Speech Animation for TalkingCoach

Report on Architecture Design for TI2806 Context Project

EPVA4 - Speech Animation

Erik Mekkes, emekkes, 4100190
Lucile Nikkels, lrmnikkels, 4564804
Emma Sala, esala, 4555910
Joshua Slik, jslik, 4393007
Muhammed Imran Özyar, mozyar, 4458508

Project under guidance of:
Willem-Paul Brinkman
Romi Kharisnawan

Delft University of Technology
The Netherlands
June 26, 2018

Abstract

TalkingCoach is a software framework which software developers can use to create a virtual character which can speak in natural language (de Jong, 2017). This project focuses on extending the human-like capabilities of the virtual agent, like its ability to make fitting body gestures and facial expressions, and to make the correct lip movements when speaking.

This document provides an overview of the design goals of the system and its architecture from various design perspectives. Perspectives covered by this document include a subsystem decomposition, a mapping between hardware and software, an overview of how the data required by the system is managed, and finally a description of how concurrency aspects are managed within the system.

Contents

Chapter 1
Introduction

Designing an architecture for software is key for its development, deployment and maintenance. For the TalkingCoach Agent, the additional software developed is aimed to provide lip-synchronization. This Virtual Agent has been programmed within Unity 5.6.6f2 and can be interacted with using JavaScript calls from within a browser (WebGL).

Unity has more support for scripted animations and the WebGL build comes in very handy, making it possible to make it directly available on the internet, such that it can be viewed and tested on almost all platforms. This is why Unity will be used along with the WebGL build for this project.

Design Goals

The ultimate design goal is to have a stable, yet flexible architecture such that it supports and fosters the workflow in a way that eventual changes in the developed software, which may happen to be big, should be supported by the environment. For example, a change in the algorithm, which may involve reading from online streams, should be supported by the environment and should also be testable in the language provided by the environment.

In the case of Unity along with WebGL, all modifications to the architecture design should be built on top of the current environment, in a way that the modification is minimal. If modifications to the architecture would really be impacting the original design, a case could appear where basic, already implemented functionality would not be working since they would not be supported by the modified environment. It would also be harder to adapt to the renewed environment and to continue the work within it, since a lot of time goes into getting familiar with the environment. Ideally, it is desirable for the initial architecture to stick for all latter versions of the developed software. This could, however, not be the case if different approaches are used for the algorithm.

The aimed architecture can be found in figure A.1 (see Appendix A). The Unity build will generate WebGL files. These files, along with a NodeJS server and the text-to-phonemes API, will form the server. The server serves a web page, containing a WebGL container. This container will display the virtual agent. When text input is entered through the web page, this input will get transcribed to phonemes, through a direct request to the NodeJS server. Those phonemes will be converted to visemes through another algorithm, after which the durations of the visemes will be calculated through the VTC (Viseme Timing Calculation) algorithm. The reason that a VTC algorithm has been constructed is that the used text-to-speech (TTS) engine does not contain the functionality to have callbacks/async functions to let the user know where the speech is in its pronunciation. During calculation, the virtual agent will start the speech. While this happens, the lipsync algorithm continuously checks whether a viseme has been pronounced. When this happens, the algorithm requests the next viseme and its animation. The animation will then be rendered on the agent. Ideally, it all takes place synchronized in real time.

Chapter 2
Software Architecture Views

This chapter provides an overview of the system architecture from various design perspectives, starting with a description of the composition of the system, followed by a mapping from the software to executed processes and hardware, along with a description of the communication. Finally, a section on data management and concurrency aspects within the system is included.

2.1 Subsystem Decomposition

This product extends the TalkingCoach virtual agent with speech animation, the final product of which is a WebGL build produced by the Unity editor. This section gives an overview of the subsystems of this product and the dependencies between them.

2.1.1 Unity Classes

All interactions with the 3D model are performed by C# scripts written within Unity. The design of the product includes an API that exposes specific actions for the avatar, such as speaking, changing models or moving models. The classes implementing the lip synchronization API have been separated over three folders: 'Models', 'JSON' and 'Synchronization'. The 'Models' folder holds the phoneme and viseme classes, enabling conversion from phonemes to visemes in a clean, yet easy way; the 'JSON' folder provides the necessary JSON interaction for the use of the API (text-to-phonemes response); and the 'Synchronization' folder holds all the base information and procedures for the lipsync algorithm. The main function, which sets everything in motion, is the LipSynchronization:synchronize(text, lang) function. This function sends a request for the phonemes of the inputted text in the language specified in the language paramter, converts the phonemes to visemes and then plays all the visemes using the SpeechAnimationManager.

Viseme Animation Interface The product should be accessible to other developers that wish to improve it or incorporate it into their own web application(s). Since it isn't efficient to use Unity's interface for the specification of over dozens of visemes, a change has been made during the development process, wherein the visemes are loaded through the file system.

Viseme Duration Interface Within the same interface, the user of the product can now also select their own custom file for viseme durations. This field makes sure the user can easily implement new viseme durations for different languages.

2.1.2 Text to Phoneme conversion

The application receives text input from the user through the web page. The first step in producing speech animation is to convert this text to the right phonemes, depending on the language selected by the user on the web page. To implement this conversion, an external open source application will be used that takes these inputs and produces a representation in phoneme representation as output. The open source application will run on the NodeJS server, as depicted in figure A.1. Functions in the TalkingCoach product will be created to call upon this application and handle its output. The style of phonemes will eventually be ARPABET. IPA, another style of phonemes, can be translated to ARPABET.

2.1.3 Phoneme to viseme conversion

Phonemes represent distinct units of audible speech. Distinct units of visible speech are also required for the application; visemes. Included within the application is a fixed set of visemes with animations per supported language (viseme animations can also overlap between languages). The conversion from phonemes to visemes will be implemented as a function within the NodeJS server that takes a set of phonemes as input and produces a set of visemes as output. This function uses a mapping based on known rules for phoneme to viseme conversion. The visemes output will get sent to the client, thus the web page, where the visemes will be converted into numbers, such that they are recognizable by the application, in order for the application to find the correct animations for the visemes.

2.1.4 Viseme Timing Calculation

The VTC takes in the generated visemelist and calculates the duration per viseme, based on a pre-written file that contains a map of every used viseme and the duration of its corresponding pronunciation. The VTC then makes this dictionary available for all other classes. The Viseme class accesses this information such that the viseme that is held in that class knows what its duration is, such that it can be retrieved easily when playing the viseme's animation and holding on to its duration.

2.1.5 Lifecycle

A use-case scenario is depicted in appendix B.1. The user has a plain text input. This input is entered through Javascript, after which the Unity back-end is called. The TalkingCoachAPI class calls the TextManager class, which played the synthetic speech right away in the original product. However, in the extended version, the TextManager sends a lip synchronization request to the LipSynchronzation class. This class makes a text-to-phoneme request to the NodeJS server, which in its turn returns a response. It then proceeds to request a phoneme-to-viseme conversion from the implemented models (along with the mappings). When the viseme response is returned, a direct request for the animations is made to the SpeechAnimationManager. Then, the speech is played directly through the TextManager, while the SpeechAnimationManager plays the animations of the Virtual Agent.

2.1.6 Web elements

HTML, WebGL and JavaScript The product includes an html+css+javascript web page that is generated from a template within Unity. This web page includes input methods such as buttons and input fields, as well as a WebGL graphics window showing the avatar model in its environment. Unity contains a WebGL build option that allows Unity to publish content as JavaScript programs, which use HTML5 technologies and the WebGL rendering API to run Unity content in a web browser. (Unity-Technologies, 2018a). The input methods on the web page call JavaScript functions included with the web page, which send commands to the Unity script API that can control the model. This link between JavaScript and Unity scripts is created by the Unity WebGL plugin (Unity-Technologies, 2018b).

Web Speech API In order to produce speech the product makes use of the Web Speech API implemented within most web browsers. The browser produces audible speech using its built in functions that are exposed through this API. These API functions can be called by JavaScript code of a web page. The link provided by the WebGL plugin mentioned above also works the other way around. The scripts within Unity can call upon browser JavaScript functions such as the Web Speech API.

2.1.7 Components of the Virtual Agent

The final product is a Virtual Agent that should appear, move and sound human-like. To achieve this visually, models of the agent's body and its environment, and definitions for animation of this body model within the environment are required.

Models The final product includes 3D representations of (a) human-like character(s), as well as various environments that they can move around in. These characters are represented by structured parts that form a human-like body model. These model parts are created, connected and positioned within the Unity editor or by using an external program such as Maya. In order for the virtual character to have a human-like appearance, this human-like model needs to be accurate. The environment or scene for the character to move around in currently consists of a chosen character model and a two-dimensional background image that creates the illusion of having the character positioned within the scene.

Animations For the characters to be human-like, they should not only be structured, but also move in a human-like manner. For each model part a set of key control points are defined of which the position can be altered in 3D space relative to the model's position. When the position of these joints are changed, the model part moves and stretches or shrinks to the new position. A sequence of these relative position changes for a set of points can be stored as a single animation, which allows for easily manageable and very fine-grained control of model movement.

Viseme Animations A language can be animated using a set of visemes, which are distinct representations of mouth movements. A set of viseme animations is included for each language included in the application. These animations have a set base duration that roughly matches the expected duration of the spoken sound. They consist of a single position that best represents a viseme. When called upon the mouth control, joints move to the specified position over the set duration. It is possible to alter or replace these viseme animations using the provided interface in the Unity Editor.

2.2 Hardware and Software Requirements

The product system is designed for two different groups; users who will interact with the virtual agent and developers who will build their own systems using the virtual agent as a base. This also results in two scenarios of hardware and software being used. The hardware involved in a deployed product for users consists of two types: clients and servers. The product is also available to developers, which requires version control / integration machines.

Frontend server A NodeJS React server, running on Delft University of Technology hardware, contains a React UI and build version of the Unity product as data, which includes JavaScript, HTML, CSS and compressed versions of the models, scripts and animations. The React server responds to requests made by clients by sending them the build version of the product with a user friendly UI around the Unity build. It should be configured to use compression and have access to significant bandwidth as the full set of build files approaches 22MB in it's current state.

Backend server When the client interacts with the WebGL build and the front end webpage, the entered text is sent to a NodeJS Express backend server, running on Delft University of Technology hardware, in order to translate the text to phonemes. The phonemes that are sent back, are translated by the compiled Unity WebGL in the client's browser to visemes. Delft University of Technology is in possession of the server as long as the project is only used within the university. As soon the project is used by another company, it should set up its own front and backend servers in order to satisfy the security and privacy of their clients's data.

System requirements for servers Servers are tested on Windows 10 1803 and Ubuntu 16, though should work on all operating systems supported by eSpeak-NG and Node.js. Servers require the latest version of the eSpeak-NG software with the executable added to their path and at least Node.js version 10.2.x.

Client The software is built in such a way that the only requirement for the client is a modern desktop-based web browser and a network connection to the frontend/backend servers. The web browser should support recent versions of HTML, CSS and JavaScript, WebGL graphics and the Web Speech API. Internet Explorer and mobile browsers do not currently support WebGL, but the latest versions of Google Chrome, Mozilla Firefox and Microsoft Edge are supported. When pointed to the server, the browser requests the build version from the server and produces the interactive web page for the user. The user interacts with the data by typing sentences and clicking on buttons.

Developer Machine A developer machine is capable of modifying the product. Required for this, is a copy of the source code, preferably from a Version Control System (Git). The machine should include version 5.6.6f2 of the Unity editor and an editor for the C# scripts and web content. The project team suggests the latest version of JetBrains Rider, but Visual Studio should also work. The machine should also include at least one of the browsers mentioned above in order to perform local tests and debugging. The developer machine should also have at least Node version 10.2.x and npm version 6.1.x. If the developer wants to test the backend server on their machine, they should also have the most recent version of eSpeak-NG installed and the executable on their path, according to installation instructions.

Version Control / Integration Machines For version control a git service such as GitHub or GitLab is suggested. Included with these version control systems are project management tools, which can further aid in managing the development process. Ideally the version control system should include an automated build and testing system.

2.3 Persistent Data Management

Joint positions For the lipsyncing part of the TalkingCoach, 40 different positions of the mouth are stored, corresponding to 40 different viseme-animations. For each position, the following joints are important; upper teeth, lower teeth, tongue, left and right cornerlip, 9 joints for the lower lip and 11 joints for the upper lip. Each joint consists of the transformation, scale and rotation in the x-, y- and z-axis, which are applied to the original position of silence. The positions are acquired through modeling and exported to Unity, saved as animationfiles. No external database is needed for the small amount of data.

Algorithms Of course, a text-to-phonemes, a phoneme-to-viseme and a viseme timing calculation algorithm should be used, which should also be static after it has been developed or used. Modifying either the algorithm or the positions for the lower face and jaw will result in high inaccuracy for the software. The text-to-phonemes algorithm requires command line commands, but since a web browser does not have the rights to access the command line, a NodeJS server will be used, with the text-to-phonemes API stored on the server side. These algorithms are depicted in figure A.1.

Coaches When forgetting about the feature of lipsyncing and looking at the whole codebase, the backgrounds and coaches also need to be stored somewhere. As of now, there are only two backgrounds and six different coaches, which are stored as scenes and prefabs in Unity. Because of the small amount of data, no external database is used.

2.4 Concurrency

Jobs within any modern operating system are typically performed by processes that execute concurrently. This concurrency often proves difficult for a system's design, introducing challenges such as deadlocks, starvation and problems with priority, timing and shared resources in general.

Threading One source of these problems is multi-threading, where a process shares its data and execution context among separately executing instances to perform its job. Threads are not supported in WebGL due to the lack of threading supporting in JavaScript (Unity-Technologies, 2018a), so this removes some aspects of concurrency management, but is also a slight drawback in terms of efficiency. A small bit of threading could have been implemented in the product, since we have a running server on which the text-to-phonemes and phonemes-to-visemes algorithms depend on. So during a call to the server, the application gets a small amount of time to run processes in parallel, yet, this is currently not the case. In the current model, the application waits for a response from the server, since it wouldn't be able to do anything without one, so the model works sequentially and not parallel.

The VA can be seen silent after clicking on the 'Speak' button. The length of this silence in linear with the length of the input text. The longer the input will be, the longer the silence. This silence is introduced because of the wait for the server's response, since this is a sequential process. This will, with high probability, be left as a sequential process, since there is nothing the system can process while waiting on the server's response.

Communication The designed system does involve separate processes within the browser, so careful design decisions still need to be made to ensure that these processes communicate with each other in a safe and timely manner.

Chapter 3
Glossary

Included here is a list of terms and concepts used within this document.

- **API** : Application Programming Interface, a set of clearly defined methods of communication between various software components.
- **Animation** : Specifically 3D animation, the movement of a 3D model.
- **Build** : Process of converting source code into software artifact(s) that can be run on a computer.
- **Continuous Integration** : Development practice that requires developers to integrate code into a shared repository, verified by an automated build, allowing teams to detect problems early.
- **Git** : A versioning control system supporting branching.
- **Phoneme** : A phoneme is a group of sounds consisting of an important sound of the language (i.e. the most frequently used member of that group) together with others which take its place in particular sound (Twaddell, 1935).
- **TalkingCoach** : Unity-based software framework which software developers can use to create a virtual character which can speak in natural language.(de Jong, 2017)
- **Unity** : Unity is a cross-platform game engine developed by Unity Technologies, which is primarily used to develop both three-dimensional and two-dimensional video games and simulations.
- **Virtual agent**: In computer science, the term 'agent' usually applies to a smart piece of software that can perform tasks in a somewhat intelligent way. In a virtual environment, the agent is bound to virtual physical constraints, like a body. ("Embodied Agents", 2018)
- **Virtual avatar**: An icon or figure representing a person, or some of the physical characteristics of a person on a display screen. (Ross, 2016)
- **Viseme**: A viseme is a virtual speech which has a unique lip shape of mouth so it is similar to phoneme of speech domain. But between phoneme and viseme there exists m to 1 relation, but not the 1 to 1 relation (Jong Won, Gwang Chol, Hyok Chol, & Kum Song, 2014).
- **WebGL** : WebGL (Web Graphics Library) is a JavaScript API for rendering interactive 3D and 2D graphics within any compatible web browser without the use of plug-ins

Bibliography

de Jong, R. (2017, May). Talking coach. Retrieved May 2, 2018, from https://github.com/ruuddejong/TalkingCoach/

Embodied Agents. (2018). http://ai-depot.com/GameAI/Embodied.html. Accessed: June 2018.

Jong Won, H., Gwang Chol, L., Hyok Chol, K., & Kum Song, L. (2014). Definition of visual speech element and research on a method of extracting feature vector for korean lip-reading. *College of Computer Science*, 2.

Ross, F. (2016). Fashion-technology and change in product development and consumption for the high-end menswear sector. *3D Printing*, 251–280. doi:10.4018/978-1-5225-1677-4.ch014

Twaddell, W. F. (1935). On defining the phoneme. *Language*, *11*(1), 5–62. Retrieved from http://www.jstor.org/stable/522070

Unity-Technologies. (2018a, April). Getting started with webgl development. Retrieved May 2, 2018, from https://docs.unity3d.com/Manual/webgl-gettingstarted.html

Unity-Technologies. (2018b, April). Webgl: Interacting with browser scripting. Retrieved May 2, 2018, from https://docs.unity3d.com/Manual/webgl-interactingwithbrowserscripting.html

Appendix A

Targeted Architecture

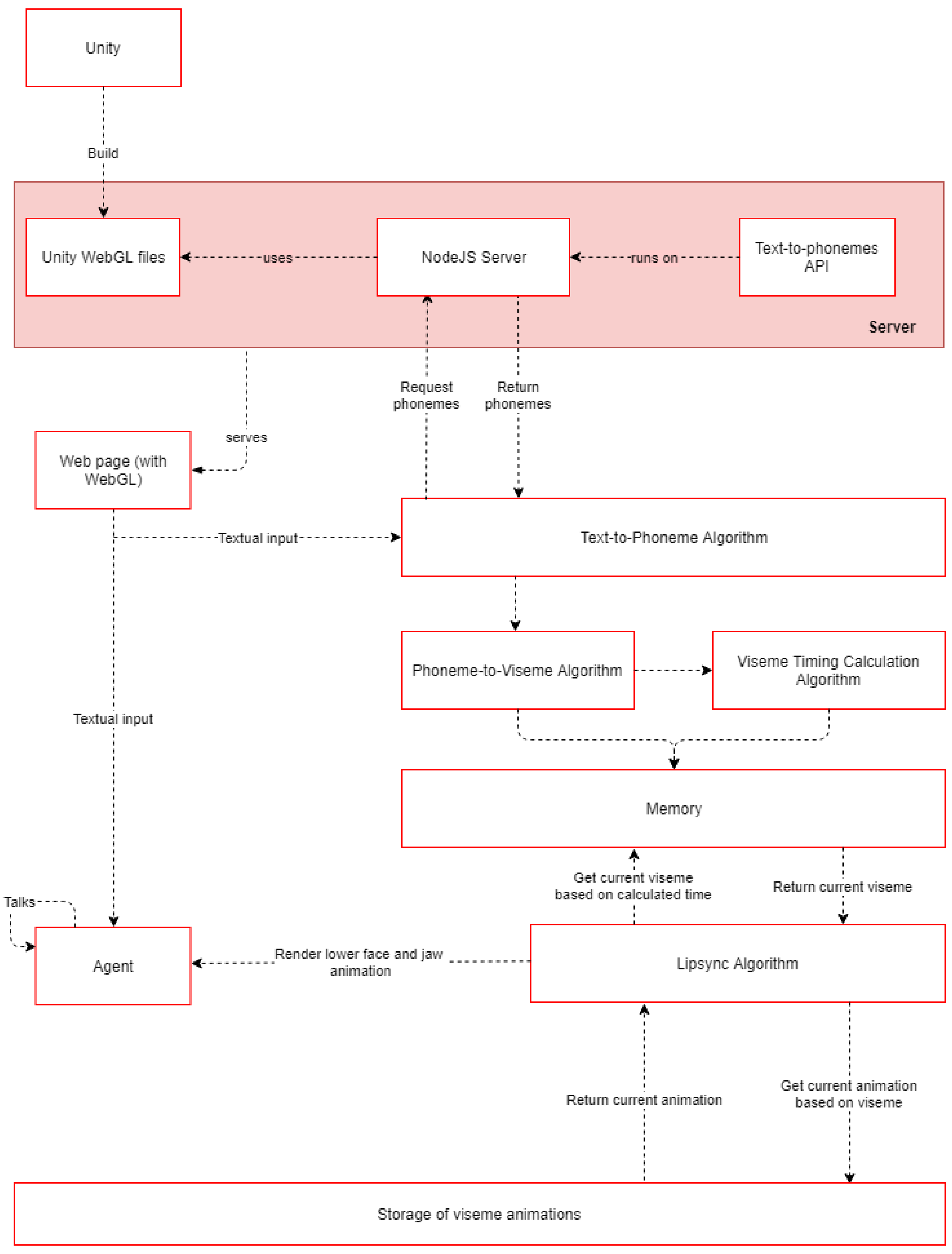

Figure A.1: The targeted architecture

Appendix B

Use case life cycle

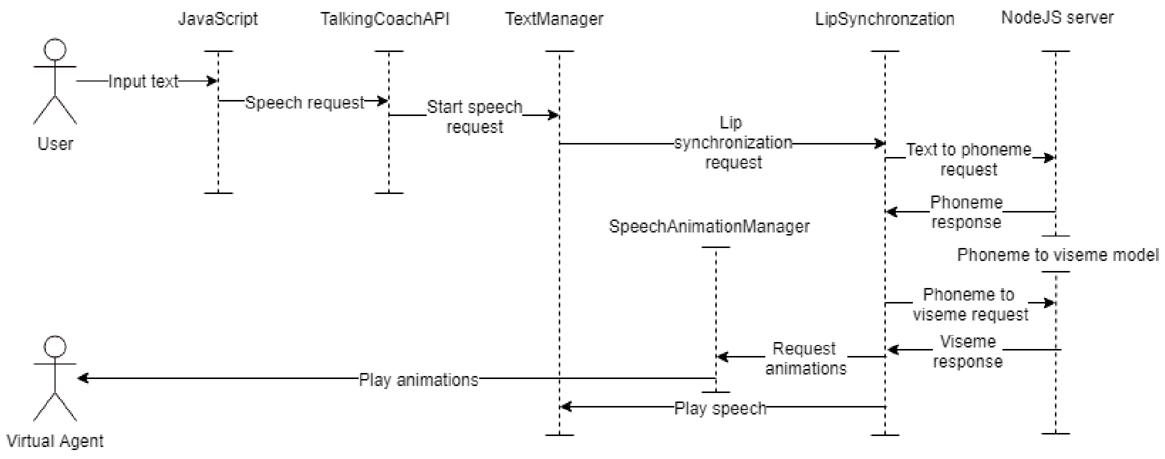

Figure B.1: The life cycle of a use case

Appendex C

API for Speech Animation (EPVA4)

For Unity developers wishing to modify the product

Speech animation can be controlled with the SpeechAnimationManager class located in the scripts folder. It provides the functions listed below, which can be called from the runtime SpeechAnimationManager instance as : SpeechAnimationManager.instance.functionName

public void loadCoach(GameObject coach)
Specifies the GameObject for which speech animations should be played, this allows SpeechAnimationManager to access Animations.
This GameObject must contain an Animator component, which must contain an animation layer for speech. The animation layer must be specified using it's index with the setVisemeLayer() function (defaults to layer 1). The animation layer should contain animation states for each of the Visemes listed in the Viseme documentation.

SpeechAnimationManager currently only a allows single GameObject to have active speech animation.

Suggested use : To be called from ApplicationManager function after a new GameObject has been loaded.

public void frameUpdate(float lastFrameDuration)
Notifies the SpeechAnimationManager that a new frame is being rendered, with the duration of the previous frame. This is used by SpeechAnimationManager to decide when to play the next animation.

Suggested use : To be called from the Update function from ApplicationManager with Time.deltaTime.

public void setVisemeLayer(int layerIndex)
Updates the layer in the Animator component where SpeechAnimationManager looks for viseme animations.
Suggested use : To be called when Unity has loaded, from Unity's start function for example.

public void setText(string text)
Sets the text that should be animated for the SpeechAnimationManager
Suggested use : To be called from the StartSpeech function called in Textmanager.

public void onBoundary(int charIndex)
Set most recently spoken word for SpeechAnimationManager.
Suggested use : To be called from the onboundary event generated by the web speech API

public void startSpeechAnimation(int charIndex)

Instruct SpeechAnimationManager to start speech animation.
Suggested use : To be called from the startevent generated by the web speech API

`public void stopSpeechAnimation(int charIndex)`
Instruct SpeechAnimationManager to stop speech animation.
Suggested use : To be called from the end event generated by the web speech API

`public void pauseSpeechAnimation(int charIndex)`
Instruct SpeechAnimationManager to pause speech animation.
Suggested use : To be called from the pause event generated by the web speech API

`public void resumeSpeechAnimation(int charIndex)`
Instruct SpeechAnimationManager to resume speech animation.
Suggested use : To be called from the resume event generated by the web speech API

`public void pauseSpeech()`
Instruct SpeechAnimationManager to pause speech synthesis and animation. Relies on onboundary events which browsers do not generate for some languages.
Suggested use : To be called through TalkingCoachAPI

`public void resumeSpeech()`
Instructs SpeechAnimationManager to resume speech synthesis and animation.
Suggested use : To be called through TalkingCoachAPI after a call to pauseSpeech()

Web backend API documentation

API url: `<hostname>:<port|3001>/api/v1/`

Translate a string to phonemes for a specific language

HTTP Request

`GET /api/v1/phoneme`

Query Parameters

Parameter	Default	Options	Description
text	N/A, mandatory	Any string	Text to be translated to phonemes
lang	N/A, mandatory	`en-us`, `nl`	Language for transcriber

Example

Translate the text `Hello World` to phonemes for the English US language:
`localhost:3001/api/v1/phoneme?text=Hello World&lang=en-us`