UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE IGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA INFORMÁTICA

ADAPTACIÓN DE CHATBOTS A AVATARES VIRTUALES CON METAHUMAN

COMUNICACIÓN DE CHATBOT CON UNREAL ENGINE Y CONVERSIÓN TEXTO A VOZ CON API DE READSPEAKER

ADAPTING CHATBOTS TO VIRTUAL AVATARS WITH METAHUMAN

CHATBOT COMMUNICATION WITH UNREAL ENGINE AND TEXT-TO-SPEECH CONVERSION WITH READSPEAKER API

Realizado por

Ignacio Jiménez del Castillo

Tutorizado por

David Bueno Vallejo

Departamento Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, FEBRERO 2022

1 Resumen

En la actualidad se está extendiendo el uso de chatbots y asistentes a través de páginas web, whatsapp, algunos con voz como el Asistente de Google, Cortana o Siri. A todos ellos les falta un toque más humano que daría una interfaz 3D. En este proyecto se ha trabajado para dar una interfaz 3D genérica en formato de Avatar a esos chatbots. Utilizando tecnologías de texto a voz, analizando los movimientos de la boca que estarían asociados a cada sonido (visema) para generar una interacción en tiempo real lo más realista posible.

Este Proyecto ha consistido en el desarrollo de un avatar virtual 3D en Unreal Engine 4 utilizando la nueva tecnología MetaHumans.

Este avatar puede conectarse con un chatbot y, utilizando la tecnología ReadSpeaker Text to Speech, puede obtener los pares de audio y duración de visema a partir de la respuesta del chatbot. El avatar utiliza estos resultados para generar una animación facial de la respuesta.

Este TFG en concreto se encarga del envío del texto al Chatbot así como del procesamiento de la respuesta para obtener el audio y la información de los visemas.

Todo esto se hace con el propósito de ofrecer a los usuarios de este tipo de servicios una experiencia más realista, aprovechando las tecnologías más modernas de generación de voz y creación de avatares virtuales.

En la memoria también se detallará como utilizar cada tecnología que ha sido empleada para el desarrollo de este proyecto, ya que gran parte de este ha consistido en aprender a utilizar tecnologías que se encuentran todavía en desarrollo.

1.1 Palabras clave

ReadSpeaker, ChatBot, Visemas, Text-To-Speech, Unreal Engine, Avatares 3D

2 Abstract

Nowadays the use of Chatbots is being expanded through web assistants, whatsapp and some Voice assistants like Google Assistant, Cortana or Siri. A lot of them lack a bit of human touch that could be provided by a 3D interface. In this project we have aimed to give a generic 3D interface to those Chatbots by adding an Avatar. Using Text-to-Speech and analyzing facial movement associated to each sound (viseme) we aim to generate the most realist real time interaction posible.

This Project has consisted in the development of a 3D virtual avatar in Unreal Engine 4 using the new MetaHumans technology.

This avatar can connect with a chatbot and using ReadSpeaker Text to Speech technology, it can obtain the audio and viseme-duration pairs from the answer of the chatbot. These results are used by the avatar to generate a facial animation of the answer.

This project is responsible of sending the input text to the chatbot and processing the answer provided by it, obtaining this way the audio and viseme information.

All of this is done with the purpose of offering the users of this kind of service a more realistic experience, using the most recent voice generating and virtual avatar technologies.

In this document it will be also detailed how to use each technology that has been used in the project, because a big part of it has been the learning process of using technologies thar are still in development.

2.1 Keywords

ReadSpeaker, ChatBot, Visemas, Text-To-Speech, Unreal Engine, 3D Avatar

3 Contenido

4 Introducción

4.1 Motivación

Los chatbots son sistemas conversacionales capaces de simular cierta inteligencia al conversar con un usuario.

La mayoría de ellos son textuales (Whatsapp, Telegram o Web) o con Voz (Alexa, Cortana o Asistente de Google). En pocos casos, estos asistentes están representados por un Avatar 3D y esta representación, si existe, suele ser de baja calidad: poco realista y con poca o ninguna expresividad.

En este contexto hemos visto que recientemente ha aparecido la novedosa tecnología de MetaHumans, que son avatares muy realistas con cientos de atributos asociados al rostro que permiten generar expresiones de la cara y boca que conectados a un chatbot supondrían una experiencia muy realista al usuario.

El objetivo de estos proyectos es preparar estos avatares para poder asociarles un chatbot. Se utilizará como prueba el chatbot Victoria la Malagueña, que fue desarrollado en Dialogflow para el Asistente de Google por nuestro tutor con el propósito de ofrecer información sobre la ciudad de Málaga a los turistas y ciudadanos.

4.2 Objetivos

Al tratarse de un trabajo en grupo, los objetivos se pueden separar en dos categorías:

- **Grupal**: El objetivo final del trabajo es realizar un avatar virtual en 3D (utilizando Metahumans de Unreal Engine) capaz de conectarse con un chatbot a través de servicios JSON. Las frases en texto que se reciban del chatbot, se convertirán en voz con la API de textToSpeech de ReadSpeaker. Analizando el audio y el texto se generarán las posiciones de la boca (visemas) y el Avatar

reproducirá el sonido asociado. En el otro sentido, se estudiará la entrada de texto para comunicarse con el avatar.

- **Individual**: De todo el proceso descrito anteriormente, en este proyecto se recibirá la salida de texto del Chatbot a través de JSON y se enviará a la API readSpeaker para así obtener el audio y los visemas que procesará en Unreal Engine el otro proyecto. También se hará la comunicación hacia el chatbot para enviarle la entrada del usuario.

4.3 Metodología

Debido a la parte tan alta de innovación del proyecto y a que hay más de una persona implicada en el mismo, la metodología a utilizar ha sido una metodología Ágil basada en Scrum con Sprints de unos 15 días. Se definirá el Product Backlog con las tareas que se detecten de todo el proyecto y se irán asignando a los distintos Sprint.

4.4 Herramientas utilizadas

Para este proyecto se han utilizado estas herramientas ordenadas por orden alfabético:

- **CodeBlocks**: CodeBlocks es un IDE gratuito de C/C++ y Fortran fácil de usar y de configurar además de extensible mediante *plugins*. Es el IDE que se ha utilizado en este proyecto debido a su sencillez y a la familiaridad con este al ser uno de los IDEs que se utilizan en la UMA.

- **Inih:** el proyecto inih [1], que consiste en un archivo `.cpp` y un *header*, nos permite manejar de forma sencilla los archivos `.ini` que para este TFG se han usado con la finalidad de guardar parámetros de las funciones de la API de ReadSpeaker además de los nombres de los distintos directorios que se necesitan para el correcto funcionamiento de nuestra API. Se ha optado por esta forma de pasar los parámetros para así poder mantener la interacción del usuario con nuestra API lo más sencilla posible teniendo que introducir solo el texto del cual quiera conseguir el audio y los visemas o, en caso de querer comunicarse con un Chatbot, la url, protocolo, token y texto de entrada, sin renunciar a que aquel usuario que desee interactuar más con la API de ReadSpeaker pueda hacerlo.

- **libcurl:** libcurl [2] es una librería de transferencia multiprotocolo de archivos gratuita que soporta un gran número de protocolos de transferencia, certificados, *proxis*, *cookies* y mucho más. Es muy portable y está disponible para la mayoría sino la totalidad de sistemas.

- **Live Link Face**: Aplicación de iOS para captura facial [3]. Esta herramienta se encuentra en desarrollo, por lo que todavía no es capaz de grabar fielmente los movimientos faciales. Se ha utilizado para la grabación de las diferentes animaciones de visemas sobre el MetaHuman.

- **nlohmann json**: Este proyecto [4] el cual podemos encontrar en este repositorio de github ofrece una solución sencilla, intuitiva y bien documentada al manejo y creación de objetos JSON [5] en C++.

- **Postman**: Postman [6] es una herramienta que nos permite crear peticiones sobre APIs de una forma muy sencilla y poder, de esta manera, probar las APIs. Todo basado en una extensión de Google Chrome. Postman proporciona herramientas para documentar los APIs, realizar una monitorización sobre las APIs, crear equipos sobre un API para que trabajen de forma colaborativa y otras muchas más. En este proyecto se ha usado para simular peticiones a un Chatbot creando para ello un *mock* de un servidor y añadiendo peticiones customizadas que permitieran probar la conexión de la API con internet.

- **Quixel Bridge**: Se trata de una herramienta que permite el manejo de diferentes *assets* 3D y su importación a diferentes programas. Para este proyecto se ha utilizado para importar un MetaHuman a Unreal Engine.

- **ReadSpeaker**: ReadSpeaker [7] es la pieza central de este TFG y gracias al cual podemos obtener tanto los visemas como el audio correspondiente al texto del cual queramos obtener dichos atributos. ReadSpeaker ofrece soluciones de voz sintética y *text-to-speech* que pueden ser aplicadas a los sitios web, las aplicaciones móviles, los libros digitales, las herramientas de aprendizaje electrónico y los documentos en línea entre otros para dotarlos de voz propia haciéndolos más atractivos e interactivos.

- **Unreal Engine**: Motor gráfico para videojuegos y producciones virtuales. Gracias a Unreal Engine se ha podido hacer uso de los nuevos avatares hiperrealistas de Metahuman a los cuales se les ha dotado de movimiento facial gracias a animaciones generadas a partir de grabaciones faciales utilizando Live Link Face. Se ha utilizado la versión 4.27 como entorno de trabajo para el proyecto.

En los siguientes capítulos se realizará una breve explicación o tutorial de uso de aquellas herramientas o aspectos de las herramientas que durante el desarrollo de este TFG requirieron de cierto grado de investigación, fueron propensas a generar errores, eran poco intuitivas, resultó complicado obtener la documentación adecuada o de las que no son frecuentemente utilizadas ya sea por pertenecer a un ámbito relativamente específico o por su novedad. Dichos tutoriales están ordenados por orden alfabético según el nombre de la herramienta.

5 Guías de las herramientas utilizadas

5.1 Cómo utilizar librerías y headers en Codeblocks

CodeBlocks ha sido el IDE que se ha utilizado en este proyecto debido a su sencillez y a la familiaridad con este al ser uno de los IDEs que se utilizan en la UMA.

Este capítulo pretende servir de breve guía a aquel usuario que no haya usado este IDE con anterioridad.

5.1.1 Incluir librerías

Para incluir una nueva librería seguiremos los siguientes pasos:

1. Hacemos clic derecho en nuestro proyecto -> "Build Options".

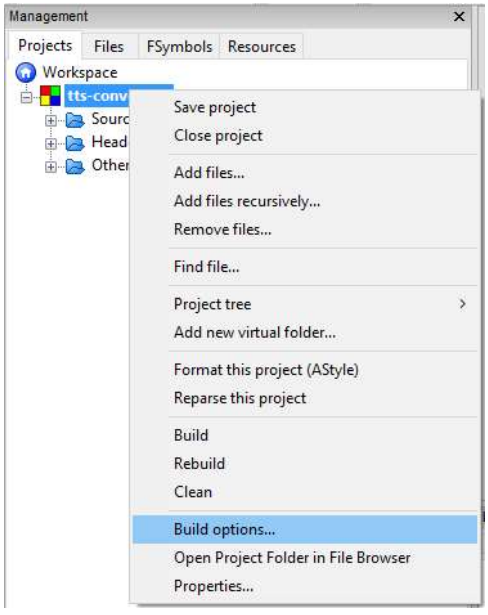


Ilustración 1: Menú del proyecto

2. Nos dirigiremos a la pestaña "Linker Settings".

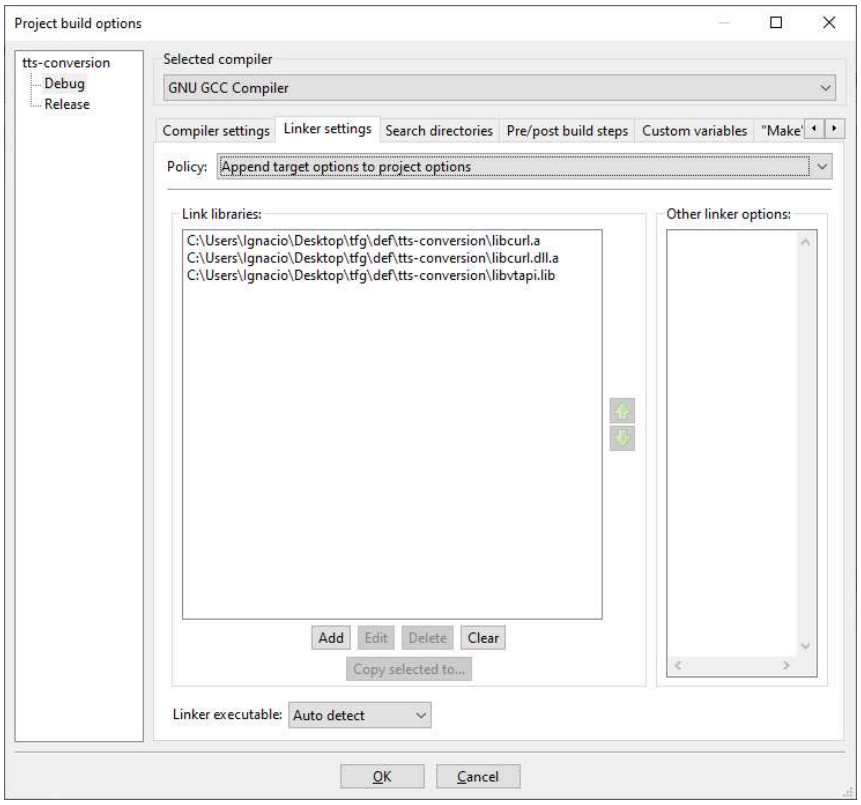


Ilustración 2: Pestaña Linker Settings

3. Hacemos clic en "Add" y seleccionamos el icono de la carpeta para acceder al explorador de archivos.

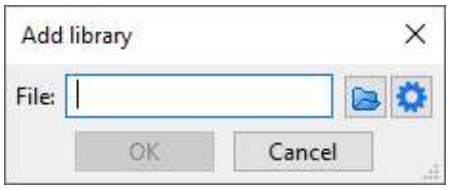


Ilustración 3: Menú Add library

4. Seleccionamos la librería o las librerías que queramos añadir desde el explorador de archivos y hacemos clic en abrir.

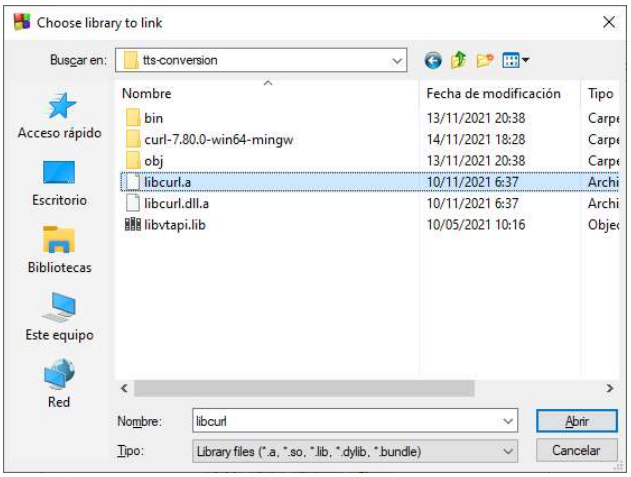


Ilustración 4: Elegimos la/as librería/as a añadir

5. Se nos mostrará una ventana preguntando si queremos mantener la ruta como ruta relativa, esta decisión dependerá del usuario, en nuestro caso y ya que ha sido la opción que se utilizó durante este TFG haremos clic en no, guardando así la ruta como ruta absoluta dentro de los archivos.

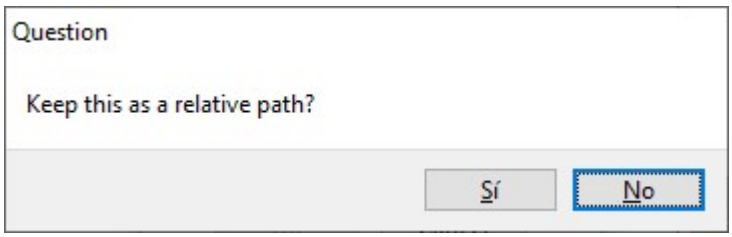


Ilustración 5: Opción de guardar la ruta como relativa

6. Hacemos clic en Ok para añadir la/as librería/as y volvemos a hacer clic en Ok más abajo para confirmar los cambios.

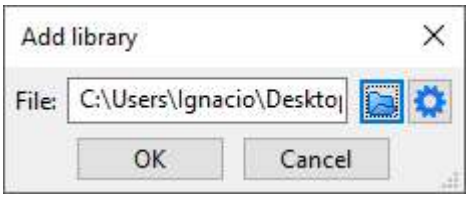


Ilustración 6: Librería seleccionada

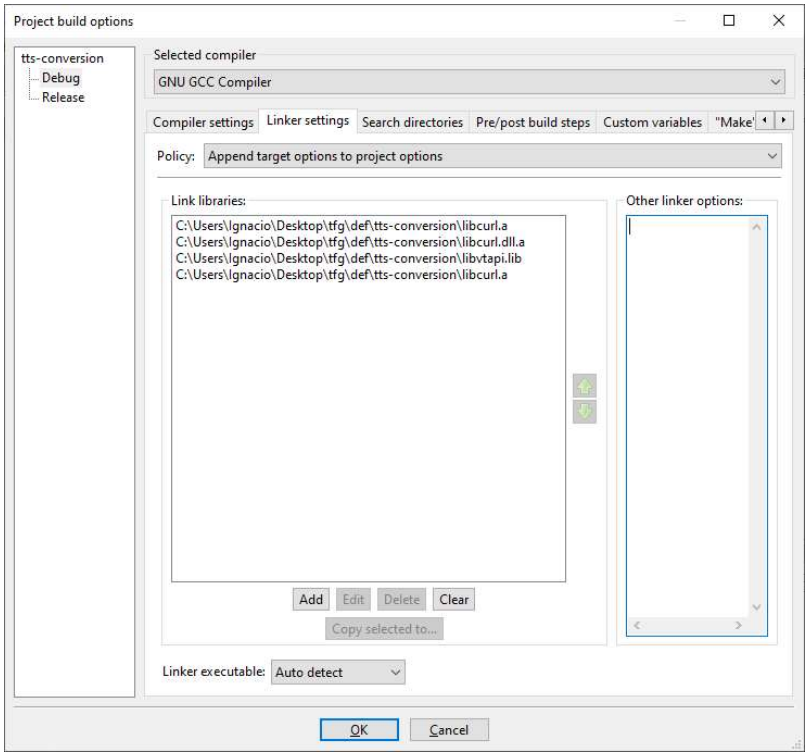


Ilustración 7: La librería ha sido añadida y confirmamos los cambios

5.1.2 Incluir headers

La forma más sencilla de incluir *headers* y la que se ha usado durante todo el desarrollo del TFG es colocándolos en el directorio donde está situado nuestro proyecto y escribiendo en el archivo donde vamos a necesitar las declaraciones `#include` "archivo_a_incluir.h" donde archivo_a_incluir.h será el archivo.h que queramos incluir.

5.2 Guía básica de inih

El proyecto inih [1] nos permite manejar de forma sencilla los archivos `.ini` que para este TFG se han usado con la finalidad de guardar parámetros de las funciones de la

API de ReadSpeaker además de los nombres de los distintos directorios que se necesitan para el correcto funcionamiento de nuestra API.

5.2.1 Ficheros .ini

Los ficheros `ini` también conocidos como ficheros de configuración son ficheros que como su nombre da a entender tienen como función guardar parámetros de configuración. Son esencialmente archivos de texto y permiten comentarios.

Tienen una estructura muy sencilla cuyos elementos centrales son:

- Secciones: las cuales permiten agrupar datos relacionados entre ellos
- Pares Nombre-Valor: que asocian un valor a un nombre para poder ser identificado posteriormente.

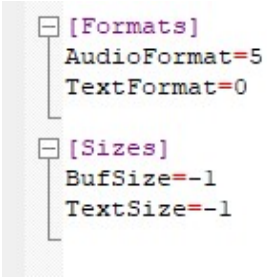


Ilustración 8: Estructura de un fichero ini

En la imagen anterior podemos apreciar un fichero `ini` con dos secciones *Formats* y *Sizes* ambas con dos pares Nombre-Valor.

5.2.2 Ficheros a incluir

El proyecto inih [1] cuenta con dos *headers* y dos archivos *sources* los cuales son `ini.h` y `ini.c` para el lenguaje C y `INIReader.h` y `INIReader.cpp` para C++.

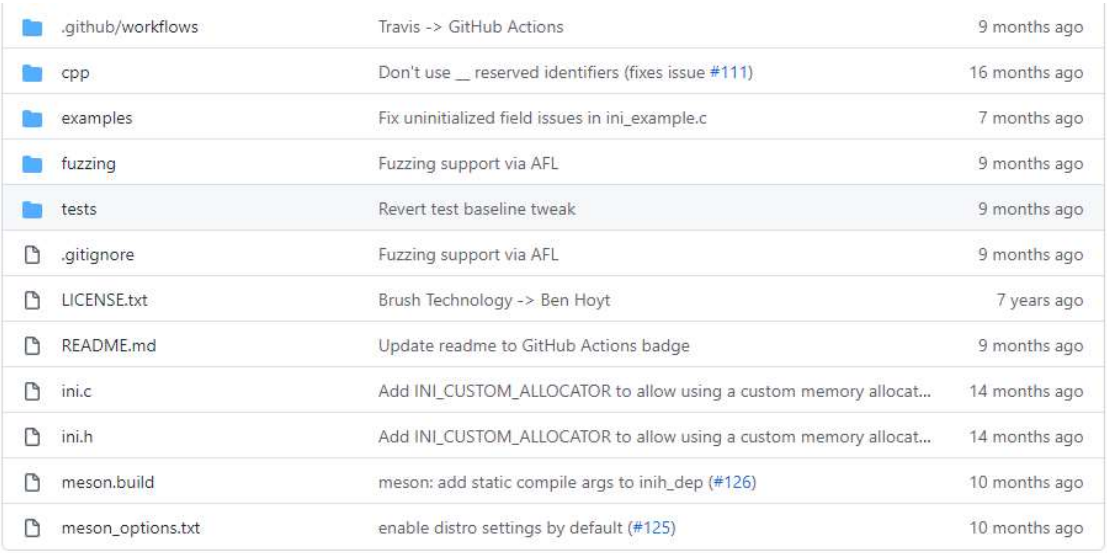


Ilustración 9: Projecto inih en github [1]

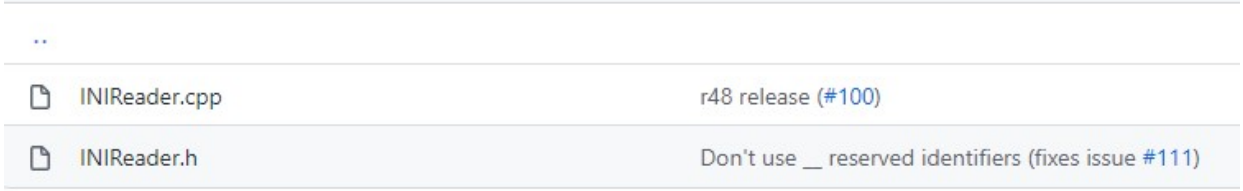


Ilustración 10: Ficheros a incluir dentro de la carpeta cpp [1]

5.2.3 Uso básico de inih

Una vez añadidos los ficheros necesarios podremos hacer uso de los métodos [1] que nos permitirán interactuar con los archivos `.ini`. Dentro del repositorio GitHub podemos observar el siguiente código de ejemplo en el que podemos ver algunos de los métodos y el tipo de dato que devuelven:

```cpp
#include <iostream>
#include "INIReader.h"

int main()
{
    INIReader reader("../examples/test.ini");

    if (reader.ParseError() < 0) {
        std::cout << "Can't load 'test.ini'\n";
        return 1;
    }
    std::cout << "Config loaded from 'test.ini': version="
              << reader.GetInteger("protocol", "version", -1) << ", name="
              << reader.Get("user", "name", "UNKNOWN") << ", email="
              << reader.Get("user", "email", "UNKNOWN") << ", pi="
              << reader.GetReal("user", "pi", -1) << ", active="
              << reader.GetBoolean("user", "active", true) << "\n";
    return 0;
}
```

Ilustración 11: Ejemplo de inih en C++ [1]

Como se puede observar comienza llamando al constructor INIReader al que se le pasa como parámetro el archivo `.ini` del que se sacarán los datos. Si continuamos leyendo podremos ver varios métodos que contienen la palabra `Get` a los que se les pasa como parámetro la sección, el nombre del valor al que queremos acceder y un valor por defecto en caso de no encontrar dicho par sección-nombre.

Hay varios métodos `Get` dependiendo del tipo de dato que queramos obtener del `.ini`, en el ejemplo se pueden apreciar métodos `Get` para los tipos de datos básicos, siendo el método `Get` que no va acompañado de ningún nombre de dato el método que usaremos para conseguir los datos de tipo `char`.

Además de estos métodos, inih cuenta con otros métodos como `HasSection` o `HasValue` que permiten comprobar la existencia de una sección o valor respectivamente.

5.3 Guía básica de libcurl

El proyecto libcurl ha sido utilizado para enviar las peticiones al chatbot.

5.3.1 Ficheros a incluir

Los ficheros a incluir dependen del sistema operativo que se esté usando y de la versión que se quiera dentro del mismo. En la página oficial de libcurl [2] podemos encontrar en enlace a la página de descargas [3] y dentro de esta un *wizard* [4] que nos ayudará a elegir la versión que más nos convenga. Para este proyecto por ejemplo se eligió la opción Windows 64 bit 7.79.1 *binary the curl Project* que contenía varias carpetas entre las que se pueden encontrar la carpeta "include" y la carpeta "lib" que contienen los *headers* y las librerías respectivamente y cuyo contenido importaremos a nuestro proyecto.

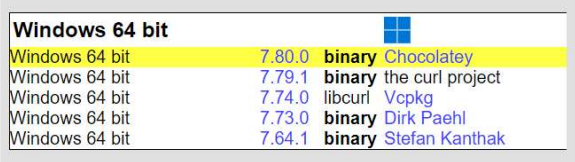
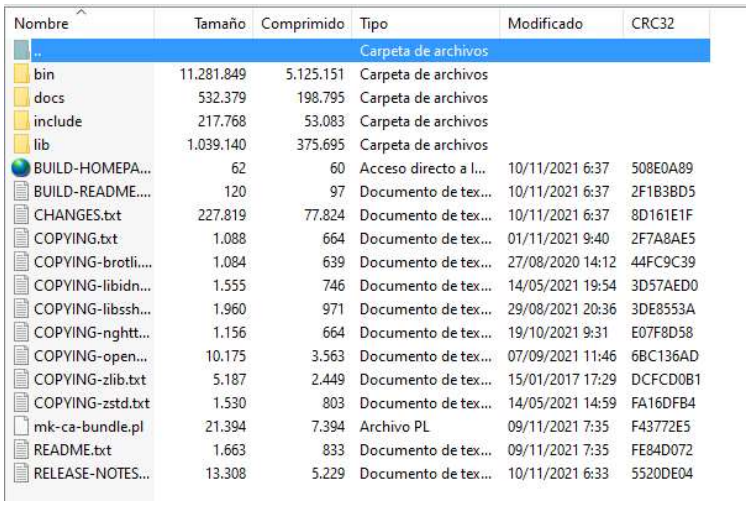
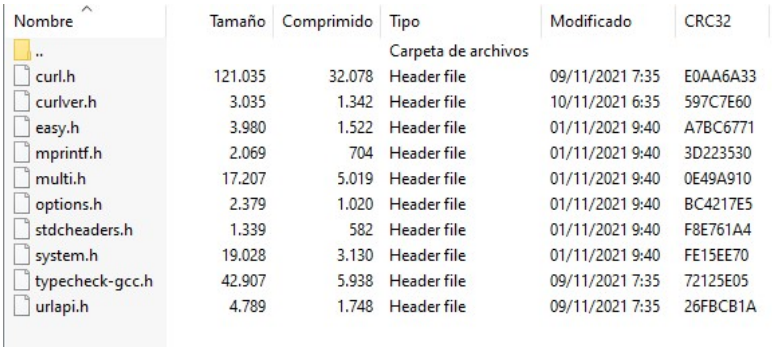


Ilustración 12: Opciones de descarga de libcurl para windows 64 [3]



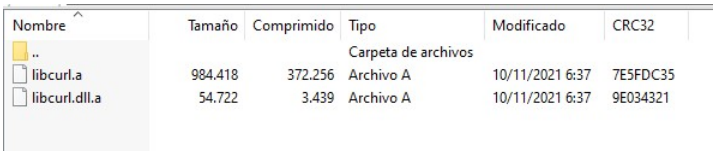
Nombre	Tamaño	Comprimido	Tipo	Modificado	CRC32
..			Carpeta de archivos		
bin	11.281.849	5.125.151	Carpeta de archivos		
docs	532.379	198.795	Carpeta de archivos		
include	217.768	53.083	Carpeta de archivos		
lib	1.039.140	375.695	Carpeta de archivos		
BUILD-HOMEPA...	62	60	Acceso directo a l...	10/11/2021 6:37	508E0A89
BUILD-README....	120	97	Documento de tex...	10/11/2021 6:37	2F1B3BD5
CHANGES.txt	227.819	77.824	Documento de tex...	10/11/2021 6:37	8D161E1F
COPYING.txt	1.088	664	Documento de tex...	01/11/2021 9:40	2F7A8AE5
COPYING-brotli....	1.084	639	Documento de tex...	27/08/2020 14:12	44FC9C39
COPYING-libidn...	1.555	746	Documento de tex...	14/05/2021 19:54	3D57AED0
COPYING-libssh...	1.960	971	Documento de tex...	29/08/2021 20:36	3DE8553A
COPYING-nghtt...	1.156	664	Documento de tex...	19/10/2021 9:31	E07F8D58
COPYING-open...	10.175	3.563	Documento de tex...	07/09/2021 11:46	6BC136AD
COPYING-zlib.txt	5.187	2.449	Documento de tex...	15/01/2017 17:29	DCFCD0B1
COPYING-zstd.txt	1.530	803	Documento de tex...	14/05/2021 14:59	FA16DFB4
mk-ca-bundle.pl	21.394	7.394	Archivo PL	09/11/2021 7:35	F43772E5
README.txt	1.663	833	Documento de tex...	09/11/2021 7:35	FE84D072
RELEASE-NOTES...	13.308	5.229	Documento de tex...	10/11/2021 6:33	5520DE04

Ilustración 13: Archivos contenidos en la descarga del proyecto libcurl



Nombre	Tamaño	Comprimido	Tipo	Modificado	CRC32
..			Carpeta de archivos		
curl.h	121.035	32.078	Header file	09/11/2021 7:35	E0AA6A33
curlver.h	3.035	1.342	Header file	10/11/2021 6:35	597C7E60
easy.h	3.980	1.522	Header file	01/11/2021 9:40	A7BC6771
mprintf.h	2.069	704	Header file	01/11/2021 9:40	3D223530
multi.h	17.207	5.019	Header file	01/11/2021 9:40	0E49A910
options.h	2.379	1.020	Header file	01/11/2021 9:40	BC4217E5
stdcheaders.h	1.339	582	Header file	01/11/2021 9:40	F8E761A4
system.h	19.028	3.130	Header file	01/11/2021 9:40	FE15EE70
typecheck-gcc.h	42.907	5.938	Header file	09/11/2021 7:35	72125E05
urlapi.h	4.789	1.748	Header file	09/11/2021 7:35	26FBCB1A

Ilustración 14: Headers libcurl



Nombre	Tamaño	Comprimido	Tipo	Modificado	CRC32
..			Carpeta de archivos		
libcurl.a	984.418	372.256	Archivo A	10/11/2021 6:37	7E5FDC35
libcurl.dll.a	54.722	3.439	Archivo A	10/11/2021 6:37	9E034321

Ilustración 15: Librerías libcurl

5.3.2 Uso básico de libcurl

Libcurl posee dos interfaces distintas dentro de su API [5], la interfaz *easy* y la interfaz multi. Como dan a entender los nombres, el propósito de la interfaz *easy* es el de hacer peticiones de la forma más sencilla posible y rápida sin entrar en demasiado detalle que probablemente requieran de un conocimiento más profundo de las peticiones. La interfaz *multi* está destinada a aquellos usuarios más experimentados que deseen personalizar sus peticiones con más detalle.

En el desarrollo de este TFG se utilizó la interfaz *easy* debido a que las peticiones que se realizan tienen una estructura sencilla, por este motivo será en dicha interfaz en la que nos centraremos en esta guía.

La interfaz tiene numerosos métodos [5], pero los principales y los que nos permitirán programar rápidamente una petición son los siguientes:

- `curl_easy_init()`: Es la primera función que debe llamarse y devuelve un manipulador o *handler* que será usado en el resto de funciones.
- `curl_easy_cleanup()`: Será la función que llamemos al final de la "sesión" de peticiones para cerrar todas las conexiones además del *handler* el cual ha de pasarse como parámetro.
- `curl_easy_setop()`: Permite cambiar los parámetros de la conexión. Esta será la función que utilicemos para introducir la url, *headers*, protocolo, *body* y otros campos de la petición.
- `curl_easy_perform()`: Realiza la petición con los parámetros especificados en `curl_easy_setop()` y con el *handler* creado en `curl_easy_init()`.
- `curl_easy_getinfo()`: Nos permite obtener datos de la petición que habrán sido almacenados en el *handler*.

A continuación, un ejemplo de una petición con libcurl:

```
CURL *curl = curl_easy_init();
if(curl) {
  CURLcode res;
  curl_easy_setopt(curl, CURLOPT_URL, "https://example.com");
  res = curl_easy_perform(curl);
  curl_easy_cleanup(curl);
}
```

Ilustración 16: Ejemplo de petición en libcurl [6]

Como se aprecia en la imagen el primer paso es iniciar el *handler* con `curl_easy_init()`, posteriormente se establecen los parámetros para esa conexión con `curl_easy_setop()` y a continuación se llama a `curl_easy_perform()` para realizarla. Una vez terminada la sesión se llama a `curl_easy_cleanup()` para finalizar las conexiones.

5.4 Guía básica de Live Link Face

Live Link Face se ha utilizado para la grabación de las diferentes animaciones de visemas sobre el MetaHuman.

Para utilizar Live Link Face harán falta dos dispositivos, un dispositivo móvil en el que utilizará la cámara a través de la aplicación y un ordenador personal con Unreal Engine. Ambos han de estar en la misma red local.

5.4.1 Dispositivo móvil

Una vez descargada la aplicación de Live Link Face en el dispositivo nos encontraremos con la siguiente pantalla.

Ilustración 17: Pantalla principal de Live Link Face [7]

Entraremos a ajustes utilizando el botón que se encuentra en la parte superior izquierda de la pantalla y veremos la pantalla de opciones. Tras esto, pulsaremos la primera opción, llamada Live Link.

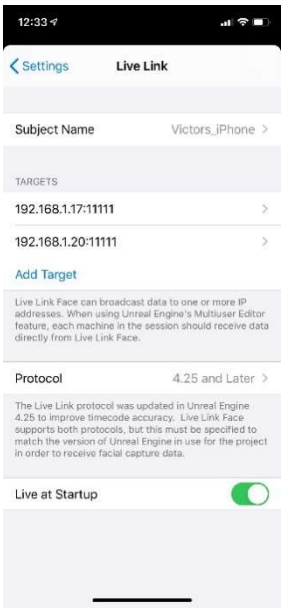


Ilustración 18: Pantalla de opciones de Link Live Face [7]

En esta pantalla definiremos un objetivo, introduciendo un nombre identificativo, la dirección IP del PC en el que estaremos ejecutando Unreal Engine y un puerto.

Tras esto podemos volver a la pantalla inicial y lo que capture la cámara será enviado al PC objetivo.

5.4.2 Ordenador personal

Lo primero que hay que tener en cuenta para configurar Unreal Engine es que hay que tener habilitados los siguientes *plugins*:

- Live Link
- ARKit

- ARKit Face Support

En el motor abriremos la pestaña de Live Link desde **Window > Live Link**. El dispositivo debería aparecer listado.

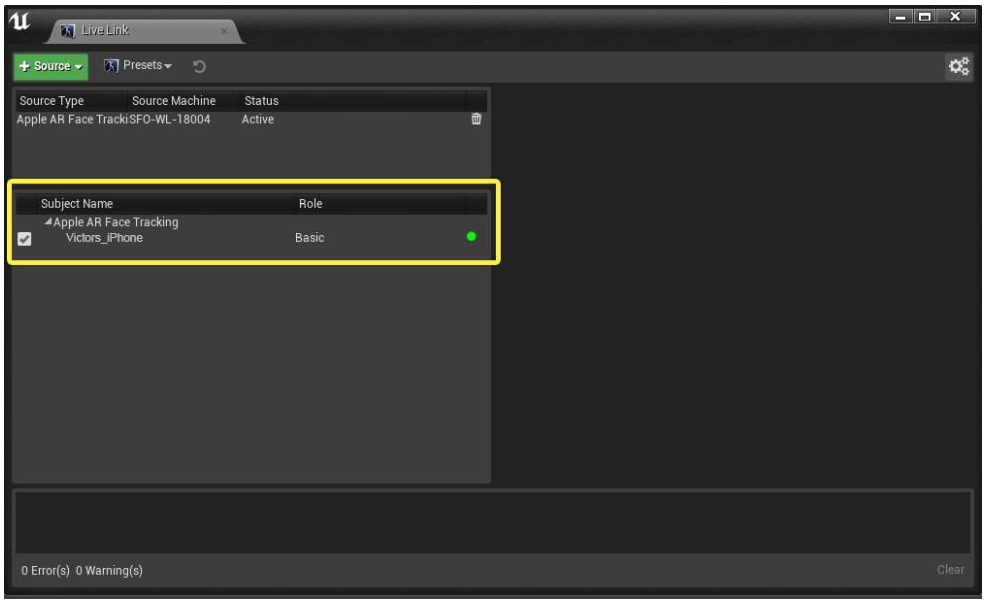


Ilustración 19: Conexión con el dispositivo móvil [7]

Tras esto, iremos a la raíz del *blueprint* de nuestro MetaHuman. En la sección "Default", entrada "LLink Face Subj" seleccionaremos el dispositivo que hemos conectado.

Tras esto el MetaHuman imitará lo reconocido por la aplicación móvil, lo que nos permitirá grabar tomas utilizando la función "Take Recorder" de Unreal Engine.

5.5 Guía básica nlohmann JSON

El proyecto nlohmann JSON se ha utilizado para todo lo relacionado con el formato JSON. Gracias a ello se puede crear el archivo que contiene el id y la duración de los visemas además de obtener los campos necesarios de la respuesta del chatbot.

5.5.1 Formato JSON

El formato JSON [1] es un tipo de formato ligero de intercambio de datos fácil de escribir para los usuarios y de interpretar y generar para las máquinas.

Está constituido por dos estructuras:

- Una colección de pares nombre-valor
- Una lista ordenada de valores.

5.5.2 Ficheros a incluir

El único archivo que deberemos incluir será el *header* `json.hpp` [2].

5.5.3 Uso básico de nlohmann JSON

El proyecto cuenta con numerosos métodos [3], pero en este apartado vamos a centrarnos en la creación rápida de un JSON y en el acceso a sus campos.

Los objetos JSON pueden crearse de forma similar a un array sustituyendo el número que iría tradicionalmente entre los corchetes por un nombre al que se le asignará un valor.

```
// create an empty structure (null)
json j;

// add a number that is stored as double (note the implicit conversion of j to an object)
j["pi"] = 3.141;

// add a Boolean that is stored as bool
j["happy"] = true;

// add a string that is stored as std::string
j["name"] = "Niels";

// add another null object by passing nullptr
j["nothing"] = nullptr;

// add an object inside the object
j["answer"]["everything"] = 42;

// add an array that is stored as std::vector (using an initializer list)
j["list"] = { 1, 0, 2 };

// add another object (using an initializer list of pairs)
j["object"] = { {"currency", "USD"}, {"value", 42.99} };
```

Ilustración 20: Método 1 de creación de un JSON [2]

El proyecto también permite crearlo utilizando la sintaxis propia de JSON.

```
json j2 = {
  {"pi", 3.141},
  {"happy", true},
  {"name", "Niels"},
  {"nothing", nullptr},
  {"answer", {
    {"everything", 42}
  }},
  {"list", {1, 0, 2}},
  {"object", {
    {"currency", "USD"},
    {"value", 42.99}
  }}
};
```

Ilustración 21: Método 2 de creación de un JSON [2]

Para acceder a un valor asociado a un nombre dentro de un JSON bastará con llamar al método `at()` [3].

expression	value
`j`	`{"name": "Mary Smith", "age": 42, "hobbies": ["hiking", "reading"]}`
`j.at("name")`	`"Mary Smith"`

Ilustración 22: Acceso a un valor de un JSON [3]

Un último método que ha sido utilizado durante este TFG y que consideramos digno de mención es el método `dump()` [3] que nos permite transformar un JSON a un `string`.

```
// explicit conversion to string
std::string s = j.dump();    // {"happy":true,"pi":3.141}
```

Ilustración 23: Conversión de JSON a string [2]

5.6 Crear un mock de un servidor en Postman

Postman ha sido utilizado para simular las respuestas de un Chatbot durante la fase de desarrollo debido a un problema con la generación de tokens de verificación que impedía comprobar el funcionamiento de las llamadas.

El primer paso [12] será crear una colección dentro de nuestro *workspace* si no la tenemos ya creada. Para ello haremos clic en el símbolo + que aparece en la zona superior izquierda teniendo seleccionada la ventana "Collections".

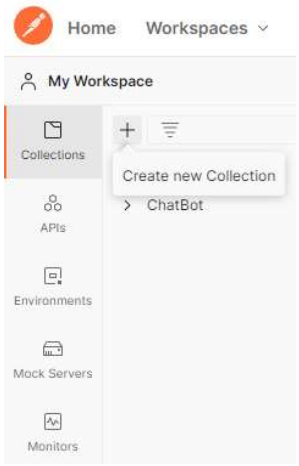


Ilustración 24: Crear nueva colección en Postman

Una vez hecho esto podremos cambiar el nombre en la zona central superior.

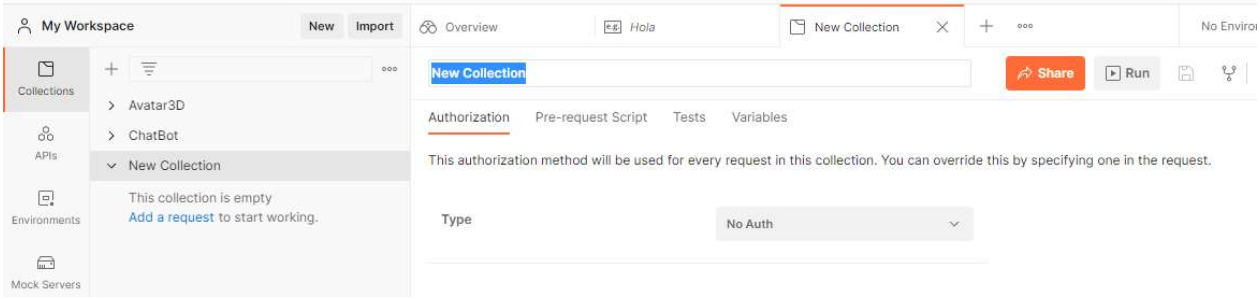


Ilustración 25: Menú de la colección

Dentro de la colección podremos crear carpetas para almacenar *request* relacionadas entre sí, aunque no es obligatorio.

Ahora hacemos clic en la pestaña de "mock servers" que encontraremos en el menú de la izquierda y una vez más haremos clic en el icono + en la zona superior izquierda para crear un *mock server*. Se nos presentará el menú de creación del *mock server* en el que como primer paso elegiremos si queremos crear una nueva colección para el server o usar una existente, en nuestro caso usaremos la que hemos creado con anterioridad a la que hemos llamado Server.

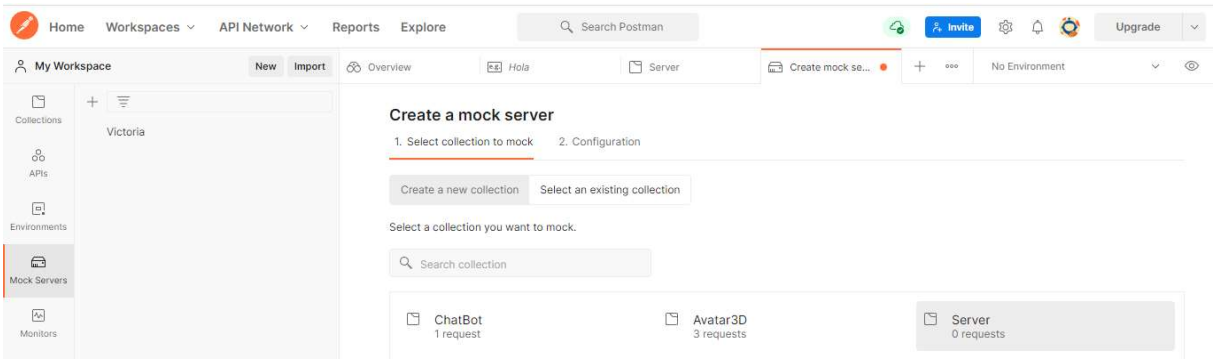


Ilustración 26: Menú de creación de un mock server

Una vez hecho esto se nos mostrará lo siguiente:

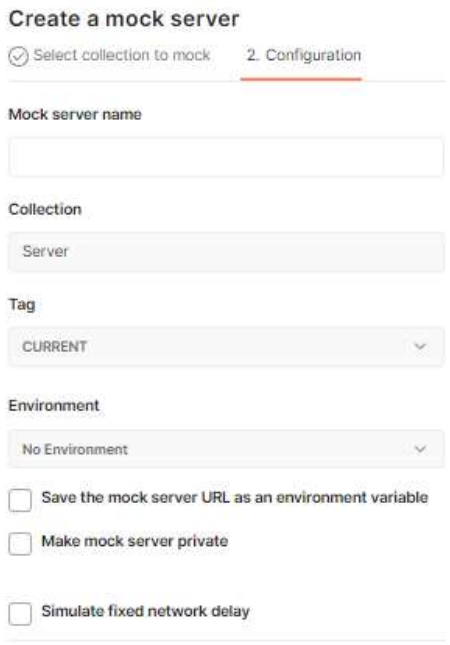


Ilustración 27 Menú de configuración del mock server

Aquí podremos ponerle nombre al server, que en nuestro caso será MockServer y elegir un *environment* que sirve para mantener guardadas variables que van a ser usadas con frecuencia en las peticiones.

Si no tenemos creado un *environment* podemos hacerlo de forma sencilla haciendo clic en el icono del ojo en la zona superior izquierda junto al desplegable de los *environments* y a continuación haciendo clic en "Add". En el nuevo menú que se abrirá podremos cambiarle el nombre al *environment*, en nuestro caso se llamará EnviromentMock.

Volviendo a la pestaña de creación del *mock server* podremos elegir en el desplegable de *environments* nuestro recién creado *environment*.

Por último, haremos clic en "Create Mock Server" para terminar el proceso de creación.

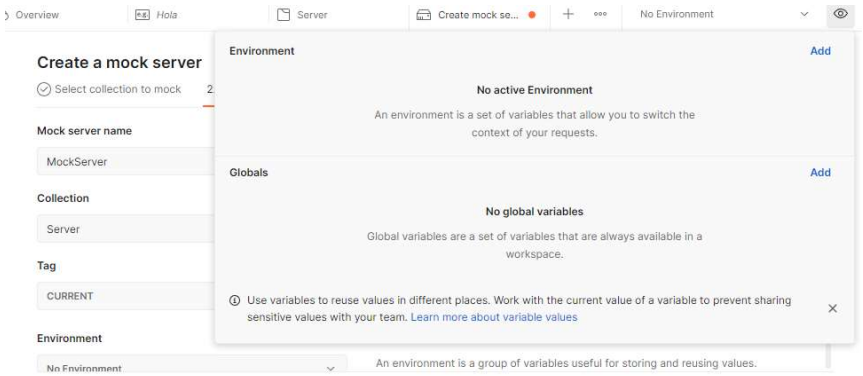


Ilustración 28: Menú de creación de un environment

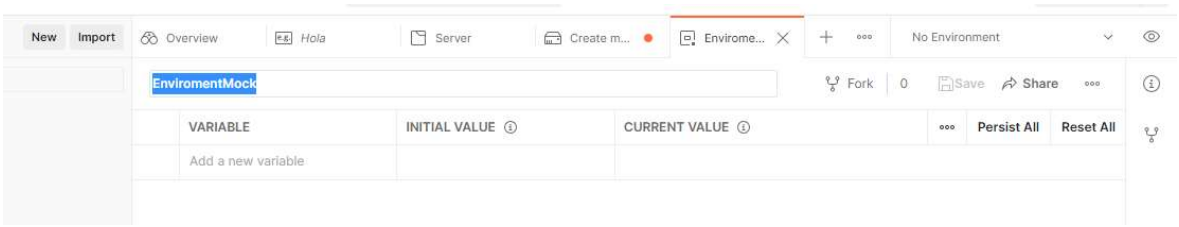


Ilustración 29: Menú del environment seleccionado

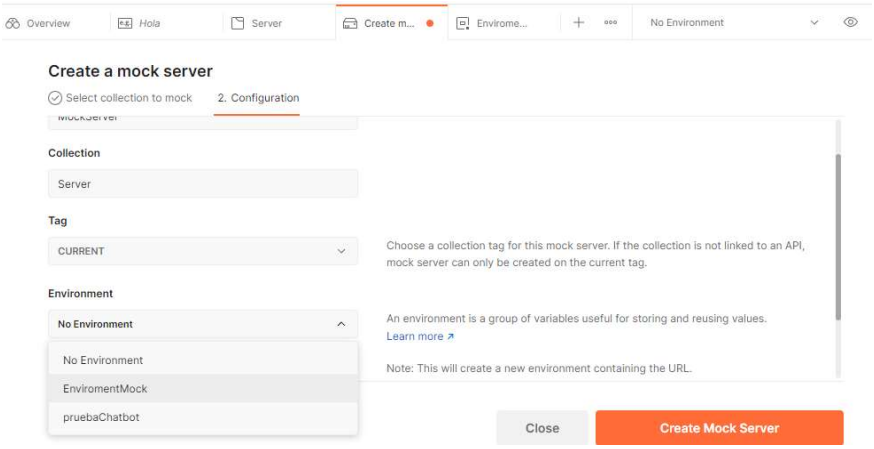


Ilustración 30: Selección del environment a usar en el mock server

Una vez creado el *mock* nos aparecerá la opción de copiar la url, la copiamos y nos dirigimos a nuestro *environment* desde el icono del ojo que usamos para crearlo y

teniéndolo seleccionado en el desplegable. Una vez en el *environment* añadimos la url dándole un nombre a la variable en la sección VARIABLE, en nuestro caso url, y copiando en la sección de INITIAL VALUE la url. En nuestro caso añadiremos también otra variable llamada autenticación cuyo valor será Bearer "token" donde "token" será la cadena de texto usada para la verificación. Una vez añadidas las variables haremos clic en "save" representado como un icono de guardado para guardarlas.

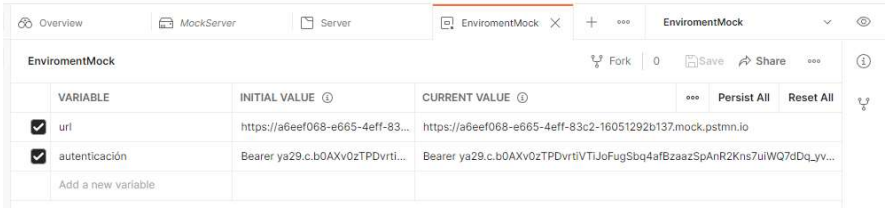


Ilustración 31: Variables creadas en el environment

Una vez creado el server y el *environment* solo queda añadir *requests*, para ello haremos clic en "Add a request" dentro del desplegable de nuestra colección.

En este ejemplo se va a crear una petición tipo post con un *body* tipo json simulando un chatbot.

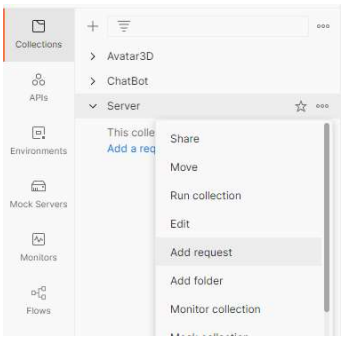


Ilustración 32: Añadir una request

Al crear la *request* se nos abrirá el menú de edición de la misma en la cual le cambiaremos el nombre, en nuestro caso Request, se establecerá el tipo de petición, en nuestro caso Post, y se rellenarán los siguientes campos:

- El campo que se encuentra al lado del tipo de petición se rellenará con el nombre de nuestra url a la que podemos acceder con {{"url"}} teniendo seleccionado nuestro *environment* donde "url" será el nombre que le pusiéramos a dicha variable. Además deberemos añadirle /"operación" donde "operación" será el nombre para esta operación en nuestro caso será Hola.
- En el apartado "Headers" se añadirán Content-Type application/json y Authorization {{"autenticación"}} siendo "autenticación" el nombre que le hayamos dado a la variable.
- En el apartado "Body" seleccionaremos raw y JSON en el desplegable y añadiremos el cuerpo del JSON.

Una vez añadidos los campos hacemos clic en el icono de guardar.

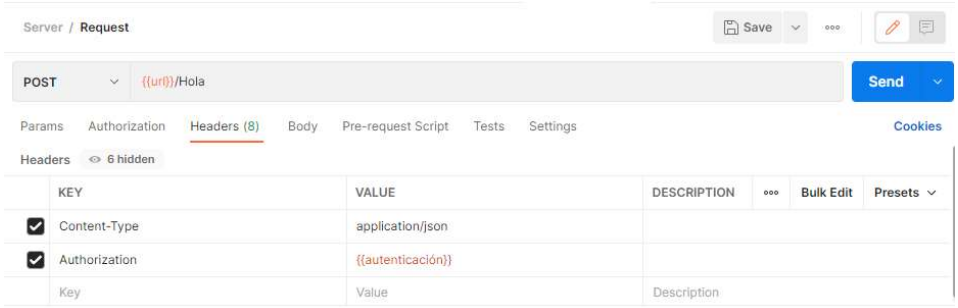


Ilustración 33: Pestaña Headers

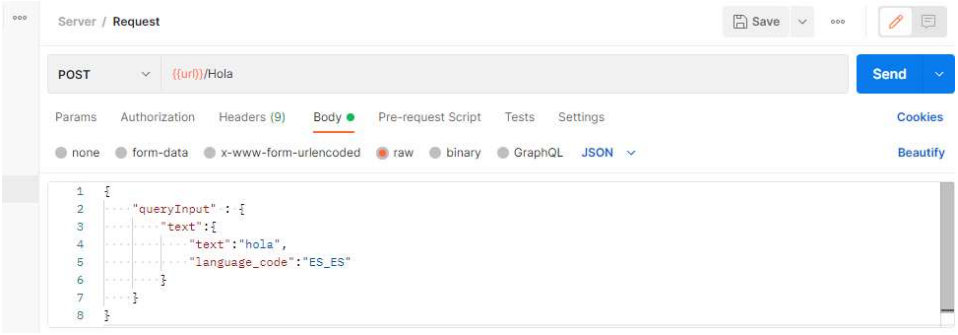


Ilustración 34: Pestaña Body

Por último, añadiremos una respuesta a dicha petición haciendo clic en "Add example" en el desplegable de nuestra petición.

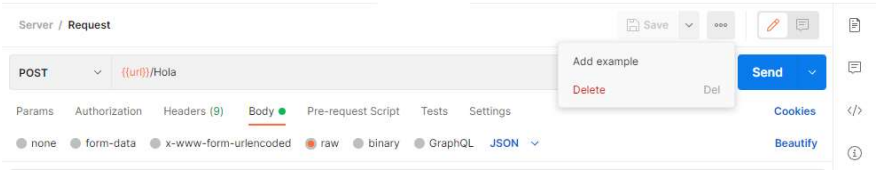


Ilustración 35: Añadir una respuesta

Dentro del menú de la respuesta solo nos quedará añadir en el *body* lo que queramos que nos devuelva la petición y el código 200 Ok. Una vez añadidos los parámetros la guardamos.

Ilustración 36: Body de la respuesta

Podemos probar el funcionamiento haciendo clic en el botón azul "Send" dentro de nuestra *request*.

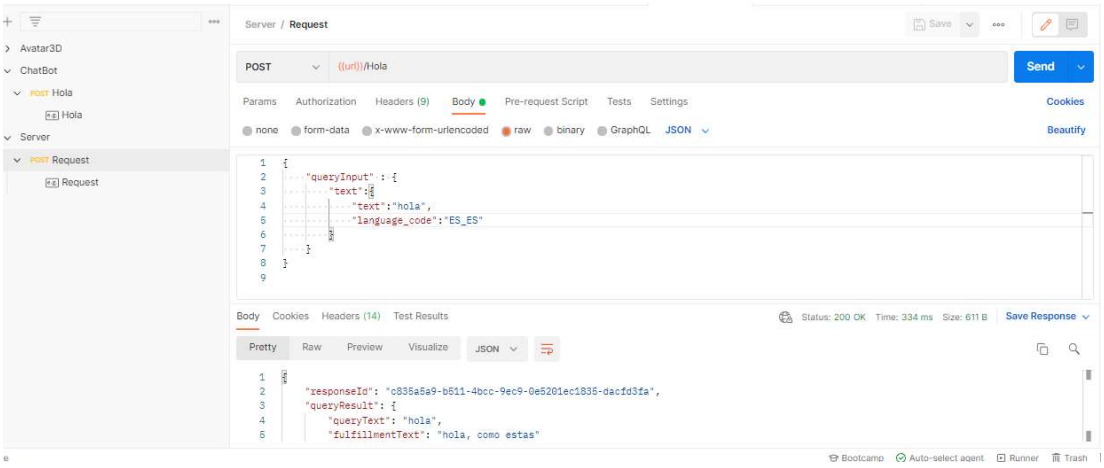


Ilustración 37: La petición se ha enviado correctamente y se ha recibido la respuesta

5.7 Cómo usar MetaHumans en Unreal Engine

5.7.1 Creación de un MetaHuman

En este proyecto se ha utilizado uno de los MetaHuman que se encuentran en Quixel Bridge, aun así, debido al gran potencial de esta herramienta creemos que resultará útil mostrar cómo podemos crear un MetaHuman desde cero.

Para esto accederemos a la página de MetaHuman Creator [11].

En esta página podremos personalizar un MetaHuman completamente, aunque sigue en desarrollo.

Primero seleccionaremos uno de los MetaHuman base.

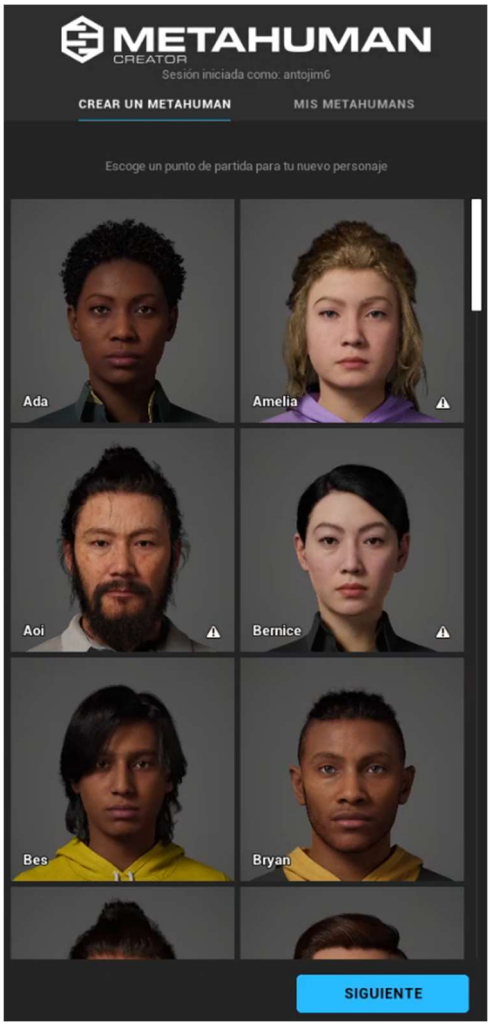


Ilustración 38: MetaHuman base

Una vez hecho esto podremos comenzar a modificar el MetaHuman.

Los usuarios pueden configurar:

- Características faciales
- Tez de la piel
- Tipo de cuerpo
- Maquillaje
- Dientes
- Ropa
- Pelo
    - Cabeza
    - Facial
    - Cejas
    - Pestañas

También tenemos la opción de seleccionar un MetaHuman "padre" en el que se basará el nuestro.

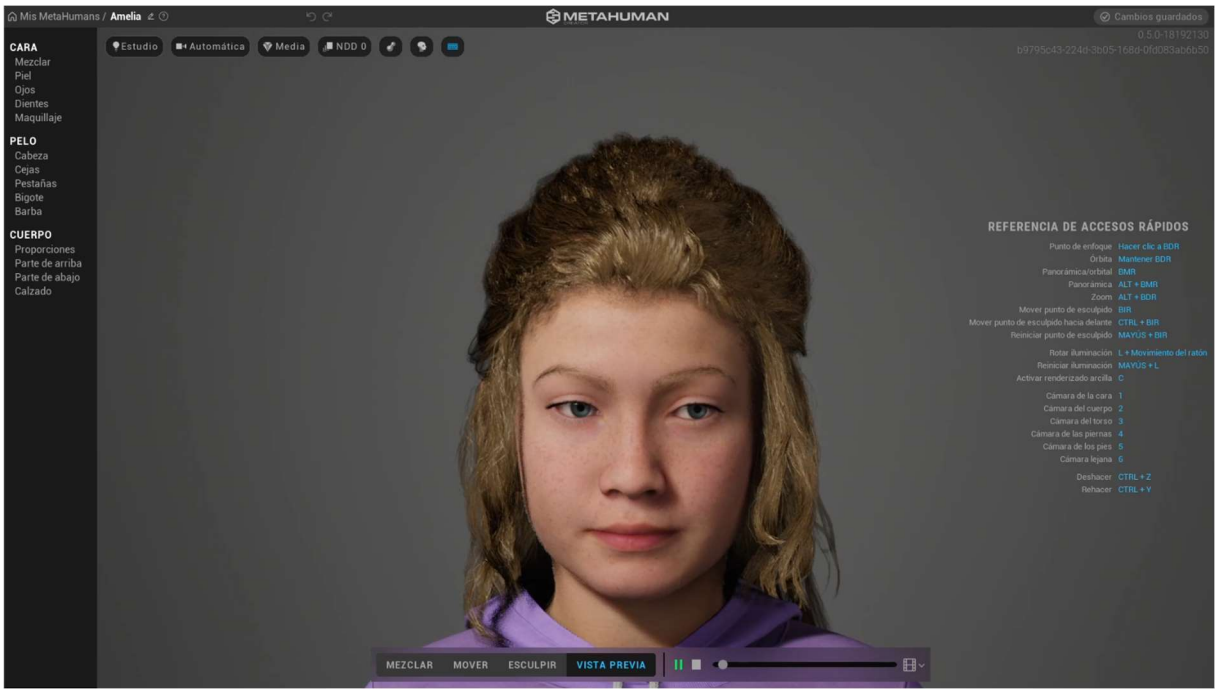


Ilustración 39: Creador de MetaHuman

5.7.2 Importar un MetaHuman a Unreal Engine

Accedemos a la sección de MetaHuman desde Bridge.

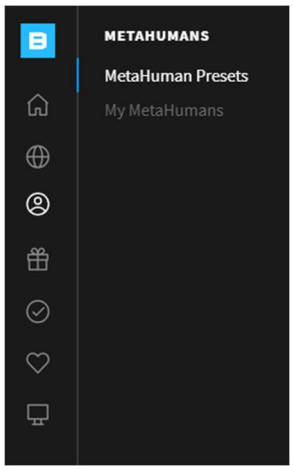


Ilustración 40: Menú de Bridge

En este menú seleccionaremos el MetaHuman que queremos usar y haremos clic en "Download".

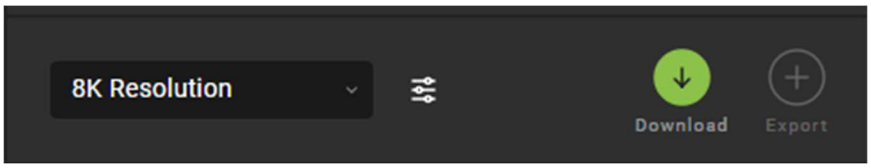


Ilustración 41: Descarga de un MetaHuman

Una vez descargado podremos exportarlo haciendo clic en "Export". Unreal Engine deberá estar seleccionado como objetivo Unreal Engine como objetivo y tiene que estar seleccionada la versión del motor que estamos utilizando, además del lugar de instalación del plugin y el proyecto por defecto.

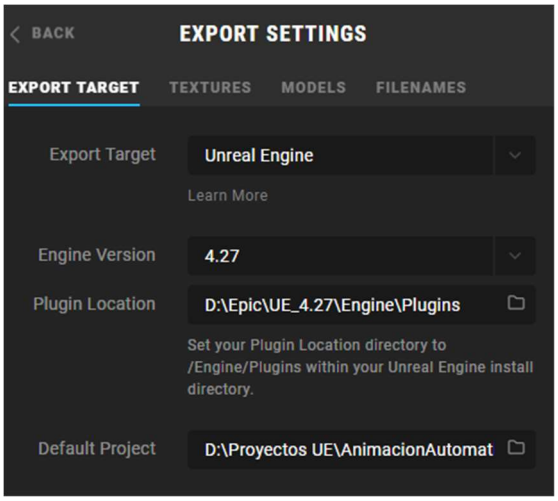


Ilustración 42: Opciones de exportación de Bridge

Una vez hecho esto podremos encontrar nuestro MetaHuman en la carpeta Content del proyecto.

5.7.3 Partes de un MetaHuman

Los MetaHumans disponen de una gran cantidad de variables faciales modificables que permitirán crear expresiones faciales con gran precisión.

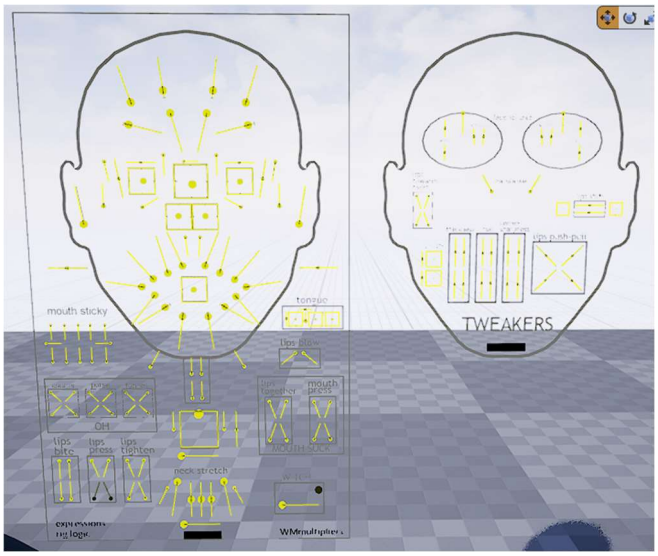


Ilustración 43: Partes de un MetaHuman

Todas estas variables son editables desde la interfaz flotante que aparece en la ilustración, desde el secuenciador, desde los *blueprints* y desde código C++.

5.8 Guía básica de ReadSpeaker

5.8.1 Instalación

Tras ponernos en contacto con el equipo de ReadSpeaker, nos proporcionarán los archivos necesarios para la instalación de los motores Text-To-Speech y para el uso de su API. Para este TFG se proporcionó lo siguiente:

- ReadSpeaker speechEngine SDK para la voz “lola”
- ReadSpeaker speechEngine SAPI para la voz “lola”
- SDK ReadSpeaker

De los cuales en el TFG se usaron:

- ReadSpeaker speechEngine SDK para la voz “lola”
- SDK ReadSpeaker

Una vez instalada el motor de voz hay que obtener el archivo de verificación, para ello nos dirigiremos a https://verification.readspeaker.com y obtendremos el archivo `.txt` con la verificación que habrá que usar más adelante en el proyecto.

5.8.2 Ficheros a incluir

En el sistema operativo Windows los ficheros a incluir serán:

- *Headers*: `vtapi.h` y `vtapi_extend.h`
- Librerías: `libvtapi.lib`, `libvtapi.dll`, `libvtconv.dll`, `libvtsave.dll`, `libvtplay.dll`, `libvteffect.dll` y `libvtssml.dll`

5.8.3 Uso básico de ReadSpeaker

ReadSpeaker ofrece un gran número de métodos destinados a funciones muy diversas e incluso alejadas de aquellas que se usaron en este proyecto, por ello la guía se centrará en aquellos métodos que se han utilizado en este proyecto y los abordará de forma superficial debido a que una explicación más detallada estará disponible en el manual del SDK otorgado por ReadSpeaker.

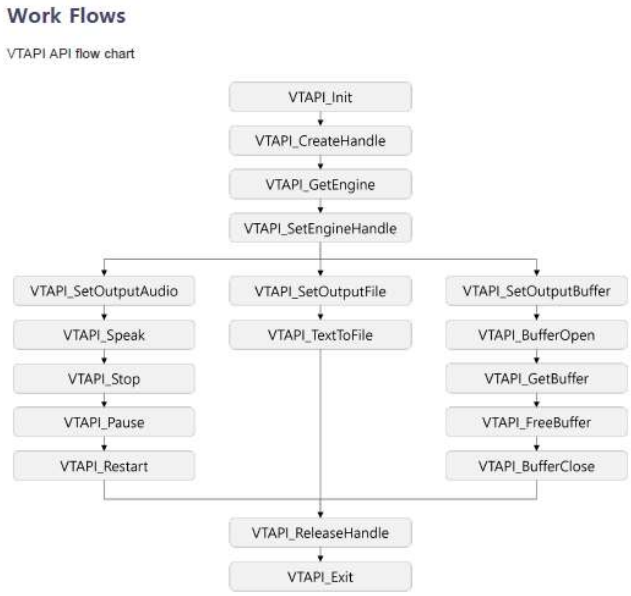


Ilustración 44: Work Flow de ReadSpeaker

Siempre se comenzará llamando al método `VTAPI_Init`, al cual pasaremos como parámetro el directorio de trabajo en el que tendremos las librerías necesarias para que ReadSpeaker funcione.

De forma similar a libcurl, el SDK se ReadSpeaker gira en torno a su *handler* VTAPI_HANDLE el cual será inicializado con `VTAPI_CreateHandle`. Hay que recordar que debe llamarse a `VTAPI_ReleaseHandle` cuando se termine de usar el *handler* para liberar el espacio de memoria asignado a este.

`VTAPI_GetEngine` será utilizado para obtener el motor instalado a partir de su tipo y su nombre, en nuestro caso p16 y lola respectivamente.

Con `VTAPI_SetEngineHandle` estableceremos nuestro *handler* como *handler* del motor de voz.

A continuación, podremos elegir entre varios cursos de acción dependiendo de nuestros objetivos:

- En la primera rama que corresponde a la situada más a la izquierda en la imagen podremos controlar el habla, las paradas pausas y reinicios de nuestro motor de voz. Esta rama no fue usada en el proyecto.
- En la segunda rama que corresponde a la situada en la zona central de la imagen podremos crear ficheros de audio a partir de texto. Es la que nos permite generar el audio a partir del texto de entrada.
- En la última rama podremos obtener información precisa del texto que queramos procesar. Es la que nos permite generar los visemas y obtener su duración

Estas dos últimas ramas serán explicadas con mayor profundidad en el capítulo 6.3.3 Tts.cpp.

Por último, tenemos las funciones `VTAPI_ReleaseHandle` y `VTAPI_Exit` que finalizarán todos los procesos y liberarán la memoria utilizada.

6 Diseño e Implementación

6.1 Investigación previa

6.1.1 Visemas

Similar a cómo los sonidos del lenguaje pueden ser descritos utilizando fonemas, el movimiento y posición de la boca pueden ser recogidos con visemas.

Los visemas son una herramienta utilizada en el campo de la lingüística para estudiar cómo se forman los diferentes sonidos utilizados en el habla. No tienen una correspondencia directa con los fonemas, si no que muchas veces múltiples fonemas están relacionados con un solo visema. Por ejemplo, los fonemas /k, g, ŋ/ corresponden al visema /k/.

Se han realizado investigaciones previamente sobre qué visemas se utilizan en el castellano con el propósito de investigar la sincronización labial de personajes virtuales.

Visema									
Modo / Punto Articulación	Bilabial	labiodental	Linguo-Interdental	Linguo-Dental	Linguo-Alveolar	Post-Alveolar	Liguo-Palatal	Linguo-velar	Uvular
Oclusiva	Sonora [b]			[d]				[g]	
	Sorda [p]			[t]				[k]	
Aproximante Oclusiva	Sonora			[ð]				[ɣ]	
	Sorda								
Fricativa	Sonora [β]				[z]		[ʝ]		
	Sorda	[f]	[θ]		[s]			[x]	[χ]
Nasal	Sonora [m]	[ɱ]			[n]		[ɲ]	[ŋ]	
	Sorda								
Lateral	Sonora				[l]		[ʎ]		
	Sorda								
Africada	Sonora					[ʤ]			
	Sorda						[ʧ]		
Vibrante Simple	Sonora				[ɾ]				
	Sorda								
Vibrante Múltiple	Sonora				[r]				
	Sorda								

Ilustración 45: Tabla de visemas del castellano [15]

6.1.2 ReadSpeaker

El contacto con ReadSpeaker lo realizó nuestro tutor que ya conocía esta tecnología y que nos puso en contacto con el director de ReadSpeaker en Latinoamérica y España Antonino Sistac Aznárez que a su vez nos puso en contacto con el soporte técnico.

Desde entonces se mantuvo comunicación directa con el soporte que fueron los que nos guiaron en la utilización de la API además de proporcionarnos los archivos necesarios y ayudarnos en la resolución de dudas.

6.2 Captura de visemas

Para la captura de los visemas se hizo uso de Live Link Face utilizando como referencia la siguiente tabla de visemas del castellano.

IPA	X-SAMPA	Descripción	Ejemplo	Visema
Consonantes				
ɾ	4	vibrante simple alveolar	pe**r**o	t
b	b	oclusiva bilabial sonora	**b**estia	p
β	B	fricativa bilabial sonora	be**b**é	B
d	d	oclusiva alveolar sonora	cuan**d**o	t
ð	D	fricativa dental sonora	ar**d**er	T
f	f	fricativa labiodental sonora	**f**ase	f
g	g	oclusiva velar sonora	**g**ato	k
ɣ	G	fricativa velar sonora	tri**g**o	k
j	j	aproximante palatal	hac**i**a	i

ʝ	j\	fricativa palatal sonora	enh**i**elar	J
k	k	oclusiva velar sorda	**c**aña	k
l	l	aproximante alveolar lateral	**l**ino	t
ʎ	L	aproximante palatal lateral	**ll**ave	J
m	m	nasal bilabial	**m**adre	p
n	n	nasal alveolar	**n**ido	t
ɲ	J	nasal palatal	caba**ñ**a	J
ŋ	N	nasal velar	ci**n**co	k
p	p	oclusiva bilabial sorda	**p**ozo	p
r	r	vibrante alveolar	pe**rr**o	r
s	s	fricativa alveolar sorda	**s**aco	s
t	t	oclusiva alveolar sorda	**t**amiz	t
t͡ʃ	tS	africada postalveolar sorda	**ch**ubasco	S
θ	T	fricativa dental sorda	cere**z**a	T
w	w	aproximante velo-labial	f**u**ego	u
x	x	fricativa velar sorda	**j**amón	k
z	z	fricativa alveolar sonora	ra**s**go	s
Vocales				
a	a	vocal abierta anterior no redondeada	t**a**nque	a
e	e	vocal semicerrada anterior no redondeada	p**e**so	e
i	i	vocal cerrada anterior no redondeada	c**i**nco	i
o	o	vocal semicerrada posterior redondeada	b**o**sque	o
u	u	vocal semicerrada anterior no redondeada	p**u**blicar	u

Ilustración 46: Visemas del castellano detallados [16]

Estos visemas fueron grabados utilizando la herramienta “Take Recorder” de Unreal Engine. Una guía detalla del proceso de grabación de los visemas se encuentra en la memoria del TFG de Antonio Jiménez.

Los visemas han sido guardados con nombres acordes a la lista de valores que asigna ReadSpeaker a cada visema. Puede ser consultada en la página de la “Speech API de Microsoft” [17].

- 0 = silence
- 1 = ae ax ah
- 2 = aa
- 3 = ao
- 4 = ey eh uh
- 5 = er
- 6 = y iy ih ix
- 7 = w uw
- 8 = ow
- 9 = aw
- 10 = oy
- 11 = ay
- 12 = h
- 13 = r
- 14 = l
- 15 = s z
- 16 = sh ch jh zh
- 17 = th dh
- 18 = f v
- 19 = d t n
- 20 = k g ng
- 21 = p b m

Esta lista contiene más visemas de los que se utilizan en el castellano, pero todos los del castellano están incluidos en ella, por lo que nos es de utilidad para el proyecto.

La relación entre los visemas de la **¡Error! No se encuentra el origen de la referencia.** y la numeración de ReadSpeaker ha sido recogida en esta tabla.

Número	Fonema	Visema
31	sil	
1	ae ax ah	a
2	aa	
3	ao	o
4	ey eh uh	e
5	er	
6	y iy ih ix	i
7	w uw	u
8	ow	
9	aw	
10	oy	
11	ay	
12	h	
13	r	r
14	l	
15	s z	s
16	sh ch jh zh	J/S
17	th dh	T
18	f v	B/f
19	d t n	t

20	k g ng	k
21	p b m	p

Ilustración 47: Numeración de visemas

El tener los montajes nombrados según la numeración que utiliza ReadSpeaker permite simplificar el código que los accederá.

6.3 Implementación del proyecto

6.3.1 Estructura del proyecto

El proyecto consiste en una librería estática `libtts.a`, un *header* `tts.h` y un archivo `parameters.ini` donde el usuario establecerá los parámetros de los métodos. Esta estructura está pensada para que el usuario tenga la menor interacción posible con el funcionamiento interno del proyecto y solo deba preocuparse por elegir entre procesar un texto o enviar un texto a un Chatbot para que se procese la respuesta del mismo.

El proyecto requiere de las librerías y *headers* mencionados en las herramientas utilizadas.

El proyecto se creó con la idea de ser lo más sencillo posible para el usuario por ello cuenta con solo dos métodos `ttsText` y `ttsAnswer`:

- `ttsText` recibirá como entrada un texto a procesar y creará dos archivos en el directorio especificado en el archivo parameters.ini. Los archivos creados son:
    - `visemas.json`: Archivo JSON que contiene pares idVisema-duración.
    - `audio.wav`: Archivo `.wav` que contiene el audio generado a partir del texto.

- `ttsAnswer` recibirá como entrada un texto a enviar al Chatbot, realizará una petición POST a dicho Chatbot y procesará el texto recibido como respuesta llamando a la función `ttsText`.

Existe un tercer método utilizado por `ttsAnswer` llamado `getUrlData` cuya función es manejar la respuesta recibida del Chatbot. Este método no tiene utilidad directa para el usuario.

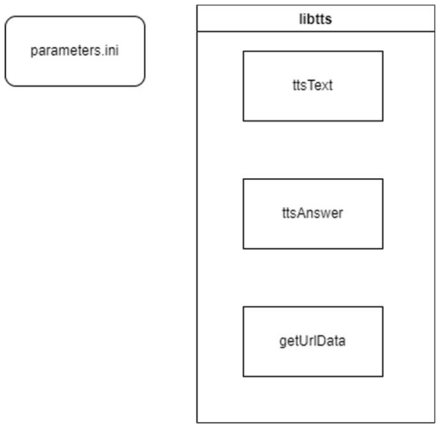


Ilustración 48: Estructura del proyecto

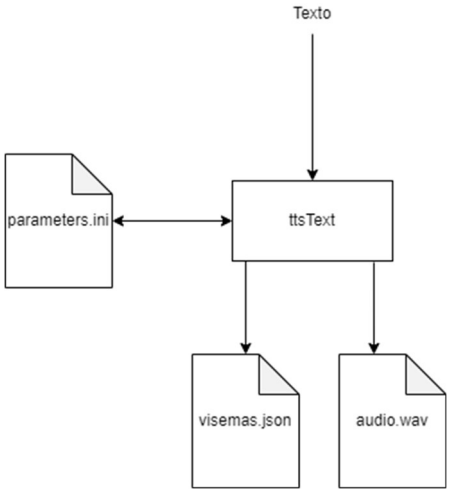


Ilustración 49: Diagrama de flujo de ttsText

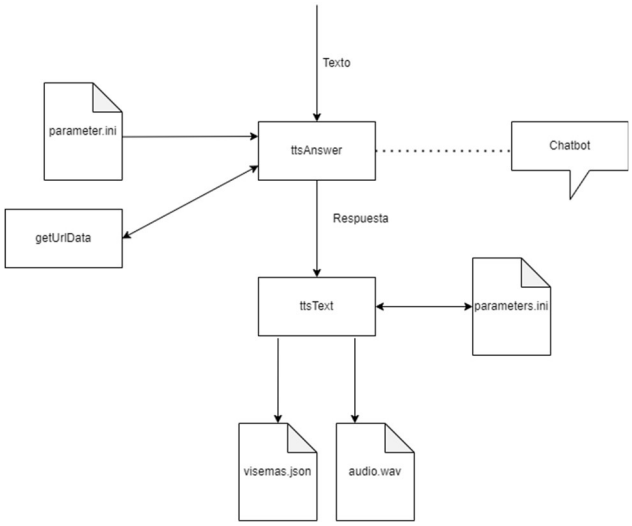


Ilustración 50: Diagrama de flujo ttsAnswer

6.3.2 Tts.h

Es el *header* de nuestro proyecto y tiene la siguiente estructura:

```
#ifndef VISEMAS_H_INCLUDED
#define VISEMAS_H_INCLUDED

#include"vtapi_extend.h"
#include<iostream>
#include"ini.h"
#include"INIReader.h"
#include"json.hpp"
#include"curl.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/timeb.h>

using json = nlohmann::json;
using jsonMap = nlohmann::ordered_json;
using namespace std;

//estructura para almacenar la respuesta de la petición de libcurl
struct memory {
   char *response;
   size_t size;
```

```
 };
//función que permite manejar el resultado de las peticiones realizadas por
libcurl
size_t getUrlData(void *data, size_t size, size_t nmemb, void *userp);
//función que procesa un texto y crea un fichero JSON con pares idvisema-
duración y un fichero de audio
//con el Text-To-Speech del texto
int ttsText(char * str);
//función que obtiene la respuesta de un chatbot mediante libcurl pasando
como parametro la url, el protocolo,
//el token de verificación y el texto que se le envía
int ttsAnswer(char* text);

#endif // VISEMAS_H_INCLUDED
```

Como podemos observar cuenta con un bloque `ifndef` para evitar que se incluya varias veces.

A continuación, están incluidas los *headers* necesarios para el proyecto, aunque se debe tener en cuenta que estos *headers* llaman a su vez a otros. Todos los *headers* necesarios están nombrados en los capítulos de las respectivas herramientas y están incluidos en los archivos del proyecto. Deben estar en el mismo directorio que el proyecto o indicar al compilador donde encontrarlos.

También se observa la estructura `memory`, la cual almacena la respuesta de la petición al chatbot.

Por último, tenemos los 3 métodos que componen el proyecto:

- `getUrlData`: función auxiliar para el manejo de la respuesta de la petición del Chatbot.
- `ttsText`: función que procesa un texto de entrada y crea un fichero `.wav` con el Text-To-Speech del texto y un fichero `.json` con los id y duración de los visemas.
- `ttsAnswer`: función que recibirá como parámetro la url del chatbot, el protocolo de la petición, el token de verificación y el texto a enviar, realizará una petición post con dichos parámetros y llamará a `ttsText` para procesar la respuesta.

6.3.3 Tts.cpp

Se trata del fichero *source* del proyecto y el que contiene la implementación de los métodos. A partir de este archivo se ha creado la librería `libtts.a`. A continuación, se realizará un resumen de sus partes.

6.3.3.1 getUrlData

Función auxiliar cuya finalidad es manejar la respuesta de la petición al chatbot. Está programada según lo requerido por libcurl en su documentación para ser pasada como parámetro en `curl_easy_setopt` el cual la establecerá como función encargada de manejar la respuesta de la petición.

Tiene la siguiente estructura:

```
size_t getUrlData(void *data, size_t size, size_t nmemb, void *userp){

   size_t realsize = size * nmemb;
   struct memory *mem = (struct memory *)userp;

   char *ptr = (char*)realloc(mem->response, mem->size + realsize + 1);
   if(ptr == NULL) return 0;

   mem->response = ptr;
   memcpy(&(mem->response[mem->size]), data, realsize);
   mem->size += realsize;
   mem->response[mem->size] = 0;

   return realsize;

 }
```

A grandes rasgos guarda en `mem`, que es la estructura destinada a guardar la información de la petición definida en `tts.h`, lo devuelto por la petición al chatbot y devuelve el tamaño de la misma para ser comprobado por libcurl más adelante.

6.3.3.2 ttsAnswer

Función que recibirá como parámetro el texto a enviar, realizará una petición post con los parámetros obtenidos en `parameters.ini` y llamará a `ttsText` para procesar la respuesta. Al ser un poco más extensa se explicará por partes:

- Fase de Inicialización:

```
int ttsAnswer(char*text){

    INIReader file("parameters.ini");

    if (file.ParseError() < 0) {
            return -1;
      }

      string url = file.Get("Conexion","Url","noUrl");

      if(url == "noUrl"){
        return -1;
      }

      string token = file.Get("Conexion","Token","noToken");

      if(url == "noToken"){
        return -1;
      }

      string protocol = file.Get("Conexion","Protocol","https");

    //creamos un JSON

    json jq;

    //se le da el formato de una petición a un chatbot

    jq["queryInput"]["text"]={{"languaje_code","ES_ES"},{"text",text}};

    //se convierte en string

    string s = jq.dump();

    //se crea el string que contendrá el token de la petición

    char* ttoken = &token[0];
    char* ttoken2 = "Authorization: Bearer ";

    int sizetoken = sizeof(ttoken2)+sizeof(ttoken)+50;
    char ttokendef[sizetoken];
    sprintf(ttokendef,"Authorization: Bearer %s",ttoken);

    //se inicia la estructura donde recibiremos la información de la petición
```

```
    struct memory chunk = {0};
```

En esta fase se observa cómo se crea el json `jq` que será enviado como *body* en nuestra petición. En el *body* se introduce el texto a enviar.

En esta fase también se crea la cadena de texto `ttokendef` donde se añadirá el token de verificación y se inicializa la estructura de tipo memoria `chunk`.

- Fase curl:

```
 CURL *curl;
    CURLcode res;
    curl = curl_easy_init();

    if(curl) {

        curl_easy_setopt(curl,CURLOPT_SSL_VERIFYPEER,0L);

        //establecemos getUrlData como función de manejo del resultado de la
petición y a chunk como estructura de escritura

        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, getUrlData);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, (void *)&chunk);

        //se establece la petición de tipo Post, se establece la url, el
protocolo, el tipo JSON los headers y el body

        curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "POST");
        curl_easy_setopt(curl, CURLOPT_URL, &url[0]);
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
        curl_easy_setopt(curl, CURLOPT_DEFAULT_PROTOCOL, protocol);
        struct curl_slist *headers = NULL;
      headers= curl_slist_append(headers, "Content-Type: application/json");
        headers = curl_slist_append(headers, ttokendef);
        curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS, &s[0]);

        //se realiza la petición

        res = curl_easy_perform(curl);

    }
```

En esta fase creamos el *handler* `curl` y el `curlcode res` para comprobar que la petición se realice con éxito.

Mediante `curl_easy_setop` se establecen:

- La función que manejará la respuesta a la petición, así como la estructura que la almacenará.
- El tipo de petición que será POST.
- La url y el protocolo.
- Los *headers*, el *token* y el *body*.

Por último, `curl_easy_perform` realiza la petición.

- Fase final:

```
if(res!=CURLE_OK){
        curl_easy_cleanup(curl);
        return -1;
      }

    //se libera el handler

    curl_easy_cleanup(curl);

    //se obtiene la respuesta de la petición

    string answer = chunk.response;

    //obtenemos los campos requeridos

    json j = json::parse(answer);
    json j2 = j.at("queryResult");
    json j3 = j2.at("fulfillmentText");
    string aux = j3.dump();
    char* c = &aux[0];

    //llamamos a ttsText para que procese el texto

    return ttsText(c);
}
```

Comprobamos que no ha habido ningún error y en caso contrario devolvemos `-1` para indicarlo y finalizar la ejecución.

Llamamos a `curl_easy_cleanup` para liberar la memoria asignada al *handler*.

Accedemos a la respuesta de la petición, la convertimos en un objeto json `j` y accedemos a los campos requeridos, en este caso `fulfillmentText` dentro de `queryResult`, para pasar el texto de respuesta como parámetro a `ttsText` cuyo valor de retorno será también el de `ttsAnswer`.

6.3.3.3 ttsText

Función que procesa un texto de entrada y crea un fichero `.wav` con el Text-To-Speech del texto y un fichero `.json` con los id y duración de los visemas. Al ser la más extensa estará dividida en varias partes:

- Fase de inicialización:

```
//Leemos el archivo parameter.ini

    INIReader file("parameters.ini");

    if (file.ParseError() < 0) {
            return -1;
      }

      //obtenemos los parametros necesarios de paraemeters que más adelante
usaremos en el resto de métodos

String ReadSpeakerLicense = file.Get("Paths","ReadSpeakerLicense","./");
string WorkingDirectory = file.Get("Paths","WorkingDirectory","./");
string AudioDir = file.Get("Paths","AudioDir","./");
string SpeakerName = file.Get("Engine","SpeakerName","noEngine");
string TypeName = file.Get("Engine","TypeName","noType");
string VisemesDir = file.Get("Paths","VisemesDir","./");

    if (SpeakerName == "noEngine") {
            return -1;
```

```
    }

    if (TypeName == "noType") {
            return -1;
    }

      int AudioFormat = file.GetInteger("Formats","AudioFormat",5);
      int TextFormat = file.GetInteger("Formats","TextFormat",0);
      int BufSize = file.GetInteger("Sizes","BufSize",-1);
      int TextSize = file.GetInteger("Sizes","TextSize",-1);

char *framebuffer = NULL;

      // puntero al fichero donde guardaremos los visemas

      FILE *visemeInfoFile = NULL;
      int size = 0;
      long total = 0;
      int i = 0;

      //objeto json donde guardaremos los visemas

      json visemeInfo;

      //iniciamos el handler de ReadSpeaker, el motor de voz y el syncHandler
a NULL

      VTAPI_HANDLE vtapi_handle = NULL;
      VTAPI_ENGINE_HANDLE engine = NULL;
      VTAPI_SYNC_HANDLE syncHandle = NULL;
      int wordCnt = 0;

    int nRet = 0;

    //establecemos el directorio de trabajo donde estarán las librerías
necesarias de ReadSpeaker

      nRet = VTAPI_Init(&WorkingDirectory[0]);

      //establecemos el path de la verificación del motor de voz

      VTAPI_SetLicenseFolder(&ReadSpeakerLicense[0]);

      //iniciamos el handler

      vtapi_handle = VTAPI_CreateHandle();
```

En esta fase se abre y lee `parámetros.ini` para conseguir los *paths* necesarios, el tipo y nombre del motor y algunos parámetros de funciones de ReadSpeaker. El método `get` de `inih` permite devolver un valor por defecto en caso de no encontrar un valor para el nombre buscado dentro del `.ini`. En el caso de los *paths* el valor por defecto será el

directorio en el que se encuentre el `cpp`, para los valores de ReadSpeaker están establecidos los valores por defecto de las funciones excepto para el tipo y el nombre ya que sin ellos no se puede identificar el motor instalado por lo que su ausencia provoca un error que termina la ejecución.

En esta fase también se crean el `handle` el `synchandle` y el `engine` que utilizaremos más adelante, se inicializa el `handle`, se establece el directorio donde se encontrarán las librerías que necesita ReadSpeaker y se establece el *path* donde encontraremos nuestra licencia para el motor de voz.

- Fase intermedia:

```
if (vtapi_handle != NULL){

        // iniciamos en motor de voz con el nombre y tipo de motor proveido
por ReadSpeaker

        engine = VTAPI_GetEngine(&SpeakerName[0], &TypeName[0]);
        if (engine <= 0)
            {

                  VTAPI_ReleaseHandle(vtapi_handle);
                  VTAPI_Exit();
                  return EXIT_FAILURE;
            }

            //establece el handler como handler del motor de voz

            nRet = VTAPI_SetEngineHandle(vtapi_handle, engine);

            if (nRet != VTAPI_SUCCESS)
        {

                  VTAPI_ReleaseHandle(vtapi_handle);
                  VTAPI_Exit();
                  return EXIT_FAILURE;
            }

            //establecemos la ruta del archivo de verificación necesario para
usar el motor de voz

            nRet = VTAPI_GetEngineLicensePath(engine, licenseFile);

            if (nRet != VTAPI_SUCCESS)
        {
```

```
                VTAPI_ReleaseHandle(vtapi_handle);
                VTAPI_Exit();
                return EXIT_FAILURE;
            }
```

En la fase intermedia se obtendrá el `engine` mediante `VTAPI_GetEngine` pasándole como parámetros el nombre y tipo del motor de voz que se obtienen de parameters.ini.

Se definirá a nuestro `handler vtapi_handler` como `handler` del `engine` con `VTAPI_SetEngineHandle` al que pasaremos como parámetros el `handler` y el `engine`.

Por último, obtiene la licencia asociada al `engine` con `VTAPI_GetEngineLicensePath`.

- Fase de generación de audio:

```
nRet = VTAPI_SetOutputFile(vtapi_handle, &AudioDir[0], AudioFormat);

        if (nRet != VTAPI_SUCCESS)
        {
                VTAPI_ReleaseHandle(vtapi_handle);
                VTAPI_Exit();
                return EXIT_FAILURE;
            }

        //generamos el audio a partir del texto de entrada

      nRet=VTAPI_TextToFile(vtapi_handle,textToProcess,TextSize,TextFormat;

        if (nRet != VTAPI_SUCCESS)
        {
            VTAPI_ReleaseHandle(vtapi_handle);
                VTAPI_Exit();
                return EXIT_FAILURE;
            }
```

Mediante `VTAPI_SetOutputFile` estableceremos donde se creará el fichero de audio correspondiente a nuestro texto y su formato.

A continuación, con `VTAPI_TextToFile`, generaremos el audio a partir del texto introducido, el tamaño del texto y el formato.

- Fase de obtención de visemas:

```
nRet = VTAPI_SetOutputBuffer(vtapi_handle, TextFormat);

            if (nRet != VTAPI_SUCCESS)
        {
                  VTAPI_ReleaseHandle(vtapi_handle);
                  VTAPI_Exit();
                  return EXIT_FAILURE;
            }

        //inicia el buffer para la síntesis de texto

      nRet=VTAPI_BufferOpen(vtapi_handle,textToProcess,TextSize,TextFormat,
BufSize);

            if (nRet != VTAPI_SUCCESS)
        {
                  VTAPI_ReleaseHandle(vtapi_handle);
                  VTAPI_Exit();
                  return EXIT_FAILURE;
            }

        int nSampling;

        //obtenemos la frecuencia de la muestra y la guardamos en nSampling,
en este caso es de 16Khz

nRet=VTAPI_GetEngineInfoFieldEx(engine,NULL,NULL,NULL,NULL,NULL,NULL,NULL,&
nSampling,NULL,NULL);

        if (nRet != VTAPI_SUCCESS)
        {
                  VTAPI_ReleaseHandle(vtapi_handle);
                  VTAPI_Exit();
                  return EXIT_FAILURE;
            }

            //el syncHandler obtiene la información del handler

            do
            {
                  framebuffer = NULL;

                  //obtiene el buffer de sintesis abierto previamente


      size=VTAPI_GetBuffer_Sync(vtapi_handle,&framebuffer,&syncHandle);
```

```
                if (size < 0)
                {
                      break;
                }

                if (size > 0)
                {
                      total += size;
                      VTAPI_FreeBuffer(framebuffer);
                }
                if (BufSize == VAL_ONEBUF)
                      break;

          } while (size > 0);

          //se obtiene el número de palabras del syncHandler

          wordCnt = VTAPI_GetWordCount(syncHandle);

          //recorremos las palabras y obtenemos los visemas

          for (i = 0; i < wordCnt; i++)
          {
                VTAPI_WORDINFO wordInfo;
                memset(&wordInfo, 0x00, sizeof(wordInfo));

                //obtenemos una palabra

                VTAPI_GetWordInfo(syncHandle, i, &wordInfo);

                int j;

                //obtenemos los fonemas

          Int phoneCnt = VTAPI_GetPhoneCount(syncHandle, wordInfo.nIndex);

                //recorremos los fonemas y obtenemos los visemas

                for (j = 0; j < phoneCnt; j++) {
                      VTAPI_PHONEINFO phoneInfo;
                      memset(&phoneInfo, 0x00, sizeof(VTAPI_PHONEINFO));

                      //se obtiene el visema


     VTAPI_GetPhoneInfo(syncHandle,wordInfo.nIndex,j,&phoneInfo);

                      //se obtiene su duración siguiendo la formula
longitud_del_phonema/frecuencia ya que estamos en monocanal

               double soundLenght = (double)phoneInfo.nLength/nSampling;

               //se introduce el par id visema-duración en el JSON con
formato [id,duración]

               visemeInfo.push_back({phoneInfo.nId_viseme,soundLenght});
                 }
          }
```

Esta fase comienza estableciendo la opción de salida del *handler* a `buffer` con `VTAPI_SetOutputBuffer`.

A continuación, con `VTAPI_BufferOpen` se inicia el `buffer` para procesar el texto que se le pasa por parámetro junto al tamaño del texto, el formato y el tamaño del `buffer`.

Mediante `VTAPI_GetEngineInfoFieldEx` podremos obtener información sobre el motor de voz, en nuestro caso nos interesa la frecuencia de la muestra que para nuestro motor son 16Khz y que se almacena en `nSampling`.

Proseguimos con el bucle por el medio del cual el `synchandler` obtiene la información del buffer del `handler` gracias a la función `VTAPI_GetBuffer_Sync`. Una vez terminado el bucle, el `synchandle` tendrá la información de todas las palabras del texto.

Para terminar, iteramos sobre las palabras que hemos obtenido en el `synchandler` y, para cada una de ellas, obtendremos el número de fonemas. Estos fonemas también serán iterados y obtendremos la información de cada uno gracias a `VTAPI_GetPhoneInfo` donde obtendremos el id del visema y su longitud. Siguiendo la fórmula longitud/frecuencia de la muestra, pues estamos en monocanal, se obtendrá la duración en segundos del visema. El par id-duración será añadido al `json` creado en la fase de inicialización.

- Fase final:

```
visemeInfoFile = fopen(&VisemesDir[0], "w");
            if (visemeInfoFile == NULL) {

                  VTAPI_BufferClose(vtapi_handle);
                  VTAPI_ReleaseHandle(vtapi_handle);
                  VTAPI_Exit();
```

```
                return EXIT_FAILURE;
            }

            //se transforma el JSON a string

        string serialized_string = visemeInfo.dump();

        //se escribe en el fichero

            fputs(&serialized_string[0],visemeInfoFile);

            //cerramos el fichero

            fclose(visemeInfoFile);

            //cerramos el buffer del handler

            VTAPI_BufferClose(vtapi_handle);

            //liberamos la memoria del handler

            VTAPI_ReleaseHandle(vtapi_handle);

      }

      //termina todo lo relacionado con ReadSpeaker

      VTAPI_Exit();


      return 0;

}
```

En la fase final se creará el archivo `json` indicando el *path* que obtendremos de `parámetros.ini` y se copiará el contenido del objeto `json` en el archivo.

Para terminar, se cerrará el buffer de ReadSpeaker con `VTAPI_BufferClose`, se liberará la memoria del `handler` con `VTAPI_ReleaseHandle` y se finalizará la sesión con `VTAPI_Exit`.

Si no se ha producido ningún fallo se devolverá un `0` que indica que el programa se ha ejecutado correctamente.

6.3.4 Implementación de proyectos externos

6.3.4.1 Implementación de nlohmann JSON

Este proyecto fue el que encontramos más recomendado para el manejo sencillo de los JSON. Una vez incluido los archivos necesarios se utilizó para dar formato al fichero JSON de salida, así como para obtener los campos dentro de las respuestas a las peticiones.

El fichero JSON de salida tiene la siguiente estructura:

```
[[3,0.095625],[19,0.06375],[1,0.178875],[31,0.2]]
```

Se trata de una lista donde cada par entre corchetes corresponde al id y a la duración de un visema.

6.3.4.2 Implementación de inih

Tras establecer que el objetivo de este proyecto sería ofrecer una interacción sencilla entre el usuario y los métodos y tras haber obtenido conocimiento sobre el funcionamiento de los métodos de ReadSpeaker y los parámetros necesarios para hacerlos funcionar, se decidió que los parámetros de estos métodos serían definidos en un archivo.ini para evitar así que el usuario tuviera manejar dichos parámetros.

Se realizó una búsqueda de proyectos de manejo de archivos `ini` y se eligió inih por su sencillez y claridad.

El fichero parámetros.ini solo debe ser modificado para añadir los *paths* requeridos, así como el tipo y nombre del motor de voz de ReadSpeaker y los parámetros de la petición al chatbot que son url, protocolo y token de verificación. El resto de los parámetros

solo deberán ser modificados en caso de que el usuario tenga conocimiento de cómo funcionan los métodos de ReadSpeaker.

A continuación, se muestran los parámetros que deben ser modificados por el usuario:

`ReadSpeakerLicense`: La ruta en la que se encuentre nuestro archivo de verificación de ReadSpeaker.

`WorkingDirectory`: La ruta en la que se encuentren las librerías de ReadSpeaker.

`AudioDir`: La ruta en la que se creará el archivo de audio. Debe incluir también el nombre del archivo y su extensión `.wav`.

`VisemesDir`: La ruta en la que se creará el archivo de texto con los visemas. Debe incluir también el nombre del archivo y su extensión `.json`.

`SpeakerName`: Nombre del motor ReadSpeaker.

`TypeName`: Tipo del motor ReadSpeaker.

`Url`: Url del Chatbot al que queremos conectarnos.

`Protocol`: Protocolo de aplicación usado.

`Token`: Token de identificación de la conexión.

Para las pruebas del proyecto se usaron los siguientes parámetros.

```
[Paths]
ReadSpeakerLicense=./
WorkingDirectory=./
AudioDir=./audio.wav
VisemesDir=./visemas.json
```

```
[Engine]
SpeakerName=lola
TypeName=p16

[Formats]
AudioFormat=5
TextFormat=0

[Sizes]
BufSize=-1
TextSize=-1

[Conexion]
Url = https://f49411fb-07b7-41da-b3f7-f5d0ebd6803e.mock.pstmn.io/Tiempo
Protocol = https
Token = token
```

6.3.4.3 Libcurl

Es probablemente el proyecto utilizado para enviar peticiones en C++ más conocida y no necesitó de comparación con otros proyectos similares. Gracias a libcurl se realizan las peticiones al chatbot.

6.3.5 Implementación en otros proyectos

Para cualquier otro proyecto se proporcionará la librería estática libtts.lib que contendrá la implementación de los métodos, así como el *header* `tts.h` el cual requiere.

6.3.6 Entregables

Se hará entrega de la librería `libtts.a` que contiene la implentación de los métodos, el *header* `tts.h` que contiene las declaraciones de los métodos, el archivo de configuración parameters.ini que contiene parámetros usados por los métodos, los archivos *source* que han sido compilados para crear la librería `libtts.a` y todos los archivos necesarios para el correcto funcionamiento del proyecto.

7 Conclusiones y líneas futuras

Creemos que este campo tiene muchas posibilidades de expansión y muchísimas aplicaciones útiles. El objetivo inicial fue dotar de habla y animación realista al avatar del ChatBot perteneciente a nuestro tutor, pero pensamos que puede expandirse a: otros chatbots, animaciones de guías virtuales, animaciones de personajes de videojuegos, a la industria del cine, a clases virtuales, a realidad virtual y a numerosas opciones más.

Al ser un campo tan novedoso puesto que, a día de hoy, no conocemos ningún otro proyecto que haga lo mismo, pensamos que tiene un gran potencial de mejora y que puede servir como impulsor del campo.

Por suerte hemos contado con buena documentación en general excepto en el caso de Unreal, donde no estaba demasiado detallada y provocó varios retrasos en el avance.

8 Ilustraciones

9 Bibliografía

[1] B. Hoyt, «GitHub inih,» [En línea]. Available: https://github.com/benhoyt/inih.

[2] libcurl, «libcurl,» [En línea]. Available: https://curl.se/libcurl/.

[3] Unreal Engine, «Recording Facial Animations from an IOS Device,» [En línea]. Available: https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/FacialRecordingiPhone/.

[4] N. Lohmann, «GitHub nlohmann JSON,» [En línea]. Available: https://github.com/nlohmann/json.

[5] ECMA International, «ECMA-404,» [En línea]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.

[6] Postman, «Postman,» [En línea]. Available: https://www.postman.com/.

[7] ReadSpeaker, «ReadSpeaker,» [En línea]. Available: https://www.readspeaker.com/es/.

[8] libcurl, «Releases and Downloads,» [En línea]. Available: https://curl.se/download.html.

[9] libcurl, «curl Download Wizard,» [En línea]. Available: https://curl.se/dlwiz/.

[10] libcurl, «The libcurl API,» [En línea]. Available: https://curl.se/libcurl/c/.

[11] libcurl, «curl_easy_cleanup - end a libcurl easy handle,» [En línea]. Available: https://curl.se/libcurl/c/curl_easy_cleanup.html.

[12] N. Lohmann, «JSON for Modern C++,» [En línea]. Available: https://json.nlohmann.me/.

[13] Asperos Geek, «Como crear un Mock Server en Postman con un ejemplo claro,» [En línea]. Available: https://www.youtube.com/watch?v=WtT1ZhXNYWU&t=491s.

[14] Unreal Engine, «MetaHuman Creator,» [En línea]. Available: https://www.unrealengine.com/en-US/metahuman-creator.

[15] M. L. Morales-Rodríguez, J. A. Radilla-Avila, A. Hernández Ramírez, J. A. Ramírez-Saldivar, J. J. González Barbosa y H. J. Fraire-Huacuja, «Silabeo Automático para la Generación de Vi-Silabas en Español,» de *17th International Congress on Computer Science Research (CIICC'11)*, Morelia, Michoacán, 2011.

[16] Amazon, «Amazon Polly,» [En línea]. Available: https://docs.aws.amazon.com/es_es/polly/latest/dg/ph-table-spanish.html.

[17] Microsoft, «SPVISEMES (SAPI 5.4),» [En línea]. Available: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ee431873(v=vs.85).

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga