasonable inputs. We will use the the VR headsets microphone to receive the voice commands.

These two function are our key functions. If we have some time left, we'll plan to use sensors to assist in controlling the car and reducing the number of collisions and accidents. Depending on the required distance between the car and the user, there may be a need to create a race track. With the already mentioned sensors we could establish checkpoints the user has to drive through.

B. Materials

As I already mentioned before we need a model car and we got provided with a race-car. That is a MuSHR race-car, which has a camera already installed. There is a Nvidia Jetson Xavier NX built in which is used as a built in PC. There are python scripts on that PC which were used for an app. We are allowed to use these scripts. We decided to use the Meta Quest 2 VR headset because we had some work experiences with that headset. We are also given a Windows PC to work with because we have reached some understanding during working with Unity and the Meta Quest 2. It is required to use a Windows PC, if we want to use the Oculus Integration SDK and the Oculus app. The usage of sensors was a goal we set up, if we get finished way faster, but that was optimistic.



C. Concept

Before getting to work we had to decide on some concepts and design decisions. There are two ways to control a race-car. It is either dynamic or static. Controlling the car in static way would mean that the user needs exact information about the distance and direction the car needs to drive. That's why we decided to use dynamic controls. That means the car uses one command and executes that command until „stop“ or another command is put in.

Another thing we had to think about is what application do we want to produce. We had two choices. We could make a web application or an unity application. After some research we decided to make a unity application because we already have some experiences working with a VR headset in unity. It's easier to create an Android application in Unity, which is getting launched on the VR headset. We need Unity to translate the input by the VR headset and send it in a way to the race-car, so the python scripts can be used. This has to work the other way too because the camera can be controlled through a python script which needs to send the video to the Unity application. The video needs to be made usable for the Quest 2, there..

III. Development

A. First Approach

The first plan we had involved installing ROS. To do that we had to install Docker and make some initial set up. We needed an Nvidia graphics driver to be able to run ROS on the system. Unfortunately, this driver was not available for macOS, so we had to find another solution. During that time we started to realize how valuable the existing python scripts are and we cancelled the idea of using ROS from scratch. After failing the first approach we decided on testing the existing python scripts.

B. Testing the python scripts

We had a lot of scripts on the race-car and it took some time to get through them. We found two scripts which were usable for us. The first one was named „car.py“ and the second one was called „carServer.py“. These scripts contain all the controls for the race car and got used in an app developed by another student. With the first script we were able to test the controls via the command line, while with the „carServer.py„ script the controls were controllable via the API endpoints.

We encountered two problems while testing the scripts. The car didn't react to any input we made, but after changing some ports we were able to give commands via console. The other problem we had to solve is the camera stream. Firstly, we tested the camera functionality in the browser which didn't work at the beginning. After some bug fixing we got the video, but only in form of pictures and not as a video. We had to rewrite the script to get a video stream. That worked and we could get the video stream on our MacBooks.

C. Meta Quest 2

After checking the scripts, we wanted to put all the pieces together. But we didn't expect the following problems. The racecar has a Linux operating system and we wanted to install the Oculus SDK onto the racecar. This didn't work out because it was only available for Windows, but for MacOS it still had some problems.

We skipped testing the VR headset for now and split ourselves in two groups. We split the following tasks into two parts. The first task was to adjust the python scripts to be used by the Unity application and the second task was to set up a Unity project and do the basic implementations to use the VR headset in our project.

D. First problems and challenges

At this point we got access to a Windows PC because working with Unity on our MacBooks didn't seem to work without problems. We wanted to install the newest Nvidia graphics driver on the PC to be able to use SteamVR, which would get us access to the Meta Quest 2.

We also changed the camera functions on the Flask server. This had to be done because the functions only send pictures, but no video. Setting up the Unity project went according to our plans and we could already do changes to the Unity script. This enabled us to use the camera and speech recognition.

But the next problems appeared. After installing the Oculus Integration SDK we got the info that it was deprecated since a few months. To use the Meta Quest 2 in combination with Unity we had to put it into developer mode. For this, we installed the Meta Quest App on a smartphone to connect it with the Meta Quest 2 via Bluetooth and put the headset into developer mode. We also had to use our own Meta account on the Quest 2 for this. This worked after deleting the existing accounts on the Meta Quest 2. On the PC, we installed the Meta Quest Developer Hub with which we could launch apps on the Meta Quest 2.

Now that we had installed the necessary software, we wanted to display the camera stream from the car in our Unity project. To do this, we tried to display the stream in our app, which we could retrieve using a get request via the API. Our first approach was to convert the video stream into a byte array, but this didn't work at first.

At this point in time we thought about scrapping the VR component of our project because it seemed like too much of a task to make it work.

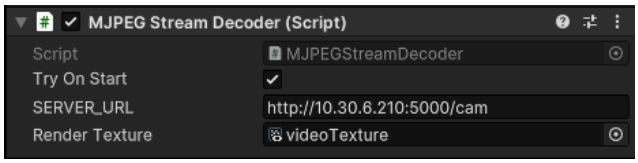
E. Access to video stream finished

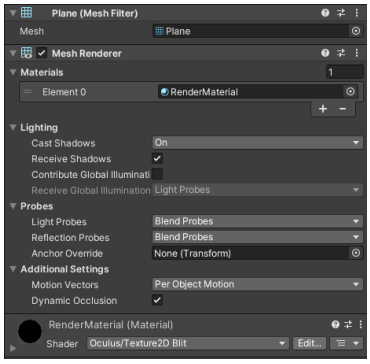
But after a short replanning we changed our approach. Firstly we completely set up our unity project in the most basic way. We didn't want to run into any problems we aren't responsible for.

Then we checked again the way to connect to the racecar and using its scripts. We also confirmed our suspicion that we have a little setup problem. If we want to start the scripts on the racecar through our application, we will need a SSH connection to start the scripts or something similar. Because that isn't that important, we decided to work on voice commands and camera stream before giving that more attention.

The camera stream was accessible through HTTP requests which worked just fine when accessing the racecar from our MacBook. But we had difficulties setting it up in unity. After some testing and debugging we managed to make it work.

With the help of a MJPEG stream decoder, which we found on GitHub, we could stream the camera from the racecar to the unity project. This decoder let us take the video-stream and converted it into 2D Texture. We set up the plane as a child of the camera, so whenever the user moves his head he will always see the screen in front of him.





We created a Render Material and selected the standard Shader at first, which didn't work until we found out that the correct Shader was the Oculus/Texture2D Blit. After that, we added this Render Material to our Mesh Renderer on our plane object. We added the MJPEG stream decoder to the parent object which contains our plane. This stream decoder streams the camera from the race-car to our video texture.

F. Voice Control

With the first task completed we shifted our attention to the voice control part. At first we tried a keyword recognizer from Unity's own Windows.Speech library. With this method, however, no POST requests were sent and we thought that this was due to the microphone. Therefore, we used Unity's microphone interface to get access to the microphone of the VR goggles. After the microphone is started, it should record for ten seconds. This recording should then be converted to a byte array and sent to Wit.ai.

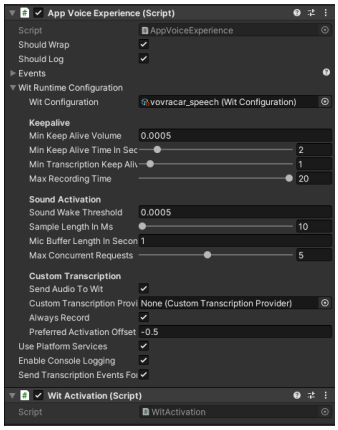
But first we had to set up Wit.ai which is a NLP model from Meta that can process speech. On the Wit.ai website we created an app called vovracar_speech and set the visibility

to „open“. In this Wit-app we defined a few utterances. Based on these we defined appropriate entities and intents.

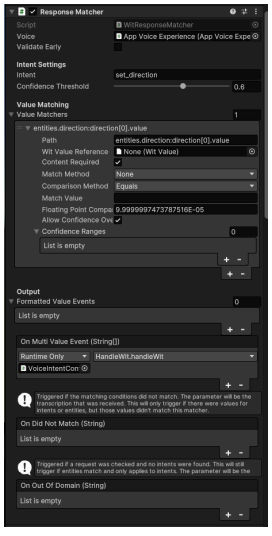
Entities are key terms or concepts that are to be identified in the texts entered by the user. They represent the important information that is to be extracted from the user input. In our case, the entities contain keywords such as "start" or "stop", i.e. the commands for controlling the car. Intents represent the intention or purpose behind the user input. They describe what the user wants to achieve with their input. By combining entities and intents, an NLP model can understand what the user wants to say and which actions should be executed based on this. When an input is sent to Wit.ai, it is checked for the keywords and a response gets sent back in JSON format. This response contains the respective entities that were found in the voice input.

Entity	Roles	Intents
action	action	set_action, set_direction
direction	direction	set_direction, set_value
value	value	set_action, set_direction, set_value

Name	Entities
set_action	action:action, value:value
set_direction	value:value, direction:direction, action:action
set_value	value:value, direction:direction



The Meta Voice SDK, which is included in the Meta All-in-One SDK, has already provided us with scripts for using Wit.ai. Wit.ai first had to be configured in Unity, for example the personal server access token of our Wit-app had to be entered in order to gain access to the NLP model. To be able to receive responses from Wit.ai at all, a App Voice Experience is required as a game object. This contains the Wit Configuration, which is started when the Unity app is opened via a script "WitActivation.cs" created by ourselves.



Since we still had no access to our microphone, we wanted to try using the LipSync demo included in the Meta Voice SDK. This allowed us to check if Unity could in theory get access to the Meta Quest microphone. When it worked in this demo project from Meta, we hoped to be able to implement it in our project as well. Because that didn't work either, we planned to use a self-defined speech-to-text script instead of text verification with Wit.ai. This was to convert the audio input from the microphone into text, which we could then check for the keywords.

After this didn't work either, we found out through a YouTube tutorial that the Meta Voice SDK automatically uses the correct microphone and that our error must lie elsewhere. It turned out that we were not using Wit.ai correctly. To handle the requests correctly, we needed response matchers in our Unity project to provide us with the responses from Wit.ai.

This responses are then passed to our "HandleWit.cs" script or its "handleWit" method. The method checks the passed response for its value and then executes the corresponding POST request within a coroutine to the Flask server. These POST requests can be used to control the end points of the car's API, which then influence the motor and wheels.

G. Further improvements

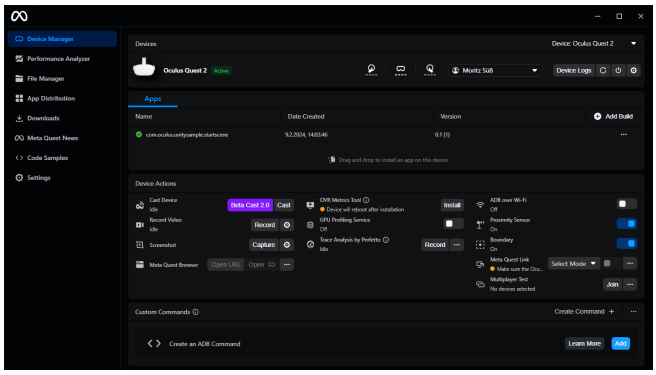
Now that the app was basically working, we wanted to implement further functionalities. On the one hand, we wanted to be able to specify the angles for steering to the left or right and the desired driving speed as a number in the respective voice commands. Secondly, we wanted our car to be able to drive certain shapes such as circles or a figure of eight using voice commands.

However, processing the angle or speed turned out to be difficult, as the handleWit method only ever receives one value at a time. As the number is an additional value alongside the direction, for example, it cannot be processed in the same interation of the handleWit method and therefore cannot be passed in the POST request. Instead of using request handlers, we tried to use utterance handlers in order to be able to process not only the value, but also the entire language input including the number.

IV. Instructions

First of all the car must be started. An SSH connection is used to access the car. The script can also be started on the car, in which case a screen, mouse and keyboard must be connected. When starting the vovracar.py script via the terminal, it is possible that the port 5000 on which the script will run must first be cleared. This will be indicated in a message in the terminal. Clearing the port (in our case 5000) is done with the command "lsof -i:[Port]". The application running on the port that is displayed must then be terminated with "kill [PID]". The script can then be executed with "python vovracar.py" in the "Desktop/Vovracar" directory.

If the app is to be started on the Meta Quest 2 via the PC, the Meta Quest Developer Hub is required on the PC. You must log in to the Meta Quest Developer Hub with a Unity Developer account, which is also used to log in to the VR goggles and the Meta Quest smartphone app. With the Meta Quest smartphone app, the VR glasses must be set to developer mode so that apps can be installed on them via USB from the PC.



If no APK file of the Unity project exists yet, it can be built under File / Build Settings in Unity. Make sure that the

app is created for Android, that "Use Player Settings" is selected for Texture Compression, that the correct device is selected and that the Compression Method is set to "LZ4". The APK file created must be added as an app in the Device Manager of the Meta Quest Developer Hub. Then the app can be launched from there.

If there is already an APK file on the Meta Quest 2, it can be started in the menu under Library. To find the file easier in the library, you can filter for "unknown sources".

The voice commands "left", "right", "forward", "backward", "start" and "stop" are available for controlling the car. "Left", "right" and "forward" change the angle of the front axle, while "start" and "backward" move the car forwards and backwards respectively. "Stop" ends all current actions and brings the car to a standstill.

V. Potential Problems

There could be problems regarding the IP address. We encountered cases where the IP address of the race-car changed. We didn't find a solution for this case. If the car doesn't connect to the application, you'll probably have to change the IP address in the Unity script.

There may also be connection problems between the car and Meta Quest 2, especially when there is a lot of network traffic. The exact cause of this is unknown, but it is most likely a network problem.

Voice input should be in English, commands in other languages will most likely not be processed correctly by Wit.ai. In addition, problems can occur with unclear voice commands.

It must be ensured that the microphone of the Meta Quest 2 is not muted and that the app is allowed to access the microphone.

VI. Future Projects

We made a pretty basic application, so improving the application in any way would be ideal. It doesn't have a UI. Displays for speed and battery life would be very helpful. A warning pop up when driving too far away form the user would be a good addition.

The voice commands could be trained more and new commands could be added. Our commands are set to have default values, so there could be improvements, which could led to the user being able to make specific left and right turns or decreasing and increasing the speed while driving.

VII. Lessons Learned

At the start of the project we thought that this would be way an easier task. We have worked on an unity project with VR integration before, so we thought that adding voice commands and letting a race-car drive would be the only challenges where we needed to invest a lot of time.

But this assumption couldn't be more wrong. The biggest problem we had to face is the incompatibility of operating systems were working on. We realized pretty fast that the project is very hard to do on Macs and with the help of a separate PC, which we got later, we managed to progress more smoothly. But the most annoying thing is that Meta devices and applications don't support Linux, but our race-car only has a Linux server installed. Therefore we needed to make an unity application to connect the Meta Quest 2 with our race-car's Linux server.

This led us to have regular complication and some of these were very special interactions. We don't really know whether more planning would have changed something. The one thing we learned from this is that we are wary of using different operating systems in combination.